

**DESARROLLO DE UN PROTOTIPO DE UNA RED INTSERV6 EN EL
SIMULADOR NS-2**

**CAROLINA JIMÉNEZ GUTIÉRREZ
JORGE ALEXANDER BOTIA GONZÁLEZ**

**UNIVERSIDAD PONTIFICIA BOLIVARIANA
SECCIONAL BUCARAMANGA
ESCUELA DE INGENIERIAS Y ADMINISTRACION
INGENIERIA ELECTRONICA
BUCARAMANGA
2008**

**DESARROLLO DE UN PROTOTIPO DE UNA RED INTSERV6 EN EL
SIMULADOR NS-2**

**CAROLINA JIMÉNEZ GUTIÉRREZ
JORGE ALEXANDER BOTIA GONZÁLEZ**

**TRABAJO DE GRADO PRESENTADO COMO REQUISITO PARA OPTAR
AL TÍTULO DE INGENIERO ELECTRONICO**

**Director
Mcs. JHON JAIRO PADILLA AGUILAR**

**UNIVERSIDAD PONTIFICIA BOLIVARIANA
SECCIONAL BUCARAMANGA
ESCUELA DE INGENIERIAS Y ADMINISTRACION
INGENIERIA ELECTRONICA
BUCARAMANGA
2008**

Nota de aceptación:

Presidente del jurado

Jurado

Jurado

Ciudad y fecha:

Primero que todo a Dios, a
nuestras familias y amigos
que no dudaron de nuestras
capacidades y nos apoyaron
para seguir adelante hasta
esta etapa del camino de
nuestras vidas

AGRADECIMIENTOS

Con estas líneas pretendemos agradecer a todas las personas que de alguna forma nos ayudaron, nos animaron y estuvieron pendientes de este proceso tan importante en nuestras vidas.

Primero que todo queremos agradecerle a nuestro director de tesis, Jhon Jairo Padilla Aguilar, por su apoyo, a todos nuestros compañeros de carrera que estuvieron al tanto del proceso, a nuestros padres que tuvieron tanta paciencia, a todos nuestros familiares que nos dieron apoyo cuando ya estábamos cansados.

INDICE

1. INTRODUCCIÓN.....	1
1.1. Objetivos.....	1
1.1.1. Objetivo General.....	1
1.1.2. Objetivos Específicos.....	1
1.2. Justificación.....	2
1.3. Resumen.....	3
2. MARCO TEÓRICO.....	10
2.1. Calidad de Servicio en Internet (QoS).....	10
2.1.1. Introducción.....	10
2.1.1.1. Antecedentes.....	10
2.1.1.2. Origen de Calidad de Servicio.....	11
2.1.2. Tipos de Tráfico.....	12
2.1.3. Parámetros de Calidad de Servicio.....	13
2.1.3.1. Ancho de Banda.....	14
2.1.3.2. Retraso Temporal.....	14
2.1.3.3. Probabilidad de Error.....	15
2.1.3.4. Jitter.....	15
2.1.4. Soluciones de Calidad de Servicio.....	16
2.1.4.1. Servicio Best-Effort.....	17
2.1.4.2. Servicios Integrados.....	17
2.1.4.3. Servicios Diferenciados.....	19
2.2. Arquitectura de Servicios Integrados (IntServ).....	24
2.2.1. Introducción.....	24
2.2.2. Principios Básicos.....	25
2.2.3. Componentes Claves.....	25
2.2.4. Protocolo de Señalización RSVP.....	27
2.2.4.1. Control de la Reserva.....	29
2.2.4.2. Estilo de la Reserva.....	31
2.2.5. Control de Admisión.....	31
2.2.5.1. Introducción.....	31
2.2.5.2. Aproximaciones de Control.....	31
2.2.6. Identificación de Flujos.....	32
2.2.7. Planificación de Paquetes.....	34
2.2.7.1. Disciplina de Planificación de Cola WFQ.....	35
2.2.8. Modelos de Servicio.....	37
2.2.8.1. Especificación de Flujos.....	38
2.2.8.2. Servicio Garantizado.....	39
2.2.8.3. Servicio de Carga Controlada.....	39
2.2.8.4. Algoritmo de Token Bucket.....	41
2.2.9. Ventajas del Modelo IntServ.....	42
2.2.10. Desventajas del Modelo IntServ.....	42
2.3. Protocolo de Internet Version 6.....	43

2.3.1.	Introducción a IPv6.....	43
2.3.1.1.	Nuevas Características de IPv6.....	44
2.3.2.	La Cabecera IPv6.....	45
2.3.3.	El Campo de Siguiete Cabecera (Next Header Field).....	46
2.3.4.	Direccionamiento IPv6.....	48
2.3.5.	Modelos de Direccionamiento.....	49
2.3.6.	Autoconfiguración.....	50
2.3.7.	Movilidad.....	51
2.4.	Propuesta de IntServ6.....	52
2.4.1.	Clasificación de Flujos en IntServ6.....	52
2.5.	NS2 – Network Simulator.....	54
2.5.1.	Introducción.....	54
2.5.1.1.	Instalación del NS-2.....	54
2.5.2.	Descripción Interna del NS-2.....	55
2.5.2.1.	Scripts de Entrada.....	56
2.5.2.2.	Event Scheduler Object (Planificador de Eventos).....	56
2.5.2.3.	Network Component Object.....	57
2.5.2.4.	Network Setup Holding Module.....	58
2.5.3.	Ejecución de un Script.....	58
2.5.4.	Nam.....	59
2.5.5.	Xgraph.....	60
2.5.6.	RSVP/NS.....	61
2.5.6.1.	Configuración de un enlace RSVP.....	61
2.5.6.2.	Creación y Configuración de Agentes RSVP.....	62
2.6.	Programación Orientada a Objetos.....	63
2.6.1.	Introducción.....	63
2.6.2.	Definición.....	63
2.6.3.	Clases y Objetos.....	64
2.6.4.	Mensajes y Métodos.....	65
2.6.5.	Constructores.....	66
2.6.6.	Destrucción.....	66
2.6.7.	Herencia.....	67
2.7.	Programación C++.....	68
2.7.1.	Introducción.....	68
2.7.1.1.	Historia del Lenguaje C++.....	68
2.7.2.	Comentarios de una sola línea de C++.....	69
2.7.3.	Flujo de Entrada/Salida de C++.....	70
2.7.4.	Declaraciones de C++.....	71
2.7.5.	Creación de nuevos tipos de datos en C++.....	71
2.7.6.	Prototipos de función y verificación de tipo.....	72
2.7.7.	Funciones en Línea.....	72
2.7.8.	Parámetros por Referencia.....	73
2.7.9.	Asignación Dinámica de Memoria mediante new y delete.....	74
2.8.	Programación OTcl.....	75
2.8.1.	Introducción a Tcl.....	75

2.8.2.	Características de Tcl.....	75
2.8.3.	Variables y Valores.....	76
2.8.4.	Estructuras de Control.....	77
2.8.5.	Operaciones con Cadenas de Texto.....	78
2.8.6.	Arrays.....	79
2.8.7.	Listas.....	79
2.8.8.	Procedimientos.....	81
2.8.9.	Comandos de Entrada/Salida.....	82
2.8.10.	Otros Comandos.....	83
3.	DESARROLLO DE LA TESIS.....	84
3.1.	Metodología de la Tesis.....	84
3.2.	Investigación y Documentación.....	85
3.3.	Capacitación sobre Lenguajes del Simulador NS-2.....	86
3.4.	Capacitación en Herramientas de Simulación.....	87
3.5.	Diseño y Construcción de los Modelos de Simulación.....	88
3.5.1.	Descripción del proceso de Clasificación en IntServ.....	93
3.5.1.1.	Clasificación en IntServ.....	93
3.5.1.2.	Clasificación en IntServ6.....	95
3.5.2.	Desarrollo del Protocolo en C++.....	95
3.5.2.1.	Diseño del Clasificador IntServ.....	97
A.	Módulo del Clasificador en C++.....	100
B.	Cambios Realizados al Simulador NS-2.....	114
3.5.2.2.	Diseño del Clasificador IntServ6.....	130
A.	Modulo HostOrigen.....	133
B.	Módulo del Clasificador IntServ6 en C++.....	133
C.	Cambios Realizados al Simulador NS-2.....	147
3.5.3.	Desarrollo del Protocolo en OTcl.....	159
3.5.3.1.	Planificador WFQ.....	160
3.5.3.2.	Diseño de la Red IntServ.....	162
3.5.3.3.	Diseño de la Red IntServ6.....	172
4.	PRUEBAS REALIZADAS Y RESULTADOS.....	180
4.1.	Pruebas Realizadas a los Módulos de C++.....	180
4.2.	Pruebas Realizadas a los scripts de Tcl.....	182
4.3.	Resultados Obtenidos.....	197
5.	CONCLUSIONES Y RECOMENDACIONES.....	212
6.	ANEXOS.....	213
7.	GLOSARIO.....	241
8.	BIBLIOGRAFIA.....	247

RESUMEN

Con este proyecto se pretende realizar una comparación de la Arquitectura de Servicios Integrados (IntServ) y la Arquitectura de Servicios Integrados sobre IPv6 propuesta por el Ingeniero Jhon Jairo Padilla Aguilar en su tesis doctoral, mediante el Simulador de Redes NS-2.

Esta nueva propuesta le deja la responsabilidad al Host Origen de calcular el Número Hash y almacenarlo en la Etiqueta de Flujo de la cabecera IPv6 del paquete. Con este cálculo se ahorraría tiempo en el Clasificador, ya que sólo se haría la Búsqueda del Número Hash y su respectiva reserva. En caso de que existan dos Números Hash iguales se presenta una Tabla de Colisiones donde se encuentran los Número Hash Colisionados con sus respectivas Quíntuplas y reservas previamente pactadas mediante el Control de Admisión.

Para realizar la comparación de estas Arquitecturas (IntServ e IntServ6), se debe realizar primero la simulación de una red Intserv en el Simulador NS-2, creando un nuevo Objeto Clasificador IntServ mediante el Lenguaje de C++. Al finalizar la creación de este objeto se debe crear un script en Tcl que incluya en su topología el nuevo Clasificador IntServ y además que configure los parámetros de la simulación (topología, retardos de enlaces, tipos de enlaces, tiempos de inicio y finalización de la simulación, anchos de banda, generadores de tráfico, entre otros). Con la ejecución del script equivalente a la red IntServ se calculan además los retardos en la Clasificación de los paquetes.

En segundo lugar se crean dos objetos necesarios en el Simulador NS-2 para crear una red IntServ6. El primer objeto hace referencia al Cálculo del Número Hash en el Host Origen y su almacenamiento en la cabecera del paquete. Y por último el segundo objeto equivale al Clasificador IntServ6, el cual clasifica los paquetes con respecto al Número Hash ingresado anteriormente en su cabecera IPv6. Al finalizar la creación de los objetos de IntServ6 se crea nuevamente un script en Tcl configurando todos los parámetros de la simulación y la topología con los nuevos objetos añadidos. En esta parte también se generan gráficas de retardos de Clasificación de los paquetes.

Finalmente se comparan los retardos obtenidos para los dos casos y se verifica la efectividad de la nueva Propuesta IntServ6.

ABSTRACT

This project aims to realize a comparison of Integrated Services Architecture (*IntServ*) and the Integrated Services Architecture on *IPv6* proposed by the engineer Jhon Jairo Padilla Aguilar in his doctoral thesis by *Network Simulator NS-2*.

This new proposal will leave the responsibility to Source Host of calculating the Hash number and store it in the *Flow Label header IPv6* packet. With this calculation would save time in the Classifier of data packets.

To compare these architectures, first, we must make the simulation of *IntServ* in the Network Simulator *NS-2*, creating a new Object Classifier *IntServ* through the language C + +. Then, we have to create a script to configure the parameters of the new topology, to set de parameters of the simulation and include the new object.

Secondly we have to create two new objects in the Network Simulator needed to create an *IntServ6* Network. The first object refers to the calculation of the Hash number on the Source Host and storage in the packet header. And the second object classifies the data packet using the Hash number calculated on the Source Host.

Finally runs the *Xgraph* tool and generate the graphs of delays in the *IntServ* classifier and the *IntServ6* classifier. The resulting graphs help to conclude that the architecture *IntServ6* has better performed than current technology.

1. INTRODUCCION

1.1. OBJETIVOS

1.1.1. Objetivo general

Desarrollar una simulación en un simulador de redes que permita comparar los comportamientos y tasas de retardos de un router IntServ con un router IntServ6, dando a conocer las ventajas y desventajas de los dos protocolos utilizados.

1.1.2. Objetivos específicos

- Diseñar y construir un clasificador de un Router IntServ por medio de un simulador de redes (NS2).
- Diseñar y construir un clasificador de un Router IntServ6 por medio de un simulador de redes (NS2).
- Diseñar y construir un planificador tipo WFQ (Weighted Fair Queueing) por medio de un simulador de redes (NS2).
- Diseñar y construir un Control de Admisión (CAC) por medio de un simulador de redes (NS2).
- Integrar todas las partes diseñadas para simular la operación completa de un router IntServ y de un router IntServ6.
- Construir una herramienta de simulación de redes IntServ e IntServ6 de fácil manejo para el usuario final y documentar el manual de usuario de la herramienta.
- Documentar los análisis y resultados del proyecto.

1.2. JUSTIFICACIÓN

Este proyecto se realiza con el fin de evaluar por medio de simulaciones, una propuesta de Calidad de Servicio de Internet (IntServ) que es planteada por parte del Ingeniero Jhon Jairo Padilla Aguilar en su tesis doctoral. Algunos de los planteamientos de esta propuesta se encuentran detallados en [1]. Así mismo, este proyecto contribuirá como base para la sustentación de las ventajas que presenta la nueva propuesta con respecto a la tecnología actual. Uno de los principales objetivos de esta propuesta es mejorar la calidad con la que hoy en día es soportada la red mundial de Internet, disminuyendo el tiempo de transmisión de datos de un origen a su destino.

El análisis de los resultados y conclusiones que se obtendrán de este proyecto investigativo serán de gran importancia para futuras aplicaciones o investigaciones de esta misma tecnología tales como redes de sensores[2], redes inalámbricas, etc. Los resultados de este trabajo se utilizarán para simular ciertas situaciones posibles en redes de sensores dentro del proyecto de investigación IRHMA (Interconexión de Redes Heterogéneas Malladas Autoconfigurables) patrocinado por la Unión Europea en que está participando la UPB (Universidad Pontificia Bolivariana) Seccional Bucaramanga, en conjunto con la UPC (Universidad Politécnica de Cataluña) de España.

Además este proyecto pretende dar un soporte para futuros proyectos del Grupo de Investigación de Telecomunicaciones y dar un gran soporte tecnológico a nivel mundial con este tipo de investigaciones que dejan en alto el nombre de la Facultad de Ingeniería Electrónica de la UPB.

1.3. RESUMEN

Con este proyecto se pretende realizar una comparación de la Arquitectura de Servicios Integrados (IntServ) y la Arquitectura de Servicios Integrados sobre IPv6 propuesta por el Ingeniero Jhon Jairo Padilla Aguilar en su tesis doctoral, mediante el Simulador de Redes NS-2.

Esta nueva propuesta le deja la responsabilidad al Host Origen de calcular el Número Hash y almacenarlo en la Etiqueta de Flujo de la cabecera IPv6 del paquete. Con este cálculo se ahorraría tiempo en el Clasificador, ya que sólo se haría la Búsqueda del Número Hash y su respectiva reserva. En caso de que existan dos Números Hash iguales se presenta una Tabla de Colisiones donde se encuentran los Número Hash Colisionados con sus respectivas Quíntuplas y reservas previamente pactadas mediante el Control de Admisión.

Para realizar la comparación de estas Arquitecturas (IntServ e IntServ6), se debe realizar primero la simulación de una red Intserv en el Simulador NS-2, creando un nuevo Objeto Clasificador IntServ mediante el Lenguaje de C++. Al finalizar la creación de este objeto se debe crear un script en Tcl que incluya en su topología el nuevo Clasificador IntServ y además que configure los parámetros de la simulación (topología, retardos de enlaces, tipos de enlaces, tiempos de inicio y finalización de la simulación, anchos de banda, generadores de tráfico, entre otros). Con la ejecución del script equivalente a la red IntServ se calculan además los retardos en la Clasificación de los paquetes.

En segundo lugar se crean dos objetos necesarios en el Simulador NS-2 para crear una red IntServ6. El primer objeto hace referencia al Cálculo del Número Hash en el Host Origen y su almacenamiento en la cabecera del paquete. Y por último el segundo objeto equivale al Clasificador IntServ6, el cual clasifica los paquetes con respecto al Número Hash ingresado anteriormente en su cabecera IPv6. Al finalizar la creación de los objetos de IntServ6 se crea nuevamente un script en Tcl configurando todos los parámetros de la simulación y la topología con los nuevos objetos añadidos. En esta parte también se generan gráficas de retardos de Clasificación de los paquetes.

Finalmente se comparan los retardos obtenidos para los dos casos y se verifica la efectividad de la nueva Propuesta IntServ6.

LISTA DE FIGURAS

- Figura 1. Dirección de envío de mensajes
- Figura 2. Modelo de Referencia de Servicios Integrados
- Figura 3. Operación Básica RSVP
- Figura 4. Sistema GPS
- Figura 5. Planificación WRR.
- Figura 6. Planificador WRR bit a bit con Re-ensamblador
- Figura 7. Planificador WFQ
- Figura 8. Componentes de un Filtro Token Bucket
- Figura 9. El Paquete IPv6
- Figura 10. La Siguiente Cabecera
- Figura 11. Cabecera IPv6 básica y datos
- Figura 12. Cabecera IPv6 básica, fragmento y datos
- Figura 13. Ejemplo comportamiento unicast
- Figura 14. Ejemplo comportamiento anycast
- Figura 15. Ejemplo comportamiento multicast
- Figura 16. Tabla de Manejo de las Reservas de IntServ6
- Figura 17. Vista simplificada del funcionamiento de NS
- Figura 18. Planificador de Eventos
- Figura 19. Componente de red - Link
- Figura 20. Linkage entre bibliotecas C++ y Otel

- Figura 21. Pasos para realizar la simulación
- Figura 22. Ejemplo de animación visualizada con el Nam
- Figura 23. Panel de Control del Nam
- Figura 24. Ejemplo de visualización de resultados con Xgraph
- Figura 25. Pasos de la Metodología del Proyecto
- Figura 26. Ingreso de nuevos Objetos al archivo *makefile.in*.
- Figura 27. Ubicación en la carpeta donde se encuentra instalado el simulador NS-2
- Figura 28. Ingreso del comando *./configure* en el terminal.
- Figura 29. Respuesta del terminal al comando *./configure*.
- Figura 30. Ingreso del comando *make depend* en el terminal.
- Figura 31. Respuesta del terminal al comando *make depend*.
- Figura 32. Ingreso del comando *make* en el terminal.
- Figura 33. Respuesta del terminal al comando *make*.
- Figura 34. Modelo de Referencia de Servicios Integrados
- Figura 35. Procedimiento de Hashing en IntServ
- Figura 36. Clase Classifier
- Figura 37. Clase HashClassifier y nueva Clase IntServClassifier
- Figura 38. Diagrama de Transición de estados del clasificador IntServ.
- Figura 39. Control de Admisión del Clasificador IntServ.
- Figura 40. Topología ejemplo para IntServ
- Figura 41. Diagrama del Corrimiento de Bits del Cálculo del Número Hash
- Figura 42. Cálculo del Número Hash en el Clasificador IntServ.

- Figura 43. Diagrama de Flujo del método *registrar()*.
- Figura 44. Almacenamiento de Datos en la Tabla Hash en IntServ.
- Figura 45. Búsqueda de un Número Hash específico en la Tabla Hash
- Figura 46. Comparación de un Número Hash con los ingresados anteriormente en la Tabla Hash
- Figura 47. Diagrama de Flujo del método *resolver()*.
- Figura 48. Almacenamiento de un Número Hash Colisionado en la Tabla de Colisiones
- Figura 49. Búsqueda de un Número Hash Colisionado en la Tabla de Colisiones
- Figura 50. Clase *Classifier*
- Figura 51. Las nuevas Clases IntServ6Classifier y HostOrigen6
- Figura 52. Diagrama de Transición de estados del Clasificador IntServ6.
- Figura 53. Control de Admisión del Clasificador IntServ6
- Figura 54. Topología Ejemplo para IntServ6
- Figura 55. Cálculo del Número Hash en el Clasificador IntServ6.
- Figura 56. Diagrama de Flujo del método *registrar_6()*
- Figura 57. Ingreso de un Número Hash en la Tabla Hash y Verificación de Colisión
- Figura 58. Búsqueda de un Número Hash Específico en la Tabla Hash
- Figura 59. Diagrama de Flujo del método *resolver_6()*
- Figura 60. Ingresa un Número Hash en la Tabla de Colisiones
- Figura 61. Búsqueda un Número Hash en la Tabla de Colisiones
- Figura 62. Topología Ejemplo para IntServ
- Figura 63. Topología Ejemplo para IntServ6

- Figura 64. Retardo Individual de los Paquetes Ingresados al Clasificador IntServ
- Figura 65. Retardo Promedio del Clasificador IntServ con Tráfico CBR
- Figura 66. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red IntServ con Tráfico CBR.
- Figura 67. Retardo Individual de los Paquetes que ingresaron al Clasificador IntServ6.
- Figura 68. Retardo Promedio del Clasificador IntServ6 con Tráfico CBR
- Figura 69. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red IntServ6 con Tráfico CBR.
- Figura 70. Comparación de Retardos Promedios de Clasificación en las redes IntServ e IntServ6 con Tráfico CBR
- Figura 71. Retardo Individual de los Paquetes ingresados al Clasificador IntServ
- Figura 72. Retardo Promedio del Clasificador IntServ con Tráfico Exponencial.
- Figura 73. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red IntServ con Tráfico Exponencial.
- Figura 74. Retardo Individual de los Paquetes ingresados al Clasificador IntServ6
- Figura 75. Retardo Promedio del Clasificador IntServ6 con Tráfico Exponencial.
- Figura 76. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red IntServ6 con Tráfico Exponencial.
- Figura 77. Comparación de Retardos Promedios de Clasificación en las redes IntServ e IntServ6 con Tráfico Exponencial
- Figura 78. Comportamiento del Planificador WFQ

LISTA DE TABLAS

Tabla 1.	Parámetros de Calidad de Servicio de algunas aplicaciones de Internet.
Tabla 2.	Parámetros de mensajes IntServ.
Tabla 3.	Estructura del campo 'Differentiated Services'.
Tabla 4.	Grupos de 'Code Points' del campo DS
Tabla 5.	'Code Points' utilizados en el Servicio Assured Forwarding
Tabla 6.	Estructura del campo 'Tipo de Servicio'
Tabla 7.	Correspondencia del campo precedencia con los servicios DiffServ
Tabla 8.	Opciones presentes en la siguiente Cabecera.
Tabla 9.	Rutas de los archivos que necesitan modificaciones en NS-2.
Tabla 10.	Métodos de la clase HashClassifier que son heredados de la clase Classifier.
Tabla 11.	Atributos de la clase HashClassifier que son heredados de la clase Classifier
Tabla 12.	Resultados de Simulación de una Red IntServ con Tráfico CBR
Tabla 13.	Resultados de Simulación de una Red IntServ6 con Tráfico CBR
Tabla 14.	Resultados de Simulación de una Red IntServ con Tráfico Exponencial.
Tabla 15.	Resultados de Simulación de una Red IntServ6 con Tráfico Exponencial.

LISTA DE ANEXOS

Anexo A. Programas de las Simulaciones

2. MARCO TEÓRICO

2.1. CALIDAD DE SERVICIO EN INTERNET

2.1.1. INTRODUCCIÓN

2.1.1.1. Antecedentes

Inicialmente la comunicación de datos utilizaba las redes existentes, que era fundamentalmente la red telefónica, hasta que durante los años 60 se propone una nueva técnica llamada “Conmutación de Paquetes”[3]. La conmutación de paquetes es mucho más eficiente que la conmutación de circuitos para la mayoría de los servicios de telecomunicación actuales, por lo que esta nueva técnica inicia un proceso de transformación profunda en el mundo de las telecomunicaciones.

La primera Internet nace financiada por *DARPA* [4] (*Defense Advanced Research Project Agency*), durante la guerra fría. Ésta fue creada por el ejército norteamericano en 1957, para asegurar la supremacía de EEUU en tecnologías de uso militar. En 1968 aparece *ARPANET*[5], promovida por el Departamento de Defensa (*DOD, Department Of Defense*) de Estados Unidos. En 1983, *ARPANET* cambia su arquitectura a *TCP/IP*, empezando la expansión imparable de Internet que todos conocemos. La creación de *ARPANET* tenía como objetivo principal, el desarrollo de una red tolerante a fallos de sus nodos, ya que en ese entonces la red debía sobrevivir a un ataque nuclear, es decir, que en caso de que uno de sus nodos dejaba de operar, las redes resultantes fuesen capaces de reconfigurarse y seguir operando normalmente.

En la década de los 70 surge lo que hoy se conoce como “Internet”. Su objetivo fue desarrollar una arquitectura y unos protocolos que permitieran interconectar varias redes heterogéneas con el fin de formar una única red. Esta nueva arquitectura es denominada *TCP/IP*[6], la cual es la base de Internet actualmente.

En la base de esta arquitectura está el protocolo IP [7] (Internet Protocol). La nueva red pasa a denominarse “Internet”, debido al papel fundamental que este protocolo tiene en la nueva arquitectura. Los routers[8] (enrutadores, encaminadores) son los dispositivos encargados de encaminar paquetes entre redes y su comportamiento se rige por las reglas que marca el protocolo IP.

La fecha exacta de este acontecimiento fue el 1 de Enero de 1983. A partir de entonces, se instalan en todos los routers y ordenadores de la antigua Arpanet, la arquitectura *TCP/IP*, y empieza el crecimiento global de Internet.

2.1.1.2. Origen de Calidad de Servicio

Uno de los factores de éxito de la Internet actual, está en la aceptación de los protocolos *TCP/IP*, como estándar para todo tipo de servicio y aplicaciones. La Internet ha desplazado a las tradicionales redes de datos y ha llegado a ser el modelo de la red pública del siglo XXI [9]. Una falla fundamental de la actual Internet es la imposibilidad de dar diferentes tipos de servicio para los diferentes tipos de aplicación de usuario.

Actualmente dentro del tráfico normal entre un Host origen y un Host destino existen congestiones en donde cada paquete que transporta información compete por obtener un ancho de banda mejor para poder llegar a su destino final. Este tipo de servicio tradicional es llamado *best-effort*[10], el cual no ofrece ningún tipo de garantía de calidad de servicio (*Quality of Service, QoS*) y donde todo el tráfico tiene igual prioridad de ser entregado a tiempo. En ciertos casos, ocurren congestiones en la red por su saturación que conlleva a su bloqueo y es donde ocurre la pérdida de paquetes de información que hacen de éste un servicio deficiente.

A partir de 1994 aproximadamente, comienza el crecimiento de aplicaciones de videoconferencia y multimedia en tiempo real en Internet, muy sensibles a situaciones de congestión. En estos casos es importante que la información no tenga ningún tipo de pérdida, la disponibilidad de un gran ancho de banda y que el envío de paquetes de datos no presente ningún retraso o en otros casos que éste sea mínimo. Es por ello que surge la necesidad de mejorar la Calidad de Servicio (*QoS*) en el nivel de transporte de datos.

Calidad de Servicio[11] (*QoS*) consiste en la capacidad de la red para reservar algunos de los recursos disponibles con la intención de proporcionar un determinado servicio, cumpliendo a su vez con los requerimientos de ciertos parámetros relevantes para el usuario. Es decir, *QoS* es el cumplimiento de un conjunto de requisitos estipulados en un contrato (*SLA: Service Level Agreement*) entre un proveedor de servicios de Internet (*ISP: Internet Service Provider*) y sus clientes [12].

Estos requerimientos de *QoS*, y por tanto de reserva de recursos, dependen del tipo de flujo de datos que necesite enviar determinada aplicación para proporcionar el servicio que un determinado usuario solicite. De esta manera, habrá aplicaciones que necesiten enviar toda la información garantizando una determinada *QoS*, por lo que deberán reservar recursos siempre. Otras no necesitarán garantías de servicio, y el servicio del mejor esfuerzo será suficiente [13]. Implementando *QoS* en una red, hace al rendimiento de ésta más predecible, y la utilización del ancho de banda más eficiente.

Realmente desde el principio de *IPv4*[3] (*Internet Protocol Version 4*), ya existía cierto tipo de Calidad de Servicio, pues en la cabecera del paquete de una red específica, se encontraba el campo denominado *TOS (Type Of Service)* de 8 bits[14], de los cuales los 3 primeros bits representaban su tipo de prioridad basado en la importancia que tenía el paquete en la red respecto a los demás paquetes. [15] Esta marcación permitía establecer en principio políticas o prioridades en la transmisión de los paquetes por la red y el comportamiento de los routers que conformaban esta red.

Aunque esta prioridad representaba cierta Calidad de Servicio y clasificaba los paquetes en categorías según su importancia, no estaba en la capacidad de ofrecer una garantía estricta, donde fuera posible reservar un ancho de banda determinado para un paquete, flujo o una aplicación específica. Esta marcación de los paquetes según su prioridad, no garantizaba un ancho de banda y retardo mínimo ya que podría sufrir congestiones en la red por el exceso de flujos o de paquetes. En este caso la red se saturaba y es donde ocurrían los retardos en los routers y en ciertas ocasiones habían pérdidas de paquetes de información.

Al transcurrir el tiempo fue creciendo la expectativa de crear nuevas arquitecturas que pudieran soportar Calidad de Servicio para cualquier tipo de redes. Por esta razón en 1986 fue creado en EEUU la *IETF* [16](*Internet Engineering Task Force*), es decir, el Grupo de Trabajo en Ingeniería de Internet, el cual es una Organización Internacional Abierta de Normalización, que tiene como objetivos el contribuir a la ingeniería de Internet, actuando en diversas áreas, tales como el transporte, encaminamiento, seguridad, entre otros aspectos importantes. Esta institución esta formada básicamente por técnicos en Internet e informática cuya misión es velar para que la arquitectura de la red y los protocolos técnicos que unen a millones de usuarios de todo el mundo funcionen correctamente.

2.1.2. TIPOS DE TRÁFICO

Los programas al ejecutarse en cualquier red e Internet no están exentos de generar un tráfico que puede clasificarse según las necesidades de proceso[17].

Tráfico de mejor esfuerzo es el flujo de datos habitual en IP, en donde los paquetes circulan por el mejor camino posible en cada momento. El correo electrónico sería un ejemplo de aplicaciones de este tipo de tráfico.

Tráfico sensible a la tasa es el provocado por aplicaciones que están dispuestas a sacrificar el retardo entre los extremos a cambio de tener garantizada la tasa de transmisión. El típico ejemplo de aplicaciones de este tipo es la videoconferencia *H.323*, que si bien fue diseñada para utilizar *ATM* (Modo de Transferencia Asíncrona) o *RDSI* (Red Digital de Servicios Integrados) puede introducirse en Internet. La codificación *H.323* requiere de una tasa de transporte constante para su correcto funcionamiento.

Tráfico sensible al retardo es el provocado por las aplicaciones en donde la puntualidad en la entrega de los paquetes es fundamental para su correcto funcionamiento. El referente inmediato de programa de esta categoría es el video *MPEG-II*, en donde los cambios entre las tramas utilizadas para las imágenes varían poco en lo que se refiere a las tasas de transmisión utilizadas, pero si requieren que lleguen en un tiempo preciso si se quiere que el Codec funcione correctamente. Para estos servicios sensibles al retardo, hay que distinguir los que no son en tiempo real, servicios de retardo controlado y los que si son en tiempo real, servicios predictivos.

La prestación de servicios de calidad a los usuarios de una red depende de una gran cantidad de factores que involucran tanto aspectos de eficiencia como de seguridad[18]. En el aspecto de la eficiencia el ancho de banda disponible y la utilización que se haga del mismo representa un factor crítico. Mientras que en el caso de la seguridad, es importante conocer el tipo de tráfico que está siendo cursado en la red, así como tener la capacidad de detectar tráfico de carácter malicioso.

En relación al aspecto de eficiencia, el administrador y el diseñador de redes deben estar en capacidad de determinar el ancho de banda requerido de acuerdo a las necesidades de la organización y la forma efectiva como éste se utiliza cuando la red se encuentra en producción. Es importante recordar que los costos de comunicación de las empresas es uno de los rubros presupuestarios por servicios más importante. Por lo que durante el diseño de una red se deben realizar planes pilotos para monitorear y determinar los anchos de banda que serán requeridos durante su funcionamiento. Los monitoreos se requieren tanto para el tráfico que se genera hacia la misma red de área local como al exterior.

Una vez que la red entra en producción, el administrador debe efectuar monitoreos de forma constante con el fin de determinar el tipo de tráfico que es enviado por medio de ésta y cuáles tipos de tráfico genera cada usuario o dispositivo. Lo anterior con el fin de detectar posibles fallas de dispositivos específicos, de diseño de la topología y de seguridad. En muchos casos reales los problemas de calidad de servicio no se deben a un limitado ancho de banda, sino a problemas de diseño de la topología y de la seguridad de la misma. Un rediseño de la topología y la seguridad, puede resultar en una buena solución.

2.1.3. PARAMETROS DE CALIDAD DE SERVICIO

Ya que Calidad de Servicio[19] se refiere a la habilidad de la red, de ofrecer prioridad a unos determinados tipos de tráfico, sobre diferentes tecnologías, también hay que tener en cuenta ciertos parámetros que son importantes en el momento de ofrecer QoS a diferentes aplicaciones. Existen cuatro parámetros los cuales son: ancho de banda (o caudal), retraso temporal (o latencia), variación de retraso (o jitter) y probabilidad de error (o pérdida de paquetes o fiabilidad)[20]. Para poder gestionar estos parámetros de forma eficiente debemos hacer uso de la prioridad y gestión del tráfico por medio de colas.

2.1.3.1. Ancho de Banda

Es la cantidad mínima de ancho de banda requerida por un flujo de una aplicación. Debe especificarse el intervalo de tiempo para medir el ancho de banda, ya que diferentes intervalos arrojan diferentes resultados.

La asignación del ancho de banda es garantizada por los algoritmos de planificación de paquetes. Por ejemplo, la clase *WFQ* de algoritmos de planificación es capaz de proveer garantías de ancho de banda mínimo sobre intervalos de tiempo muy cortos.

2.1.3.2. Retraso temporal (Latencia)

Es el tiempo requerido por un elemento de dato individual para moverse a través de un caudal desde un origen a un destino[21]. Esta puede variar dependiendo del volumen de otros datos en el sistema y de otras características de la carga del sistema. Esta variación se denomina fluctuación (*jitter*); formalmente, la fluctuación es la primera derivada de la latencia.

El retraso absoluto puede ser ocasionado por el mal procesamiento de la señal en el *MTA (Message Transport Agent, Agente de Transporte de Mensajes)*, en el *MG (Media Gateway)* o por exceso de tráfico entre un extremo y otro. Típicamente, se permite un retraso máximo de 150 ms en una vía y de 300 ms en ambos sentidos para considerar que la calidad de voz se encuentra dentro de parámetros aceptables.

En función del retraso se pueden distinguir los siguientes tipos de tráfico:

- Asíncrono: Retraso de transmisión sin restricciones.
- Síncrono: El retraso de transmisión está acotado para cada mensaje.
- Isócrono: El retraso de transmisión es constante para cada mensaje.

Este retardo extremo a extremo es especialmente crítico en aplicaciones de audio interactivas como telefonía o videoconferencia, además de que retardos mayores a 400 ms pueden dañar la interactividad de la conversación seriamente, por lo que suelen implicar descartes en el receptor.

El retardo de un paquete tiene tres componentes:

- Retardo de Propagación: Debido a la velocidad de la luz y depende de la distancia.
- Retardo de Transmisión: Tiempo para enviar un paquete sobre un enlace.
- Retardo en cola: Tiempo de espera que experimenta un paquete.

2.1.3.3. Probabilidad de Error (Fiabilidad)

La probabilidad de error es la razón entre los paquetes perdidos y el total de paquetes transmitidos. Las redes tradicionales proporcionan comunicación fiable entre emisor y receptor. Los protocolos de transmisión tienen sistemas de control de errores y de reenvío de paquetes que aseguran que esta fiabilidad es transparente a los niveles superiores.

Para la transmisión en tiempo real esta gestión puede ser negativa, debido al retraso que produciría la retransmisión de un paquete de nuevo. Para evitar este problema se plantea que el tratamiento y gestión de los errores sea a niveles superiores.

Las prestaciones de una transmisión multimedia puede ser medida en dos dimensiones: Latencia y Fidelidad. La latencia puede ser vital para aplicaciones interactivas como conferencias mientras que la transmisión de una película no lo es. La fidelidad de la transmisión es variable. Hay aplicaciones que no toleran ninguna variación en la fidelidad de la imagen como podrían ser la transmisión de imágenes médicas y otras en que esta variación solo produce una cierta distorsión tolerable como la transmisión de películas o música.

La pérdida de paquetes se debe principalmente a la congestión de la red o al exceso de ruido en algún canal que interfiere la óptima transmisión de la información. Estas pérdidas pueden ser prevenidas asignando suficiente ancho de banda y buffers para los flujos de tráfico.

2.1.3.4. Jitter

El retraso variable o *jitter* es la máxima diferencia entre el mayor retardo y el menor retardo que experimentan los paquetes y se debe básicamente a las diferentes rutas que siguen los paquetes IP durante su trayectoria debido a las condiciones de tráfico existentes en la red. Este retraso tiene un impacto directo sobre la calidad de la voz. La congestión en la red ocasiona que los paquetes se almacenen en *buffers* antes de ser retransmitidos. Esto genera un retraso en la llegada de los paquetes que altera la calidad de la conversación. La variación del retardo no debería ser mayor que el peor caso de los retardos de transmisión y de cola.

La Tabla 1 muestra algunas de las aplicaciones mas frecuentes en Internet y sus respectivos parámetros de Calidad de Servicio:

Aplicación	Fiabilidad	Retardo	Jitter	Ancho de Banda
Correo Electrónico	Alta (*)	Alto	Alto	Bajo
Transferencia de ficheros	Alta (*)	Alto	Alto	Medio
Acceso a la Web	Alta (*)	Medio	Alto	Medio
Login Remoto	Alta (*)	Medio	Medio	Bajo
Audio bajo demanda	Media	Alto	Medio	Medio
Vídeo bajo demanda	Media	Alto	Medio	Alto
Telefonía	Media	Bajo	Bajo	Bajo
Videoconferencia	Media	Bajo	Bajo	Alto

Tabla 1. Parámetros de Calidad de Servicio de algunas aplicaciones de Internet.

(*) La fiabilidad alta en estas aplicaciones se consigue automáticamente al utilizar el protocolo de transporte TCP[15].

2.1.4. SOLUCIONES DE CALIDAD DE SERVICIO

La Calidad de Servicio[22] de una comunicación a través de una red, se mide mediante tres aspectos importantes: la velocidad máxima de envío de información, las pérdidas de paquetes de información en la transmisión de datos, y el retardo introducido por esta transmisión. La grande eficiencia de Internet anteriormente se debía a que el protocolo IP[23] se diseñó para dar un servicio que no garantizaba la calidad de servicio, ni la entrega de los paquetes a su destinatario, conocido como el servicio del mejor esfuerzo o “**Best- Effort**” mencionado anteriormente. La gran ventaja de este servicio es su gran simplicidad, pero en momentos de congestión los routers pueden perder paquetes de información. En los casos donde no hay congestión la transmisión es de buena calidad, pero cuando los paquetes llegan a un router congestionado la calidad se degrada y se producen pérdidas y retrasos muy significativos.

Debido a estas congestiones surgió como solución el aumento del ancho de banda en Internet, pero no esto no fue suficiente para finiquitar con este inconveniente de la red. Esto se debe a que cuando un paquete llega a un router donde hay congestión, este automáticamente degrada la calidad de la transmisión sin importar que tan grande sea el ancho de banda en ese tramo de la red y en el resto de la ruta que lleva ese flujo de paquetes.

La mayoría de las aplicaciones de Internet utilizan el protocolo *TCP*[6] (*Transmisión Control Protocol*) para evitar la pérdida de paquetes en casos de congestión, por lo que la existencia de pérdidas de paquetes en la red nunca ha sido un problema importante. En cambio, los protocolos de la arquitectura multimedia de Internet envían los flujos de audio y video como paquetes *UDP* (*User Datagram Protocol*)[24] que no tiene la protección como *TCP* contra las pérdidas en la red, por lo que en este caso se degradan las conversaciones de audio y video a tal punto que no se entienden por los espacios en blanco en los

archivos de audio o video. La razón por la que se usaba *UDP* para la transmisión de audio y video es que introduce menor retardo que *TCP*, ya que en este caso es indispensable, pero con la desventaja de la pérdida de información [4].

Por estas razones, tanto *IETF*, como investigadores de todo el mundo, llevan tiempo trabajando en el desarrollo de técnicas que permitan controlar la Calidad de Servicio directamente a nivel del protocolo IP, de forma que las aplicaciones de envío de audio y vídeo puedan disponer de calidades de servicio IP sin pérdidas ni retrasos significativos en la transmisión de paquetes de datos.

IETF ha desarrollado dos propuestas de control de la calidad del servicio de transporte de información para la transmisión de voz y video con calidad a través de Internet, que han alcanzado un apoyo considerable y son conocidas como *IntServ (Integral Services)*[25] o Servicios Integrados y *DiffServ (Differentiated Services)*[26] o Servicios Diferenciados.

2.1.4.1. Servicio Best- Effort

El Protocolo de Internet provee un servicio de paquetes no fiable (también llamado del *mejor esfuerzo (best effort)*, el cual hará lo mejor posible pero garantizando poco). IP no provee ningún mecanismo para determinar si un paquete alcanza o no su destino y únicamente proporciona seguridad (mediante *checksums* o sumas de comprobación) de sus cabeceras y no de los datos transmitidos. Por ejemplo, al no garantizar nada sobre la recepción del paquete, éste podría llegar dañado, en otro orden con respecto a otros paquetes, duplicado o simplemente no llegar. Si se necesita fiabilidad, ésta es proporcionada por los protocolos de la capa de transporte, como *TCP*.

El Servicio de *Best-Effort* es un modelo simple de servicio, en el cual una aplicación envía información cuando ella lo desea, en cualquier cantidad, sin ningún permiso requerido, y sin informar previamente a la red. La red reparte o envía la información si puede, sin asegurar ningún retraso ni fiabilidad.

2.1.4.2. Servicios Integrados

IntServ fue la primera propuesta en aparecer[27]. La idea fundamental de esta arquitectura radica en que las aplicaciones se ven como un flujo dentro de la Internet y por cada flujo se deberá crear un estado ("*soft state*") en cada uno de los routers [9]. En estos estados se realiza la reserva de recursos para ofrecer QoS a las aplicaciones. En la arquitectura *IntServ* es necesario contar con un protocolo que crea, mantenga y elimine estos estados. Este protocolo de reserva de recursos es llamado *RSVP (Resource Reservation Protocol)* [28] y se encuentra presente tanto en los nodos extremos como en los routers. El nodo extremo de transmisión lo utiliza para informar las características de QoS que requiere la aplicación y el nodo extremo de recepción lo utiliza para realizar la reserva de

recursos en la red. Los routers hacen uso de *RSVP* para transportar los requerimientos de *QoS* entre routers vecinos. Como características básicas del protocolo *RSVP* están su operación en aplicaciones *multicast*; y su aplicación a los protocolos *IPv4* e *IPv6*[29].

La arquitectura de servicios integrados se define entonces, como un conjunto de mensajes[13] para crear, mantener y eliminar los estados en cada nodo de la red, para reservar recursos a solicitud de una aplicación. Una aplicación en el nodo transmisor envía un mensaje "*Path*" hacia el nodo receptor para especificar los requerimientos de tráfico y definir el trayecto que seguirán los paquetes de datos. Cuando el mensaje "*Path*" llega al nodo receptor, este envía hacia el nodo transmisor un mensaje "*Resv*" para realizar la reserva de recursos en cada nodo de la red definido por el mensaje "*Path*". Si un nodo, al recibir un mensaje "*Path*" o "*Resv*" no puede interpretar (por ejemplo, tipo de reserva no definida), este nodo deberá generar un mensaje de error "*PathErr*" o "*ResvErr*" respectivamente. Cuando los mensajes "*Path*" y "*Resv*" se reciben por los nodos correspondientes, los paquetes de datos se envían a la red. Estos mensajes se deben enviar a la red cada 30 segundos para refrescar las reservas asignadas. Cuando se realiza una reserva, se informa al nodo receptor con el envío del mensaje "*ResvConf*". Una vez finalizado el envío de datos, los estados creados y refrescados en cada nodo por los mensajes "*Path*" y "*Resv*" deberán ser eliminados por los mensajes "*PathTear*" y "*ResvTear*" respectivamente tal como se ilustra en la Tabla 2 y la dirección de envío de estos mensajes es ilustrada en la figura 1.

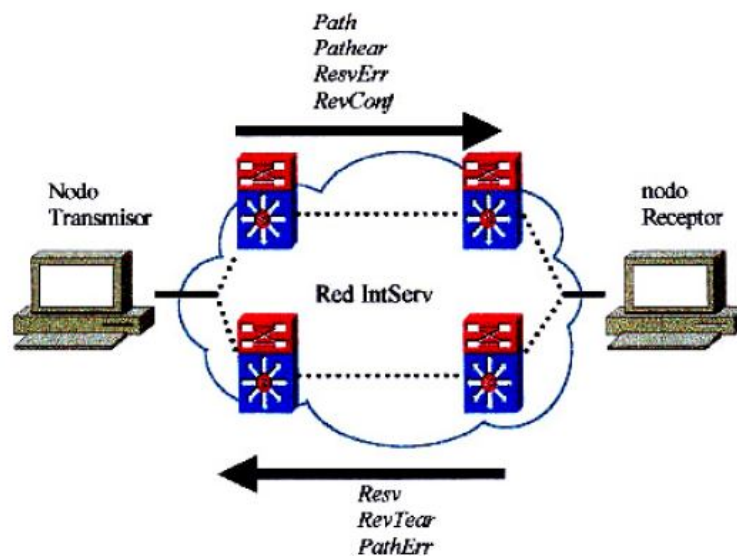


Figura 1. Dirección de envío de mensajes. Fuente: [9]

Nombre	Descripción
<i>Path</i>	Especifica los requerimientos de tráfico, y define el trayecto que seguirán los paquetes de datos.
<i>Resv</i>	Realiza la reserva de recursos en cada nodo de la red.
<i>PathErr</i>	Mensaje de error, al enviar el mensaje.
<i>ResvErr</i>	Mensaje de error, al no poder realizar la reserva de recursos en cada nodo de la red.
<i>ResvConf</i>	Cuando se realizó una reserva satisfactoriamente
<i>PathTear</i>	Para finalizar el envío de datos <i>path</i> .
<i>ResvTear</i>	Para finalizar la reserva de recursos en los nodos de la red.

Tabla 2. Parámetros de mensajes IntServ. Fuente: [9]

2.1.4.3. Servicios Diferenciados

La arquitectura *DiffServ* [15] se basa en la idea de que la información sobre calidad de servicio se escribe en los paquetes y no en los routers. Esta es la diferencia fundamental con *IntServ* y es la que permite implementar una calidad de servicio escalable a cualquier cantidad de flujos.

Para escribir la información sobre la calidad de servicio de cada paquete se utiliza un campo de un byte en la cabecera denominado *DS (Differentiated Services)*[26]. El campo *DS* esta estructurado como se muestra en la Tabla 3:

Subcampo	Longitud (bits)
DSCP(Differentiated Services Code Point)	6
ECN(Explicit Congestion Notification)	2

Tabla 3. Estructura del campo 'Differentiated Services'. Fuente: [15]

El subcampo *ECN (Explicit Congestion Notification)* tiene que ver con la notificación de situaciones de congestión. En cuanto al subcampo *DSCP (Differentiated Services Code Point)* nos permite definir en principio hasta $2^6 = 64$ posibles categorías de tráfico, aunque en la práctica se utilizan menos. Los valores de *DSCP* se dividen en tres grupos. (Ver Tabla 4)

Codepoint	Posibles Valores	Uso
xxxyy0	32	Estándar
xxxx11	16	Local/Experimental
xxxx01	16	Reservado

Tabla 4. Grupos de 'Code Points' del campo DS. Fuente: [15]

Así, se contemplan 32 posibles categorías de paquetes, correspondientes a los cinco primeros bits del campo *DS*.

En *DiffServ* se definen tres tipos de servicio, que son los siguientes:

- **Servicio ‘*Expedited Forwarding*’ o ‘*Premium*’:** Este servicio es el de mayor calidad. Debe garantizar un caudal mínimo, una tasa máxima de pérdida de paquetes, un retardo medio máximo y un *jitter* máximo. El valor del subcampo *DSCP* relacionado con este servicio es ‘101110’.
- **Servicio ‘*Assured Forwarding*’:** Este servicio asegura un trato preferente, pero no garantiza caudales, retardos, etc. Se definen cuatro clases posibles asignándole a cada clase una cantidad de recursos en los enrutadores (ancho de banda, espacio en buffers, etc.). La clase se indica en los tres primeros bits del *DSCP*. Para cada clase se definen tres categorías de descarte de paquetes (probabilidad alta, media y baja) que se especifican en los dos bits siguientes (cuarto y quinto). Existen por tanto 12 valores de *DSCP* diferentes asociados con este tipo de servicio, estos se observan en la Tabla 5.

	Precedencia de descarte		
Clase	Baja	Media	Alta
4	10001	10010	10011
3	01101	01110	01111
2	01001	01010	01011
1	00101	00110	00111

Tabla 5. ‘Code Points’ utilizados en el servicio *Assured Forwarding*. Fuente: [15]

Se puede imaginar la prioridad de descarte como algo equivalente al bit de *Frame Relay* o al bit *CLP* (*Cell Loss Priority*) de *ATM* (*Asynchronous Transfer Mode*), solo que en este caso se pueden marcar tres prioridades de descarte diferentes en vez de dos. En muchas implementaciones se ignora el quinto bit del campo *DSCP*, con lo que las precedencias media y alta son equivalentes. En estos casos el cuarto bit del *DSCP* desarrolla un papel equivalente al bit *DE* de *Frame Relay* o al *CLP* de *ATM*[30].

En el servicio *Assured forwarding* el proveedor puede aplicar *traffic policing* al usuario, y si el usuario excede lo pactado el proveedor puede descartar paquetes, o bien aumentar la precedencia de descarte.

- **Servicio *Best Effort*:** Este servicio se caracteriza por tener ceros en los tres primeros bits del *DSCP*. En este caso los dos bits restantes pueden

utilizarse para marcar una prioridad, dentro del grupo '*best-effort*'. En este servicio no se ofrece ningún tipo de garantías.

El servicio *Expedited Forwarding* es aproximadamente equivalente al Servicio Garantizado de *IntServ*, mientras que el *Assured Forwarding* corresponde más o menos al Servicio de Carga Controlada de *IntServ*.

Algunos *ISPs* (proveedores de servicio de Internet) ofrecen los servicios denominados 'olímpicos' con categorías denominadas, oro, plata y normal (o tiempo-real, negocios y normal). Generalmente estos servicios se basan en las diversas clases del servicio *Assured Forwarding*.

El campo *DS* es una incorporación reciente en la cabecera IP. Anteriormente existía en su lugar un campo denominado *TOS* o 'Tipo de Servicio' que tenía la estructura que se observa en la Tabla 6:

Subcampo	Longitud (bits)
Precedencia	3
Flags D, T, R, C	4
Reservado	1

Tabla 6. Estructura del campo 'Tipo de Servicio'. Fuente: [15]

El subcampo 'Precedencia' permitía especificar una prioridad entre 0 y 7 para el paquete (7 máxima prioridad). Este campo es en cierto modo el antecesor del campo *DS*. A continuación se encontraba un subcampo compuesto por cuatro bits que actuaban como indicadores o '*flags*' mediante los cuales el usuario podía indicar sus preferencias respecto a la ruta que seguiría el paquete. Los flags denominados D, T, R y C permitían indicar si se prefería una ruta con servicio de bajo retardo (D=*Delay*), elevado rendimiento (T=*Throughput*), elevada fiabilidad (R=*Reliability*) o bajo costo (C=*Cost*). El campo *TOS* ha sido muy impopular ya que el subcampo precedencia se ha implementado muy raramente en los enrutadores. En cuanto a los flags D, T, R, y C prácticamente no se han utilizado y su inclusión en la cabecera IP ha sido muy criticada. Estos problemas facilitaron evidentemente la 'transformación' del campo *TOS* en el *DS*, aunque existen todavía enrutadores en Internet que interpretan este campo con su antiguo significado de campo *TOS*. Dado que DiffServ casi siempre utiliza solo los tres primeros bits del *DSCP* para marcar los paquetes, y que los servicios de mas prioridad, como es el caso del *Expedited Forwarding*, se asocian con los valores más altos de esos tres bits, en la práctica has bastante compatibilidad entre el nuevo campo *DSCP* del byte *DS* y el antiguo campo de Precedencia del byte *TOS*, como puede verse en la Tabla 7.

Valor Campo Precedencia	Servicio DiffServ correspondiente
7	Reservado
6	Reservado
5	Expedited Forwarding
4	Assured Forwarding Clase 4
3	Assured Forwarding Clase 3
2	Assured Forwarding Clase 2
1	Assured Forwarding Clase 1
0	Best Effort

Tabla 7. Correspondencia del campo precedencia con los servicios DiffServ. Fuente: [15]

Evidentemente esta compatibilidad no es accidental. Tradicionalmente el campo Precedencia no hacía uso de los dos niveles de prioridad más altos, que quedaban reservados para mensajes de gestión de red, como los paquetes del protocolo de routing. En DiffServ se ha reservado también los dos valores más altos de los tres primeros bits con lo que se mantiene la compatibilidad con el campo precedencia.

➤ **Ventajas del Modelo *DiffServ***

- o *Escalabilidad*: No es necesario mantener información de estado o flujos.
- o *Performance*: El contenido del paquete solo se inspecciona una sola vez para clasificarlo. En ese momento, el paquete es marcado y todas las decisiones de QoS posteriores se hacen de acuerdo al valor en un campo fijo de cabecera, reduciendo los requerimientos de procesamiento.
- o *Interoperabilidad*: Todos los fabricantes ya están ejecutando IP.
- o *Flexibilidad*: El modelo *DiffServ* no dictamina que se implemente ninguna funcionalidad particular en un nodo de red (como una técnica de encolado). El nodo puede utilizar cualquier técnica que optimice su hardware y su arquitectura, mientras sea consistente con el comportamiento esperado definido en los *PHBs* (*Per-Hop Behavior*, Comportamiento por salto) [31].

➤ **Desventajas del modelo *DiffServ***

- o No hay reservaciones de ancho de banda extremo a extremo, por lo tanto, las garantías de servicio pueden ser imparciales en los nodos de la red que no implementen los *PHBs* correctamente sobre enlaces congestionados o por nodos que no están correctamente diseñados para el volumen de tráfico esperado de una clase específica.

- o La ausencia de control de admisión por flujo o por sesión hace posible que las aplicaciones se congestionen unas con otras. (Por ejemplo, si solamente hay ancho de banda para 10 llamadas de voz y se permite una llamada más (11), todas las once llamadas sufren un deterioro de la calidad).

2.2. ARQUITECTURA DE SERVICIOS INTEGRADOS[32]

2.2.1. Introducción

A principios de los años 90 se comenzó a diseñar *MBone* (IP *Multicast Backbone*), entendida como una subred experimental dentro de Internet para trabajar con tráfico IP *Multicast* [33]. La idea fundamental del IP *Multicast* que lo hacía tan viable para desarrollar servicios de videoconferencia multipunto se basaba en la utilización de direcciones IP *multicast*, en vez de direcciones IP *unicast* tradicionales. Gracias a esto se tenían las siguientes ventajas:

- Optimización del ancho de banda utilizado al circular por cada enlace entre el origen y el destino(s) de datos, un único paquete.
- La complejidad de la distribución a todos los destinos recae en los equipos de red y no en los equipos finales. Esto simplificaba su trabajo y los liberaba de carga.

Sin embargo, también presentaba otras carencias, que fueron las que limitaron el crecimiento de este servicio, entre las cuales están:

- Las aplicaciones de tiempo real no trabajaban bien sobre Internet, debido al retardo variable y a las pérdidas de información por congestión de redes.
- El mal uso del multicast podía causar grandes interrupciones a una gran porción de Internet.
- Aplicaciones como video digital eran capaces de generar una alta tasa de tráfico, pues en los años 90 subió hasta saturar algunos *backbones*.

Las anteriores fueron las causas por las cuales *MBone*[33] fue un intento fallido por parte de la *IETF* para lograr la utilización de aplicaciones de tiempo real en Internet. Años después surgió la necesidad de implementar nuevas técnicas y métodos para soportar cierta calidad de servicio en Internet con el fin de que las aplicaciones de tiempo real tuvieran un excelente desempeño en Internet con sus especificaciones necesarias.

El primer intento de estandarizar la Calidad de Servicio apareció en los años 90 cuando la *Internet Engineering Task Force (IETF)* publicó un *Request for Comment* referido a los Servicios Integrados ([RFC1633])[31]. Estos documentos describen un protocolo de señalización denominado *Resource Reservation Protocol (RSVP)*. Este protocolo implementa una señalización antes que el flujo de datos sea enviado a la red, construyendo un canal virtual a lo largo del cual se reservan los recursos necesarios. Esta arquitectura opera sobre flujos individuales

reservando recursos suficientes en los routers de punta a punta para satisfacer los requerimientos de QoS del flujo.

El objetivo de este Servicio es preservar el modelo de paquetes de las redes basadas en *IP* y al mismo tiempo soportar reservas de recursos para aplicaciones de tiempo real. Esto quiere decir que la red se basará en el mismo sistema de paquetes que se identificarán mediante el protocolo *IP*, y que a su vez esta red será capaz de hacer reservas por cada flujo que quiera solicitar el servicio.

2.2.2. Principios Básicos

Como se ha dicho anteriormente en esta arquitectura se hace reserva de recursos por flujos[34]. Un flujo es una cadena de paquetes que fluyen a través de la red desde una aplicación de un host origen hasta una aplicación de un host destino. La reserva de recursos debe establecerse previamente con cada uno de los routers que establecen la comunicación entre los dos nodos extremos. Para ello cuando una aplicación quiere iniciar una comunicación en la red deben seguirse los siguientes pasos:

- A. La fuente inicia el establecimiento de la reserva dando a conocer a la red los requerimientos de la aplicación que desea mandar la información para así darle los recursos adecuados.
- B. La red decide si existen suficientes recursos para asignar a este nuevo flujo con respecto a lo solicitado.
- C. Cuando ya este establecida la reserva, la aplicación del host origen comienza a enviar su flujo de paquetes de información a través de los nodos con los cuales se reservaron recursos.

Un supuesto implícito de la reserva de recursos es que la demanda de ancho de banda es excedida por la suministrada. *IntServ* asume que la principal Calidad de Servicio[35] a la que la red se compromete es el retardo por paquete. Las razones de asumir este parámetro son:

- ✓ El tiempo de entrega es una de las cantidades más importantes y de interés para las aplicaciones.
- ✓ Las aplicaciones *Playback* son más sensibles a los paquetes que han experimentado el máximo retardo.
- ✓ Es más fácil para la red comprometerse con este parámetro que con el valor promedio de retardo por paquete.

2.2.3. Componentes Claves

El modelo de referencia de *IntServ*[36] (Figura 2) se puede dividir en dos partes: (a) El plano de Control, que establece la reserva de recursos, y (b) El plano de Datos, que re-envía los paquetes de datos basado en el estado de la reserva.

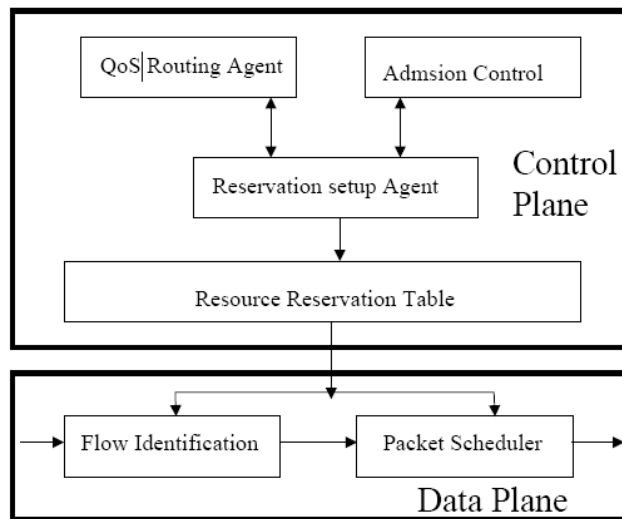


Figura 2. Modelo de Referencia de Servicios Integrados. Fuente: [32]

El funcionamiento del Plano de Control se da de la siguiente manera:

1. Para establecer la reserva de recursos, una aplicación primero caracteriza su flujo de tráfico y especifica los requerimientos de QoS. A este proceso se le llama en *IntServ: Especificación del flujo (Flow Specification)*.
2. La solicitud de establecimiento de reserva de recursos es entonces enviada a la red mediante mensajes de señalización.
3. Cuando un router[37] recibe la solicitud, realiza dos tareas:
 - a) Interactúa con el módulo de enrutamiento para determinar el siguiente salto al que debe ser enviado la solicitud de reserva.
 - b) Tiene que coordinar con el control de admisión para decidir si hay suficientes recursos para comprometerse con los recursos solicitados.
4. Cuando se completa el establecimiento de la reserva, la información del flujo reservado es instalada en la *Tabla de Reserva de Recursos*.

La información en la reserva de recursos es usada entonces para configurar el módulo de identificación de flujos (*flow identification*) y el módulo de planificación de paquetes (*Packet Scheduler*) en el plano de datos.

Cuando llegan los paquetes al router, el módulo de identificación de flujos selecciona los paquetes que pertenecen a los flujos reservados y los coloca en las

colas apropiadas para darles el servicio pactado. Y el planificador de paquetes[38] asigna los recursos a los flujos basado en la información de las reservas.

Selección de la ruta

En cada nodo la red debe determinar cuál camino usar para establecer la reserva de recursos y éste debe ser seleccionado de manera que tenga suficientes recursos para alcanzar los requerimientos.

La selección óptima de la ruta es difícil con el enrutamiento *IP* actual, pues no necesariamente el único camino que satisface el ancho de banda debe ser el más corto. Este es un problema que aún no se ha resuelto e *IntServ* supone que hay un módulo de enrutamiento en el router que suministra el próximo salto.

Establecimiento de la Reserva

Para hacer la reserva, se requiere de un protocolo de establecimiento de reservas que va instalando el estado de reserva en los routers salto por salto a lo largo del camino. Este protocolo trasporta información sobre caracterización del tráfico y requerimientos de los recursos, por lo que cada nodo a lo largo del camino puede determinar si se puede aceptar o no una nueva solicitud de reserva. El protocolo de establecimiento de la reserva debe seguir los cambios en la topología de la red en el tiempo.

La reserva de recursos involucra transacciones financieras usualmente, es decir que hay un completo conjunto de características relacionadas con: autorización, autenticación y facturación al cliente por parte del proveedor del servicio. Antes de que se inicie una reserva, debe ser antes autorizado por la misma persona que paga el servicio de reserva[39], y este usuario debe ser autenticado y la reserva es grabada para llevar las cuentas de estos movimientos financieros.

En *IntServ*, se ha desarrollado el protocolo *RSVP* como el protocolo de establecimiento de la reserva para Internet. *RSVP* está basado en una aproximación iniciada por el receptor y está diseñado para trabajar con *IP Multicast*. Este a su vez, permite diferentes tipos de estilos de reserva y usa la aproximación del “*soft state*” para seguir los cambios de las rutas.

2.2.4. El protocolo de Señalización RSVP

RSVP (Reservation Protocol)[17], protocolo de reservación de recursos. Es un protocolo de control de la red que permite que los programas que van a trabajar en Internet puedan obtener la calidad de servicio que sus flujos de datos puedan requerir. Se trata de un protocolo totalmente emergente que se encuentra aún en fase de normalización por parte del *IETF*, que esta desarrollando su estandarización.

Este protocolo no es, en contra de lo que pudiera parecer, un protocolo de enrutamiento. Es un protocolo que se inscribe dentro de la capa de Transporte del modelo de conectividad *OSI* (*Open System Interconnection*, Modelo de referencia de Interconexión de Sistemas Abiertos), y se apoya en las tablas de rutas dinámicas que manejan los protocolos de enrutado clásicos para establecer una conexión a modo de circuito virtual entre emisor y receptor o receptores implicados.

Para *RSVP*[40] el flujo de datos es simplemente una secuencia de paquetes que tienen un mismo origen, uno o varios destinos, según sea la difusión, unicast o multicast, y una calidad de servicio, todo ello caracterizado mediante sesiones. Una sesión *RSVP* es cada torrente de datos que el protocolo maneja de forma independiente.

Las especificaciones de operación de este protocolo se materializan en un programa *RSVP* estructurado en módulos, cada uno de ellos con unas funciones específicas. Por una parte están el módulo de Control de Admisión y el módulo de Política. El primero se encarga de determinar si el nodo tiene los recursos solicitados disponibles para soportar la Calidad de Servicio pedida. El Control de Política determina si el solicitante tiene los permisos necesarios para poder disponer de los recursos que solicita. En otro lado se encuentra el motor de la reserva, el módulo de Clasificación, encargado de recibir los paquetes para determinar su ruta y QoS necesaria y el módulo Esquemático, el cual se encarga de la transmisión de paquetes.

El host, en donde se ejecuta la aplicación que genera el tráfico en la red, inicia una sesión con el enrutador con capacidad *RSVP*, especificando la dirección destino, el identificador del protocolo y puerto a utilizar. Si recibe respuesta, obtiene un identificador de sesión que marcará el camino que seguirán los paquetes que genere el programa. Cuando se activa la sesión, y los paquetes son enviados el enrutador recibirá, junto con ellos, mensajes *RSVP* en donde se especifican la reserva de recursos que ha de realizar a lo largo de toda la trayectoria.

El protocolo *RSVP* recibe los paquetes y los clasifica, encolándolos según criterios temporales. Se les asigna ruta, Calidad de Servicio y en función del temporizador se colocan en la interface del enrutador adecuado. Si la capa de enlace del puerto seleccionado tiene su propia gestión de QoS, el programa del protocolo de reservación, negociará con él la obtención de la Calidad de Servicio requerida. Si no dispone de esta capacidad, es el propio programa quien puede encargarse de reservar la capacidad necesaria para la transmisión, pudiéndose ocupar no sólo de los parámetros que afecten a la línea de comunicación, si no que puede abarcar CPU y almacenamiento.

El inicio del proceso de reservación de este protocolo comienza realmente cuando el programa consulta a los protocolos de enrutado local, las rutas que ha de

utilizar. En cada salto, en cada nodo del camino, el programa *RSVP* local tiene que aplicar el mismo procedimiento, es decir, calcular si se puede ofrecer Calidad de Servicio que le han solicitado. Si este Control de Admisión y Control de Política es satisfactorio, el nodo local clasifica y encola los paquetes para darles la QoS que requieren. Si no le es posible proporcionar los recursos que le han sido pedidos, la aplicación que originó el flujo de datos recibirá una indicación de error.

En el contexto de operación de *RSVP*[17], la distribución de datos puede hacerse por difusión en *unicast*, en donde sólo hay un emisor y un receptor, o por *multicast*, en cuyo caso hay un emisor y varios receptores. El flujo de datos que se maneja en una sesión se este protocolo siempre tiene un solo emisor y los paquetes de cada sesión en particular irán dirigidos a la misma dirección IP y a un puerto. Si hay difusión *multicast*, la dirección IP será la dirección del grupo. Si para este protocolo cada emisor y receptor debe corresponderse con un host único, no hay inconveniente para que un mismo host pueda contener varios emisores o receptores lógicos que se identifiquen por los puertos.

En lo que respecta a QoS, los requerimientos que pueda necesitar la aplicación se definen mediante la especificación del flujo de datos, que es la estructura de datos utilizada por los hosts para solicitar servicios especiales a la red. Mediante un atributo que va incorporado en los mensajes con los que se relacionan *RSVP* y los programas, se determina de qué forma han de intercambiar datos los actores de la transmisión. Así, el host utiliza su parte *RSVP* para solicitar el nivel de Calidad de Servicio que necesita para cada ráfaga de datos. El enrutador utiliza su parte de protocolo para propagar esas necesidades a los otros encaminadores de la ruta a seguir.

2.2.4.1. Control de la Reserva

La labor de gestión de recursos la realiza este protocolo mediante el intercambio de mensajes que se incorporan en los paquetes *RSVP*, que le permiten gestionar ese circuito virtual que se llega a establecer entre emisor y receptores. Hay cuatro tipos de mensajes.

El primer tipo de mensajes que ha de manejarse es el de solicitud de reservación, mensaje que tiene que generar el receptor. Es decir, el host de destino ha de enviar un mensaje que recorra la ruta de llegada a la inversa para que el host emisor pueda determinar qué control de flujo es necesario para que pueda iniciarse el tráfico desde el primer salto. El protocolo no mantiene informado al emisor sobre el éxito del establecimiento de la conexión con el receptor.

Para que esos mensajes de solicitud de reservación se produzcan, el emisor tiene que lanzar un mensaje de trayectoria que trace la ruta según la información que le proporcionan los protocolos de enrutamiento y que cada nodo del camino ha de conocer para mantenerla. Sobre la viabilidad de la ruta, sobre la reservación de

recursos y su estado, el protocolo maneja mensajes de error y confirmación. Se producirán mensajes de error en la reservación si la solicitud de reserva no se puede realizar por alguna circunstancia de la red, como puede ser la no disponibilidad del ancho de banda necesario, que el servicio solicitado no esté disponible o algún nodo no sea capaz de admitirlo en las condiciones requeridas. Así mismo, si el trazado de la ruta no tiene éxito, el emisor recibirá un mensaje de error en la trayectoria. Los mensajes de confirmación se dan cuando se exige en el mensaje de solicitud de reservación que ésta sea confirmada y viaja hacia el receptor de la comunicación.

La validez de las reservas realizadas y sus trayectorias se administran mediante mensajes de “limpieza”, de revocación, que se encargan de eliminar los estados de estos dos aspectos de la transmisión, trayectoria y reserva, sin tener que esperar a que expire su tiempo de vida. Pueden ser generados por cualquiera de los elementos implicados, el emisor, el receptor o el enrutador como resultado de alguna variación de las condiciones del estado de cualquiera de ellos. Un mensaje de eliminación de trayectoria implica, lógicamente, la eliminación del estado de reservación.

Así, el formato de los paquetes de este protocolo consta de dos partes, un encabezado de mensajes y objeto *RSVP*. La primera parte lleva la información de control que el protocolo requiere para manipular correctamente el paquete, como puede ser versión de protocolo, tiempo de vida del IP con el que se envió el mensaje, tamaño del paquete completo, identificador del mensaje y otros. La parte de objeto lleva información específica de la función de reservación de recursos, como puede ser la dirección IP y puerto destino, la dirección IP del nodo *RSVP* que envía el mensaje, el estilo de la reservación, la especificación de *QoS*, y otros más.

Para un mayor entendimiento de este proceso de reservas se incluye una figura de la operación en general del protocolo de señalización *RSVP*.

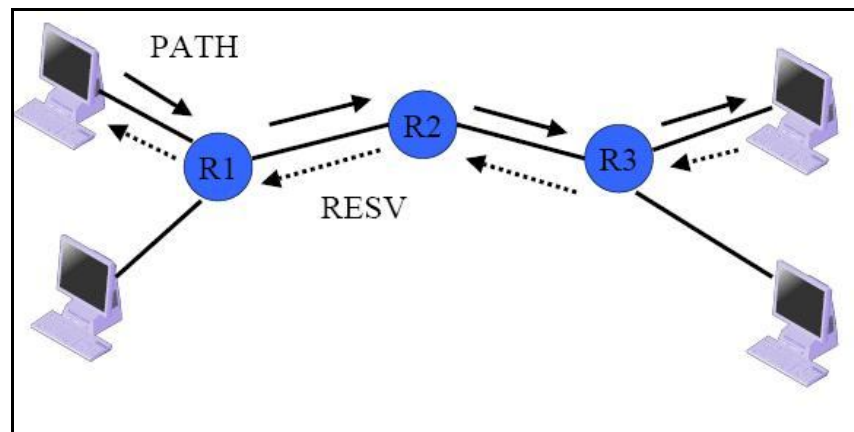


Figura 3. Operación Básica RSVP. Fuente: [41]

2.2.4.2. Estilo de Reserva

Para llevar a cabo su función *RSVP*, utiliza una serie de controles determinados por los parámetros que soporta para especificar como se van a reservar los recursos para los elementos que los solicitan. La reserva de recursos que este protocolo maneja puede ser compartida cuando se da el caso que la sesión es compartida por varios emisores simultáneamente sin que se interfieran entre ellos, o bien será una reservación de recursos exclusiva cuando se establece una sesión independiente para cada uno de los emisores. Esta característica define el estilo de la reserva de recursos que hace el protocolo.

Se puede plantear el fluido de datos que manejan este protocolo, la sesión, como una tubería por la que fluye la información y que su torrente puede llegar a uno o varios recipientes. La cañería que se tiende desde la fuente a los recipientes, puede contemplarse, siguiendo con la analogía, en función del material con el que está hecha, flexible o rígido, que permitiría acomodar sobre la marcha más caudal o no. Así, en el estilo de reservación *RSVP* se contempla que en el estilo, exclusivo o compartido, de la reservación se puede utilizar como una tubería rígida, filtro fijo, en donde se define un determinado canal, una determinada reservación que será utilizada por una sola fuente emisora o una lista de ellas, pero sin oportunidad de variar. La tubería flexible, el filtro comodín, se entiende cuando los recursos reservados son comunes a todos los emisores y el canal tiene una “anchura” que será la del que mayor requerimientos tenga, la necesiten o no.

El estilo de reserva no permite mezclar aquellas que son compartidas con las exclusivas y cada una de ellas será apropiada para las aplicaciones en función de las características que requiera el flujo de datos que vayan a utilizar.

2.2.5. Control de Admisión

2.2.5.1. Introducción

Para ofrecer recursos garantizados para los flujos reservados, una red debe monitorear el uso de sus recursos. La red debe negar solicitudes de reserva cuando no hay suficientes recursos disponibles. Estas tareas las realiza un *Agente de Control de Admisión*. [25]

Antes de que una solicitud de reserva pueda ser aceptada, esta debe pasar una *prueba de control de admisión*. Las funciones básicas del Control de Admisión son: (1) determinar si una reserva puede ser establecida basándose en las políticas de control de admisión y (2) monitorear y medir los recursos disponibles.

2.2.5.2. Aproximaciones de Control

Hay dos aproximaciones para el control de admisión:

- ❖ Aproximación basada en parámetros.

- ❖ Aproximación basada en medidas.

- **Aproximación basada en parámetros**

En esta aproximación se usa un conjunto de parámetros para caracterizar los flujos de tráfico de forma precisa, y con estos parámetros el Agente de Control de admisión calcula los recursos requeridos. Pero hay que tener en cuenta que es difícil obtener cálculos ajustados a los modelos de tráfico, pues entre mayores movimientos generan más tráfico, ejemplo: tasa de tráfico de un codec de vídeo).

Si se reservan recursos para el peor caso, se tendrá una baja utilización de los recursos de la red, y es aun peor cuando hay tráfico a ráfagas.

- **Aproximación basada en medidas**

En esta aproximación no se mantiene una caracterización del tráfico “a priori”. La red mide la carga de tráfico real y usa esto para el control de admisión.

Esta aproximación es de naturaleza probabilística y no puede ser usada para proveer garantías fijas en acuerdos de recursos. Es usada para fuentes a ráfagas y aplicaciones que toleran algún grado de variación en el retardo, ofrece un buen balance entre el nivel de garantía de recursos y la utilización de los recursos.

2.2.6. Identificación de Flujos

Dentro del procesamiento de los paquetes en los routers existen ciertos pasos que hay que seguir para una eficiente identificación de flujos:

- Se debe examinar cada paquete entrante y decidir si el paquete pertenece a uno de los flujos *RSVP* reservados.
- La identificación del flujo IP se hace mediante 5 campos de la cabecera del paquete (llamados *Quíntupla*):
 - ✓ Dirección IP fuente
 - ✓ Dirección IP destino
 - ✓ Identificación del protocolo
 - ✓ Puerto fuente
 - ✓ Puerto destino
- Para determinar si un paquete coincide con un flujo *RSVP*, el motor de identificación de flujos debe comparar la *Quíntupla* del paquete entrante con la *Quíntupla* de todos los flujos en la tabla de reservas. Si hay coincidencia, se obtiene el estado de la reserva de la tabla de reservas y el paquete es enviado al planificador de paquetes junto con el estado de la reserva asociado con el flujo. Este proceso se hace para cada paquete entrante en

un tiempo determinado. Este tiempo es muy corto ya que el número de flujos en los enlaces troncales puede ir de cientos a miles.

Los routers modernos[8] deben soportar altas velocidades y deben funcionar para el peor caso. En el peor caso pueden llegar múltiples paquetes, que pertenecen a flujos reservados, a la velocidad del enlace entrante.

Hay severas restricciones de tiempo para el procesamiento de los paquetes:

- Cada paquete es procesado por una serie de módulos de procesamiento formando una línea de ensamble: revisión dirección IP, identificación de flujos, planificación, retransmisión.
- Este procesamiento debe hacerse para el tiempo asignado a un paquete. Ejemplo: paquetes de 64 Bytes en un enlace de 622 Mbps, el tiempo de procesamiento por paquete es menos de 1 microsegundo.
- El número de flujos concurrentes puede ser muy grande.

Hay diferentes aproximaciones posibles y todas tienen un compromiso entre velocidad y espacio. Una de esas es la tabla de memoria, la cual ocupa mucho espacio, es rápida y solo tiene un acceso a memoria. Y la otra es la búsqueda binaria, la cual ocupa menos memoria, es lenta, y tiene 17 accesos a memoria para identificar uno de 64 K flujos reservados.

Existe también una solución *basada en hashing* [42] la cual guarda una buena relación velocidad-espacio. Esta solución funciona de la siguiente manera.

En el router transmisor:

- Cuando se hace una reserva *RSVP*, un router aplica una función *hash* a la *quíntupla* que es usada para identificar el flujo.
- Si el valor de salida hash no se ha usado para otro flujo reservado, el valor hash es asignado al estado de este flujo en la tabla de reservas.
- Si el valor hash ya ha sido asignado a otro flujo, se dice que hay una colisión hash y se fija un bit para indicar que hubo una colisión.
- El valor hash apunta a una tabla de resolución de colisiones que almacena *quíntuplas* de todos los flujos reservados con el mismo valor hash.
- Cada *quíntupla* en la tabla de resolución de colisiones tiene otro apuntador a los datos en la tabla de reservas.

En el router receptor:

- Cuando el router procesa un paquete, primero aplica la misma función hash a la *quíntupla* del paquete entrante.

- Si el valor hash no tiene bit de colisión, el valor hash es usado para encontrar la entrada en la tabla de reservas y obtener el estado de la reserva para el flujo.
- Si el bit de colisión está fijado, se realiza otra búsqueda en la tabla de resolución de colisiones para encontrar la coincidencia exacta y su estado de reserva asociado.

Es un aspecto crítico reducir las colisiones al mínimo. Generalmente al aumentar el tamaño de la tabla hash, se reduce la tasa de colisiones (sintonía fina en el compromiso velocidad-espacio).

Para cualquier diseño hay dos métricas importantes de desempeño:

- ❖ **La tasa de colisión** (porcentaje de flujos que tienen al menos una colisión). Determina el desempeño promedio del sistema.
- ❖ **El número de flujos colisionados** de peor caso (máximo número de flujos que podrían tener el mismo valor hash). Mide el desempeño del sistema en el peor caso.

Si la tasa de colisión es extremadamente baja, sería aceptable permitir algún pequeño porcentaje de colisiones, por lo que se podría saltar el paso de la resolución de colisiones (compromiso entre precisión y velocidad).

Existen varios esquemas basados en *hashing* entre los cuales están:

- XOR bit por bit entre dirección IP fuente y destino (por grupos de m bits).
- XOR bit por bit entre dirección IP destino y puerto destino.
- XOR bit por bit con la Quíntupla del flujo.
- CRC de 32 bits: es más seguro ante errores pero es más complejo.

2.2.7. Planificación de paquetes

Este es el último paso de la reserva de recursos y es uno de los más importantes. La planificación de los paquetes es el proceso responsable de la asignación de los recursos. Este proceso afecta directamente el retardo que experimentan los paquetes y afecta indirectamente en cuál paquete se pierde cuando el buffer se llena.

La tarea central del planificador es seleccionar un paquete para transmitirlo cuando el enlace saliente está listo.

El planificador utilizado en el servicio llamado comúnmente “*best-effort*” es el *FCFS* (*First Come First Send*, Primero en llegar, primero en enviar), ya que este no puede soportar garantías de servicio. En *IntServ* se requieren algoritmos de planificación más avanzados para poder soportar calidad de servicio para cualquier tipo de aplicaciones.

El algoritmo más conocido es el WFQ[43] (Weighted Fair Queueing) y es el que se utiliza en la arquitectura de servicios integrados. WFQ es una clase de algoritmos de planificación que comparten una aproximación común pero que difieren en los detalles de implementación.

El propósito de un planificador es permitir compartir un recurso común de forma controlada. Hay dos aspectos importantes a tener en cuenta:

- o **Aislamiento de flujos:** El caso extremo sería el de la conmutación de circuitos (los recursos reservados se sub-utilizan).
- o **Distribución de recursos:** El manejo más eficiente de un recurso se logra con multiplexación estadística, usada en conmutación de paquetes.

Hay un compromiso entre estos dos aspectos. Los algoritmos de distribución justa (fair queueing) logran un balance.

Hay dos tipos de límites de retardos; los *determinísticos* (ej: retardo del peor caso en IntServ), los cuales requieren fuertes garantías de reservas, y los retardos *estadísticos* (ej: retardos debido al crecimiento de las colas por un tiempo máximo), los cuales permiten multiplexación estadística.

Cuando no hay suficientes recursos para satisfacer las demandas de tráfico, debe asignarse de forma justa el ancho de banda para todos los flujos en competencia. Una política de distribución justa de recursos (criterio de justicia) que ha sido ampliamente considerada en la literatura es llamada max-min fair sharing. Este criterio trata de suavizar la mínima parte asignada a un flujo cuya demanda no ha sido completamente satisfecha. Los principios usados en este criterio son: que los recursos se distribuyen en función de la demanda de los usuarios; a un usuario no se le asignan más recursos de los que ha solicitado; y los usuarios que no han satisfecho sus demandas, obtienen igual distribución de los recursos[35].

2.2.7.1. Disciplina de planificación de cola WFQ (*Weighted Fair Queueing*)

WFQ ó PGPS [43](*Packet GPS*) lleva a cabo la distribución justa del ancho de banda para paquetes de tamaño variable aproximando un sistema GPS (*Generalized Processor Sharing*) ó modelo de fluidos.

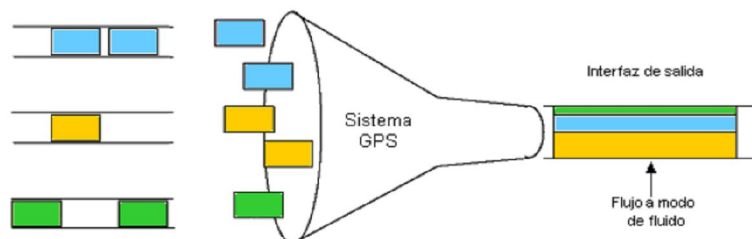


Figura 4. Sistema GPS. Fuente: [44]

En *GPS* todo el tráfico se enviaría simultáneamente mezclado de forma que todos obtienen un flujo constante. *GPS* es un planificador teórico que no puede ser implementado, pero su comportamiento es aproximado al de una disciplina de planificación *WRR* (*Weighted Round-Robin*) bit a bit. En esta disciplina los bits individuales de los paquetes de la cabeza de cada cola son servidos siguiendo el orden asociado a una disciplina *Round-Robin* en la que se ha tenido en cuenta los pesos de cada cola. En un ciclo completo *Round-Robin* se servirá a todas las colas al menos una vez, y las colas de más prioridad podrían ser servidas más de una vez.

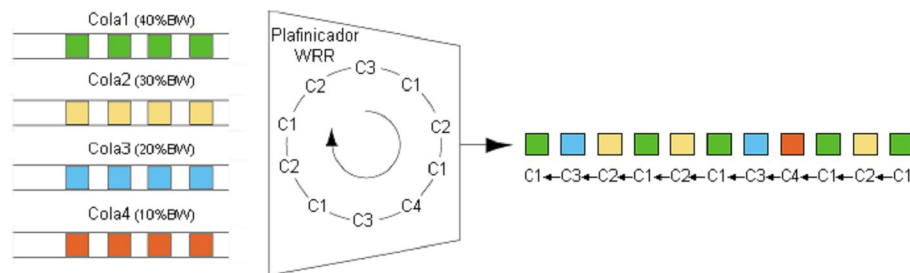


Figura 5. Planificación WRR. Fuente: [44]

Esta aproximación hace posible un reparto justo de ancho de banda, porque tiene en cuenta el tamaño del paquete. De esta manera, cada cola recibe en todo momento la parte de ancho de banda que tiene asignada por configuración. La transmisión de los paquetes desde las diferentes colas bit a bit puede ser soportada por las redes *TDM* (*Time Division Multiplexed*), pero no por redes multiplexadas estadísticamente que trabajen a nivel de conmutación de paquetes ó celdas completas tales como *ATM*, *Frame Relay* ó *TCP/IP*.

Para adaptar el modelo a este tipo de redes se va a suponer la existencia de un reensamblador de paquetes justo al final de la interfaz de salida. Así, el orden en el que cada paquete estaría totalmente ensamblado vendría dado por el orden en que el último bit de cada paquete es desencolado. Esto es lo que se define como tiempo de acabado ó final (*finish time*) y determina el orden en el que los paquetes debe ser desencolados para una disciplina de planificación que trabaje paquete a paquete en vez de bit a bit, como es el caso de *WFQ*.

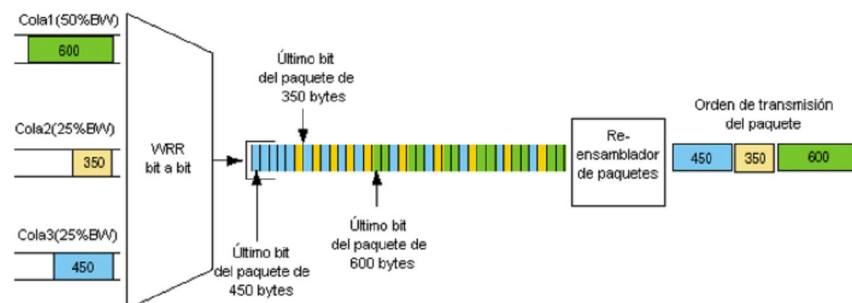


Figura 6. Planificador WRR bit a bit con Re-ensamblador. Fuente: [44]

En la Figura 6 se muestra un planificador *WRR* bit a bit sirviendo 3 colas. Se ha supuesto que la cola1 tenga asignado el 50 por ciento del ancho de banda de la interfaz de salida y que las colas 2 y 3 tendrán asignado el 25 por ciento cada una. El planificador transmite de esta forma, 2 bits de la cola1, 1 de la cola2, 1 de la cola3, y vuelve de nuevo a la cola1. Como resultado de esta disciplina de planificación, el último bit del paquete de 600-bytes de la cola 1 es transmitido antes que el último bit del paquete de 350-bytes de la cola 2, y el último bit del paquete de 350-bytes es transmitido antes que el último bit del paquete de 450-bytes de la cola 3. Esto hace que el paquete de 600-bytes acabe (sea completamente reensamblado) antes que el de 350-bytes, y que éste a su vez acabe antes que el de 450-bytes.

WFQ aproxima esta disciplina de planificación teórica calculando y asignando un tiempo de acabado a cada paquete. A partir de la tasa de bit que permite la interfaz de salida, del número de colas activas, del peso relativo asignado a cada cola, y de la longitud de cada uno de los paquetes de las colas, *WFQ* calcula y asigna un tiempo de acabado a cada paquete que llega a las colas. El planificador selecciona y desencola el paquete que tenga el menor tiempo de acabado de todos los paquetes almacenados en las colas. El tiempo de acabado no es el tiempo que tarda en transmitirse el paquete, sino un número asignado por la disciplina de planificación a cada paquete que se usa únicamente para saber el orden en el que los paquetes deberían ser desencolados.

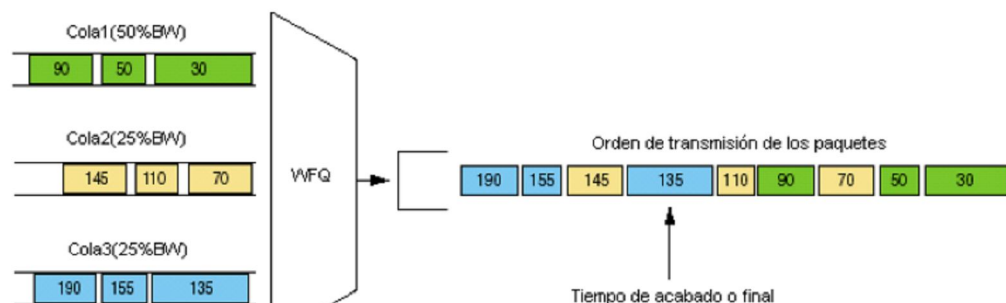


Figura 7. Planificador WFQ. Fuente: [44]

2.2.8. Modelos de Servicio

Los modelos de servicio describen la interfaz entre la red y sus usuarios en la arquitectura de asignación de recursos, es decir, cuales usuarios de servicios pueden hacer solicitudes a la red y qué clase de acuerdos de recursos puede ofrecer la red[25].

IntServ estandarizó dos modelos de servicio básico:

- Servicio Garantizado
- Servicio de carga controlada

2.2.8.1. Especificación de Flujos

Para hacer una reserva, una aplicación debe caracterizar el tráfico que inyectará a la red y especificar los requerimientos del servicio para el flujo. En *IntServ* esto se hace en la “*especificación del flujo*” (flow specification).

La especificación del flujo es en esencia en contrato de servicios que especificara: el tráfico que la fuente enviará, y los recursos y servicios que la red se compromete a prestar[39]. En caso de que la fuente viole su descripción del tráfico (enviando una tasa mayor que la acordada), la red obviamente no será capaz de mantener su compromiso.

Típicamente, el tráfico es analizado por un *control de policía* (o de políticas) antes de que entre a la red para asegurar que el tráfico está conforme con la descripción de tráfico acordada.

Un flujo puede ser caracterizado de muchas formas; la forma más exacta depende del tipo de control de admisión y tipo de planificación de paquetes usados. Los parámetros más comunes de la especificación del flujo son:

- Tasa Pico (*Peak Rate*)
- Tasa promedio (*Average Rate*)
- Tamaño de la ráfaga (*Burst Size*)

➤ Tasa Pico (*Peak Rate*)

Es la tasa máxima a la que puede generar tráfico una fuente y está limitada por los dispositivos de Hardware (no se pueden generar a más de 10 Mbps en una Ethernet de 10 Mbps). En algunos casos es recortada para reducir la tasa pico de la fuente.

Esta tasa puede ser calculada a partir del tamaño de los paquetes y el espaciamiento entre los paquetes consecutivos.

➤ Tasa Promedio (*Average Rate*)

Es la tasa promedio de transmisión sobre un intervalo de tiempo. Puede ser calculada de muchas formas y los resultados pueden ser diferentes.

Típicamente se calcula con una ventana de tiempo movable, por lo que el intervalo de tiempo para promediar puede iniciar en cualquier instante de tiempo.

➤ **Tamaño de la Ráfaga (*Burst Size*)**

Es la cantidad máxima de datos que pueden ser inyectados en la red a la tasa pico. Este tamaño refleja la proporción de ráfagas de la fuente de tráfico.

Para evitar pérdidas de paquetes. El router del primer salto podría tener que asignar un buffer para fuentes más grandes que el tamaño de la ráfaga.

2.2.8.2. Servicio Garantizado

Este servicio provee garantía de ancho de banda y límites estrictos en los retardos de colas *End-to-End* para los flujos que lo conforman. Este servicio fue creado para las aplicaciones que requieren alto aseguramiento del ancho de banda y el retardo (Ej: Aplicaciones de *Playback*)[45].

El servicio garantizado también puede ser usado para aplicaciones que tienen requerimientos fuertes de tiempo real, tales como sistemas de control de misión.

➤ **Comportamiento *End-to-End***

El comportamiento *End-to-End* de un camino que soporta servicio garantizado puede ser visto como un circuito virtual con ancho de banda garantizado.

La partición de ancho de banda entre circuitos virtuales es lógica, por lo que los límites de las particiones pueden ser expandidos. Los flujos de tráfico *best-effort* podrían ser oportunistas y usar el ancho de banda no consumido por flujos reservados.

El servicio garantizado también provee límites estrictos en el retardo. No se controla el retardo mínimo o el retardo promedio, sino el retardo en cola máximo. La garantía de retardo en cola máximo asume que el tráfico entrante esta conforme con los parámetros especificados del *Token Bucket*.

Las aplicaciones deben tomar en cuenta el retardo adicional el cual es el retardo de propagación del camino más el retardo del recorte para acondicionar los flujos de tráfico a los parámetros del *Token Bucket*. El retardo en cola podría aumentar si los flujos violan los parámetros del *Token Bucket*, lo que aumentaría el retardo debido al acondicionamiento (*shaping*) en el regulador *Token Bucket*.

2.2.8.3. Servicio de Carga Controlada

En el servicio garantizado, los recursos deben ser reservados para el peor caso. Para tráfico de ráfagas, esto conlleva a una baja utilización de la red y un costo elevado de la reserva de recursos. Además, es difícil conocer exactamente los

requerimientos del ancho de banda y retardo para una aplicación dada. Para ciertas aplicaciones, se podrían servir sus necesidades con modelos de servicio menos estrictos en las garantías y más bajos en costos.

El Servicio de Carga Controlada no provee ninguna garantía cuantitativa en límites de retardo o de ancho de banda. Trata de emular una red cargada ligeramente para aplicaciones que requieren este servicio. Además, permite multiplexado estadístico, por lo cual permite implementarse de forma más eficiente que el servicio garantizado.

Este servicio es adecuado para aplicaciones adaptativas que requieren algún grado de aseguramiento del desempeño pero sin límites absolutos.

➤ **Comportamiento *End-to-End***

El comportamiento *End-to-End* de carga controlada es algo vago con respecto al servicio garantizado. El servicio de carga controlada es un servicio entre el servicio *best-effort* y el servicio garantizado. Este servicio usa mecanismos de control de admisión y de aislamiento de tráfico. Y se conoce como “servicio mejor que *best-effort*”.

El comportamiento de este servicio puede ser descrito como similar a aquel de una red *best-effort* ligeramente cargada, es decir, que un alto porcentaje de los paquetes transmitidos será entregado exitosamente por la red a los receptores, y el retardo de tránsito en cola experimentado por un alto porcentaje de paquetes entregados no excederá mucho el retardo mínimo.

El servicio de carga controlada no hace uso de valores específicos para controlar los parámetros como el retardo y las pérdidas. La aceptación de una solicitud para este servicio implica que la red tiene suficientes recursos para acomodar el tráfico sin causar congestión.

Otra forma de describir el servicio de carga controlada es describir los eventos que se espera ocurran con alguna frecuencia:

- Retardo en cola promedio pequeño o no existente sobre todas las escalas de tiempo significativamente más grandes que el tiempo de ráfaga (el tiempo requerido para transmitir el máximo tamaño de ráfaga a la tasa solicitada).
- Pérdidas pequeñas por congestión o no existentes sobre las escalas de tiempo significativamente más largas que el tiempo de ráfaga.

En esencia, el servicio de carga controlada permite un pico ocasional de retardos o pérdidas. Sin embargo, la probabilidad de que ocurran tales eventos debe ser

suficientemente baja por lo que el retardo promedio en cola y la tasa de pérdidas promedio, sobre un periodo razonable, es cercana a cero.

2.2.8.2. Algoritmo de *Token Bucket*

El algoritmo *Token Bucket* (Algoritmo del Cubo) es un mecanismo sencillo que se limita a dejar pasar paquetes que lleguen a una tasa que no exceda una impuesta administrativamente, pero con la posibilidad de permitir ráfagas cortas que excedan esta tasa.

La implementación de un filtro *token bucket* (*TBF-Token Bucket Filter*) consiste en un *buffer* (el *bucket* o balde), que se llena constantemente con piezas virtuales de información denominadas *tokens*, a una tasa específica (*token rate*). El parámetro más importante del *bucket* es su tamaño, que es el número de *tokens* que puede almacenar.

Cada *token* que llega toma un paquete de datos entrantes de la cola de datos y se elimina del *bucket*. Asociar este algoritmo con los dos flujos (*tokens* y datos), nos da tres posibles situaciones:

- ✓ Los datos llegan al *TBF* a una tasa que es igual a la de *tokens* entrantes. En este caso, cada paquete entrante tiene su *token* correspondiente y pasa a la cola sin retrasos.
- ✓ Los datos llegan al *TBF* a una tasa menor a la de los *tokens*. Solo una parte de los *tokens* se borran con la salida de cada paquete que se envía fuera de la cola, de manera que se acumulan los *tokens*, hasta llenar el *bucket*. Los *tokens* sin usar se pueden utilizar para enviar datos a velocidades mayores de la tasa de *tokens*, en cuyo caso se produce una corta ráfaga de datos.
- ✓ Los datos llegan al *TBF* a una tasa mayor a la de los *tokens*. Esto significa que el *bucket* se quedara pronto sin *tokens*, lo que causaría que el *TBF* se acelere a si mismo por un rato. Esto se llama una situación sobre límites. Si siguen llegando paquetes, empezarían a ser descartados.

Esta última situación es muy importante, porque permite ajustar administrativamente el ancho de banda disponible a los datos que están pasando por el filtro.

La acumulación de *tokens* permite ráfagas cortas de datos extralimitados para que pasen sin pérdidas, pero cualquier sobrecarga restante causaría que los paquetes se vayan retrasando constantemente, y al final sean descartados.

Se debe tener en cuenta que en la implementación actual, los *tokens* se corresponden a bytes no a paquetes.

En la siguiente figura se muestran los componentes de un filtro *token bucket*.

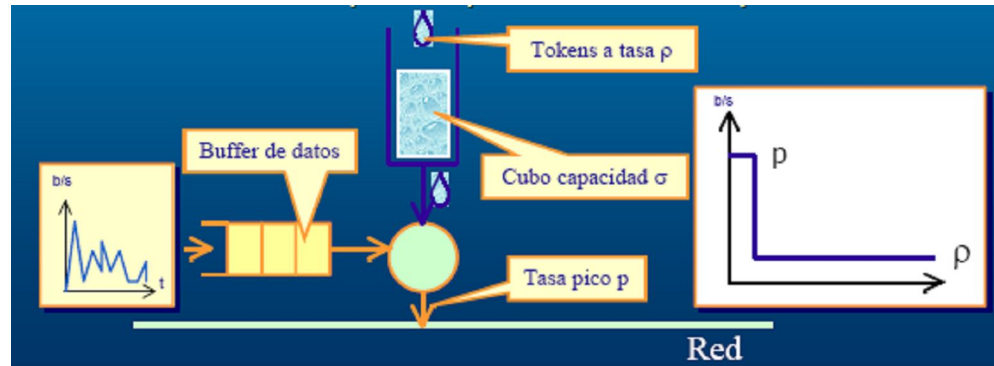


Figura 8. Componentes de un Filtro Token Bucket. Fuente: [46]

2.2.9. Ventajas del Modelo IntServ

- Simplicidad conceptual, facilitando la integración con la administración de políticas de red.
- QoS discreta por flujo, haciéndolo arquitecturalmente conveniente para control de admisión de llamadas de voz, lo cual permite avisar a los puntos finales si el ancho de banda necesario está disponible[47].

2.2.10. Desventajas del Modelo IntServ

- Todos los elementos de la red deben mantener e intercambiar mensajes de estado y señalización por cada flujo, lo cual puede requerir de una cantidad de ancho de banda significativo en redes grandes.
- Se utilizan mensajes de refresco periódicos, lo cual puede requerir protección frente a pérdida de paquete para mantener la(s) sesión(es) intacta(s).
- Todos los nodos intermediarios deben implementar el protocolo de señalización *RSVP*.

2.3. PROTOCOLO DE INTERNET VERSION 6

2.3.1 Introducción a IPv6

La comunicación en las redes de información no sería posible sin los protocolos de red existentes[29]. Un protocolo es un paquete de bits con una cierta estructura que tiene como objetivo permitir que uno o más dispositivos se puedan comunicar entre sí. Los protocolos son la parte de software más importante en las telecomunicaciones para que las aplicaciones instaladas en los dispositivos puedan comunicarse con otras a través de una red.

Para que el navegador de Internet pueda cargar las páginas *Web*, existe detrás un protocolo de comunicación conocido como *HTTP*, que se encarga de establecer comunicación entre el computador (cliente) y en donde se originan las páginas *Web* (servidor).

Los protocolos, además son los encargados del establecimiento y la liberación de una comunicación, así mismo el de establecer el flujo de la información entre dos o más nodos. Una tarea muy importante de los protocolos es la verificación y control de error de los paquetes que transitan por la red, ya que esto permite que la información que se envía a través de la red llegue sin errores, y en caso de suceder algún inconveniente en la transmisión, la información es reenviada nuevamente por el emisor, hasta que ésta llegue a su destino libre de error[48].

En el mundo de Internet, el protocolo más utilizado es el *TCP/IP* (*Transfer Control Protocol / Internet Protocol*). *TCP/IP* es una pila de protocolos que establecen la comunicación de dispositivos a través de Internet que hace posible que las computadoras de diferentes arquitecturas se puedan comunicar entre si (ejemplo, *Macs* con *PCs*). El protocolo más conocido de la pila *TCP/IP* es el de IP (*Internet protocol*).

El protocolo de Internet *IPv4* (*Internet Protocol Version 4*), creado a comienzo de los años 80 por el *DOD* (Departamento de Defensa) de los EEUU para una red militar del gobierno norteamericano llamada *ARPANET*, se extendió a lo que actualmente conocemos como la gran Internet, una red pública mundial con millones de usuarios[49].

Internet ha tenido desde ese momento un crecimiento abrumador implementando nuevos servicios y aplicaciones, en donde *IPv4* se adaptó a estas exigencias con distintas técnicas pero que actualmente no logra cumplir a cabalidad. Algunas de estas deficiencias eran el agotamiento de direcciones disponibles, el no tener incorporado el concepto de *QoS* (*Quality of Service* – Calidad de servicios), movilidad, seguridad entre otros aspectos importantes. Ante estos incumplimientos, la *IETF* (*Internet Engineering Task Force*), que desarrolla los

protocolos estándar de Internet, en 1992 convocó a las comunidades de investigadores a estudiar alternativas para mejorar *IPv4*.

El resultado llegó en 1995 y se llamó *IPv6*[50] (*Internet Protocol Version 6*). Con este protocolo mejorado se agregaron muchas de las funcionalidades que actualmente son muy útiles. Además de aumentar la cantidad de direcciones, también se incluyeron mejoras con respecto a la implementación nativa de seguridad de datos (*Ipsec*), calidad de servicios (*QoS*), autoconfiguración de nodos (*Neighbour Discovery*), movilidad e innovaciones en dispositivos de red que mejoraron la calidad y velocidad de los datos.

2.3.1.1. Nuevas Características de *IPv6*

El nacimiento de este nuevo protocolo no ha venido solo propiciado por la escasez de direcciones *IPv4*, sino que además se añaden nuevas características y se mejoran las existentes. A continuación detallaremos un poco más cada una de estas nuevas características[49].

➤ Aumento del espacio de direcciones

El protocolo *IPv4* estaba basado en una arquitectura que utiliza direcciones de 32 bits. Con la nueva versión del protocolo las direcciones constan de 128 bits, permitiendo así una cantidad de 10^{38} o más direcciones. Podemos decir que una “desventaja” de estas nuevas direcciones es su dificultad para recordarlas dado a su tamaño.

➤ Autoconfiguración

Cuando un nodo se conecta a la red, éste recibe los datos necesarios para empezar a comunicarse por parte del enrutador: dirección *IPv6*, máscara de red y rutas. Hay que recordar que este nuevo protocolo trata de simplificar. Con el *IPv4* tenemos el *DHCP* (*Dynamic Host Configuration Protocol*, Protocolo de Configuración Dinámica de Nodo) para conseguir algo similar.

➤ Movilidad

Con esta funcionalidad podremos “saltar” de una red a otra sin apenas percibir ningún cambio. Si bien esto ya era posible con *IPv4* de una manera más bien ardua, en *IPv6* fue uno de los requerimientos de diseño. Esta característica es de gran importancia en el funcionamiento de las nuevas redes de telefonía con tecnología *UMTS* (*Universal Mobile Telecommunications System*).

➤ Seguridad

Este fue otro de los requerimientos del diseño del nuevo protocolo: todas las aplicaciones se deben beneficiar de las facilidades de autenticación y encriptación de datos de forma transparente. El estándar escogido para esto fue IPsec.

➤ Encaminamiento Jerárquico

El encaminamiento bajo *IPv6* es bastante similar al de *IPv4* con *CIDR*, es decir, jerárquico y sin clases. Con esto se pretende conseguir que las entradas en las tablas de rutas en los backbones no abunden más de lo necesario. Al mismo tiempo, se consigue simplificar el enrutamiento y se espera que los enrutadores sean más rápidos.

➤ Multi-Homing

Esta funcionalidad se consigue con direcciones *anycast*. Una dirección *anycast* identifica a un conjunto de distintas interfaces, encontrándose estas, por norma general, en distintos nodos. Un paquete a una dirección *anycast* será entregado a un solo miembro del conjunto. En principio, el paquete será entregado al miembro más cercano según el concepto de cercano de los protocolos de encaminamiento.

➤ Calidad de Servicio (QoS)

Si bien con *IPv4* tenemos unos pocos bits para el control del tipo de servicio, *ToS*, con *IPv6* disponemos de campos más amplios para definir la prioridad y flujo de cada paquete. Según el contenido de este campo, el enrutador deberá darle un trato más o menos especial.

2.3.2. La Cabecera *IPv6*

La cabecera de un paquete *IPv6*[51] es, sorprendentemente, más sencilla que la del paquete *IPv4*, además de que la funcionalidad del protocolo *IPv6* es mucho mayor.

La cabecera de un paquete *IPv4* es variable, por lo que necesita un campo de tamaño o *length*. Sin embargo, para simplificar la vida de los routers, *IPv6* utiliza un tamaño de cabecera fijo de 40 bytes, que componen un total de 8 campos:

- ❖ Versión (4 bits), sirve para que el enrutador se entere de que es un paquete *IPv6*.
- ❖ Dirección origen y de destino (128 bits cada una), son direcciones de los nodos *IPv6* que realizan la comunicación.

- ❖ Clase de tráfico (8 bits), para poder diferenciar entre servicios sensibles a la latencia, como *VoIP*, de otros que no necesitan prioridad, como tráfico *http*.
- ❖ Etiqueta de flujo (20 bits), permite la diferenciación de flujos de tráfico. Esto tiene importancia a la hora de manejar la calidad de servicio (*QoS*).
- ❖ Siguiente cabecera (8 bits), este campo permite a los enrutadores y hosts examinar con más detalle el paquete. A pesar de que el paquete básico *IPv6* tiene cabecera de tamaño fijo, el protocolo puede añadir más para utilizar otras características como encriptación y autenticación.
- ❖ Tamaño de *payload* (16 bits), describe el tamaño en octetos de la sección de datos del paquete. Al ser este campo de 146 bits, podremos usar paquetes de hasta más de 64000 bytes.

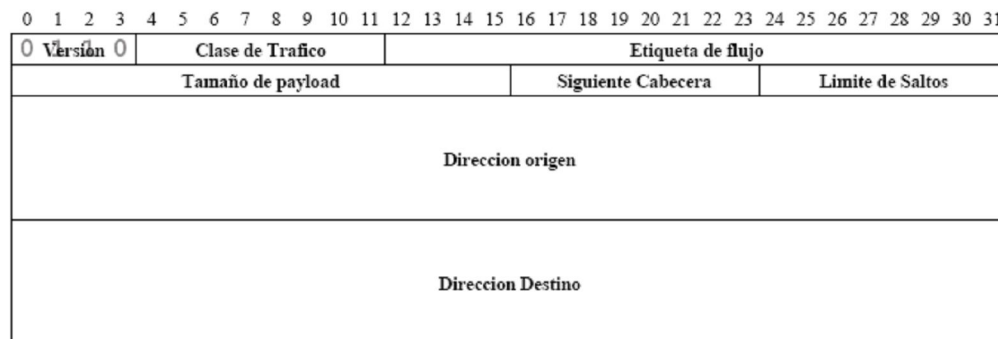


Figura 9. El paquete IPv6. Fuente: [49]

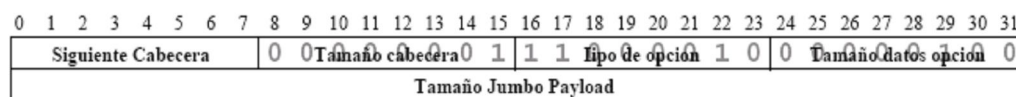


Figura 10. La siguiente cabecera. Fuente: [49]

- ❖ Límite de saltos (8 bits), especifica el número de saltos al enrutador que puede hacer el paquete antes de ser desechado. Con 8 bits se puede tener un máximo de 255 saltos.

2.3.3. El Campo de Siguiente Cabecera (*Next Header Field*)

Como se dijo anteriormente, el tamaño de la cabecera *IPv6*[52] básica es fijo. Dentro de esta cabecera existe un campo llamado de siguiente cabecera que permite describir con más detalle las opciones del paquete. Esto quiere decir que

en realidad tendremos una cabecera de tamaño fijo por norma general y otra cabecera de tamaño variable en caso de que utilicemos alguna de las características avanzadas. En la tabla 8 se muestran las opciones presentes en la siguiente cabecera:

Siguiente cabecera	Valor del Campo
Opciones de Hop-by-Hop	0
Opciones de destino	60
Encaminamiento	43
Fragmento	44
Autenticación	51
Encapsulación	50
Ninguna	59

Tabla 8. Opciones presentes en la Siguiente Cabecera. Fuente: [49]

Esta arquitectura es muy flexible, ya que cada cabecera tiene un campo de siguiente cabecera, con lo que se pueden tener varias opciones agregadas. Con la cabecera de encaminamiento conseguimos la funcionalidad equivalente de *IPv4* de *Source-Routing*, es decir, especificar los nodos intermedios por los que ha de pasar el paquete.

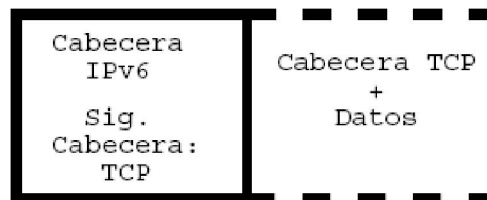


Figura 11. Cabecera IPv6 básica y datos. Fuente: [49]



Figura 12. Cabecera IPv6 básica, fragmento y datos. Fuente: [49]

Una cosa que debe quedar clara es que los nodos intermedios o enrutadores no deben examinar más que la cabecera *IPv6* básica. Existen excepciones como en el caso de que existan cabeceras de opciones *Hop-by-Hop* o, como en el caso anterior, que exista una cabecera de encaminamiento en el que sólo los nodos en ella definidos deberán alterar el paquete[53].

La especificación recomienda además el siguiente orden para las cabeceras adicionales:

- ✓ Cabecera *IPv6* básica.
- ✓ Opciones *Hop-by-Hop*.
- ✓ Opciones de destino.
- ✓ Encaminamiento.
- ✓ Fragmento.
- ✓ Autenticación.
- ✓ Encapsulación.
- ✓ Opciones de destino.
- ✓ Cabecera nivel superior.

Las opciones de destino pueden ser procesadas en momentos distintos dependiendo de si el paquete atraviesa un nodo intermedio o llega al nodo destino. La única restricción de la especificación es que las opciones de *Hop-by-Hop* han de ir siempre en la cabecera básica.

2.3.4. Direccionamiento *IPv6*

Como se ha dicho anteriormente, las direcciones *IPv6*[23] son identificadores de interfaces y/o conjuntos de interfaces de 128 bits. Tenemos tres tipos de direcciones:

- *Unicast*: identificará una solo interfaz. Un paquete enviado a una dirección unicast se entregará a una solo interfaz.
- *Anycast*: Identificará un conjunto de interfaces, probablemente en distintos nodos. Un paquete enviado a una dirección de este tipo será entregado sólo a unos de los nodos, que debería ser, en principio, el más cercano.
- *Multicast*: Igual que en el caso anterior, identificará a un conjunto de interfaces que estarán seguramente en nodos distintos. Pero, en este caso, el paquete será enviado a todos los nodos del conjunto.

En las siguientes figuras se puede ver un ejemplo de comunicación entre tres nodos con direcciones A, B y C. Y los distintos comportamientos según el tipo de comunicación.

Con *IPv6* dejan de existir las direcciones *broadcast*, cuya funcionalidad es absorbida por las direcciones *multicast*.

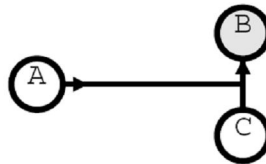


Figura 13. Ejemplo comportamiento unicast. Fuente: [49]

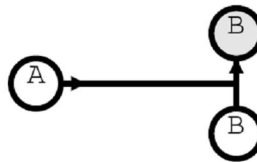


Figura 14. Ejemplo comportamiento anycast. Fuente: [49]

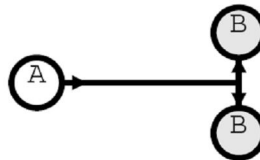


Figura 15. Ejemplo comportamiento multicast. Fuente: [49]

2.3.5. Modelos de direccionamiento

Cualquier tipo de dirección se asigna a interfaces, no nodos. Todas las interfaces tienen, por lo menos, una dirección de enlace local (*Link-Local*) de tipo *unicast*. Una misma interfaz pueden tener asignadas múltiples direcciones de cualquier tipo (*unicast*, *anycast*, *multicast*) o ámbito (*scope*, que se explica más adelante). Direcciones *unicast* con ámbito mayor que el de enlace no son necesarias para interfaces que no son usados como origen y destino de paquetes *IPv6* hacia o desde vecinos. Esto significa que para la comunicación dentro de una *LAN* no nos hacen falta direcciones *IPv6* globales, sino que tenemos más que suficiente con direcciones de ámbito local. De hecho, es lo aconsejable para enlaces punto a punto.

Respecto a los prefijos de subred, *IPv6* sigue el mismo modelo que *IPv4*, es decir, un prefijo se asocia a un enlace, pudiendo haber varios prefijos en un mismo enlace.

El protocolo *IPv6* añade soporte para direcciones de distintos ámbitos, lo que quiere decir que se tiene direcciones globales y no globales. Si bien con *IPv4* ya se había empleado direccionamiento no global con la ayuda de prefijos de red privados, con *IPv6* esta noción forma parte de la propia arquitectura de direccionamiento. Cada dirección *IPv6* tiene un ámbito, que es un área dentro de la cuál ésta puede ser utilizada como identificador único o varios interfaces. El ámbito de cada dirección forma parte de la misma dirección, con lo que se puede diferenciar a simple vista.

2.3.6. Autoconfiguración

Como se ha dicho anteriormente, ésta es otra de las nuevas características que trae consigo *IPv6* respecto a *IPv4*. El proceso de autoconfiguración consiste en lo siguiente: creación de una dirección de enlace local (*link-local*) y comprobar su unicidad en el enlace, determinar qué tipo de información se debe autoconfigurar (direcciones, otros datos o los dos) y, en el caso de direcciones, determinar qué mecanismo se debe usar para obtenerlas (con control de estado, sin él o con los dos). La obtención automática de configuración sólo se aplica a hosts y nunca a enrutadores (si bien éstos pueden generar su propia dirección de enlace local), ya que los hosts necesitan información anunciada por éstos y alguien tendrá que configurarlos.

IPv6 define tanto el mecanismo de autoconfiguración con el control de estado como sin control de estado. En el caso de sin control de estado, la configuración necesaria es nula en los nodos y prácticamente nula en los enrutadores. El mecanismo permite al host obtener una dirección a partir de información local (el identificador de interfaz) e información anunciada por los enrutadores (el prefijo de subred). En caso de que no haya enrutadores en la red, los hosts pueden generar sus propias direcciones de enlace local (*link-local*), siendo esto suficiente para comunicarse entre sí.

En el caso de autoconfiguración con control de estado, los hosts obtienen sus direcciones y otra información de algún servidor. Este servidor puede mantener un control preciso de qué direcciones han sido asignadas a cada host. Se ha desarrollado una versión específica de *DHCP* para *IPv6* llamada *DHCPv6* para tal efecto.

No necesitamos control de estado cuando un 'sitio' no se preocupa en exceso por las direcciones que usa cada host mientras éstas sean válidas y encaminables. Sí lo necesitamos en el caso de que necesitemos saber qué host usa qué dirección en todo momento por cualquier razón (quizá inventario). En cualquier caso, siempre se puede usar los dos métodos para configurar según qué cosas.

Para asegurarnos de que las direcciones sean únicas, los nodos ejecutan un algoritmo de detección de direcciones duplicadas.

A partir de ahora hablaremos exclusivamente del mecanismo sin control de estado, lo que no significa que algunas de las cosas sean comunes.

2.3.7. Movilidad

Se entiende por movilidad como la facilidad para cambiar de red, tanto a nivel físico como a nivel lógico, sin perder el transporte ni las conexiones establecidas por capas de nivel superior al IP. Para que esto sea posible, se debe mantener una misma dirección *IPv6* no importa el lugar donde se encuentre y los paquetes enviados hacia nosotros se encaminaran a este lugar.

Todo nodo móvil (*Mobile Node, MN*) tendrá una dirección 'de casa' (*Home Address, HA*), que será su dirección en su red origen. Esta dirección se mantendrá aunque cambiemos de red. Los paquetes que se envíen al nodo móvil estando éste en su red origen serán encaminados de forma normal, como si el soporte de movilidad no existiese.

En el momento en que el nodo móvil pasa a una red que no es la suya de origen, éste obtendrá una nueva dirección 'de invitado' (*Care-of-Address, CoA*). A partir de ese momento el nodo puede ser contactado también a través de esta *CoA* y lo siguiente que hace el nodo móvil es contactar con un enrutador de su red origen (*Home Agent, HA*) y comunicarle cual es su *CoA* actual. De esta forma, cuando un paquete es enviado a la 'dirección de casa', el enrutador sabe que tendrá que interceptarlo con destino a la *CoA* del nodo móvil.

Lo que en realidad hace el *MN* cuando se mueve es mandar un mensaje de *Binding Update (BU)* al *HA*. El *BU* asocia la *CoA* con la dirección 'de casa' del nodo móvil durante un cierto periodo de tiempo.

Se le llama nodo correspondiente (*Correspondent Node, CN*) a cualquier nodo, ya sea fijo o móvil que se comunique con un *MN*. Cuando un nodo móvil se comunica con un *CN*, el *MN* envía directamente los paquetes utilizando la dirección 'de invitado' que ha obtenido en la red que se encuentre. Sin embargo, el *CN* envía los paquetes a la dirección 'de casa' del *MN*, que serán interceptados por el *HA* y reenviados a la *CoA* del nodo móvil. En este caso se tiene una ruta triangular, que no es ningún problema, pero es ineficiente. Para resolver esto, *MobileIPv6* presenta el concepto de optimización de ruta. Este mecanismo permite al *MN* avisar al *CN* de que puede enviarle los paquetes directamente a su *CoA* utilizando para ello mensajes de *Binding Update*.

2.4. PROPUESTA DE INTSERV6

El fundamento principal de esta propuesta [35] es el uso de *IntServ* para soportar Calidad del servicio en redes *IPv6*. Además, se busca que el desempeño de los Routers de Internet que soportan *IntServ* sea mayor que el que se ofrece actualmente. Para ello se utiliza la etiqueta de flujo de *IPv6* para agilizar el proceso de clasificación de paquetes en los Routers.

2.4.1. Clasificación de flujos en *IntServ6*

El fundamento principal de *IntServ6* es el uso de una *Arquitectura de Servicios Integrados* para soportar *Calidad de servicio* en redes *IPv6*. Además, se busca que el desempeño de los *Routers de Internet* que soportan *Servicios Integrados* sea mayor que el que se ofrece actualmente. Para ello *IntServ6* utiliza el campo *Etiqueta de Flujo* de la cabecera *IPv6* para agilizar el proceso de *Clasificación de Paquetes* en los *Routers*. Este proceso es realizado dentro del modulo de *Identificación de Flujos* y es un aspecto clave para diferenciar nuestra propuesta con respecto a el estándar *IntServ*.

IntServ6 [54] agiliza el proceso de *clasificación de paquetes* en los *Routers IntServ* para el caso de *IPv6*. Para ello, se descarga la responsabilidad del cálculo del *Número Hash* en el *Host* origen del flujo. Esto puede hacerse sin problema ya que mientras el algoritmo de cálculo sea el mismo siempre, si la semilla es la misma (la *Quíntupla*), no importa el lugar donde se aplique la función *Hash*, siempre arrojará el mismo resultado. El número *Hash* es llevado a lo largo del camino de los paquetes durante el establecimiento de la reserva y para su transporte se usan los mensajes *PATH* y *RESV* de *RSVP*. Una vez establecida la reserva en el camino de paquetes, el *Número Hash* asignado al flujo se utiliza como *Identificador del Flujo* de paquetes y es transportado en el campo *Flow Label (Etiqueta de Flujo)* de la cabecera de los paquetes *IPv6*. De esta forma, cada vez que un *Router* toma un paquete entrante (Ver Figura 16), revisa su *Etiqueta de Flujo* y luego utiliza este número como un apuntador a la *Tabla Hash* para encontrar los datos de la *Reserva* y el enlace saliente en la *Tabla de Reservas*.

Estos datos han sido almacenados previamente durante el proceso de establecimiento de la reserva. El manejo de la *Etiqueta de Flujo* ha sido descrito en y nuestra propuesta está acorde con las políticas allí planteadas.

Debido a que se utiliza un número *Hash* como identificador de un flujo, existe la posibilidad de que haya una *colisión*, es decir, que el número *Hash* que se calculó en el *Host* ya esté asignado en el *Router* a otro flujo. Para solucionar este inconveniente se utiliza una *Tabla de Resolución de Colisiones* (Ver Figura 16) que contiene la información de las etiquetas que han sido asignadas a varios flujos, así como los datos de la *Quíntupla* de cada uno de estos flujos y un apuntador a la *Tabla de Reservas*. Cuando ocurre una colisión, el *Router* busca en

esta tabla cuál de las quintuplas que tienen la misma *Etiqueta de Flujo* es la que coincide con la del paquete entrante para así encontrar los datos de la reserva y el enlace de salida en la *Tabla de reservas*.



Figura 16. Tabla de Manejo de las Reservas en IntServ6. Fuente: [54]

2.5. NS2 – NETWORK SIMULATOR

2.5.1 Introducción

A través de los años se ha hecho importante el modelamiento de diversos eventos antes de tomar decisiones[55]. Tal es el caso de la programación lineal, donde se plantea un problema de la vida cotidiana y mediante un modelamiento matemático es factible encontrar una o más soluciones. Sin embargo, en la realidad hay que considerar miles de factores que lamentablemente un modelo matemático no considera en el mayor de los casos, pero aproxima bastante a una solución que nos puede llevar por un buen camino. Es por ello que la idea de modelar y/o simular es bastante importante en la toma de decisiones.

En este capítulo se tratará lo referente a el manejo del simulador de redes *NS-2*[56], se describirá como se ha desarrollado, desde el inicio, el sistema cliente servidor en el *NS-2*, describiendo las principales funciones, la forma de acceder desde *Otcl* (lenguaje utilizado en el simulador junto a C++ para desarrollar escenarios de simulación) y algún *script* de ejemplo. También se detallará el entorno propuesto de simulación, así como los distintos parámetros que lo caracterizan.

NS-2 es un simulador de redes, que esta escrito en C++. La simulación se escribe, como se dijo anteriormente, en lenguaje *Otcl*, que es un lenguaje orientado a objetos de tipo intérprete, es decir, que las instrucciones del código se va traduciendo una a una conforme se va ejecutando, dándole una flexibilidad durante el desarrollo del código para hacer la simulación. De esta forma los enlaces entre los dos (*links*), son los objetos *Otcl* que influyen y que se pueden programar situaciones como retrasos, gestión de colas, módulos de pérdidas, errores, etc. Si se quiere modificar alguno de estos parámetros o incluir uno propio, se emplea el C++.

Una vez hecha la simulación, se puede realizar dos cosas para conocer los resultados: Ver el funcionamiento de la red simulada en forma de gráfico animado con un programa llamado *NAM*, y también se puede analizar la carga de la red a través de un programa que da a conocer de forma gráfica la relación tiempo-carga de la red a través de *XGRAF*.

2.5.1.1. Instalación del NS-2

El software necesario para la instalación de *NS-2* en el sistema operativo *Linux*, puede conseguirse de forma gratuita en la siguiente página Web: <http://www.isi.edu/nsman>. En esta misma página, se encuentran enlaces a

diferentes manuales, tutoriales, preguntas frecuentes (*Frequent Answer Questions, FAQ's*), acceso a foros, etc.

El simulador se puede instalar de dos formas, paquete por paquete o todo a la vez.[57] Se recomienda para aquellos usuarios que no sean expertos en NS-2 que utilicen esta segunda opción. Para ello, se deberá descargar un fichero comprimido con todos los paquetes, con “todo en uno”, que ocupa unos 50 Mbytes de espacio son descomprimir. El nombre del fichero para la versión utilizada es el siguiente: *ns-allinone-2.31.tar.gz*.

Una vez descomprimido el paquete, se obtienen todos los paquetes desplegados en el directorio *ns-allinone-2.31* en la estructura de ficheros adecuada para su correcto funcionamiento. Para compilar los ficheros fuente se deberá ejecutar el script de instalación en directorio del ns.

También se recomienda ejecutar el *script* de validación para ver si funcionan correctamente los paquetes en el sistema operativo.

Se recomienda configurar adecuadamente los *path's* del sistema para poder ejecutar ns, *Nam* y *Xgraph* desde cualquier directorio. Lo más sencillo será editar el fichero *.bashrc* añadiendo los *path's* y las librerías necesarias.

Si es la primera vez que se usa NS-2, resulta muy útil (y recomendable) estudiar el tutorial que se encuentra en la siguiente dirección: <http://www.isi.edu/nsnam/ns/tutorial/index.html>.

2.5.2 Descripción Interna del NS-2

El *Network Simulator 2 (NS-2)* es un simulador ampliamente aceptado y utilizado en el análisis de redes[55]. NS-2 es un simulador de eventos discretos que permite la simulación de entornos diversos, tales como: varios protocolos de red (transporte, *multicast*, *routing*), topologías simples o complejas (incluyendo la generación de dichas topologías), agentes (que definen como puntos extremos donde los paquetes se crean o se consumen a nivel de red), varias fuentes de generación de tráfico, simulación de aplicaciones (*FTP*, *Telnet*, *Web*), diversas políticas de gestión de colas, modelado de errores, redes de área local, redes inalámbricas, etc.

El código del simulador NS-2[58] está escrito en C++ y en *Object Tcl* (*Otcl*). Hay una correspondencia biunívoca entre una clase en C++ (jerárquica compilada de clases en NS-2) y una clase en *Otcl* (jerárquica interpretada). Esta arquitectura de programación permite una gran personalización de la simulación de las rutinas a nivel de paquete software (implementadas en C++), y una configuración y un control flexible de la simulación usando un lenguaje sencillo de interpretar, como el *Otcl*.

En la simulación se toma en cuenta lo que es la estructura (topología) de la red y el tráfico de paquetes que posee la misma, con el fin de crear una especie de diagnóstico que nos muestre el comportamiento que se obtiene al tener una red con ciertas características.

En el siguiente gráfico se muestra una vista bastante simplificada de lo que es NS.

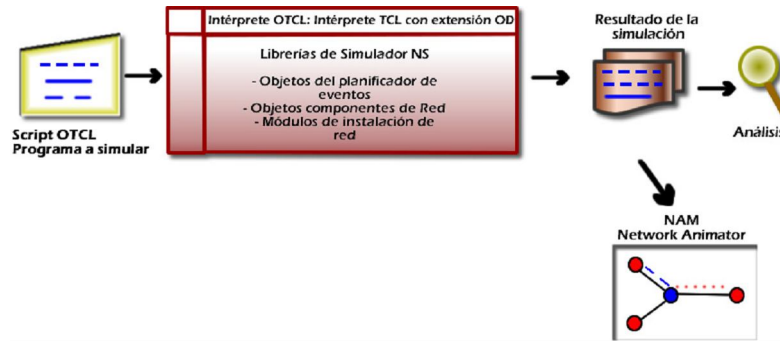


Figura 17. Vista simplificada del funcionamiento de NS. Fuente: [59]

Como se puede observar, se comienza con un script en Otcl que viene a hacer lo que el usuario codifica para simular. Es el único *INPUT* que da el usuario al programa. El resto es el procesamiento interno del NS. La simulación queda en un archivo que puede ser bastante incómodo de leer o analizar para el usuario, sin embargo, usando una aplicación especial se puede mostrar mediante una interfaz gráfica.

2.5.2.1. Scripts de entrada

Toda la información de configuración y control de una simulación en ns-2[60] está especificada en el *script Otcl* de entrada. Los objetos de simulación (nodos, enlaces, fuentes de tráfico, etc.) son creados a través de instancias en el *script*, e inmediatamente se reflejan en la jerarquía compilada de C++. El script de entrada define la topología, construye los agentes (fuentes y destinos), especifica los ficheros de trazas y los tiempos de comienzo de los eventos iniciales en la simulación. Estos eventos iniciales, probablemente creen nuevos eventos a lo largo de la simulación, estos harán lo propio, y así sucesivamente.

2.5.2.2. Event Scheduler Object (Planificador de Eventos)

Este evento en NS, es un paquete único con una calendarización o programa dado por el programador en la codificación. Internamente se identificará con un puntero al objeto que maneja este evento. En la figura 18 se muestra la forma de calendarizar los eventos.

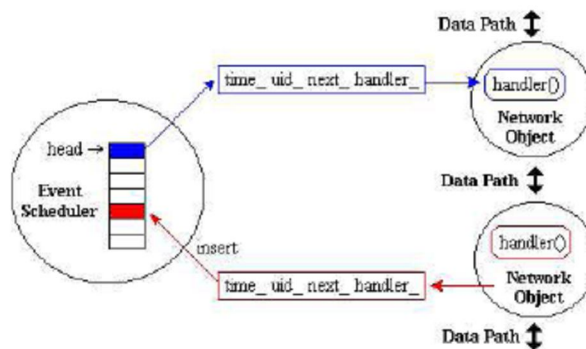


Figura 18. Planificador de eventos. Fuente: [59]

Los usuarios principales del planificador de eventos es el *Network Component* que se verá más adelante. Esto porque la transmisión de paquetes requiere de ciertos tiempos o retardos necesarios para la simulación. Por ejemplo, al declarar un *link* con un ancho de banda muy bajo, el planificador de eventos deberá realizar retardos más prolongados en ese enlace para simular que la transmisión es lenta.

Por otro lado, cada objeto de red usa un planificador de eventos, que es quien maneja el evento por el tiempo planificado. Importante es hacer notar que la trayectoria de datos entre los objetos de la red es diferente de la trayectoria del evento.

2.5.2.3. *Network Component Object*

Se encarga de hacer consistente la comunicación que hay entre distintos componentes de red, por donde pasarán los paquetes. Los componentes de red pueden ser; ancho de banda de un *link*, un *link* unidireccional o bidireccional, retardos de paquetes, etc. En el caso de los retardos también actúa el *event scheduler*.

Como un ejemplo, en la figura 19 se muestra el componente de red que permite unir dos nodos, es decir, un *link*.

En esta figura se representa un *link* simple unidireccional. En el caso de requerir uno bidireccional, simplemente se crea otro objeto con la misma estructura para el lado contrario. En la entrada al *link* el paquete deberá quedar en cola y se realizarán una serie de procedimientos dependiendo del tipo de cola que tenga ese *link*, tales como, si el tamaño del paquete supera el tamaño de la cola, o si la cola simplemente está llena, etc. Considerando esto, se tomará la decisión si el paquete es descartado, en cuyo caso pasará a *Drop* y a un agente NULO. De lo contrario, se realizará un retardo simulado (*Delay*) del que se hablaba anteriormente.

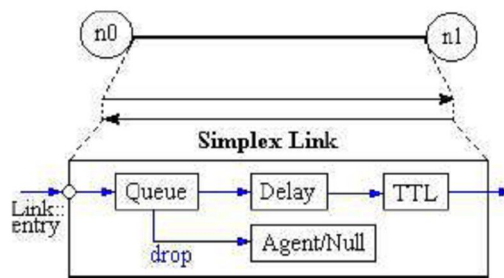


Figura 19. Componente de red – Link. Fuente: [59]

Finalmente se recalculará y actualizará el *TTL* (*time to live*, tiempo de vida) del paquete para llegar al nodo destino.

2.5.2.4. Network Setup Helping Module

Por último, el *Network Setup Helping Module* indicará las bibliotecas necesarias para realizar la simulación. Esto es necesario ya que los anteriores componentes (*Event Scheduler Object* y *Network Component Object*), están escritos y compilados en C++ y están disponibles para el intérprete *Otcl* a través de un *linkage*. La razón no es muy clara, pero tiene que ver con el tiempo de procesamiento (no de simulación). Se puede hacer la analogía entre C con C++ y *tcl* con *Otcl*.

En la siguiente figura 20 se logra mostrar la forma en que se comunican las bibliotecas compiladas de C++ y *Otcl*. Más bien, es *Otcl* que llama a estas bibliotecas[61].

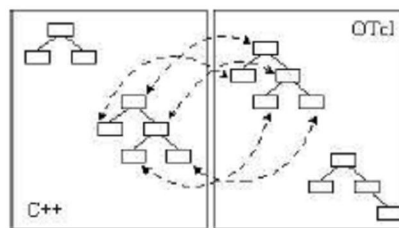


Figura 20. Linkage entre bibliotecas C++ y Otcl. Fuente: [59]

2.5.3. Ejecutar un Script

Para ejecutar la aplicación, *Network Simulator* toma como *INPUT* a un *script* de *Otcl*. En este *script* se define físicamente la configuración de la red (nodos, conexiones entre nodos), los protocolos que serán usados en las conexiones y definiciones específicas de aplicaciones usando tipos de conexiones.

El script es un archivo con extensión *.tcl*. Para hacerlo correr se debe ejecutar *ns ejemplo1.tcl* desde la línea de comandos y esto creará un archivo que contendrá el *OUTPUT* del análisis, un archivo de extensión *.nam* (o el similar *.tr* que más

adelante se analizará la diferencia). Este archivo es una completa descripción de la simulación, donde cada línea describe los paquetes recibidos, enviados, encolados, sacados de cola, etc. Sin embargo, por mucho que se mire este archivo, será muy difícil obtener una gran fotografía de lo que sucede en la simulación. Es por ello que la visualización se realiza mediante el programa *nam* y se ejecuta simplemente con el comando *nam ejemplo1.nam*. Para entender mejor se muestra la figura 21 donde se muestran los pasos para ejecutar un script:

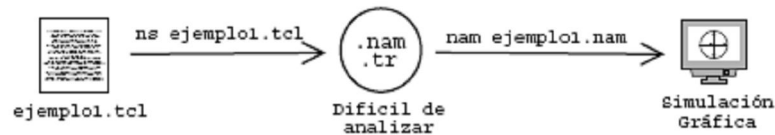


Figura 21. Pasos para realizar la simulación. Fuente: [59]

A parte de generar el archivo *.nam*, también puede generar otro archivo *.q*, que contiene información acerca de una cola de un nodo en particular durante la simulación.

2.5.4 Nam

El *Network Animador*, más conocido como *Nam*[13], viene incluido en la implementación de *NS-2* como una aplicación externa que permite visualizar de forma gráfica lo que sucede durante la simulación. Su funcionamiento es sencillo, de modo que a partir de las definiciones del entorno establecidas en el fichero *Tcl*, crea un entorno gráfico que representa los nodos y enlaces del sistema.

El *Nam* permite, además, ver qué es lo que pasa a lo largo del tiempo, pudiendo personalizar la velocidad de representación, retroceder, avanzar o manejar el tiempo a gusto propio. Esto lo consigue a partir del fichero de traza definido anteriormente, que especifica los eventos en cada instante, de modo que el *Nam* se encarga de interpretar dicho fichero en formato de salida de *NS-2* y convertirlo a un entorno visual más agradable para el usuario. Un ejemplo de una representación visualizada con el *Nam*, podemos verlo a continuación en la figura 22.

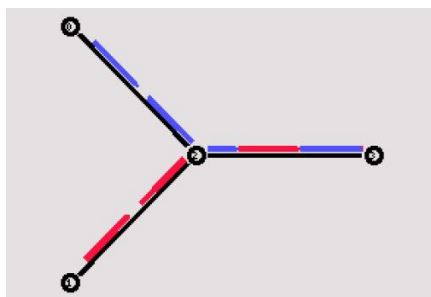


Figura 22. Ejemplo de animación visualizada con el Nam. Fuente: [13]

La simulación puede controlarse a través de un panel visual y de botones específicos, los cuales quedan representados en la figura 23, donde podemos apreciar las múltiples opciones que nos facilita este *Nam* a la hora de visualizar cómo se desarrolla la animación.

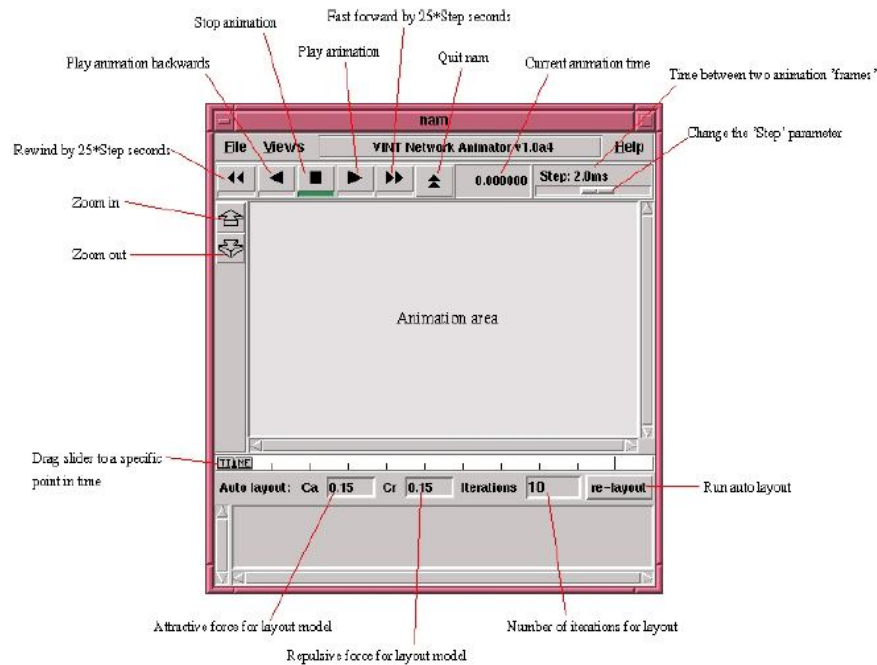


Figura 23. Panel de Control del Nam. Fuente: [13]

2.5.5. Xgraph

También es posible obtener trazas personalizadas en ficheros externos, de modo que se puedan monitorizar distintas variables del programa y su evolución en determinados momentos de la simulación. *Xgraph* [13] es una aplicación incluida en la instalación del *Network Simulator* que permite dibujar gráficas en dos dimensiones a partir de los datos de los ficheros organizados de una forma determinada. El formato que admite este programa es el de los ficheros con dos columnas de datos: la primera indica la coordenada X (abscisas) y la segunda indica su correspondiente valor en el eje Y (ordenadas).

Un uso común de esta utilidad suele ser el generar un evento que cada cierto tiempo vaya apuntando en un fichero el valor del tiempo actual de simulación y también el valor de la variable a monitorizar, de modo que al final lo que obtenemos es la evolución temporal de dicha variable y la gráfica correspondiente. Podemos ver un ejemplo de dicha representación en la Figura 24.

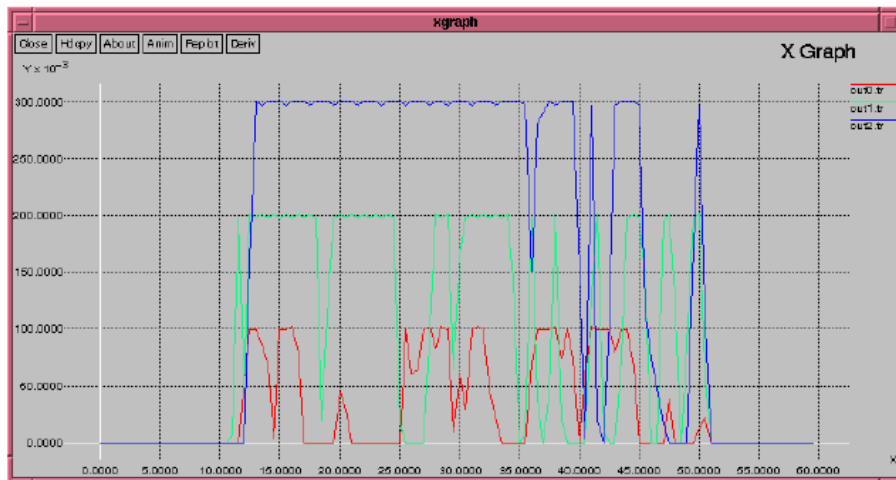


Figura 24. Ejemplo de Visualización de resultados con Xgraph. Fuente: [13]

Todo lo relacionado con este simulador de redes se puede encontrar aquí: [62]

2.5.6. RSVP/NS

El protocolo de reservas de recursos *RSVP* [63] (*Resource Reservation Protocol*) forma parte de la arquitectura de Servicios Integrados (*IntServ*), y permite a las aplicaciones llevar a cabo reservas de recursos.

Para utilizar *RSVP/NS*, básicamente lo que debemos saber es cómo crear/configurar enlaces y agentes *RSVP*.

2.5.6.1. Configuración de un enlace *RSVP*

Para crear un enlace *RSVP* entre los nodos *n1* y *n2* se utiliza el siguiente comando:

```
ns duplex-rsvp-link <n1> <n2> <bw> <delay> <reservable> <rsvp> <queue>
<adc> <est>
```

Donde cada argumento significa lo siguiente:

- *ns*: es una instancia del simulador.
- *bw*: el ancho de banda para este enlace.
- *delay*: el retardo del enlace.
- *reservable*: la porción del ancho de banda del enlace que se puede usar para *RSVP*.
- *rsvp*: el ancho de banda (en bits por segundo) que queda reservado para mensajes *RSVP*.
- *queue*: el tamaño de la cola para la clase *best-effort* (en bytes).

- *adc*: el algoritmo de control de admisión.
- *est*: el estimador usado para las medidas en las que se basan los algoritmos de control de admisión.

2.5.6.2. Creación y Configuración de agentes *RSVP*

Un agente *RSVP* se añade a un nodo con el comando:

```
n1 add-rsvp-agent
```

```
set rsvpagent [$n1 add-rsvp-agent]
```

Todas las opciones se pueden fijar para cada agente *RSVP* o globalmente mediante el siguiente comando:

```
Agent/RSVP set <option> <value>
```

2.6. PROGRAMACIÓN ORIENTADA A OBJETOS

2.6.1 Introducción

La *programación orientada a objetos* [64] (*POO*) hace modelos de los objetos del mundo real mediante sus contrapartes en *software*. Aprovecha las relaciones de clase, donde objetos de una cierta clase, como la clase de vehículos, tienen las mismas características. Aprovecha las relaciones de *herencia*, e inclusive las relaciones de *herencia múltiple*, donde clases recién creadas de objetos se derivan heredando características de clases existentes, pero poseyendo características únicas, propias de ellos mismos.

La programación orientada a objetos nos proporciona una forma más natural e intuitiva de observar el proceso de programación, es decir *haciendo modelos* de objetos del mundo real, de sus atributos y de sus comportamientos. *POO* también hace modelos de la comunicación entre los objetos. De la misma forma que las personas se envían *mensajes* uno al otro, los objetos también se comunican mediante mensajes.

POO encapsula datos (atributos) y funciones (comportamiento) en paquetes llamados *objetos*; los datos y las funciones de un objeto están muy unidos. Los objetos tienen la propiedad de *ocultar la información*. Esto significa que aunque los objetos puedan saber cómo comunicarse unos con otros mediante *interfaces* bien definidas, a los objetos por lo regular no se les está permitido saber cómo funcionan otros objetos los detalles de puesta en práctica quedan ocultos dentro de los objetos mismos.

2.6.2. Definición

Un programa tradicional se compone de procedimientos y de datos. Un programa orientado a objetos consiste solamente en objetos, entendiendo por objeto una entidad que tiene unos atributos particulares, los datos, y unas formas de operar sobre ellos, los métodos o procedimientos.

La programación orientada a objetos es una de las técnicas más modernas que trata de disminuir el costo del *software*, aumentando la eficiencia en la programación y reduciendo el tiempo necesario para el desarrollo de una aplicación. Con la programación orientada a objetos, los programas tienen menos líneas de código, menos sentencias de bifurcación, y módulos que son más comprensibles porque reflejan de una forma clara la relación existente entre cada concepto a desarrollar y cada objeto que interviene en dicho desarrollo. Donde la programación orientada a objetos toma verdadera ventaja es en la compartición y reutilización del código.

Sin embargo, no se debe pensar que la programación orientada a objetos resuelve todos los problemas de una forma sencilla y rápida. Para conseguir buenos resultados, es preciso dedicar un tiempo significativamente superior al análisis y al diseño. Pero, éste no es un tiempo perdido, ya que simplificará enormemente la realización de aplicaciones futuras.

Según lo expuesto, las ventajas de la programación orientada a objetos son sustanciales. No obstante, también presenta inconvenientes; por ejemplo, la ejecución de un programa no gana en velocidad y obliga al usuario a aprenderse una amplia biblioteca de clases antes de empezar a manipular un lenguaje orientado a objetos.

2.6.3. Clases y Objetos

Un objeto en la *POO* es cualquier entidad del mundo real que se pueda imaginar, como por ejemplo, los automóviles, los aviones, los animales mamíferos, etc. Además los objetos soportan una serie de características específicas de los mismos:

- ❖ Se agrupan en grupos denominados clases.
- ❖ Contienen datos internos que definen su estado actual.
- ❖ Soportan ocultamiento de datos.
- ❖ Pueden heredar propiedades de otros objetos.
- ❖ Pueden comunicarse con otros objetos enviando o pasando mensajes.
- ❖ Tienen métodos que definen su comportamiento.

La *POO* se basa en la observación de que, en el mundo real, los objetos se construyen a partir de otros objetos. La combinación de estos objetos es un aspecto de dicha programación, pero también incluye mecanismos y características que hacen que la creación y el uso de objetos sea sencillo y flexible. Un mecanismo importante es la *clase*, y el encapsulamiento y la herencia son dos propiedades poderosas.

Un *objeto* de una determinada clase se crea en el momento en que se define una variable de dicha *clase*. Por ejemplo, la siguiente línea declara el objeto *cuenta01* de la clase o tipo *CCuenta*.

```
CCuenta cuenta01 ; // crea el objeto cuenta01 de la clase CCuenta
```

Cuando se escribe un programa utilizando un lenguaje orientado a objetos, lo primero es definir las clases de objetos, donde una clase se ve como una plantilla para múltiples objetos con características similares. Afortunadamente no se tiene que escribir todas las clases que se necesiten en un programa, ya que la biblioteca estándar de C++ proporciona clases para realizar las operaciones más

habituales, como por ejemplo, leer datos, escribir resultados, operaciones con cadenas, trabajo con vectores, etc.

2.6.4. Mensajes y Métodos

Un programa orientado a objetos se compone solamente de objetos. Cada uno de ellos es una entidad que tiene propiedades particulares, los *atributos*, y unas formas de operar sobre ellos, los *métodos*. Por ejemplo, una ventana de una aplicación *Windows* es un objeto. El color de fondo, la anchura, la altura, etc. son atributos. Las rutinas, lógicamente transparentes al usuario, que permiten maximizar la ventana, minimizarla, son los métodos.

Cuando se ejecuta un programa orientado a objetos, los objetos están recibiendo, interpretando y respondiendo a *mensajes* de otros objetos. En la *POO* un *mensaje* está asociado con un *método*, de tal forma que cuando un objeto recibe un mensaje la respuesta a ese mensaje es ejecutar el método asociado. Por ejemplo, cuando un usuario quiere maximizar una ventana de aplicación *Windows*, lo que hace simplemente es pulsar el botón de la misma que realiza esa acción. Eso provoca que *Windows* envíe un mensaje a la ventana para indicar que tiene que maximizarse. Como respuesta a este mensaje se ejecutará el método programado para ese fin.

Un *método* se escribe en una *clase* de objetos y determina cómo tiene que actuar el objeto cuando recibe el *mensaje* vinculado con ese método. A su vez, un *método* puede también enviar *mensajes* a otros objetos solicitando una acción o información. En adición, los atributos definidos en la clase permitirán almacenar información para dicho objeto.

Desde el punto de vista de desarrollo, un *método* no es más que una unidad de código con entidad propia que en C++ generalmente denominados *función*, o si lo prefiere, *procedimientos*.

Según lo expuesto, se puede decir que la ejecución de un programa orientado a objetos realiza fundamentalmente tres tareas:

- ❖ Crea los objetos necesarios.
- ❖ Los mensajes enviados a unos y a otros dan lugar a que se procese internamente la información.
- ❖ Finalmente, cuando los objetos no son necesarios, son borrados.

2.6.5. Constructores

Un *constructor* es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de esa clase. Su función es iniciar el objeto. Cuando se crea un objeto, C++ hace lo siguiente:

- Asigna memoria para el objeto.
- Inicia los atributos de ese objeto con los valores predeterminados por el sistema.
- Llama al constructor de la clase que puede ser uno entre varios.

Un *constructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece y no puede retornar un valor (ni siquiera se puede especificar la palabra reservada **void**). Por ejemplo si se añadiera a la clase *CCuenta* un constructor, se tendría que llamar también *CCuenta*. Ahora bien, *cuando* en una clase no se escribe explícitamente un constructor, C++ asume uno por omisión. Por ejemplo, la clase *CCuenta* que se ha descrito anteriormente tiene por omisión un constructor definido así:

```
public : CCuenta ( ) { }
```

Un *constructor por omisión* de una clase C es un constructor sin parámetros que no hace nada. Sin embargo, es necesario porque, según lo que se acaba de exponer, será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema (en C++ las variables locales no son iniciadas por el sistema, por lo tanto inicialmente tendrán valores indeterminados, que genéricamente se denomina basura).

Los constructores, salvo en casos excepcionales, deben declararse siempre públicos para que puedan ser invocados desde cualquier parte.

Cuando en una clase un mismo método se define varias veces con distinto número de parámetros, o bien con el mismo número de parámetros pero diferenciándose una definición de otra en que al menos un parámetro es de un tipo diferente, se dice que el método está *sobrecargado*.

2.6.6. Destruidores

Un destructor es un método especial de una clase que se ejecuta antes de que un objeto de esa clase sea eliminado físicamente de la memoria. Un *destructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece precedido por una tilde ~. Un *destructor* no es heredado, no tiene argumentos, no

puede retornar un valor (incluyendo **void**) y no puede ser declarado **const** ni **static**, pero sí puede ser declarado **virtual**. Utilizando destructores virtuales se puede destruir objetos sin conocer su tipo.

Cuando en una clase no se especifica un destructor, el compilador añade uno por omisión, público. Por ejemplo, el destructor para la clase *CFecha* es declarado por el compilador C++ así:

```
~CFecha () { };
```

Para definir un destructor en una clase tiene que reescribir el método anterior. A diferencia de lo que ocurría con los constructores, en una clase sólo es posible definir un *destructor*. En el cuerpo del mismo puede escribir cualquier operación que quiera realizar relacionada con el objeto que se vaya a destruir.

2.6.7. Herencia

La herencia es una de las características más importantes en la *POO* porque permite que una clase herede los atributos y métodos de otra clase (los constructores no se heredan). Esta característica garantiza la reutilización del código.

Con la herencia todas las clases están clasificadas en una jerarquía estricta. Cada clase tiene su superclase (la clase superior en la jerarquía, también llamada clase base), y cada clase puede tener una o más subclases (las clases inferiores en la jerarquía; también llamadas clases derivadas).

Las clases que están en la parte inferior en la jerarquía se dice que *heredan* de las clases que están en la parte superior en la jerarquía. El término *heredar* significa que las subclases disponen de todos los métodos y propiedades de su superclase. Este mecanismo proporciona una forma rápida y cómoda de extender la funcionalidad de una clase. En C++ cada clase puede tener una superclase (o clase base), lo que se denomina *herencia simple*, o dos o más superclases, lo que se denomina *herencia múltiple*.

Una subclase puede redefinir cualquier método heredado de su clase padre, siempre que sea necesario que su comportamiento en la subclase sea diferente. Redefinir un método heredado significa volverlo a escribir en la subclase con el mismo nombre, la misma lista de parámetros y el mismo tipo del valor de retorno que tenía en la superclase; su cuerpo será adaptado a las necesidades de la subclase.

Para finalizar, el mecanismo de herencia proporciona una forma rápida y cómoda de modificar, en la dirección que se desee, la funcionalidad de una clase[64].

2.7. PROGRAMACION C++

2.7.1 Introducción

C++ es una mejora sobre muchas de las características de C, y proporciona capacidades de *P.O.O* [64](Programación Orientada a Objetos) que promete mucho para incrementar la productividad, calidad y reutilización del software.

Los diseñadores de C y los responsables de sus primeras puestas en práctica jamás anticiparon que este lenguaje resultaría en un fenómeno como éste. Cuando un lenguaje de programación se torna tan arraigado como C, nuevas necesidades demandan que el lenguaje evolucione, en lugar de que sólo sea reemplazado por un nuevo lenguaje.

C++ fue desarrollado por *Bjarne Stroustrup* en los Laboratorios Bell y originalmente fue llamado "C con clases"[65]. El nombre C++ incluye el operador de incremento (++) de C, para indicar que C++ es una versión mejorada de C.

En C, la unidad de programación es la **función**, con lo cual, se trata de una programación orientada a la acción. En cambio, en C++, la unidad es la **clase** a partir de la cual, los objetos son producidos. Se trata, pues, de una programación orientada al objeto.

Las bibliotecas estándar de C++ proporcionan un conjunto extenso de capacidades de entrada / salida. C++ usa entradas / salidas de tipo seguro; no podrán introducirse datos equivocados dentro del sistema.

Se pueden especificar entradas/salidas de tipos definidos por el usuario, así como de tipos estándar. Esta *extensibilidad* es una de las características más valiosas de este lenguaje de programación. C++ permite un tratamiento común de entradas/salidas de tipos definidos por usuario. Este tipo de estado común facilita el desarrollo de software en general y de la reutilización de software en particular.

En este capítulo se quiere dar a conocer los métodos más utilizados para programar en C++ y así conocer todos sus componentes y principios básicos. Además se pretende mostrar las grandes ventajas de la Programación orientada a Objetos[64], ya que esta será la base para la mayoría de los programas de nuestro proyecto.

2.7.1.1. Historia del Lenguaje C++

C++ [65] es un lenguaje de programación de propósito general basado en el lenguaje de programación C y ha sido diseñado para: ser mucho mejor que C,

soportar la abstracción de datos, soportar la programación orientada a objetos y soportar la programación genérica utilizando plantillas.

El lenguaje C nació en los laboratorios Bell de *AT&T* (*Dennis Ritchie*, 1972) y ha sido estrechamente asociado con el sistema operativo *UNIX*, ya que su desarrollo se realizó en este sistema y debido a que tanto *UNIX* como el propio compilador C y la casi totalidad de los programas y herramientas de *UNIX* fueron escritos en C. Su eficiencia y claridad han hecho que el lenguaje ensamblador apenas haya sido utilizado en *UNIX*.

Este lenguaje ha evolucionado paralelamente a *UNIX*. Muestra de esta evolución es que en 1980 se añaden al lenguaje C [66] características como *clases*, chequeo del tipo de los argumentos de una función y conversión, si es necesaria, de los mismos, así como otras características; el resultado fue el denominado *C con Clases*.

En 1983/84, *C con Clases* fue rediseñado, extendido y nuevamente implementado. El resultado se denominó *Lenguaje C++*. Las extensiones principales fueron *funciones virtuales*, *funciones sobrecargadas* (un mismo identificador da acceso a múltiples formas de una función), y *operadores sobrecargados* (un mismo operador puede utilizarse en distintos contextos y con distintos significados). Después de algún otro refinamiento más, C++ quedó disponible en 1985. Este lenguaje fue creado por *Bjarne Stroustrup* (*AT&T* Bell Laboratorios) y documentado en varios libros suyos.

Posteriormente, C++ fue ampliamente revisado y refinado, lo que dio lugar a añadir nuevas características, como herencia múltiple, funciones miembro ***static*** y ***const***, miembros ***protected***, tipos genéricos de datos o plantillas y manipulación de excepciones. Se revisaron características como sobrecarga, enlace y manejo de la memoria. Además de esto, también se hicieron pequeños cambios para incrementar la compatibilidad con C y se añadieron la identificación de tipos durante la ejecución y los espacios de nombres con el objetivo de convertir a C++ en un lenguaje más propicio para la escritura y utilización de bibliotecas.

Esta evolución requería una estandarización. Por eso, en 1989 se convocó el comité X3J16 de *ANSI* (*American National Standards Institute*), que más tarde, en 1991, entró a formar parte de la estandarización *ISO*. El trabajo conjunto de estos comités permitió publicar en 1998 el estándar *ISO C++*, que ha dado lugar a que este lenguaje sea estable, a que no dependa del compilador C++ utilizado, y a que el código se pueda portar entre diferentes plataformas.

2.7.2. Comentarios de una sola línea de C++

Con frecuencia los programadores insertan un pequeño comentario al final de una línea de código. C requiere que un comentario sea delimitado mediante */** y **/*. C++

le permite que empiece un comentario con `//` y que utilice el resto de la línea para texto del comentario; el fin de la línea da de manera automática por terminado el comentario. Esta forma de comentario ahorra algunos golpes de tecla, pero sólo es aplicable a comentarios que no continúen a la línea siguiente.

Por ejemplo, el comentario C requiere de ambos delimitadores `/*` y `*/`, aún para un comentario de una línea.

`/* Este es un comentario de una línea. */`

La versión en una línea de C++ de lo anterior es:

`// Este es un comentario de una línea.`

Dado que C++ es un súper conjunto de C, en un programa C++ ambas formas de comentario son aceptables[65].

2.7.3. Flujo de entrada/salida de C++

C++ ofrece una alternativa a las llamadas de función *printf* y *scanf* para manejar la entrada/salida de los tipos y cadenas de datos estándar. Por ejemplo, el diálogo simple

```
printf ( "Digite su clave: " );  
scanf ( "%d", $clave);  
printf ( "La nueva clave es: %d\n", clave);
```

se escribe en C++ como

```
cout << "Digite su nueva clave: ";  
cin >> clave  
cout << "La nueva clave es: " << clave << '\n';
```

El primer enunciado utiliza el *flujo estándar cout* y el operador `<<` (el *operador de inserción de flujo* que se pronuncia "colocar en"). El enunciado se lee

La cadena "Digite su nueva clave" es colocada en el flujo de salida cout.

Note que el operador de inserción de flujo es también el operador de desplazamiento a la izquierda a nivel de bits. El segundo enunciado utiliza el *flujo de entrada estándar cin* y el operador `>>` (el *operador de extracción de flujo*, que se pronuncia "obtener de"). El enunciado se lee

Obtener un valor para clave del flujo de entrada cin.

Note que el operador de extracción de flujo también es un operador de desplazamiento a la derecha a nivel de bits cuando el argumento izquierdo es un tipo entero. Los operadores de inserción y de extracción de flujo, a diferencia de *printf* y de *scanf*, no requieren de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos[65].

Para utilizar entradas/salidas de flujo, los programas C++ deben incluir el archivo de cabecera *iostream.h*.

2.7.4. Declaraciones en C++

En C++, las declaraciones pueden ser colocadas en cualquier parte de un enunciado ejecutable, siempre y cuando las declaraciones antecedan el uso de lo que se está declarando. Por ejemplo:

```
cout << "Digite dos enteros: ";  
int x, y;  
cin >> x >> y;  
cout << " La suma de " << x << " y " << y << " es " << x+y << '\n' ;
```

En este ejemplo se declaran las variables *x* y *y* después del enunciado ejecutable *cout*, pero antes que sean utilizadas en el enunciado subsecuente *cin*. También las variables pueden ser declaradas en la sección de inicialización de una estructura *for* cuyas variables se mantienen en alcance hasta el final del bloque en el cual la estructura *for* está definida. Por ejemplo:

```
for ( int i = 0 ; i <= 5 ; i ++)  
    cout << i << '\n' ;
```

2.7.5. Creación de nuevos tipos de datos en C++

C++ proporciona la capacidad de crear tipos definidos por el usuario mediante el uso de la palabra reservada *enum*, la palabra reservada *struct*, la palabra reservada *union* y la nueva palabra reservada *class*. Al igual que en C, las enumeraciones C++ son declaradas mediante *enum*. Sin embargo; a diferencia de C, una enumeración en C++, cuando se declara, se convierte en un tipo nuevo. Para declarar la variable del nuevo tipo la palabra reservada *enum* no es requerida. Lo mismo es cierto en el caso de *struct*, *union* y *class*. Por ejemplo, las declaraciones

```
enum Boolean {FALSE, TRUE} ;  
  
struct Name {  
    char first [10] ;  
    char last [10] ;
```

```
};

union Number {
    int i;
    float f;
};
```

crean tres tipos de datos, definidos por usuario, con los nombres de etiqueta *Boolean*, *Name* y *Number*. Los nombres de etiqueta pueden ser utilizados para declarar variables, como sigue:

```
Boolean done = FALSE;
Name student;
Number x;
```

Estas declaraciones crean la variable *Boolean done* (inicializada a *FALSE*), la variable *Name student* y la variable *Number x*.

2.7.6. Prototipos de función y verificación de tipo

El prototipo de función le permite al compilador de C verificar por tipo la exactitud de las llamadas de función. En *ANSI C*, los prototipos de función son opcionales. En C++ los prototipos de función son requeridas para todas las funciones. Una función definida en un archivo, antes de cualquier llamada a la misma, no requiere de un prototipo de función por separado. En este caso, el encabezado de función actúa como prototipo de función. C++ también requiere que se declaren todos los parámetros de función en los paréntesis de la definición de función y del prototipo. Por ejemplo, una función *square*, que toma un entero de argumento y regresa un entero tiene el prototipo:

```
Int square (int) ;
```

Las funciones que no regresan un valor se declaran con el tipo de regreso *void*. En C++ una lista vacía de parámetros se especifica escribiendo *void* o absolutamente nada en los paréntesis. La siguiente declaración especifica que la función *print* no toma argumentos y no regresa un valor.

```
Void print ( ) ;
```

2.7.7. Funciones en línea

Desde el punto de vista de la ingeniería del software es una buena idea poner en práctica un programa como un conjunto de funciones, pero las llamadas de función involucran sobrecarga en tiempo de ejecución. C++ tiene *funciones en línea* que ayudan a reducir la sobrecarga por llamadas de función especial para

pequeñas funciones. El calificador *inline* colocado en la definición de función antes del tipo de regreso de una función “aconseja” al compilador que genere una copia del código de la función “*in situ*” (cuando sea apropiado), a fin de evitar una llamada de función. La contrapartida es que en el programa se insertan muchas copias de código de la función, en vez de, cada vez se llama a la función, tener una copia de la función a la cual pasarle el control. El compilador puede ignorar el calificador *inline* y típicamente así lo hará para todas, a excepción de las funciones más pequeñas.

Cualquier modificación a una función *inline* requiere que sean recompilados todos los clientes de dicha función. Esto pudiera resultar de importancia en algunas situaciones de desarrollo y mantenimiento de programas.

Las funciones en línea ofrecen ventajas en comparación con las macros de preprocesador que expanden código en línea. Una ventaja es que las funciones *inline* son iguales a cualquier otra función de C++. Por lo tanto, en llamadas a las funciones *inline* se ejecutará una adecuada verificación de tipo; las macros de preprocesador no aceptan verificación de tipo. Otra ventaja es que las funciones *inline* eliminan efectos colaterales inesperados, asociados con un uso inapropiado de las macros de preprocesador. Por último, las funciones *inline* pueden ser depuradas mediante un programa depurador. El depurador no reconoce a las macros de preprocesador como unidades especiales, porque sólo son sustituciones de texto, efectuadas por el preprocesador antes de la compilación del programa. Un depurador puede ayudar a localizar errores lógicos resultado de sustituciones de macros, pero no puede atribuir dichos errores a macros específicas[65].

2.7.8. Parámetros por referencia

En C, todas las llamadas de función son llamadas por valor. En C las llamadas por referencia son simuladas pasando un apuntador a un objeto y obteniendo a continuación acceso al objeto desreferenciando el apuntador en la función llamada. Otros lenguajes de programación ofrecen formas directas de llamada por referencia como los parámetros *var* en pascal. ++ corrige esta debilidad de C al ofrecer *parámetros por referencia*.

Un parámetro por referencia es un seudónimo de su argumento correspondiente. Para indicar que un parámetro de función es pasado por referencia, sólo se coloca *ampersand* (&) después del tipo del parámetro en el prototipo de función (exactamente de la misma forma en que pondría un * después del tipo parámetro, para indicar que un parámetro es el apuntador a una variable). Se utiliza la misma regla convencional en el encabezado de la función al enlistar el tipo de parámetros. Por ejemplo, la declaración

Int &count

en un encabezado de función puede ser leído “*count* es una referencia a *int*”. En la llamada de función, sólo se menciona la variable por su nombre y automáticamente será pasada por referencia. Entonces, el mencionar en el cuerpo de la función la variable por su nombre local, de hecho la refiere a la variable original en la función llamadora, y la variable original puede ser modificada de manera directa por la función llamada[65].

2.7.9. Asignación dinámica de memoria mediante *new* y *delete*

Los operadores *new* y *delete* de C++ le permiten a los programas llevar a cabo la asignación dinámica de memoria. En *ANSI C*, la asignación dinámica de memoria por lo general se lleva a cabo con las funciones estándar de biblioteca *malloc* y *free*.

En C++, el enunciado

```
ptr = new typeName
```

asigna memoria para un objeto del tipo *typeName* partiendo de la *tienda libre* del programa (término utilizado por C++ para la memoria adicional disponible para el programa en tiempo de ejecución). El operador *new* crea automáticamente un objeto del tamaño apropiado, y regresa un apuntador del tipo apropiado. Si mediante *new* no se puede asignar memoria, se regresa un apuntador nulo (para representar un apuntador nulo los programadores C++ utilizan el valor 0, en vez de *NULL*).

Para liberar en C++ el espacio para este objeto se utiliza el siguiente enunciado:

```
delete ptr ;
```

En C, se invoca la función *free* con el argumento *ptr*, a fin de desasignar memoria. El operador *delete* sólo puede ser utilizado para desasignar memoria ya asignada mediante el operador *new*. Aplicar *delete* a un apuntador previamente desasignado, puede llevar a errores inesperados durante la ejecución del programa. Aplicar *delete* a un apuntador nulo no tiene efecto en la ejecución del programa.

2.8. PROGRAMACIÓN OTCL

2.8.1. Introducción a *Tcl*

Tcl (*Tool Command Language*) es un lenguaje de programación de comandos muy popular para el desarrollo de pequeñas aplicaciones en entornos *UNIX* (aunque también existe una versión disponible para *Windows*)[67]. Permite programar de forma rápida y sencilla aplicaciones no demasiado complejas. Sin embargo, la velocidad de ejecución de éstas no será muy elevada debido a que este es un lenguaje interpretado y no un lenguaje compilado.

Una característica muy importante de este lenguaje es la facilidad con la que se pueden añadir nuevos comandos a los ya existentes en el *Tcl* estándar. Estos nuevos comandos pueden implementarse utilizando el lenguaje de programación C e integrarse de manera sencilla en *Tcl*. Así, se han escrito bastantes extensiones para la realización de ciertas tareas comunes como, por ejemplo, *Otcl*[68] (*Tcl* orientado a objetos) o *Tk* (que permite crear interfaces gráficas de usuarios).

En esta parte del documento se pretende mostrar de una forma muy fácil de entender los principios básicos que se deben tener en cuenta para programar en *Tcl*, permitiendo así la implementación de muchas herramientas útiles en este proyecto.

2.8.2. Características de *Tcl*

Este lenguaje fue creado por John Ousterhout en 1998 [68]. *Tcl* se ha caracterizado por ser un lenguaje muy simple y genérico, se puede trabajar fácilmente en él para diversas aplicaciones.

Se pueden ejecutar comandos *Tcl*[69] de dos modos:

1. Modo interactivo: directamente, a través del intérprete de comandos *tclsh*, se pueden introducir y ejecutar comandos *Tcl* de forma interactiva.
2. Modo no interactivo: los comandos se guardan en un fichero. Se puede ejecutar de dos maneras:
 - a) Llamando al intérprete de comandos de *Tcl* pasándole como parámetro el nombre del fichero.
 - b) Ejecutando directamente el fichero de comandos. Para que esto sea posible, la primera línea del fichero debe incluir el *path* hacia el intérprete de comandos (por ejemplo, *#!/usr/bin/tclsh*). Además, el fichero tiene que tener los permisos adecuados (permiso de ejecución).

Hay un grupo de variables especiales, que son las que guardan los argumentos con que es invocado un *script*:

- *argc*: Contiene el número de argumentos (0 si no hay). No guarda el nombre del *script*.
- *argv*: Contiene una lista con los argumentos, el primero está en la posición 0.
- *argv0*: Contiene el nombre del *script* que fue invocado con *tclsh*.

2.8.3. Variables y valores

Tcl trata principalmente con cadenas de texto (*strings*)[68]. Cuando es necesario (por ejemplo, a la hora de realizar operaciones aritméticas), *Tcl* convierte automáticamente los *strings* en números.

Para crear una variable y asignarle un valor se utiliza el comando *set*:

```
% set a 100
```

Cuando el símbolo \$ precede al nombre de una variable, el intérprete sustituye dicha variable por su valor. Por ejemplo:

```
% puts $a
100
```

Para realizar operaciones aritméticas se utiliza el comando *expr*:

```
% exp. 2*$a
200
```

Una cadena contenida entre corchetes ([cadena]) se considera que es un comando: la cadena se evalúa como si fuese un comando *Tcl* y se sustituye por el resultado obtenido. Esto no ocurre si la cadena está entre comillas ("cadena"). Para ejecutar realmente dicha cadena se requiere el comando *eval*:

```
% set b [expr 2*$a]
200
% set b "expr 2*$a"
expr 2*100
% eval $b
200
```

2.8.4. Estructuras de Control

Los comandos de control de flujo son similares a sus equivalentes en el lenguaje C[68]. Los principales operadores de comparación son: <, >, <=, >=, ==, !=, &&, ||, !. Los ejemplos siguientes pueden ser útiles para ilustrar la utilización de estos comandos:

- **if**

```
% set i 1
% if {$i < 0} {
puts "Negativo"
} elseif {$i == 0} {
puts "Cero"
} else {
puts "Positivo"
}
Positivo
```

- **for**

```
% for {set i 1} {$i <= 3} {incr i} {
puts $i
}
1
2
3
```

El comando *break* finaliza la ejecución del bucle inmediatamente. El comando *continue* fuerza a que se ejecute la iteración siguiente (evidentemente, siempre y cuando se cumpla la condición de continuación del bucle).

- **while**

```
% set i 1
% while {$i <= 3} {
puts $i
incr i
}
1
2
3
```

2.8.5. Operaciones con cadenas de texto

- ***string length***

Devuelve el número de caracteres que forman una cadena.

```
% string length "ABCD"  
4
```

- ***string index***

Devuelve el carácter situado en una determinada posición de una cadena. Al primer carácter le corresponde el índice 0. Podemos referirnos al último carácter de una cadena mediante la palabra *end*.

```
% string index "ABCD" 1  
B  
% string index "ABCD" end  
D
```

- ***string range***

Devuelve la subcadena formada por los caracteres que se encuentran entre dos posiciones de una cadena.

```
% string range "ABCDEF" 3 end  
DEF
```

- ***string first, last***

Busca la primera (última) ocurrencia de una subcadena en una cadena y devuelve el índice a partir del cual se encuentra dicha subcadena (si no se encuentra, devuelve -1).

```
% string first "CD" "ABCDABCD"  
2  
% string last "CD" "ABCDABCD"  
6
```

- ***string map***

Reemplaza en una cadena las subcadenas indicadas por otras nuevas.

```
% string map {AB 12 C 3 D 4} "ABCD"  
1234
```

- **string match**

Devuelve 1 si la cadena coincide con el patrón proporcionado (o 0 en caso contrario). Para obtener completa información sobre la especificación de patrones se puede consultar la ayuda de la herramienta *flex*.

```
% string match "*B*" "ABC"
1
```

- **regexp**

Permite buscar patrones en una cadena y asignarlos a variables.

```
% set cadena "El resultado es 100."
% regexp {[0 - 9]+} $cadena resultado
% puts $resultado
100
```

2.8.6. Arrays

Los arrays en *Tcl* [68] son muy asociativos: cualquier cadena puede ser índice de un array.

```
% set miarray (0) 0
% set miarray (elemento) 1
% puts $miarray (0)
0
% puts $miarray (elemento)
1
```

Los comandos más importantes para manejar arrays son: *array exists*, *array get*, *array names*, *array size*...

2.8.7. Listas

Son agrupaciones de elementos de *Tcl*. Cualquier elemento *Tcl* puede formar parte de una lista, incluido otras listas. La sintaxis para crear una lista es la siguiente:

```
% set meses {enero febrero marzo }
```

- **llenght**

Devuelve el número de elementos que forman la lista.

```
% llenght $meses  
3
```

- **lindex**

Devuelve el elemento que ocupa una posición determinada en una lista. Al elemento inicial de la lista le corresponde el índice 0.

```
% lindex $meses 1  
febrero
```

- **linsert**

Permite añadir elementos a una lista a partir de una posición determinada.

```
% linsert $meses end abril mayo  
enero febrero marzo abril mayo  
% puts $meses  
enero febrero marzo
```

Atención: este comando no modifica la lista *meses*, sino que devuelve una lista con los nuevos elementos añadidos.

- **lappend**

Añade elementos a una lista a partir del final de la misma. Este comando sí modifica realmente la lista.

```
% lappend meses abril mayo  
enero febrero marzo abril mayo  
% puts $meses  
enero febrero marzo abril mayo
```

- **lsearch**

Busca elementos de la lista que verifiquen un determinado patrón. Devuelve el índice del primer elemento que cumple con el patrón o -1 si no se encuentra ningún elemento.

```
% lsearch $meses feb*  
1
```

- **lreplace**

Devuelve la lista resultado de reemplazar en una lista los elementos seleccionados por otros nuevos.

```
% lreplace $meses 0 1 ENERO FEBRERO  
ENERO FEBRERO marzo abril mayo
```

Si no se especifican los nuevos elementos, simplemente se eliminan los elementos seleccionados.

```
% lreplace $meses 0 1
marzo abril mayo
```

- **split**

Crea una lista a partir de una cadena. El delimitador por defecto es el espacio en blanco.

```
% split "uno dos tres"
uno dos tres
% split "uno, dos, tres" { , }
uno dos tres
```

- **foreach**

Este comando asigna a una variable un elemento de una lista en cada paso.

```
% foreach i {1 2 3} {
    puts $i
}
1
2
3
```

2.8.8. Procedimientos

Un procedimiento se define de la siguiente manera[68]:

```
proc nombre_proc { arg1 arg2 ...} {
...
return $var
}
```

Por defecto, todas las variables creadas dentro de un procedimiento son locales a dicho procedimiento. Para acceder a variables globales:

- **global variable:** permite acceder dentro de un procedimiento a una variable global.
- **upvar olivar newvar:** hace que la variable global *olivar* sea accesible en el procedimiento actual a través de una variable de nombre *newvar*.

2.8.9. Comandos de entrada / salida

- **Open**

Abre un fichero y devuelve un identificador de canal que se puede utilizar en futuras llamadas a otros comandos de *E/S*.

```
% set fichero [ open "fichero . txt" r ]
```

Se permiten los siguientes modos de acceso:

- **r**: sólo lectura (el fichero debe existir).
- **r+**: lectura y escritura (el fichero debe existir).
- **w**: sólo escritura (si existe, lo sobrescribe).
- **w+**: lectura y escritura (si existe, lo sobrescribe).
- **a**: sólo escritura (el fichero debe existir, añade datos al final del fichero).
- **a+**: lectura y escritura (el fichero debe existir, añade datos al final del fichero).

Si el primer carácter del nombre del fichero es el símbolo `|`, el *open* funciona como una tubería: permite ejecutar programas externos y que su salida esté disponible a través del identificador devuelto:

```
% set listado [ open "| ls" r ]
```

- **close**

Este comando cierra el canal establecido para manejar un fichero.

```
% close $fichero
```

- **gets**

Permite leer líneas de un fichero. Devuelve el número de caracteres leídos o -1 si se produce un error.

```
% ges $fichero linea
```

- **puts**

Permite escribir líneas en un fichero. Con la opción *nonewline* se evita escribir el salto de línea.

```
% puts $fichero "1 2 3"
```

- **file**

Permite comprobar el estado de un fichero. Se puede ejecutar con varias opciones: *exists*, *executable*, *extensión*, *type*, *size*.

% file exists "fichero .txt"

2.8.10. Otros comandos

❖ Comentarios

Toda línea que empieza con el carácter # se considerará un comentario ignorándose su contenido.

❖ Parámetros pasados a la aplicación

Los parámetros que se le pasan a una aplicación *Tcl* a través de la línea de comandos se guardan en una lista denominada *argv*. Por tanto, para obtener el primer parámetro:

% lindex \$argv 0

❖ Ejecución de programas externos

Se pueden ejecutar programas externos a la aplicación *Tcl* de dos maneras:

- Mediante el comando *open*.
- Mediante el comando *exec*: *exec comando*.

❖ Generación de números aleatorios

El comando *srand* permite inicializar la semilla del generador de números aleatorios. El comando *rand* genera un número aleatorio entre 0 y 1. Por ejemplo:

```
% expr srand (1)
7.82636925943e-06
% expr rand ( )
0.131537788143
```

3. DESARROLLO DE LA TESIS

3.1. METODOLOGIA DE LA TESIS

Para desarrollar el presente proyecto fue necesario seguir una serie de pasos los cuales se muestran como una breve introducción a continuación y más adelante se encuentra una profundización en cada uno de estos campos. El resumen de estos pasos se puede encontrar en la Figura 25.

a) Investigación y documentación: En esta fase, se realizó una investigación acerca de los temas de redes de datos más importantes que eran esenciales para que se pudieran cumplir todos los objetivos a cabalidad, tales como: *IPv4*, *IPv6*, conmutación de paquetes, conmutación de circuitos, Calidad de Servicio en Redes IP, *IntServ*, *IntServ6*, Protocolos de señalización.

b) Capacitación sobre lenguajes del Simulador: En esta parte del proyecto se estudiaron los diferentes lenguajes de programación y su forma de aplicación en el simulador de redes utilizado (*Network Simulator*, *NS-2.31*). Estos lenguajes de programación utilizados se dividen en dos: lenguaje intérprete que corresponde a *Perl* y el lenguaje compilador que corresponde a *C++*.

c) Capacitación en herramientas de simulación: Conocimiento y familiarización con las herramientas de trabajo como lo es el sistema operativo (*LINUX*), simulador de redes (*NS-2.31*). Dentro de la familiarización con el simulador, se conocieron las rutas de los diferentes archivos que se debían modificar para cumplir con los objetivos.

d) Diseño y construcción de los modelos de simulación: Se realizaron detalladamente los diseños de los scripts para los modelos de simulación (Redes IP) teniendo en cuenta el funcionamiento de las soluciones de *QoS* (*IntServ* e *IntServ6*). Estos diseños se hicieron por módulos (Clasificador, Planificador, Control de Admisión), para después finalmente integrarlos en uno solo mediante el *script* de *tcl*.

e) Realización de Pruebas: En esta etapa se realizan diferentes tipos de pruebas con diferentes parámetros de simulación para analizar después el comportamiento del diseño y concluir acerca de los cambios que suceden al variar ciertos detalles de la simulación. Además se utilizaron herramientas de *debugger* y compiladores de *C++* (*Dev-C++*) para observar el comportamiento paso a paso de los diferentes scripts y analizar si realmente los programas cumplían con los objetivos planteados.

f) Análisis de resultados: Una vez el simulador arrojará los resultados de los tiempos de las simulaciones y se guardaran en un archivo, se pudo proceder a

analizarlos. Para analizarlos de una mejor forma se realizaron las curvas de los mismos resultados para tener una mayor claridad a la hora de comparar los dos casos específicos.

g) Documentación: Finalmente se documentaron los análisis de los resultados, las conclusiones pertinentes y de igual forma se redactó un manual de usuario para el mayor entendimiento de la herramienta. Cabe resaltar que durante este procedimiento se tomaron en cuenta las anotaciones y observaciones que se hicieron durante el proceso de pruebas y de integración de cada modulo por aparte.

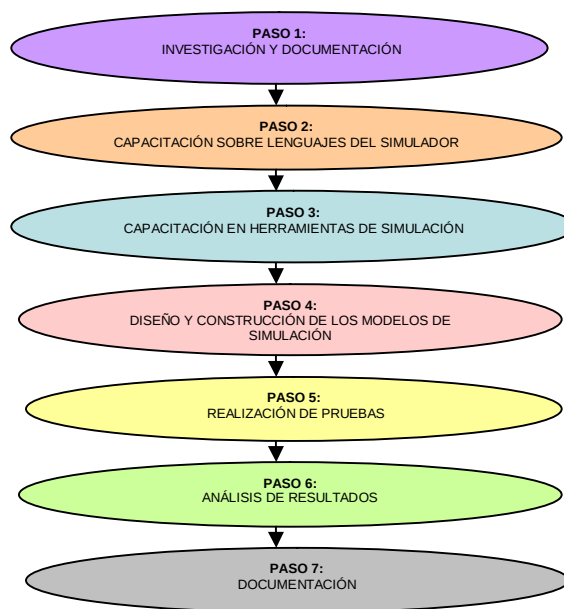


Figura 25. Pasos de la Metodología del Proyecto.

3.2. INVESTIGACIÓN Y DOCUMENTACIÓN

En esta etapa inicial se realizó una investigación global acerca de todos los temas necesarios para entender la estructura interna del proyecto, por ejemplo: Cuál es el funcionamiento interno de una red básica, porqué es importante soportar Calidad de Servicio en Internet, cual es la función de un *router* en una red, cual es la diferencia entre conmutación de circuitos y la conmutación de paquetes, entre otros temas que se debían abarcar para dar respuestas a ciertas dudas existentes del proyecto. Los temas que se tomaron en cuenta en este punto fueron: Protocolo de Internet versión 4 (*IPv4*), Protocolo de Internet versión 6 (*IPv6*), Conmutación de Circuitos, Conmutación de Paquetes, Calidad de Servicio en Redes IP (*QoS*), Servicios Integrados (*IntServ*), Protocolos de Señalización (*RSVP*) y la nueva propuesta *IntServ6*.

Este proceso de Investigación y Documentación se llevó a cabo por medio de lecturas de libros, archivos de Internet, organizando la información de una manera fácil de encontrar y muy sencilla de clasificar en el momento de adicionar las referencias bibliográficas al documento. Algunos de los temas tratados en esta etapa se encuentran organizados de la siguiente forma: En la sección 2.1 de este libro se encuentra toda la información referente a Calidad de Servicio en Internet (QoS), en la sección 2.2 esta la Arquitectura de Servicios Integrados (*IntServ*) y el protocolo de señalización *RSVP* y en la sección 2.3 se encuentra el protocolo de Internet versión 6 (IPv6) que retoma un poco las características de IPv4.

De esta forma es más fácil entender el proceso que hay que seguir para crear un nuevo protocolo en el simulador de redes (*NS-2*), investigando primero acerca de este protocolo y organizando esta información de tal forma que sea más sencilla de entender.

3.3. CAPACITACIÓN SOBRE LENGUAJES DEL SIMULADOR

El siguiente paso después de investigar acerca del protocolo y organizar la información obtenida, es la capacitación acerca de los tipos de lenguajes que utiliza el simulador de redes. En este caso el *Network Simulator*, que es el simulador con el cual trabajamos en este proyecto, se apoya en dos lenguajes de programación para su correcto funcionamiento: El lenguaje intérprete, el cual es *OTcl* (*Tcl* con extensiones de objetos), y el lenguaje compilador que corresponde a C++ (*POO*, Programación Orientada a Objetos). Estos dos lenguajes se encuentran estrechamente relacionados entre sí, ya que cada objeto presente en la jerarquía compilada, tiene su objeto equivalente en la jerarquía intérprete.

La programación en C++ o Programación Orientada a Objetos (*POO*), es la que se utiliza en la estructura interna del simulador. Mediante este lenguaje, se manejan los detalles de los protocolos de la simulación y es ideal para esta tarea por su eficiencia en el manejo de bytes y grandes conjuntos de datos siendo además un lenguaje de programación rápido para las ejecuciones.

Por otro lado, el lenguaje *OTcl* es utilizado para realizar las siguientes tareas: Para crear las topologías con sus respectivos componentes de red, darle los parámetros iniciales a los diferentes objetos que se incluyen en la simulación (como especificar el tipo de tráfico, retardos de enlace, clase de planificador de colas) y además para darle un tiempo específico a cada fase de la simulación, especificando además el tiempo de inicio y el tiempo de finalización de la simulación.

Con el simulador *NS-2* se tienen dos opciones de trabajo: Mediante el lenguaje *Otc* se pueden manejar objetos de C++ que ya existen dentro del simulador (clasificadores, cabeceras de paquetes, planificadores, enlaces, entre otros) y con estos crear un nuevo *script* de *tcl* y manipularlos como el usuario lo desee. Por

otro lado, se encuentra la opción de crear un nuevo protocolo con un comportamiento diferente al de los módulos existentes, lo cual requiere de programación de líneas en C++ y re-compilación del simulador para que se tomen los nuevos cambios. En este caso se debe tener en cuenta la adición de estos nuevos objetos dentro del fichero *makefile.in* del simulador, para que también sean compilados y en caso de tener errores sean notificados.

El objetivo de este proceso de capacitación era entender los conceptos básicos de estos dos lenguajes para más adelante crear nuestros propios programas del nuevo protocolo. Como resultado de esta capacitación se pudo concluir que necesitábamos modificar los programas de C++ del simulador, como se dijo anteriormente, creando así nuestros propios programas, para luego recompilar el *NS-2*.

Además, mediante este proceso también se obtuvo la idea de cómo, mediante el lenguaje *OTcl*, podíamos configurar nuestra propia topología especificando protocolos, estructuras físicas, tráfico y orden de los eventos dependiendo de los requerimientos de los objetivos planteados al inicio del proyecto. Para un mayor entendimiento de estos lenguajes se pueden leer los capítulos 2.5, 2.6 y 2.7 de este documento.

3.4. CAPACITACIÓN EN HERRAMIENTAS DE SIMULACIÓN

Del buen conocimiento de la herramienta depende si se cumplen los objetivos o no. Es por esta razón que dedicamos un tiempo a la capacitación en las herramientas necesarias para la simulación, es decir, el sistema operativo bajo el cual trabaja el simulador (*Linux*), la ubicación de archivos que necesitan modificaciones en el *NS-2*, métodos para compilar de forma adecuada los nuevos programas y además como se enlazan los dos lenguajes (C++ y *OTcl*) dentro de la simulación.

El simulador de redes *NS-2* en sus inicios solo podía ejecutarse en una plataforma *Linux*, sin embargo, actualmente también existe una versión que puede ser ejecutada sobre *Windows 9x/Me*. En nuestro proyecto se decidió instalar el simulador bajo una distribución de *Linux* llamada *Fedora Core 8*. Este sistema operativo es muy sencillo de manejar y la distribución de archivos y carpetas es muy similar a la de *Windows*.

Uno de los primeros pasos después de instalar el *Fedora Core 8*, fue la familiarización con el entorno, el conocimiento de sus herramientas y el manejo de archivos el cual era bastante sencillo. Al finalizar este proceso, se prosiguió a la instalación del simulador de redes *NS-2*. Al terminar esta instalación también se le dedicó el tiempo suficiente para conocer el simulador, entender su funcionamiento con la ayuda de tutoriales, y además encontrar los archivos que necesitaban modificarse para cumplir los requerimientos pedidos en los objetivos planteados.

Como una ayuda en el manejo e instalación del simulador y el sistema operativo *Fedora Core 8*, se creó un Manual de Usuario, el cual viene adjunto a este documento.

3.5. DISEÑO Y CONSTRUCCIÓN DE LOS MODELOS DE SIMULACIÓN

Para el diseño y construcción de los modelos de simulación es necesario analizar la herramienta y, tomando en cuenta que este trabaja con dos lenguajes diferentes de programación, el desarrollo del protocolo se divide en dos: Desarrollo del Protocolo en C++ y Desarrollo del Protocolo en *OTcl*.

En la primera parte del desarrollo del protocolo en C++, se crea un módulo clasificador independiente del simulador *NS-2*, con el objetivo de simular y analizar el comportamiento ideal de los routers *IntServ* e *IntServ6*. Posteriormente se realizan las modificaciones a los archivos del *NS-2* y se crea la estructura interna del router en general, teniendo en cuenta el módulo de clasificación anteriormente creado y los parámetros necesarios para realizar el acople con el simulador. Este módulo en C++ es creado en un archivo de texto con extensión *.cc* o *.cpp*, para luego ser compilado en el compilador de *Linux*. Los pasos que se necesitan para la creación del archivo ejecutable del código anteriormente creado son los siguientes:

- ✓ Se abre un terminal en *Fedora Core 8* y se accede a la carpeta en donde se encuentra el archivo *.cpp*.
- ✓ Se digita el siguiente comando:

g++ -o nombre_ejecutable nombre_archivo.cpp

- ✓ Al escribir el anterior comando se compila el código y se crea el ejecutable, de forma que al correr el programa se abren los ficheros con los resultados almacenados durante la ejecución del mismo.

Por otro lado, en el desarrollo del Protocolo en *OTcl* se realiza la estructura externa de la simulación, es decir, la topología, los routers, los enlaces con sus respectivos parámetros, los tiempos de inicio y finalización de la simulación, los generadores de tráfico, los anchos de banda, retardos para cada enlace de la red, el acople de los clasificadores creados con los routers del *NS-2*, entre otros parámetros mencionados anteriormente.

Como se había mencionado anteriormente, estos dos lenguajes con los cuales trabaja el simulador, necesitan acoplarse para poder ejecutar cualquier simulación, debido a que los dos tienen objetivos diferentes y se complementan entre sí.

Como el objetivo del proyecto requiere crear nuevos módulos en C++, se realizaron varias pruebas de modificaciones sencillas a algunos archivos en C++

para luego recompilar estos archivos. Para realizar estas modificaciones es necesario seguir ciertos pasos los cuales se explican a continuación:

- ❖ Encontrar el respectivo archivo a modificar dentro de la carpeta del simulador. Se debe tener en cuenta que estos son archivos en lenguaje C++, y que pertenecen a la parte interna del simulador. En caso de querer modificar la parte externa del simulador, es decir, cambiar la topología o los parámetros de simulación, las modificaciones se harían en los *scripts* de *tcl*. En caso de modificar la estructura interna del NS-2, los archivos se encuentran en */usuario/ns-allinone-2.31/ns-2.31/* ó en la ruta de acceso donde se halla instalado el simulador. Al modificar estos archivos es necesario crear nuevas carpetas con los nuevos cambios, para así no cambiar la estructura del NS-2 como tal, sino agregar módulos.
- ❖ Si se agregan nuevos ficheros al NS-2, es indispensable incluirlos en el fichero de *makefile.in*, para poder incluir esos nuevos programas en la compilación general del simulador. Esta carpeta se encuentra en el fichero de *ns-allinone-2.31/ns-2.31/*. Más adelante en el diseño del proceso de cada clasificador, se explica de una forma mas detallada como se ingresan estas nuevas líneas al archivo *makefile.in*.
- ❖ Después de crear los nuevos archivos e incluirlos en la carpeta de *makefile.in*, se deben compilar todos los programas del simulador para así comprobar si las nuevas modificaciones no tienen ningún error. Para realizar esta compilación correctamente, se deben digitar los siguientes comandos en un terminal:
 - El primer paso antes de digitar los comandos es acceder a la ruta donde se encuentra instalado el simulador, tal como se muestra a continuación:
cd ns-allinone-2.31/ns-2.31
 - *./configure*: este comando nos sirve para reconfigurar los ficheros recién incluidos en el *makefile.in*.
 - *make depend*: Este comando añade los nuevos cambios hechos en los archivos del simulador, para luego realizar el comando *make*.
 - *make*: este comando compila todos los ficheros existentes dentro del simulador, y muestra si hay errores de sintaxis en alguno de ellos, mostrando el fichero y la línea, para una mayor facilidad al corregir los errores.

En el siguiente ejemplo se detallan más profundamente los pasos que se deben seguir cuando se realiza un cambio en un archivo del simulador:

Inicialmente accedemos a la carpeta donde se encuentra instalado el simulador y se abre el archivo *makefile.in*. Estando dentro del archivo se busca la lista de objetos y se ingresa la línea del nuevo objeto con extensión *.o*, como se muestra en la Figura 26.

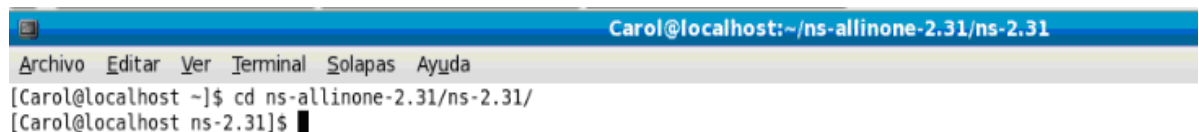
```

156 OBJ_CC = \
157     tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
158     common/scheduler.o common/object.o common/packet.o \
159     common/ip.o routing/route.o common/connector.o common/ttl.o \
160     trace/trace.o trace/trace-ip.o \
161     classifier/classifier.o classifier/classifier-addr.o \
162     classifier/classifier-hash.o \
163     classifier/classifier-IntServ.o \
164     classifier/classifier-virtual.o \
165     classifier/classifier-mcast.o \
166     classifier/classifier-bst.o \
167     classifier/classifier-mpath.o mcast/replicator.o \
168     classifier/classifier-mac.o \
169     classifier/classifier-qs.o \
170     classifier/classifier-port.o src_rtg/classifier-sr.o \
171     src_rtg/sragent.o src_rtg/hdr_src.o adc/ump.o \
172     qs/qsagent.o qs/hdr_qs.o \
173     apps/app.o apps/telnet.o tcp/tcplib-telnet.o \
174     tools/trafgen.o trace/traffictrace.o tools/pareto.o \
175     tools/expoo.o tools/cbr_traffic.o \
176     adc/tbf.o adc/resv.o adc/sa.o tcp/saack.o \
177     tools/measurmod.o adc/estimator.o adc/adc.o adc/ms-adc.o \
178     adc/timewindow-est.o adc/acto-adc.o \
179     adc/pointsample-est.o adc/salink.o adc/actp-adc.o \
180     adc/hb-adc.o adc/expavg-est.o \
181     adc/param-adc.o adc/null-estimator.o \
182     adc/adaptive-receiver.o apps/vatrcvr.o adc/consrcvr.o \
183     common/agent.o common/message.o apps/udp.o \

```

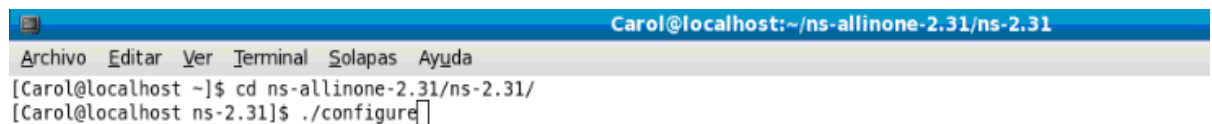
Figura 26. Ingreso de nuevos objetos al archivo *makefile.in*.

El siguiente paso es abrir un nuevo terminal y ubicarse en la carpeta donde se encuentra instalado el simulador, para luego digitar los comandos: *./configure*, *make depend* y *make*. En las figuras 27, 28, 29, 30, 31, 32 y 33 se encuentra paso a paso el procedimiento detallado de cómo responde el terminal ante los comandos ingresados.



The screenshot shows a terminal window with the title bar "Carol@localhost:~/ns-allinone-2.31/ns-2.31". The menu bar includes "Archivo", "Editar", "Ver", "Terminal", "Solapas", and "Ayuda". The command prompt shows the user has navigated to the directory "ns-2.31" using the "cd" command.

Figura 27. Ubicación en la carpeta donde se encuentra instalado el simulador NS-2



The screenshot shows the same terminal window as Figure 27, but now the command prompt shows the user has entered the command "configure" to start the configuration process.

Figura 28. Ingreso del comando *./configure* en el terminal.

```

Archivo  Editar  Ver  Terminal  Solapas  Ayuda
checking for sbrk... yes
checking for snprintf... yes
checking return type of random... long
checking for int8_t... yes
checking for int16_t... yes
checking for int32_t... yes
checking for u_int8_t... yes
checking for u_int16_t... yes
checking for u_int32_t... yes
checking for uchar... yes
checking for u_int... yes
checking for strtouq... yes
checking for strtoll... yes
checking for long... yes
checking size of long... 4
checking for __int64_t... no
checking for long long... yes
checking for int64_t... yes
checking which kind of 64-bit int to use... int64_t
checking for struct ether_header... found
checking for struct ether_addr... found
checking for addr2ascii... no
checking for Linux compliant tcphdr... found
checking for BSD compliant tcphdr... not found
checking for socklen_t... yes
checking for main in -lpcap... no
checking to make nse... no
Explicitly disabling static compilation
checking for dlopen in -ldl... yes
checking for a BSD-compatible install... /usr/bin/install -c
configure: creating ./config.status
config.status: creating Makefile
config.status: creating tcl/lib/ns-autoconf.tcl
config.status: creating indep-utils/webtrace-conv/ucb/Makefile
config.status: creating indep-utils/webtrace-conv/dec/Makefile
config.status: creating indep-utils/webtrace-conv/nlanr/Makefile
config.status: creating indep-utils/webtrace-conv/epa/Makefile
config.status: creating indep-utils/cmu-scen-gen/setdest/Makefile
config.status: creating autoconf.h
config.status: autoconf.h is unchanged
[Carol@localhost ns-2.31]$

```

Figura 29. Respuesta del terminal al comando `./configure`.

```

configure: creating ./config.status
config.status: creating Makefile
config.status: creating tcl/lib/ns-autoconf.tcl
config.status: creating indep-utils/webtrace-conv/ucb/Makefile
config.status: creating indep-utils/webtrace-conv/dec/Makefile
config.status: creating indep-utils/webtrace-conv/nlanr/Makefile
config.status: creating indep-utils/webtrace-conv/epa/Makefile
config.status: creating indep-utils/cmu-scen-gen/setdest/Makefile
config.status: creating autoconf.h
config.status: autoconf.h is unchanged
[Carol@localhost ns-2.31]$ make depend

```

Figura 30. Ingreso del comando `make depend` en el terminal.

```

Archivo  Editar  Ver  Terminal  Solapas  Ayuda
amp.cc sctp/sctp-cmt.cc sctp/sctpDebug.cc tools/integrator.cc tools/queue-monitor.cc tools/flowmon.cc tools/loss-monitor.cc queue/queue.cc queue/drop-tail.cc
adc/simple-intserv-sched.cc queue/red.cc queue/semantic-packetqueue.cc queue/semantic-red.cc tcp/ack-recons.cc queue/sfq.cc queue/fq.cc queue/drr.cc queue/s
rr.cc queue/cbq.cc queue/jobs.cc queue/marker.cc queue/demarker.cc queue/wfq.cc queue/wfqclassifier.cc queue/wfqaggreg.cc link/hackloss.cc queue/errmodel.cc
queue/fec.cc link/delay.cc tcp/snoop.cc gaf/gaf.cc link/dynalink.cc routing/rtpProtoDV.cc common/net-interface.cc mcast/ctrlMcast.cc mcast/mcast_ctrl.cc mcast/
srm.cc common/sessionhelper.cc queue/delaymodel.cc mcast/srm-ssm.cc mcast/srm-topo.cc routing/alloc-address.cc routing/address.cc lib/int.Vec.cc lib/int.RVec
.cc lib/dmalloc support.cc webcache/http.cc webcache/tcp-simple.cc webcache/pagepool.cc webcache/inval-agent.cc webcache/tcpapp.cc webcache/http-aux.cc webca
che/mcache.cc webcache/webtraf.cc webcache/webserver.cc webcache/logweb.cc empweb/empweb.cc empweb/empftp.cc realaudio/realaudio.cc mac/lanRouter.cc classifi
er/filter.cc common/pkt-counter.cc common/Decapsulator.cc common/Encapsulator.cc common/encap.cc mac/channel.cc mac/mac.cc mac/ll.cc mac/mac-802.11.cc mac/ma
c-802.3.cc mac/mac-tdma.cc mac/smac.cc mobile/mip.cc mobile/mip-reg.cc mobile/gridkeeper.cc mobile/propagation.cc mobile/tworayground.cc mobile/antenna.cc m
bile/omni-antenna.cc mobile/shadowing.cc mobile/shadowing-vis.cc mobile/dumb-agent.cc common/bi-connector.cc common/node.cc common/mobilenode.cc mac/arp.cc m
obile/god.cc mobile/dem.cc mobile/topography.cc mobile/modulation.cc queue/priqueue.cc queue/dsr-priqueue.cc mac/phy.cc mac/wired-phy.cc mac/wireless-phy.cc
mac/mac-timers.cc trace/cmu-trace.cc mac/varp.cc mac/mac-simple.cc satellite/sat-hdic.cc dsdv/dsdv.cc dsdv/rtable.cc queue/rtqueue.cc routing/rttable.cc imep
/imep.cc imep/dest queue.cc imep/imep api.cc imep/imep rt.cc imep/rxmit queue.cc imep/imep timers.cc imep/imep util.cc imep/imep io.cc tora/tora.cc tora/tora
api.cc tora/tora dest.cc tora/tora io.cc tora/tora logs.cc tora/tora neighbor.cc dsr/dsragent.cc dsr/hdr sr.cc dsr/mobicaache.cc dsr/path.cc dsr/requesttable
.cc dsr/routecache.cc dsr/add sr.cc dsr/dsr proto.cc dsr/flowstruct.cc dsr/linkcache.cc dsr/simplecache.cc dsr/sr forwarder.cc aodv/aodv logs.cc aodv/aodv.cc
aodv/aodv_rtable.cc aodv/aodv_rqueue.cc common/ns-process.cc satellite/satgeometry.cc satellite/sathandoff.cc satellite/satlink.cc satellite/satnode.cc sate
llite/satposition.cc satellite/satroute.cc satellite/sattrace.cc rap/raplist.cc rap/rap.cc rap/media-app.cc rap/utilities.cc common/fsm.cc tcp/tcp-abs.cc dif
fusion/diffusion.cc diffusion/diff_rate.cc diffusion/diff_prob.cc diffusion/diff_sink.cc diffusion/flooding.cc diffusion/omni_mcast.cc diffusion/hash_table.cc
c diffusion/routing_table.cc diffusion/rlist.cc tcp/tfrc.cc tcp/tfrc-sink.cc mobile/energy-model.cc apps/ping.cc tcp/tcp-rfc793edu.cc queue/rto.cc queue/sem
antic-rto.cc tcp/tcp-sack-rh.cc tcp/tcpscoreboard-rh.cc plm/loss-monitor-plm.cc plm/cbr-traffic-PP.cc linkstate/hdr-ls.cc mpls/classifier-addr-mpls.cc mpls/ldp
.cc mpls/mpis-module.cc routing/rtpmodule.cc classifier/classifier-hier.cc routing/addr-params.cc nix/hdr_nv.cc nix/classifier-nix.cc nix/nixnode.cc routealgo/
rnode.cc routealgo/bfs.cc routealgo/rbitmap.cc routealgo/rlookup.cc routealgo/routealgo.cc nix/nixvec.cc nix/nixroute.cc dsfserv/dsred.cc dsfserv/dsredq.cc
dsfserv/dsEdge.cc dsfserv/dsCore.cc dsfserv/dsPolicy.cc dsfserv/ew.cc dsfserv/dewp.cc queue/red-pd.cc queue/pi.cc queue/rem.cc queue/gk.cc
pushback/rate-limit.cc pushback/rate-limit-strategy.cc pushback/ident-tree.cc pushback/agg-spec.cc pushback/logging-data-struct.cc pushback/rate-estimator.cc
pushback/pushback-queue.cc pushback/pushback.cc common/parentnode.cc trace/basetrace.cc common/simulator.cc asim/asim.cc common/scheduler-map.cc common/spla
y-scheduler.cc linkstate/ls.cc linkstate/rtpProtoLS.cc pgm/classifier-pgm.cc pgm/pgm-agent.cc pgm/pgm-sender.cc pgm/pgm-receiver.cc mcast/rcvbuf.cc mcast/clas
sifier-lms.cc mcast/lms-agent.cc mcast/lms-receiver.cc mcast/lms-sender.cc queue/delay.cc xcp/xcpq.cc xcp/xcp.cc xcp-end-sys.cc mwan/p802.15.4csmaca.cc
wpan/p802.15.4fail.cc wpan/p802.15.4hlist.cc wpan/p802.15.4mac.cc wpan/p802.15.4nam.cc wpan/p802.15.4phy.cc wpan/p802.15.4sscs.cc wpan/p802.15.4timer.cc wpa
n/p802.15.4trace.cc wpan/p802.15.4transac.cc diffusion3/lib/nr/nr.cc diffusion3/lib/dr.cc diffusion3/filters/diffusion/one phase pull.cc diffusion3/filters/d
iffusion/two phase pull.cc diffusion3/lib/diffapp.cc diffusion3/ns/diffagent.cc diffusion3/ns/difftrg.cc diffusion3/ns/difftimer.cc diffusion3/filter core/fi
lter core.cc diffusion3/filter_core/iolog.cc diffusion3/filter_core/lostats.cc diffusion3/lib/main/attrs.cc diffusion3/lib/main/events.cc diffusion3/lib/main
/idev.cc diffusion3/lib/main/iobook.cc diffusion3/lib/main/timers.cc diffusion3/lib/main/message.cc diffusion3/lib/main/tools.cc diffusion3/apps/gear_exampl
es/gear_common.cc diffusion3/apps/gear_examples/gear_receiver.cc diffusion3/apps/gear_examples/gear_sender.cc diffusion3/apps/rmst_examples/rmst_sink.cc diff
usion3/apps/rmst_examples/rmst_source.cc diffusion3/apps/ping/lpp_ping_sender.cc diffusion3/apps/ping/lpp_ping_receiver.cc diffusion3/apps/ping/2pp_ping_send
er.cc diffusion3/apps/ping/2pp_ping_receiver.cc diffusion3/apps/ping/ping_common.cc diffusion3/apps/ping/push_receiver.cc diffusion3/apps/ping/push_sender.cc
diffusion3/filters/gear/gear_attr.cc diffusion3/filters/gear/gear.cc diffusion3/filters/gear/gear_tools.cc diffusion3/filters/misc/log.cc diffusion3/filters
/misc/srsrcr.cc diffusion3/filters/misc/tag.cc diffusion3/filters/rmst/rmst.cc diffusion3/filters/rmst/rmst_filter.cc delaybox/delaybox.cc packmime/packmime.H
TTP.cc packmime/packmime.HTTP rng.cc packmime/packmime.OL.cc packmime/packmime.OL_ranvar.cc packmime/packmime_ranvar.cc emulate/inet.c emulate/net-ip.cc emul
ate/net.cc emulate/tap.cc emulate/ether.cc emulate/internet.cc emulate/ping responder.cc emulate/arp.cc emulate/icmp.cc emulate/net-pcap.cc emulate/nat.cc em
ulate/iptables.cc emulate/tcptap.cc common/tclAppInit.cc common/tkAppInit.cc %> /dev/null
[Carol@localhost ns-2.31]$

```

Figura 31. Respuesta del terminal al comando *make depend*.

```

usion3/apps/rmst_examples/rmst_source.cc diffusion3/apps/ping/lpp_ping_sender.cc diffusion3/apps/ping/lpp_ping_receiver.cc diffusion3/apps/ping/2pp_ping_send
er.cc diffusion3/apps/ping/2pp_ping_receiver.cc diffusion3/apps/ping/ping_common.cc diffusion3/apps/ping/push_receiver.cc diffusion3/apps/ping/push_sender.cc
diffusion3/filters/gear/gear_attr.cc diffusion3/filters/gear/gear.cc diffusion3/filters/gear/gear_tools.cc diffusion3/filters/misc/log.cc diffusion3/filters
/misc/srsrcr.cc diffusion3/filters/misc/tag.cc diffusion3/filters/rmst/rmst.cc diffusion3/filters/rmst/rmst_filter.cc delaybox/delaybox.cc packmime/packmime.H
TTP.cc packmime/packmime.HTTP rng.cc packmime/packmime.OL.cc packmime/packmime.OL_ranvar.cc packmime/packmime_ranvar.cc emulate/inet.c emulate/net-ip.cc emul
ate/net.cc emulate/tap.cc emulate/ether.cc emulate/internet.cc emulate/ping responder.cc emulate/arp.cc emulate/icmp.cc emulate/net-pcap.cc emulate/nat.cc em
ulate/iptables.cc emulate/tcptap.cc common/tclAppInit.cc common/tkAppInit.cc %> /dev/null
[Carol@localhost ns-2.31]$ make

```

Figura 32. Ingreso del comando *make* en el terminal.

```

nv/ucb; do ( cd $i; make all; ) done
make[1]: se ingresa al directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/cmu-scen-gen/setdest'
make[1]: No se hace nada para `all'.
make[1]: se sale del directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/cmu-scen-gen/setdest'
make[1]: se ingresa al directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/dec'
make[1]: No se hace nada para `all'.
make[1]: se sale del directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/dec'
make[1]: se ingresa al directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/epa'
make[1]: No se hace nada para `all'.
make[1]: se sale del directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/epa'
make[1]: se ingresa al directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/nlanr'
make[1]: No se hace nada para `all'.
make[1]: se sale del directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/nlanr'
make[1]: se ingresa al directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/ucb'
make[1]: No se hace nada para `all'.
make[1]: se sale del directorio `/home/Carol/ns-allinone-2.31/ns-2.31/indep-utils/webtrace-conv/ucb'
[Carol@localhost ns-2.31]$

```

Figura 33. Respuesta del terminal al comando *make*.

Con la ayuda de los anteriores pasos es más fácil modificar los archivos que trae el NS-2 por defecto y crear nuestros propios programas con base en ellos. Este simulador se caracteriza por que el usuario puede crear sus propias herramientas o mejorar las existentes solo con el aprendizaje de los lenguajes necesarios (C++ y *Otc*).

Cabe resaltar que este libro incluye un manual de usuario que facilita el entendimiento de esta herramienta y una sección (2.4) donde se puede encontrar mayor información acerca del Simulador. Además en el manual de usuario se puede encontrar la instalación del Simulador *NS-2.31* y la instalación de la herramienta creada para *IntServ* e *IntServ6*.

3.5.1. DESCRIPCIÓN DEL PROCESO DE CLASIFICACIÓN EN INTSERV

El objetivo de este proyecto es simular la arquitectura de Servicios Integrados (*IntServ*) y la Arquitectura de Servicios Integrados Versión 6 (*IntServ6*) propuesta por el Ingeniero Jhon Jairo Padilla en su tesis doctoral, para así poder hacer una comparación entre las dos arquitecturas y mostrar las ventajas y desventajas de cada una. Por tanto, a continuación se describirán los procesos a simular y posteriormente se describirá la forma en que se construyeron.

3.5.1.1. Clasificación en *IntServ*

La primera arquitectura que se diseñó en este proyecto fue Servicios Integrados, y para esto se estudió detalladamente el procedimiento que realiza un router *IntServ* en una red IP cualquiera. En la figura 34 [70] se muestra el Modelo de Referencia de Servicios Integrados donde se explica de forma detallada las partes que componen un router *IntServ*.

En primer lugar, la Arquitectura de Servicios Integrados soporta calidad de Servicio por flujos, es decir, se hacen reservas de recursos para los flujos que lo soliciten. La idea es reservar estos recursos antes de empezar la transmisión de paquetes y por consiguiente se hace la petición a la red con la especificación del flujo (*Flow Specification*) y sus requerimientos. El router *IntServ*, ante esta petición, le da la tarea al Control de Admisión de calcular si es posible entregar estos recursos al usuario; finalmente se da una respuesta de afirmación o negación de la petición.

En caso de aceptarse la petición, esta reserva es guardada en una Tabla de Reservas y se prosigue a la transmisión de paquetes. En caso de negar la petición se devuelve un mensaje al usuario informándole la situación de la red. Este usuario decide si volver a enviar una petición a la red con otras especificaciones. El protocolo encargado de realizar esta reserva de recursos es el *RSVP* [28], el cual utiliza mensajes de señalización en la red para enviar peticiones y respuestas de las peticiones al usuario. A este proceso de revisar las capacidades de la red, y en caso de tenerlas asignar recursos, se le llama Plano de Control.

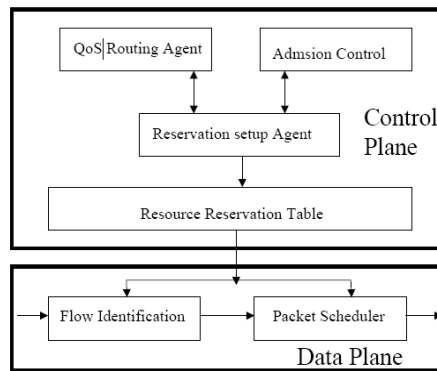


Figura 34. Modelo de Referencia de Servicios Integrados

La información de las reservas sirve para configurar los módulos de Identificación ó Clasificación de flujos (*Flow Identification*) y Planificador de flujos (*Packet Scheduler*) del Plano de Datos del Modelo de Referencia de *IntServ*. Cuando se empieza la transmisión de paquetes, estos llegan al Plano de Datos y son enviados al Clasificador. En el proceso de Clasificación se comparan los flujos de los paquetes entrantes con los flujos reservados, esto se realiza con el fin de entregarle a los flujos reservados su servicio pactado anteriormente en el Plano de Control.

La identificación de los paquetes se hace por medio de la extracción de la Quintupla de la cabecera IP de cada paquete y se calcula un Número Hash con estos 5 números. Este Número Hash, en caso de tener reserva, debe estar registrado en la Tabla de Reservas y, al llegar el paquete al router, este lo debe clasificar y darle el ancho de banda estipulado por el control de admisión. Cabe resaltar que existen diferentes formas de calcular el Número Hash. Sin embargo, en este proyecto se calcula mediante una función *XOR* con palabras de 20 bits, debido a que *IntServ6* lo realiza así [54]. Éste método es más simple que el *CRC-32* y obtiene una eficiencia similar a este [70].

Por otra parte, existe la posibilidad de que se encuentren dos flujos con el mismo Número Hash. En estos casos se acude a una Tabla de Resolución de Colisiones donde se comparan todos los valores en detalle de la Quintupla. En la figura 35 [54] se explica como es el procedimiento básico de *Hashing* en *IntServ*.

Después de ser clasificado, el paquete llega al planificador, que se encarga de poner estos paquetes en las colas correctas y darle el ancho de banda correspondiente.

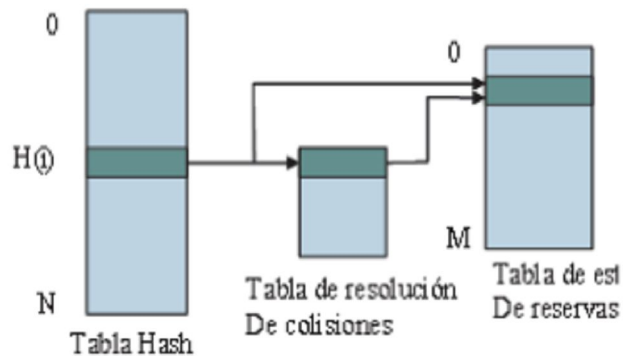


Figura 35. Procedimiento de Hashing en IntServ

3.5.1.2. Clasificación en *IntServ6*

En el caso del Clasificador *IntServ6* en [54] se presenta una nueva propuesta para agilizar el enrutamiento de los paquetes de un flujo específico dentro de un router y mejorar el desempeño del mismo. Para este objetivo se le asigna al Host origen la tarea de calcular el mismo Número Hash con la Quíntupla del paquete y almacenarlo en el nuevo campo de Etiqueta de Flujo de *IPv6*, con la finalidad de reducir el retardo dentro del Clasificador y hacer más eficiente la red [54]. De esta forma los paquetes de un flujo determinado viajarán a través de la red con su respectivo Número Hash, calculado en el origen, dentro de la cabecera IP del mismo y facilitarán la tarea del Clasificador en el momento de hacer la búsqueda de la reserva dentro de la Tabla Hash utilizando este número como un índice. Finalmente al ser clasificados los paquetes, éstos llegan al planificador, anteriormente mencionado, encargado de asignarles los recursos a los flujos en base a la información de la tabla de reservas.

En los siguientes puntos se detallará el procedimiento que se siguió para diseñar e implementar los componentes o partes de un router *IntServ* y los cambios que se tienen que realizar al router *IntServ* para diseñar el modelo de simulación de la nueva propuesta *IntServ6*.

3.5.2. DESARROLLO DEL PROTOCOLO EN C++

Debido a que en *IntServ6*, la principal diferencia con respecto a *IntServ*, radica en el procedimiento de Clasificación, fue necesario determinar los módulos de *NS-2* que realizaban esta tarea para luego ser modificados. Para una mejor ubicación de estos archivos en *NS-2* se estructuró una tabla de rutas para mayor facilidad al momento de buscarlos (Tabla 9).

Número	Nombre del Archivo	Ruta
1	classifier-hash.h	home/usuario/ns-allinone-2.31/ns-2.31/classifier
2	classifier-hash.cc	home/usuario/ns-allinone-2.31/ns-2.31/classifier
3	classifier-IntServ.h	home/usuario/ns-allinone-2.31/ns-2.31/classifier
4	classifier-IntServ.cc	home/usuario/ns-allinone-2.31/ns-2.31/classifier
5	classifier-IntServ6.h	home/usuario/ns-allinone-2.31/ns-2.31/classifier
6	classifier-IntServ6.cc	home/usuario/ns-allinone-2.31/ns-2.31/classifier
7	HostOrigen6.h	home/usuario/ns-allinone-2.31/ns-2.31/classifier
8	HostOrigen6.cc	home/usuario/ns-allinone-2.31/ns-2.31/classifier
9	ip.h	home/usuario/ns-allinone-2.31/ns-2.31/common
10	ip.cc	home/usuario/ns-allinone-2.31/ns-2.31/common
11	config.h	home/usuario/ns-allinone-2.31/ns-2.31
12	makefile.in	home/usuario/ns-allinone-2.31/ns-2.31
13	ns-default.tcl	home/usuario/ns-allinone-2.31/ns-2.31/tcl/lib

Tabla 9. Rutas de los archivos que necesitan modificaciones en NS-2 y archivos añadidos.

Los dos primeros archivos de la tabla anterior se refieren a un Clasificador Hash que trae el simulador por defecto. Este es utilizado por algunos de los procesos en general del simulador y en nuestro caso es preferible no trabajar con él debido a que son usados por otros procesos del NS-2, lo que afectaría otras partes del simulador. Por tanto, se decidió que se crearían unos nuevos procesos con nombres diferentes que sólo se usarán en nuestra herramienta. Por consiguiente, se crearon los seis archivos siguientes en la tabla (archivos 3 al 8), que hacen referencia a los clasificadores *IntServ* e *IntServ6* y que presentan las modificaciones hechas al Clasificador Hash del simulador y algunos archivos que son indispensables para el proceso de los dos clasificadores y que en consecuencia también fueron modificados. Como se puede observar en la tabla, estos nuevos archivos se encuentran ubicados en la misma ruta del Clasificador Hash, lo que representa una ventaja a la hora de añadir estos módulos a los routers estándar del NS-2 debido a que esto permite la reutilización de los métodos existentes designados para esta función. Por último se debe tener en

cuenta la importancia de incluir estos nuevos objetos en el fichero *makefile.in* para luego recompilar todos los archivos del *NS-2*.

Los siguientes archivos en la tabla (*ip.h*, *ip.cc*), deben ser modificados para agregar algunos de los parámetros de la quintupla (Dirección IP origen, dirección IP destino, puerto origen, puerto destino e Identificación del protocolo) a la cabecera IP del paquete, la cual es leída al ingresar a un router *IntServ*, y un nuevo campo que representa la Etiqueta de Flujo de *IPv6* que tiene como objetivo almacenar el Número Hash necesario en *IntServ6*. Cabe resaltar que la Quintupla es ingresada a esta cabecera IP, debido a que el *NS-2* no los utiliza para la clasificación de los paquetes. Para modificar estos archivos es indispensable adicionarles las líneas de programación faltantes y recompilar el simulador para incluir los nuevos cambios. (Para ver estos cambios remitirse al anexo A, página 225, donde se muestra el código del archivo). En caso de que halla algún error al compilar, se muestran las líneas en donde ocurrió el error para mayor facilidad al corregirlos.

Sin embargo, el archivo *config.h* al igual que los de la cabecera IP, solo necesitan modificaciones y no es necesario crear un nuevo fichero. En este archivo, se adicionan algunas estructuras que son primordiales para los nuevos diseños de clasificadores. Para dar una mejor explicación de los cambios realizados en el simulador se muestra una breve explicación de los componentes principales de un router para soportar Calidad de Servicio mediante Servicios Integrados y se hace una profundización de cada una de esas partes.

3.5.2.1. DISEÑO DEL CLASIFICADOR INTSERV

Para crear un nuevo clasificador en el simulador *NS-2* es necesario analizar primero el orden jerárquico de clases y objetos que tiene el simulador encontrando cada uno de los clasificadores que trae por defecto.

- En la figura 36 se puede observar que la Clase *Classifier* tiene por defecto 13 tipos de clasificadores diferentes, los cuales presentan variados comportamientos de red. El clasificador que más se acomoda a los comportamientos que debe tener nuestro Clasificador *IntServ* es el Clasificador Hash. Éste es usado para clasificar un paquete como un miembro de un flujo particular. Como su nombre lo indica los Clasificadores Hash usan internamente una Tabla Hash para asignar paquetes a los flujos. Estos objetos son usados donde se requiera información a nivel de flujo (por ejemplo en disciplinas de colas específicas por flujos y recolección de estadísticas).

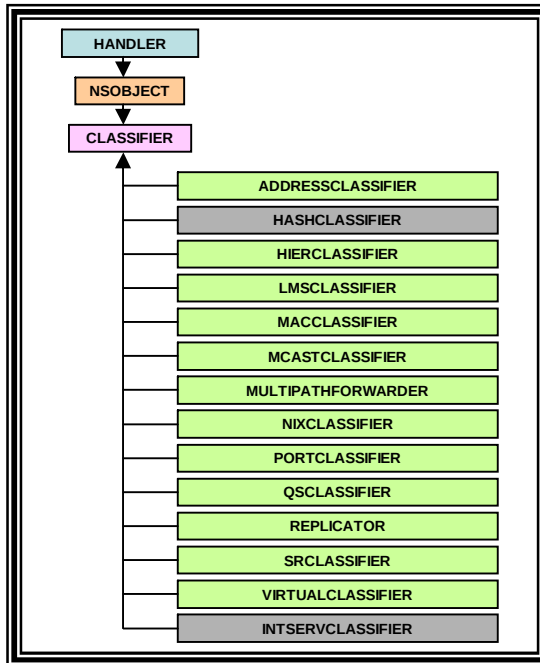


Figura 36. Clase Classifier

En el Calificador-Hash del NS-2, los paquetes pueden ser asignados a los flujos basándose en uno o varios campos:

- Flow ID
- Dirección IP Destino (32 bits)
- Direcciones IP Fuente/Destino (32 bits)
- Combinaciones de direcciones IP Fuente/Destino y Flow ID.

Este clasificador tiene 4 clases diferentes basándose en las 4 combinaciones de campos anteriores, en la Figura 37 se pueden observar estas 4 clases y además el nuevo clasificador creado para nuestro propósito.

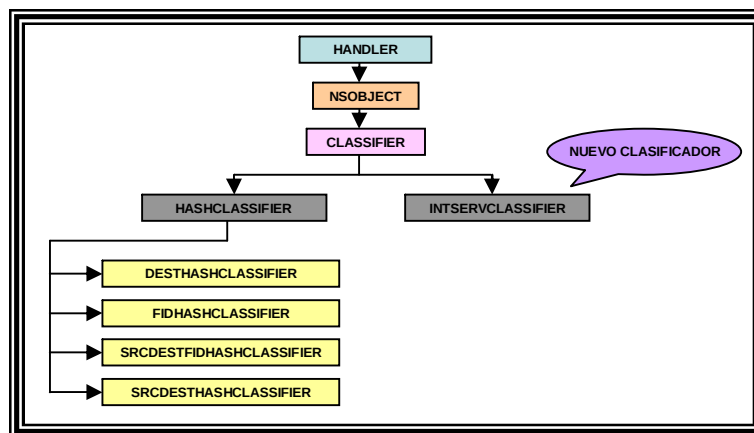


Figura 37. Clase HashClassifier y nueva Clase IntServClassifier

Como se dijo anteriormente hay 4 clases que heredan características del Clasificador Hash, pero ninguna de estas clases incluyen la Quintupla completa para el cálculo del Número Hash. Debido a esto se creó un nuevo Clasificador llamado *IntServClassifier*. Con este nuevo clasificador se incluyen nuevas características propias como la adición los campos necesarios para completar la Quintupla.

En resumen el procedimiento que se debe seguir en un router *IntServ* es el siguiente [54] (Ver Figura 38):

- ❖ Entrada del paquete al router y lectura de la Quintupla.
- ❖ Cálculo del Número Hash con la Quintupla.
- ❖ Se hace la búsqueda de este Número Hash en la Tabla de Reservas y se verifica si existe o no colisión con otro flujo diferente.
- ❖ En caso de que no exista colisión se envía el paquete al planificador para que este le de el ancho de banda pactado y enrute el paquete.
- ❖ En caso de que sí exista colisión se hace la búsqueda en la Tabla de Colisiones y se comparan todos los valores de la Quintupla hasta encontrar la reserva correspondiente a ese flujo, para después enviarlo al planificador y finalmente enrutar el paquete.

Este procedimiento se hace teniendo en cuenta que ya deben estar estipuladas las reservas en la Tabla Hash y en caso de haber colisiones, se deben tener las Quintuplas y anchos de banda correspondientes en la Tabla de Resolución de Colisiones.

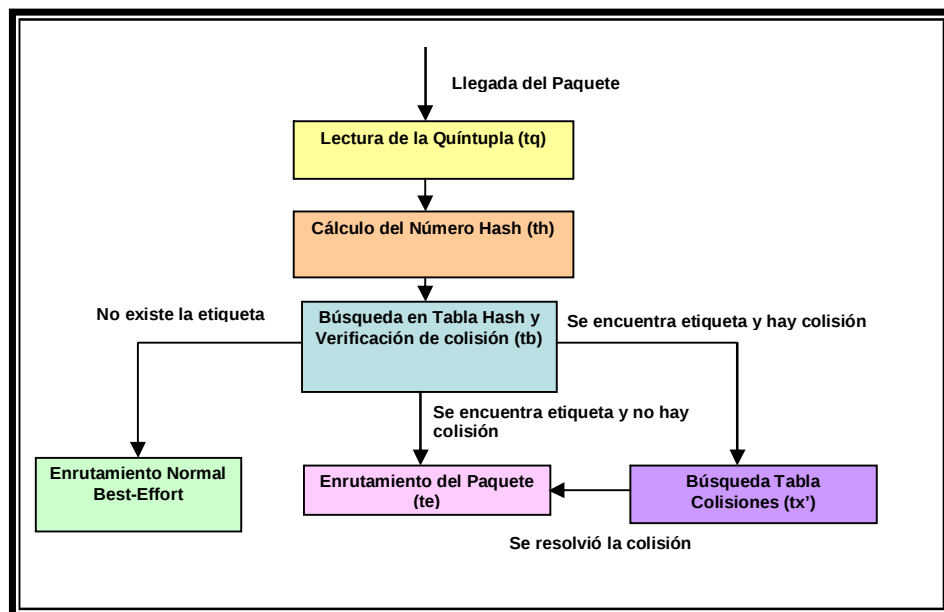


Figura 38. Diagrama de Transición de estados del clasificador IntServ.

A. MODULO DEL CLASIFICADOR INTSERV EN C++

El procedimiento anterior debe ser programado en C++ creando nuevas carpetas y modificando otras. Antes de modificar cualquier archivo en el simulador se creó un modulo en C++, en un archivo de texto con extensión *.cpp* en linux (compilable con el comando *g++* de *Linux*), que no tiene ningún tipo de enlace con el simulador y que además muestra de una forma organizada el comportamiento ideal de un Clasificador *IntServ*. Este se crea con el objetivo de obtener una base de cómo debe comportarse el clasificador y tener una ventaja a la hora de ingresar este código nuevo al simulador. Este código tiene en cuenta el Diagrama de Transición de estados del Clasificador *IntServ* y utiliza herramientas de grabación de archivos de C++ para verificar los resultados de los procedimientos de cada paso del Clasificador.

En primer lugar, en la figura 39 se ilustra un diagrama de todos los pasos que tiene el clasificador para simular el control de admisión de la red, creando una tabla hash y una tabla de colisiones que son necesarias para clasificación de los paquetes .

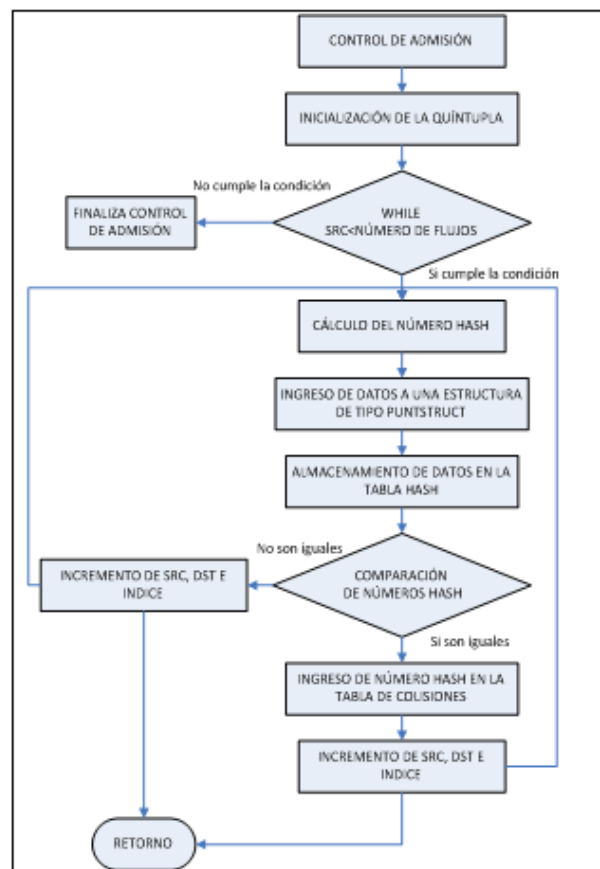


Figura 39. Control de Admisión del Clasificador IntServ.

El primer paso de este módulo, como se muestra en el diagrama anterior, es la inicialización de la Quintupla. Esta inicialización se realiza con el fin de simular la entrega de datos iniciales por medio del *script* de *Tcl*, para la creación de la Tabla Hash y la Tabla de Colisiones.

En este proceso de inicialización se le fijan los parámetros a cada variable de la Quintupla (Dirección IP Origen, Dirección IP Destino, Puerto Origen, Puerto Destino e Identificación de Protocolo), dependiendo del tipo de topología que desea el usuario. Con la ayuda de la Figura 40, se realizará la explicación detallada de toda la programación utilizada en este módulo. Cabe resaltar en esta sección, que las Direcciones IP de Origen y Destino en el simulador *NS-2* equivalen a las identificaciones de cada nodo, debido a que no se utilizan direcciones IP reales. La identificación de cada nodo es el número que acompaña a la letra *n* con la cual se crean los nodos. Por ejemplo, si el nodo es *n3*, la identificación del nodo es 3.

En este diagrama, los nodos *n0* a *n3* y *n6* a *n9* representan los computadores Origen y Destino respectivamente. Además el nodo *n4* representa el Router que soporta *IntServ* o *IntServ6* y el *n5* es un Router normal del simulador.

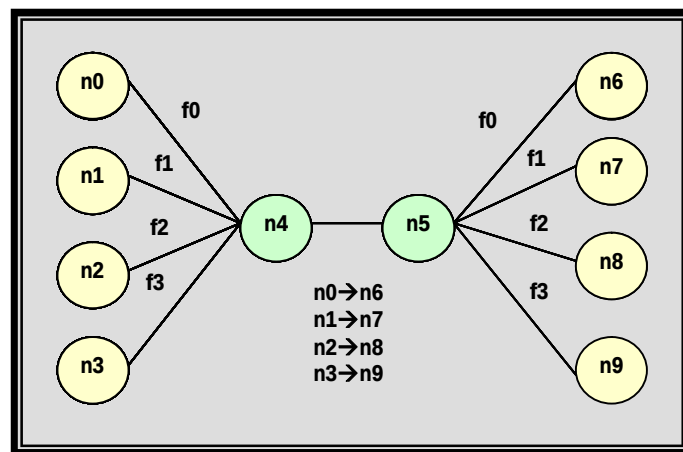


Figura 40. Topología ejemplo para IntServ

Por ejemplo, si se crean 4 nodos de origen y 4 nodos de destino, como se muestra en el diagrama anterior y teniendo en cuenta que la Identificación de Protocolo y los Puertos de Origen y de Destino siempre son iguales para cada flujo, solo variarían las Direcciones IP de Origen y Destino, haciendo más fácil la programación en C++ debido al diseño de la topología. Cabe resaltar que este tipo de topología y los parámetros de la misma pueden variar dependiendo de lo deseado por el usuario. El código implementado en este paso es el siguiente:

```

void inicializar() { // Método para inicializar valores de la quintupla
    src = 0;
    sport = 0;
}

```

```

        dst = 6;
        dport = 0;
        pid = 0;
    return;
}

```

Nótese que los Puertos de Origen y Destino se inicializan en cero al igual que la Identificación del Protocolo. Las direcciones IP Origen y Destino son cero y seis respectivamente, debido a que esos son los routers que representan, junto con los nodos *n4* y *n5*, la trayectoria del primer Flujo de paquetes que se transmite. De aquí en adelante se van incrementando las direcciones para ir variando el flujo de paquetes y calcular su respectivo Número Hash, lo cual se realiza mediante un ciclo *while* que se explica en el siguiente paso.

El segundo paso del diagrama del Control de Admisión del Clasificador IntServ (Figura 39), es la presentación de un bucle *while*, el cual tiene como condición que la dirección IP Origen sea menor al Número de Flujos, pues esa sería la cantidad de Números Hash que se calcularán. Este ciclo *while* tiene un cuerpo de instrucciones los cuales se realizan dependiendo de si se cumple con la condición al incrementar la variable *src*. A continuación se detalla el cuerpo de instrucciones de este ciclo *while* y el código utilizado:

```

while (src<4) { // Número de flujos → 4
    hashkey_1(); // Calculo del Número hash
    archivo << hash_ << "\n"; // Herramienta de ayuda para imprimir archivos con resultados
    estructura(); // Ingreso de datos a la estructura
    registrar(campos, 0, indice); //Llena los datos en la tabla hash
    registrar(campos, 2, indice); //Compara si hay algún hash igual
    ++src; // Incrementa src, dst y el indice para volver a realizar el mismo procedimiento
    ++dst;
    ++indice;
}

```

Los pasos del ciclo *while* son los siguientes:

✓ Cálculo del Número Hash

El cálculo del Número Hash se realiza utilizando una función XOR Folding entre los campos de la Quíntupla obteniendo como resultado un Número Hash con tamaño de 20 bits. Teniendo en cuenta que cada uno de los campos de la Quíntupla no poseen el mismo tamaño, se reagruparon los bits de la Quíntupla en subcampos de 20 bits. (Ver Figura 41).

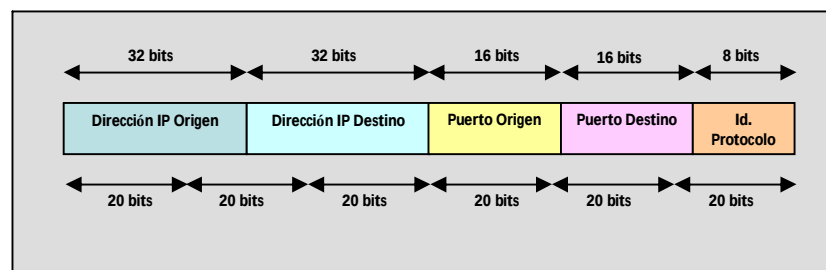


Figura 41. Diagrama del Corrimiento de Bits del Cálculo del Número Hash

Cómo una ayuda, se presenta un ejemplo de cómo se realizó el cálculo del Número Hash y cuales fueron sus principales procesos: (Figura 42).

Ejemplo: Suponiendo la Quintupla de esta forma:

- Dirección IP Origen en hexadecimal: "A8DEB624"
- Dirección IP Destino en hexadecimal: "B34A217A"
- Puerto Origen en hexadecimal: "3A2B"
- Puerto Destino en hexadecimal: "5C8A"
- Identificación del Protocolo en hexadecimal: "2A"

Se realiza inicialmente un corrimiento de Bits, mediante los signos de operación "<<" y ">>", con el fin de organizar toda la Quintupla en un arreglo de 20 bits, para luego realizarse la función *XOR Folding*. Para este proceso fue indispensable utilizar las funciones lógicas básicas como *AND* (&) y *OR* (|). Además se utilizaron unas máscaras para ayudar a la extracción por partes de la Quintupla y su almacenamiento en el arreglo. El anterior proceso se describe a continuación:

A8DEB624 FFFFF000	→ &	Dirección IP Origen Mascara

A8DEB000 000A8DEB	R1 R1 >>= 3	→ <u>Arreglo[0] = R1</u>
A8DEB624 0000FFF	→ &	Dirección IP Origen Mascara

0000624 00062400	R2 R2 <<= 2	
B34A217A FF000000	→ &	Dirección IP Destino Mascara

B3000000 000000B3	R3 R3 >>= 6	
00062400 000000B3	R2 R3	

000624B3	R2 R3 = R4	→ <u>Arreglo[1] = R4</u>
B34A217A 00FFFFF0	→ &	Dirección IP Destino Mascara

004A2170 0004A217	R5 R5 >>= 1	→ <u>Arreglo[2] = R5</u>
B34A217A 0000000F	→ &	Dirección IP Destino Mascara

0000000A 000A0000	R6 R6 <<= 4	
00003A2B 000A0000	→ R6	Puerto Origen

000A3A2B	Puerto Origen R6 = R7	→ <u>Arreglo[3] = R7</u>
00005C8A 0005C8A0	R8 R8 <<= 1	

0000002A 000000F0	→ &	Identificación de Protocolo Mascara

00000020 00000002	R9 R9 >>= 1	
0005C8A0 00000002	R8 R9	

0005C8A2	R8 R9 = R10 →	<u>Arreglo[4] = R10</u>
0000002A 0000000F	→ &	Identificación de Protocolo Mascara

0000000A	R11 →	<u>Arreglo[5] = R11</u>

De esta forma se almacenan los valores en el arreglo para finalmente realizarle la función *XOR Folding* mediante un ciclo *for*, y así calcular el nuevo Número Hash.

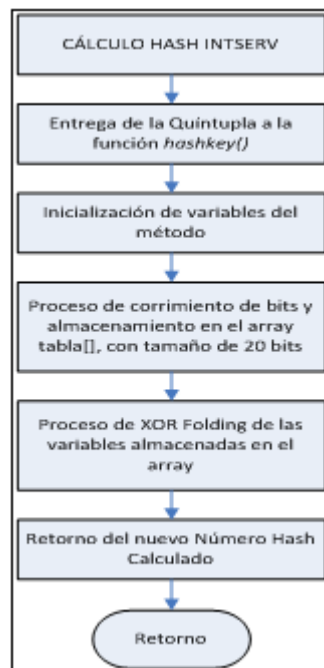


Figura 42. Cálculo del Número Hash en el Clasificador IntServ.

En el segmento de código que se ubica a continuación, el primer paso del algoritmo del Cálculo del Número Hash, el cual se realiza mediante la invocación de la función *Hashkey_1()*, es la entrega de los valores de la Quintupla al método, esto se realiza mediante la línea de código que invoca al método con los valores de entrada y el tipo de valor de retorno. El siguiente paso al recibir las variables de entrada es la inicialización de las variables que son utilizadas en el proceso del cálculo como lo son: *mask1_* que representa la variable a la cual se le van ingresando las mascarar utilizadas en el corrimiento, dos variables temporales *temp1_* y *temp2_* para el proceso en general y por último una variable llamada *hash_*, en la cual se almacenará el resultado del cálculo. La notación utilizada en este cálculo es la siguiente: *src* es la dirección IP Origen, *dst* es la dirección IP

destino, *sport* es el Puerto del Origen, *dport* es el Puerto del Destino y finalmente *pid* es la Identificación del Protocolo.

```
const char*  hashkey_10() { // Método para calcular el número hash (32 bits)

    mask1_ = 0;
    temp1_ = 0;
    temp2_ = 0;
    hash_ = 0;

    // Cálculo hash
    mask1_ = 0xffff000;           //Mask 1
    temp1_ = src;
    temp1_ &= mask1_;
    temp1_ >>= 3;
    tabla[0] = temp1_;           //Variable 1
    mask1_ = 0x00000fff;         //Mask 2
    temp1_ = src;
    temp1_ &= mask1_;
    temp1_ <<= 2;
    mask1_ = 0xff000000;         //Mask 3
    temp2_ = dst;
    temp2_ &= mask1_;
    temp2_ >>= 6;
    temp1_ |= temp2_;
    tabla[1] = temp1_;           //Variable 2
    mask1_ = 0x00ffff0;         //Mask 4
    temp1_ = dst;
    temp1_ &= mask1_;
    temp1_ >>= 1;
    tabla[2] = temp1_;           //Variable 3
    mask1_ = 0x0000000f;         //Mask 5
    temp1_ = dst;
    temp1_ &= mask1_;
    temp1_ <<= 4;
    temp2_ = sport;
    temp1_ |= temp2_;
    tabla[3] = temp1_;           // Variable 4
    temp1_ = dport;
    temp1_ <<= 1;
    mask1_ = 0x000000f0;         //Mask 6
    temp2_ = pid;
    temp2_ &= mask1_;
    temp2_ >>= 1;
    temp1_ |= temp2_;
    tabla[4] = temp1_;           //Variable 5
    mask1_ = 0x0000000f;         //Mask 7
    temp1_ = pid;
    temp1_ &= mask1_;
    tabla[5] = temp1_;           //Variable 6

    for (i=0;i<6;i++)
        hash_ ^= tabla[i];

    return (const char*) &hash_;
}
```

✓ Ingreso de Datos a la Estructura

Para mayor organización y optimización del programa se invoca a un método llamado *estructura()*, el cual realiza el ingreso del Número Hash, junto con los campos de origen, destino, colisión y Ancho de Banda a una estructura

denominada *campos*, creada al iniciar el programa, para ingresar estos datos a una Tabla Hash o a la Tabla de Colisiones dependiendo si se presenta colisión o no. En esta sección se crearon nuevos tipos de estructuras (*puntstruct* y *rescol*) utilizadas en este proyecto para acceder a la Tabla Hash o a la Tabla de Colisiones según el caso. Estas estructuras son adicionadas al simulador NS-2 sin modificar las estructuras que tiene el simulador por defecto.

Las variables *campos* y *colisiones* son de tipo *puntstruct* y *rescol* respectivamente, y se crean con el siguiente código:

```
//Estructuras

puntstruct campos;           // Nombre estructura Tabla Hash
rescol colisiones;           // Nombre estructura Tabla de Colisiones

struct puntstruct { // Estructura Utilizada en los métodos de la Tabla Hash
    int    numhash_;         // Numero hash
    int    BW_;              // Ancho de Banda
    int    colision_;        // Colision
    int    origen_;          // Dirección IP Origen
    int    destino_;         // Dirección IP Destino
};

struct rescol { // Estructura Utilizada en los métodos de la Tabla de Colisiones
    int    numhash1_;        // Numero Hash
    int    origen1_;         // Dirección IP Origen
    int    destino1_;        // Dirección IP Destino
    int    BW1_;             // Ancho de Banda
};
```

Las variables almacenadas en la estructura se describen a continuación.

- Número Hash de 20 bits recién calculado y de tipo entero.
- Direcciones IP Origen y Destino (32 bits c/u y tipo entero), ya que en este caso los Puertos y la Identificación de Protocolo son iguales para todos los flujos. Estos dos campos son necesarios más adelante en caso de que exista una colisión y sea indispensable la comparación y verificación de dichos campos.
- Campo de Colisión, utilizado en la Tabla Hash para comprobar si existe algún Numero Hash igual. Este campo es de tipo entero.
- Y por último el Ancho de Banda, que es el valor que se va a retornar al hacer una búsqueda de un paquete en la Tabla Hash.

```
void estructura() { // Método para llenar la estructura de la Tabla Hash
    campos.numhash_ = hash_;           // Número Hash
    campos.BW_ = 0;                    // Ancho de Banda
    campos.colision_ = 0;               // Colisión
    campos.origen_ = src;               // Dirección IP Origen
    campos.destino_ = dst;              // Dirección IP Destino
    return;
} // Los anchos de banda se definen más adelante cuando se acopla este modulo C++ al simulador
```

✓ Gestión de la Tabla Hash en *IntServ*

Con el ingreso de estos campos a la estructura, se invoca a un método llamado *registrar()*, en el cual los datos de entrada son la estructura anterior y algunos parámetros que definen que tipo de proceso se va a realizar.

La declaración de este método es la siguiente:

```
int registrar(puntstruct& tab, int a, int indice); // Método para acceder a la Tabla Hash (Llenar, Buscar o Comparar datos)
```

Donde *tab* es un apuntador a una estructura de tipo *puntstruct*, la variable *a* es de tipo entero y la variable *indice* de tipo entero y se utiliza como índice de la posición de la Tabla Hash.

La tarea principal de esta función es acceder a una matriz de datos que equivale a la Tabla Hash. La declaración de la matriz es la siguiente:

```
static int htabla [x] [5]; // [Filas] [Columnas]
```

Donde *x* es una variable que equivale al Número de Flujos con la cual este trabajando la simulación.

El método presenta una variable *a* que dependiendo de su valor (0, 1 o 2) da paso a un procedimiento diferente. Si *a=0*, se toman los datos que trae la estructura y se guardan en una matriz que es equivalente a la Tabla Hash. Si *a=1*, se va a buscar un Número Hash dado por la estructura en la Tabla Hash, y finalmente si *a= 2*, se va a hacer una comparación de el Número Hash recién ingresado y los valores anteriormente almacenados en la Tabla Hash. Para una mayor comprensión, en la Figura 43 se ilustran los pasos que presenta este método.

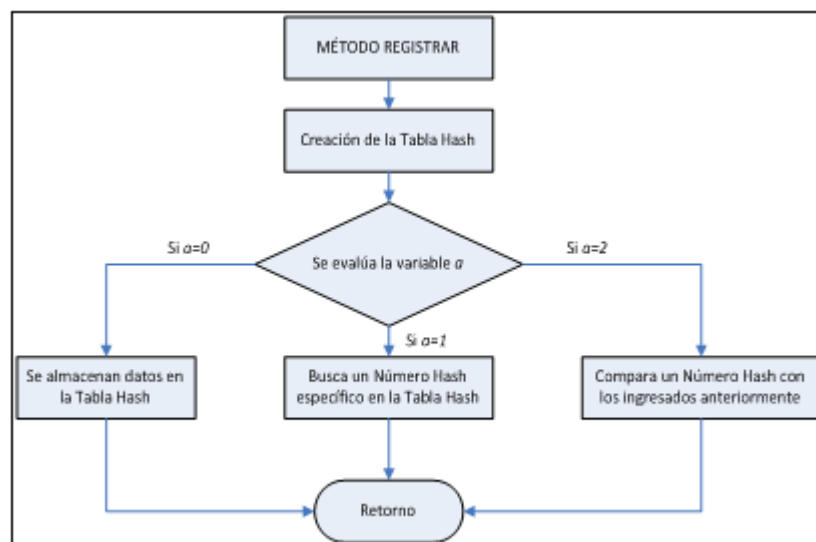


Figura 43. Diagrama de Flujo del método *registrar()*.

A continuación se realiza mediante diagrama de flujos una explicación de los pasos y secuencias que presentan los 3 casos del método *registrar()*. El primer proceso que se lleva a cabo se realiza solo cuando *a* es igual a 0 y hace referencia al almacenamiento de ciertas variables que se encuentran en una estructura y son trasladadas a una matriz llamada *htable*. En la Figura 44 se ilustra este procedimiento.

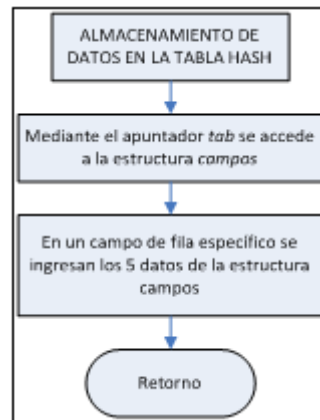


Figura 44. Almacenamiento de Datos en la Tabla Hash en IntServ

El objetivo de este caso es llenar la fila de la Tabla Hash con sus valores Hash y sus respectivos valores de origen, destino, colisión y ancho de banda, los cuales son necesarios más adelante para comparar con otros valores y tomar decisiones en el algoritmo. La fila es indicada mediante la variable *indice*. El código es el siguiente:

```

if (a == 0) {
    x = indice;
    htabla[x][1] = tab.numhash_;
    htabla[x][2] = tab.BW_;
    htabla[x][3] = tab.colision_;
    htabla[x][4] = tab.origen_;
    htabla[x][5] = tab.destino_;
    return 0;
}
  
```

El siguiente caso del método *registrar()* es la búsqueda de un Número Hash en la Tabla Hash y solo se realiza en caso de que *a* sea igual a 1. Su entrada inicial para realizar la búsqueda es una estructura donde se encuentra el Valor Hash que se desea buscar. Al finalizar la búsqueda y encontrar el número correcto se verifica si tiene colisión, y dependiendo de esta variable se decide en cual tabla (hash o colisiones) se obtiene el ancho de banda de ese flujo para finalmente darle el servicio pactado. (Ver Figura 45).

Si se encuentra una colisión en el Número Hash que se busca, esto representa una nueva búsqueda en la Tabla de Colisiones, la cual tiene su

propio método de acceso llamado *resolver()* y su respectiva estructura llamada *colisiones* mencionados anteriormente. Este método se explica más adelante con sus respectivos diagramas de flujo.

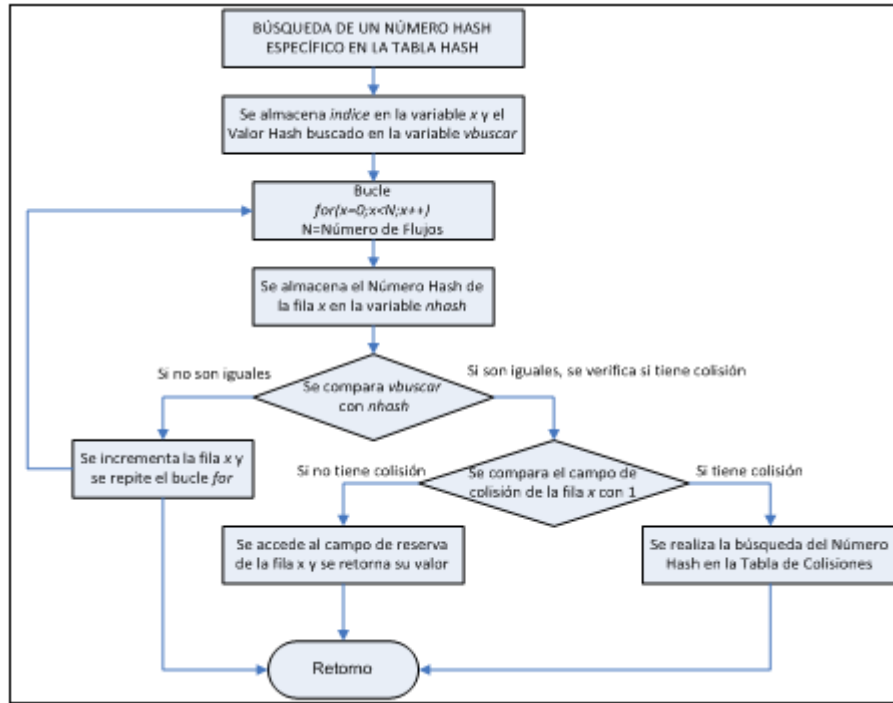


Figura 45. Búsqueda de un Número Hash específico en la Tabla Hash

Las líneas de programación en C++ utilizadas para cumplir con el diagrama de flujo anterior son las siguientes (Figura 45):

```

else if (a == 1) {
    x = indice;
    vbuscar = tab.numhash_;
    for (x=0; x<5; x++) {
        nhash = htabla [x] [1];
        if (vbuscar == nhash) {
            rtdocol = htabla [x] [3];
            if (rtdocol == 1) {
                colisiones.numhash1_ = nhash;
                colisiones.origen1_ = htabla [x] [4];
                colisiones.destino1_ = htabla [x] [5];
                y = resolver(colisiones, 1);
                rtdoflow = y;
                x = 1300;
            } else {
                rtdoflow = htabla [x] [2];
                x = 1300;
            }
        }
    }
    return (rtdoflow);
}

```

El siguiente caso del método *registrar()* es cuando *a* es igual a 2. En éste se compara un Número Hash entrante con los valores guardados en la tabla anteriormente. Este proceso se hace con el fin de encontrar las colisiones de los Números Hash y organizarlos en su respectiva Tabla Hash. De modo que en la tabla hash sólo se encuentre uno de cada Número Hash, y en caso de tener colisión verificar con el campo *colision_* y hacer la búsqueda en otra Tabla de Colisiones. Cabe resaltar que este proceso se realiza después de cada nueva entrada a la Tabla Hash y en caso de que se encuentre un Número Hash igual, se elimina la fila que se repite y se coloca en 1 la bandera de colisión en el primer Número Hash encontrado. Este proceso se encuentra de una forma más detallada en la Figura 46 y sus líneas de código se encuentran a continuación:

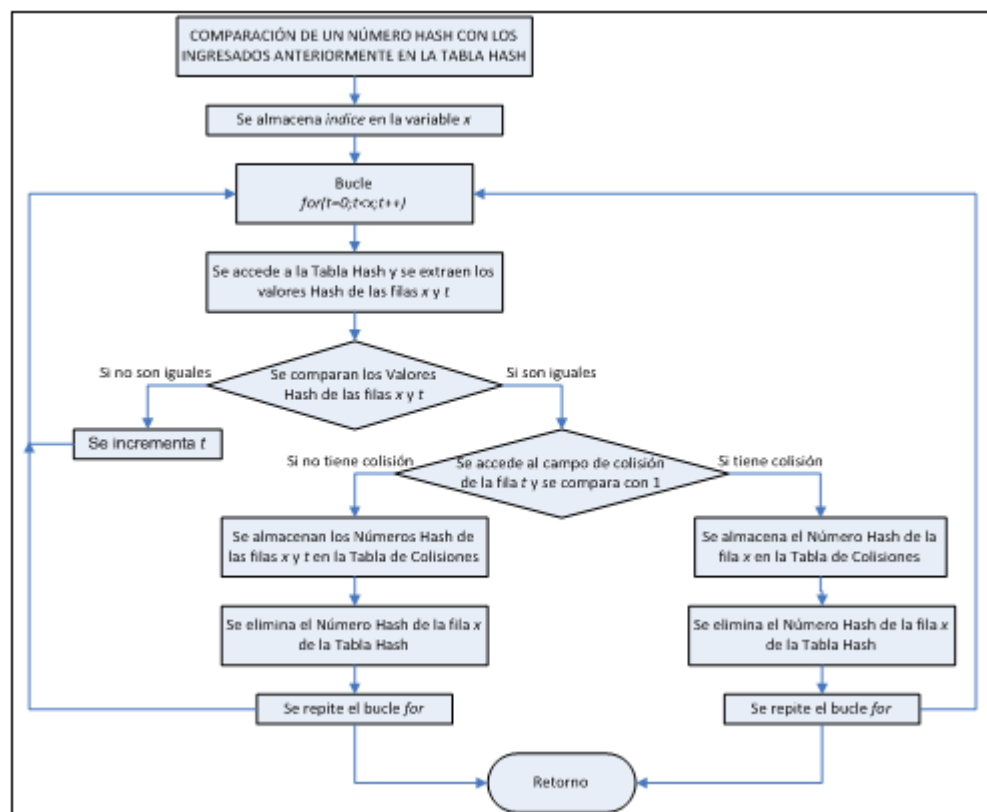


Figura 46. Comparación de un Número Hash con los ingresados anteriormente en la Tabla Hash

```

else if (a == 2) {
    x = indice;
    for (t=0; t<x; t++) {
        cmp1 = htabla[t][1];
        cmp2 = htabla[x][1];
        if (cmp1 == cmp2) {
            bit = htabla[t][3];
            if (bit == 1) { // Si ese número ya tiene colisión solo se guarda el
                           nuevo hash colisionado

```

```

        colisiones.numhash1_ = htabla [x] [1]; // Se guarda el
        numero hash en la estructura de colisiones
        colisiones.origen1_ = htabla [x] [4]; // Se guarda el Origen
        en la estructura de colisiones
        colisiones.destino1_ = htabla [x] [5]; // Se guarda el Destino
        en la estructura de colisiones
        colisiones.BW1_ = htabla [x] [2]; // BW = 0
        resolver(colisiones, 0);
        htabla [x] [1] = 0;
        htabla [x] [2] = 0;
        htabla [x] [3] = 0;
        htabla [x] [4] = 0;
        htabla [x] [5] = 0; //Borra el numero hash
        t = 1500; //Para que se salga del For
    } else if (bit == 0) {
        colisiones.numhash1_ = htabla [t] [1];
        colisiones.origen1_ = htabla [t] [4];
        colisiones.destino1_ = htabla [t] [5];
        colisiones.BW1_ = htabla [t] [2];
        htabla [t] [3] = 1; //Bandera de colisión en 1
        resolver(colisiones, 0);
        colisiones.numhash1_ = htabla [x] [1];
        colisiones.origen1_ = htabla [x] [4];
        colisiones.destino1_ = htabla [x] [5];
        colisiones.BW1_ = htabla [x] [2];
        resolver(colisiones, 0);
        htabla [x] [1] = 0;
        htabla [x] [2] = 0;
        htabla [x] [3] = 0;
        htabla [x] [4] = 0;
        htabla [x] [5] = 0; //Borra el numero hash
        t = 1500; //Para que se salga del For
    }
}
}

return 0;
}

```

Dentro del procedimiento *registrar()* se invoca al método *resolver()*, el cuál es utilizado para acceder a otra matriz llamada *tabla_col*. En esta matriz se ingresan los Números Hash que tienen colisión y sus respectivos datos con los cuales se diferencian. Este método utiliza una estructura llamada *colisiones*, a la cual se hace referencia en varias ocasiones en el método *registrar()* y que tiene como tarea organizar los valores de los datos de entrada al método *resolver()* antes de ingresar a éste.

✓ Gestión de la Tabla de Colisiones en *IntServ*

Como se mencionó anteriormente, el método *resolver()* es utilizado para acceder a la tabla de colisiones y dependiendo de una variable de entrada (*k*), se realiza un procedimiento diferente. En el diagrama de la Figura 47 se detallan las diferentes opciones que realiza este método.

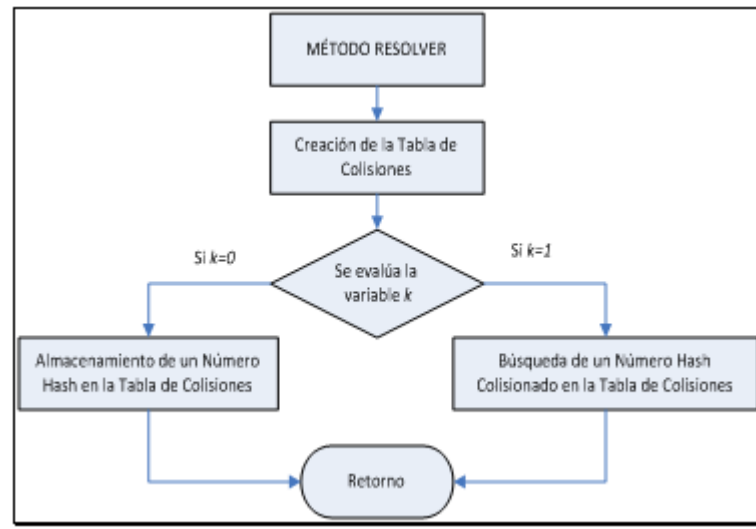


Figura 47. Diagrama de Flujo del método `resolver()`.

La primera tarea de este método es crear una matriz *tablacol* con 4 campos donde se almacenaran los Números Hash colisionados con sus respectivas direcciones IP de origen y destino, y además el valor resultante con el cual se clasificará el paquete de dicho flujo. Al finalizar la creación de esta Tabla de Resolución de Colisiones, se compara el valor entrante de *k* y se decide el proceso a seguir.

Existen 2 opciones de proceso, si $k=0$, se llena la matriz *tablacol* con los datos almacenados en una estructura llenada anteriormente llamada *colisiones*. Y en caso de que $k=1$, se realiza una búsqueda de un Número Hash Colisionado que viene almacenado en la estructura *colisiones*. Para un mejor entendimiento se presentan 2 figuras más (Figuras 48 y 49), donde se detallan a profundidad las anteriores opciones de procesos.

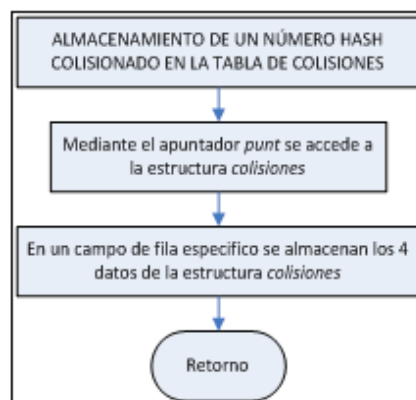


Figura 48. Almacenamiento de un Número Hash Colisionado en la Tabla de Colisiones

En el anterior diagrama se plantea un almacenamiento de datos en una matriz (*tablacol*) mediante la utilización de un apuntador (*punt*) a una

estructura (*colisiones*). De esta estructura se extraen los siguientes datos: Valor hash colisionado, Dirección IP Origen, Dirección IP destino, y el valor de la reserva que es necesario para clasificar el paquete. Cabe resaltar que cuando se encuentra una colisión, se guardan los dos Valores Colisionados con sus respectivos datos, y en caso de existir una triple colisión o más, sólo se almacena el último Número Hash que no ha sido almacenado en la Tabla de Resolución de Colisiones. El código utilizado para esta opción se encuentra a continuación:

```

if (k == 0) {
    f = f + 1;
    tablacol[f][1] = punt.numhash1_;
    tablacol[f][2] = punt.origen1_;
    tablacol[f][3] = punt.destino1_;
    tablacol[f][4] = punt.BW1_;
    return 0;
}

```

El segundo caso (Ver Figura 49) ($k=1$), se refiere a la búsqueda de un número hash colisionado en la matriz *tablacol*. Inicialmente en este proceso se fija el valor de la variable *maxf* que hace referencia al valor máximo de colisiones que tiene la matriz. Posteriormente se realiza un ciclo *for* con la variable *maxf* como límite para ejecutar un cuerpo de instrucciones que se detallan a continuación en el siguiente diagrama:

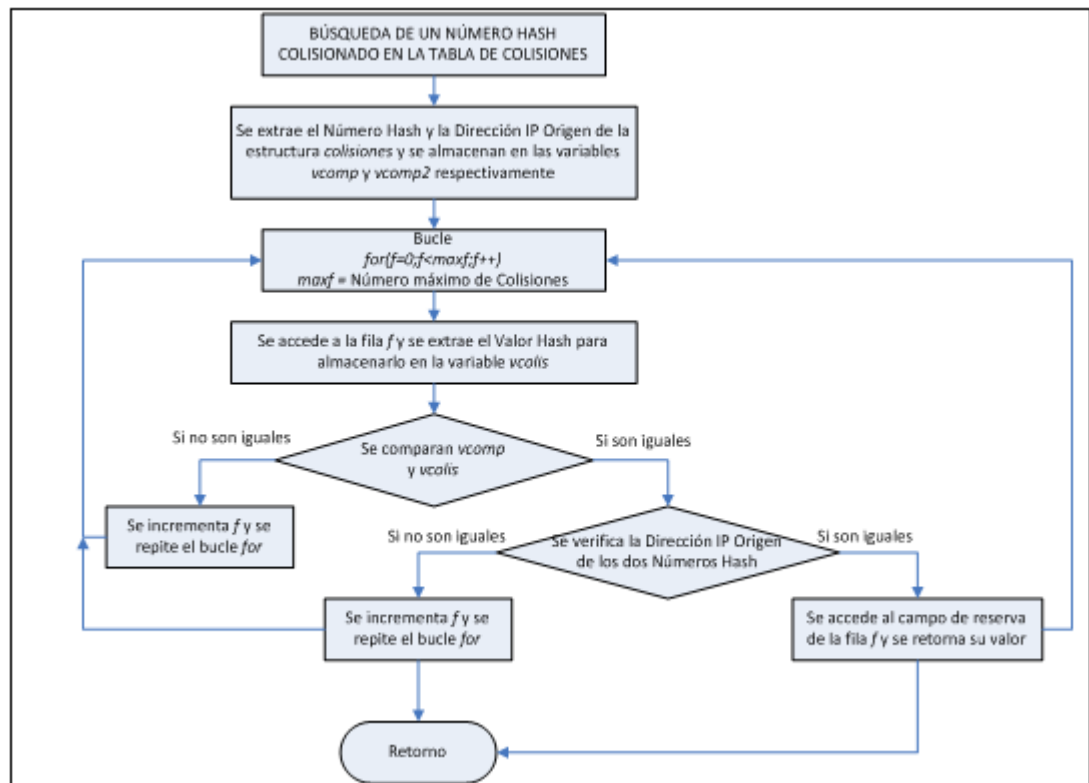


Figura 49. Búsqueda de un Número Hash Colisionado en la Tabla de Colisiones

La idea de esta opción es comparar cada Número Hash de la Tabla de Colisiones con el valor a buscar, de modo que cuando se encuentre alguna coincidencia se compare nuevamente cada una de las direcciones IP Origen para verificar que sea el flujo correcto. En caso de que coincidan las direcciones IP de los Números Hash, se retorna una variable de reserva con la cual se procede a clasificar el paquete. Si no coinciden las Direcciones IP se sigue buscando otro Número Hash Colisionado con su respectiva Dirección IP Origen.

En esta etapa del programa solo se compara la Dirección IP Origen que representa la Quintupla completa, debido a que solo va existir un flujo por cada Dirección IP Origen, lo que ahorra tiempo y optimiza el código mejorando el desempeño de la simulación.

A continuación se detalla el código en C++ de esta última opción del método *resolver()*.

```

else if (k == 1) {
    maxf = 4;
    for (f=1;f<maxf;f++) {
        vcomp = punt.numhash1_;
        vcomp2 = punt.origen1_;
        vcolis = tablacol [f] [1];
        if (vcomp == vcolis) {
            vquint = tablacol [f] [2];
            if (vcomp2 == vquint) {
                reserva = tablacol [f] [4];
                f = 1300;
            }
        }
    }
    return (reserva);
}

```

B. CAMBIOS REALIZADOS AL SIMULADOR NS-2

A continuación se explica cuales fueron los cambios realizados en el clasificador hash creando como resultado el clasificador *IntServ*.

Primero que todo se crean los nuevos archivos llamados *classifier-IntServ.h* y *classifier-IntServ.cc*, ubicados en las rutas especificadas por la Tabla 9 y se agregan a la carpeta *Makefile.in*, como se definió anteriormente. Luego se prosigue a adicionar las nuevas líneas de programación.

El primer aspecto que se debe tener en cuenta para hacer las modificaciones de los archivos es la Jerarquía de Clases de NS-2 para verificar que métodos de la Clase madre *Classifier* son utilizados en la Clase *HashClassifier* y que por lo tanto pueden ser reutilizados con sus respectivos nombres en el nuevo Clasificador

IntServ. De esta forma no es necesario crear nuevos métodos para el nuevo Clasificador, si ya están diseñados y se pueden heredar. Por lo tanto el paso siguiente es detallar los métodos del Clasificador Hash que pueden ser reutilizados.

En la Tabla 10 se pueden observar los métodos de la clase *HashClassifier* que se heredan de la clase *Classifier* y además los nombres de los métodos nuevos [71].

Tipo de Valor de Retorno	Métodos del Hash-Classifier	Heredados de la Clase Classifier	Nuevo nombre del método	Público/Protegido
virtual int	classify	Si	-----	Público
virtual int	lookup	No	busqueda	Público
virtual int	unknown	No	desconocido	Público
void	set_default	No	set_default_1	Público
int	do_set_hash	No	do_set_hash_1	Público
void	set_table_size	Si	-----	Público
int	maxslot	Si	-----	Público
NSObject*	slot	Si	-----	Público
int	mshift	Si	-----	Público
void	set_default_target	Si	-----	Público
virtual void	recv	Si	-----	Público
virtual NSObject*	find	Si	-----	Público
virtual void	clear	Si	-----	Público
virtual void	do_install	Si	-----	Público
int	install_next	Si	-----	Público
virtual void	install	Si	-----	Público
virtual void	recvOnly	Si	-----	Público
virtual void	delay_bind_init_all	Si	-----	Público
virtual int	delay_bind_dispatch	Si	-----	Público
int	isdebug	Si	-----	Público
virtual void	debug	Si	-----	Público
int	lookup	No	busqueda	Protegido
int	newflow	No	nuevoflujo	Protegido
void	reset	Si	-----	Protegido
virtual const char*	hashkey	No	hashkey_1	Protegido
int	set_hash	No	set_hash_1	Protegido
int	get_hash	No	get_hash_1	Protegido
virtual int	command	Si	-----	Protegido
virtual int	getnxt	Si	-----	Protegido
void	alloc	Si	-----	Protegido
void	handle	Si	-----	Protegido

Tabla 10. Métodos de la Clase HashClassifier que son heredados de la Clase Classifier.

Los métodos que no son reutilizables fueron reemplazados por otros nuevos dando a conocer los nombres de ellos. Esto se hace con el fin de hacer más comprensible los nuevos programas creados y comprender mejor la ubicación de

los métodos en el momento de querer desarrollar un nuevo protocolo en el simulador.

El Clasificador Hash que tiene el simulador, presenta también algunos atributos protegidos que se pueden reutilizar en el diseño del Clasificador *IntServ* y otros que son propios de la clase *HashClassifier*. A continuación se presenta una tabla con los atributos de la clase *HashClassifier* que se heredan de la clase *Classifier*.

Tipo de Valor de Retorno	Atributos Protegidos de la clase HashClassifier	Heredados de la clase Classifier	Nuevo nombre del atributo
Int	default_	No	default1_
Tcl_HashTable	ht_	No	ht1_
Hkey	buf_	No	buf1_
Int	keylen_	No	keylen1_
NsObject**	slot_	Si	-----
Int	nslot_	Si	-----
Int	maxslot_	Si	-----
Int	offset_	Si	-----
Int	shift_	Si	-----
Int	mask_	Si	-----
NsObject*	default_target_	Si	-----
Int	nsize_	Si	-----
Int	debug_	Si	-----

Tabla 11. Atributos de la clase *HashClassifier* que son heredados de la clase *Classifier*.

Guiándonos de la tabla 9, la cual muestra los archivos del simulador de *NS-2* que necesitan ser modificados para desarrollar nuestro protocolo, el primer archivo que es fundamental cambiar es *ip.h*, el cual configura la cabecera IP de cada paquete y por lo tanto la Quintupla completa. A continuación se muestra el código y la explicación de cada una de las líneas incluyendo las nuevas líneas de código necesarias para el nuevo Clasificador *IntServ*.

Las líneas resaltadas en negrita son las líneas que se adicionaron para completar la quintupla:

IP.H

```
/* Programa de la configuración de la cabecera IP para cada uno de los paquetes */
#ifndef ns_ip_h
#define ns_ip_h
```

```
/*Estas líneas incluyen archivos para su perfecto funcionamiento, ya que en estos se declaran parámetros que se necesitan en este código.*/
```

```
#include "config.h"
#include "packet.h"
```

```
/*Definición de la longitud de la cabecera IP y el tiempo de vida de cada paquete*/
```

```
#define IP_HDR_LEN    40 /*Ipv6*/
#define IP_DEF_TTL    32
```

*/*La siguiente definición se hace con el fin de evitar mensajes de advertencias en el sistema por líneas en donde se encuentra definido el IP_BROADCAST.*/*

```
#ifdef IP_BROADCAST
#undef IP_BROADCAST
#endif
```

*/*Definición del IP_BROADCAST en 32 bits.*
Static const u_int32_t IP_BROADCAST = ((u_int32_t) 0xffffffff);

*/*Estructura de la Cabecera IP*/*

```
struct hdr_ip {
    ns_addr_t      src_;
    ns_addr_t      dst_;
    int            ttl_;
```

/*Declaración de la variable que hace referencia a la identificación del protocolo*/

```
int            idprot_;
```

*/*Por ultimo estas dos variables de identificación de flujo y prioridad que son características propias de el protocolo de Internet versión 6*/*

```
int            fid_;    /* flow id */
int            prio_;
```

*/*Declaración de variable nflu_, para luego declarar el método nflu(), que retorna la variable declarada anteriormente*/*

```
static int offset_;
inline static int& offset() { return offset_; }
```

*/*Declaración del método para acceder al paquete y extraer su cabecera IP*/*

```
inline static hdr_ip* access(const Packet* p) {
    return (hdr_ip*) p->access(offset_); }
```

*/*Declaración de la funciones de acceso a los miembros por campo*/*

```
ns_addr_t& src() { return (src_); }
nsaddr_t& saddr() { return (src_.addr_); }
int32_t& sport() { return src_.port_ ; }
```

```
ns_addr_t& dst() { return (dst_); }
nsaddr_t& daddr() { return (dst_.addr_); }
int32_t& dport() { return dst_.port_ ; }
int& ttl() { return (ttl_); }
```

```
/* ipv6 fields */
int& flowid() { return (fid_); }
int& prio() { return (prio_); }
int& protid() { return (idprot_); }    // Identificación del Protocolo
```

```
};
#endif
```

En resumen en este fichero .h se añade un nuevo campo de Identificación de Protocolo necesario para completar la Quintupla y el método respectivo para acceder a este. Los 4 campos restantes (Dirección IP Origen, Dirección IP destino, Puerto Origen y Puerto Destino) de la Quintupla, son reutilizados de los existentes en esta cabecera IP. No fue necesario cambiar estos campos debido a que presentan las mismas características de la Arquitectura de Servicios Integrados (IntServ) con respecto al Protocolo de Internet Versión 4 (Ipv4).

El simulador NS-2.31 trae por defecto esta cabecera IP mencionada anteriormente, la cual presenta dos estructuras de tipo *ns_addr_t* llamadas *src_* (origen) y *dst_* (destino), cada una con 2 campos (de 32 bits y 16 bits respectivamente) llamados *addr_* (dirección) y *port_* (puerto). De esta forma quedan definidas las cinco variables de la Quintupla para el posterior cálculo del Número Hash. Inicialmente esta estructura tenía los dos campos de 32 bits, pero teniendo en cuenta que el puerto es de 16 bits se cambió el tipo de la variable de *int32_t* a *int16_t* de la variable *port_* de cada estructura *src_* y *dst_*.

El realizar estas modificaciones en el archivo *ip.h*, implica cambios en el archivo *ip.cc* que se encuentra en la misma carpeta *common* del simulador *ns-allinone-2.31/ns-2.31*. Para ver las rutas de acceso de estos archivos remitirse a la Tabla N° 9. A continuación se muestra el código de este fichero, resaltando las líneas adicionadas en negrita:

IP.CC

```
#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /cvsroot/nsnam/ns-2/common/ip.cc,v 1.8 1998/08/12 23:41:05 gnguyen Exp $";
#endif

#include "packet.h"
#include "ip.h"

int hdr_ip::offset_;

static class IPHeaderClass : public PacketHeaderClass {
public:
    IPHeaderClass() : PacketHeaderClass("PacketHeader/IP",
                                         sizeof(hdr_ip)) {
        bind_offset(&hdr_ip::offset_);
    }
    void export_offsets() {
        field_offset("src_", OFFSET(hdr_ip, src_));
        field_offset("dst_", OFFSET(hdr_ip, dst_));
        field_offset("ttl_", OFFSET(hdr_ip, ttl_));
        field_offset("fid_", OFFSET(hdr_ip, fid_));
        field_offset("prio_", OFFSET(hdr_ip, prio_));
        field_offset("idprot_", OFFSET(hdr_ip, idprot_));
    }
} class_iphdr;
```

De otra parte, la creación de las estructuras mencionadas en *ip.h* están definidas en el fichero *config.h*, el cual se describe a continuación:

CONFIG.H

```
#ifndef ns_config_h
#define ns_config_h

#define MEMDEBUG_SIMULATIONS

/* pick up standard types */
#include <sys/types.h>
#include STDC_HEADERS
```

```

#include <stdlib.h>
#include <stddef.h>
#endif
/* get autoconf magic */
#ifdef WIN32
#include "autoconf-win32.h"
#else
#include "autoconf.h"
#endif

/* after autoconf (and HAVE_INT64) we can pick up tclcl.h */
#ifndef stand_alone
#ifdef __cplusplus
#include <tclcl.h>
#endif /* __cplusplus */
#endif
/* handle stl and namespaces */
/* add u_char and u_int
 * Note: do NOT use these expecting them to be 8 and 32 bits long...
 * use {u_int{8,16,32}_t if you care about size. */

/* Removed typedef and included checks in the configure.in
typedef unsigned char u_char;
typedef unsigned int u_int;
*/
typedef int32_t nsaddr_t;
typedef int32_t nsmask_t;

/* 32-bit addressing support */
struct ns_addr_t { // Estructura existente de 32 bits(dirección, nflu)
    int32_t addr_;
    int16_t port_;
#ifdef __cplusplus
    bool isEqual(const ns_addr_t &o) {
        return ((addr_ == o.addr_) && (port_ == o.port_)) ? true : false;
    }
#endif /* __cplusplus */
};

// ESTRUCTURA PARA CREAR TABLA HASH
struct puntstruct {
    int    numhash_;           // Número Hash
    int    BW_;               // Ancho de Banda
    int    colision_;         // Colisión
    int    origen_;           // Dirección IP Origen
    int    destino_;          // Dirección IP Destino
};

ESTRUCTURA PARA CREAR LA TABLA DE COLISIONES

struct rescold {
    int    numhash1_;         // Número Hash Colisionado
    int    origen1_;          // Dirección IP Origen de la colisión
    int    destino1_;         // Dirección IP Destino de la colisión
    int    BW1_;              // Ancho de Banda de la colisión
};

/* 64-bit integer support */
#ifndef STRTOI64
#ifdef defined(SIZEOF_LONG) && SIZEOF_LONG >= 8
#define STRTOI64 strtol
#define STRTOI64_FMTSTR "%ld"
/*#define STRTOI64(S) strtol((S), NULL, 0) */
#else
#define STRTOI64 strtoll
#define STRTOI64_FMTSTR "%lld"
/*#define STRTOI64(S) strtoll((S), NULL, 0) */
#endif
#endif

```

```

#elif defined(HAVE_STRTOLL)
#define STRTOI64 strtoll
#define STRTOI64_FMTSTR "%lld"
/* #define STRTOI64(S) strtoll((S), NULL, 0) */
#endif
#endif

#define NS_ALIGN (8) /* byte alignment for structs (eg packet.cc) */
/* some global definitions */
#define TINY_LEN 8
#define SMALL_LEN 32
#define MID_LEN 256
#define BIG_LEN 4096
#define HUGE_LEN 65536
#define TRUE 1
#define FALSE 0

/*
 * get definitions of bcopy and/or memcpy
 * Different systems put them in string.h or strings.h, so get both
 * (with autoconf help).
 */
#ifdef HAVE_STRING_H
#include <string.h>
#endif /* HAVE_STRING_H */
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif /* HAVE_STRINGS_H */

#ifdef HAVE_BZERO
#define bzero(dest,count) memset(dest,0,count)
#endif
#ifdef HAVE_BCOPY
#define bcopy(src,dest,size) memcpy(dest,src,size)
#endif

#include <stdlib.h>

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif /* HAVE_UNISTD_H */

#ifdef HAVE_TIME_H
#include <time.h>
#endif /* HAVE_TIME_H */

#ifdef HAVE_ARPA_INET_H
#include <arpa/inet.h>
#endif /* HAVE_ARPA_INET_H */

#if (defined(__hpux) || defined(_AIX)) && defined(__cplusplus)
/* these definitions are perhaps vestigial */
extern "C" {
int strcasecmp(const char *, const char *);
clock_t clock(void);
#if !defined(__hpux)
int gethostid(void);
#endif
time_t time(time_t *);
char *ctime(const time_t *);
}
#endif

#if defined(NEED_SUNOS_PROTOS) && defined(__cplusplus)
extern "C" {
struct timeval;
struct timezone;

```

```

int gettimeofday(struct timeval*, ...);
int ioctl(int fd, int request, ...);
int close(int);
int strcasecmp(const char*, const char*);
int srandom(int); /* (int) for sunos, (unsigned) for solaris */
int random();
int socket(int, int, int);
int setsockopt(int s, int level, int optname, void* optval, int optlen);
struct sockaddr;
int connect(int s, struct sockaddr*, int);
int bind(int s, struct sockaddr*, int);
struct msghdr;
int send(int s, void*, int len, int flags);
int sendmsg(int, struct msghdr*, int);
int recv(int, void*, int len, int flags);
int recvfrom(int, void*, int len, int flags, struct sockaddr*, int);
int gethostid();
int getpid();
int gethostname(char*, int);
void abort();
}
#endif

#if defined(NEED_SUNOS_PROTOS) || defined(solaris)
extern "C" {
#if defined(NEED_SUNOS_PROTOS)
    caddr_t sbrk(int incr);
#endif
    int getrusage(int who, struct rusage* rusage);
}
#endif

#ifdef WIN32

#include <windows.h>
#include <winsock.h>
#include <time.h> /* For clock_t */

#include <minmax.h>
#define NOMINMAX
#undef min
#undef max
#undef abs

#define MAXHOSTNAMELEN 256

#define _SYS_NMLN 9
struct utsname {
    char sysname[_SYS_NMLN];
    char nodename[_SYS_NMLN];
    char release[_SYS_NMLN];
    char version[_SYS_NMLN];
    char machine[_SYS_NMLN];
};
typedef char *caddr_t;

struct iovec {
    caddr_t iov_base;
    int iov_len;
};

#endif

#ifndef TIMEZONE_DEFINED_
#define TIMEZONE_DEFINED_
struct timezone {
    int tz_minuteswest;
    int tz_dsttime;
};

```

```

#endif

typedef int pid_t;
typedef int uid_t;
typedef int gid_t;

#if defined(__cplusplus)
extern "C" {
#endif

int uname(struct utsname *);
int getopt(int, char * const *, const char *);
int strcasecmp(const char *, const char *);
/* these shouldn't be used/needed, even on windows */
/* #define srandom srand */
/* #define random rand */
int gettimeofday(struct timeval *p, struct timezone *z);
int gethostid(void);
int getuid(void);
int getgid(void);
int getpid(void);
int nice(int);
int sendmsg(int, struct msghdr*, int);
/* Why this is here, inside a #ifdef WIN32 ??
#ifdef WIN32
    time_t time(time_t *);
#endif*/
#define strcasecmp _stricmp
#if defined(__cplusplus)
}
#endif
#ifdef WSAECONNREFUSED
#define ECONNREFUSED WSAECONNREFUSED
#define ENETUNREACH WSAENETUNREACH
#define EHOSTUNREACH WSAEHOSTUNREACH
#define EWOULDBLOCK WSAEWOULDBLOCK
#endif /* WSAECONNREFUSED */

#ifdef M_PI
#define M_PI 3.14159265358979323846
#endif /* M_PI */

#endif /* WIN32 */

#ifdef sgi
#include <math.h>
#endif

/* Declare our implementation of snprintf() so that ns etc. can use it. */
#ifdef HAVE_SNPRINTF
#if defined(__cplusplus)
extern "C" {
#endif
    extern int snprintf(char *buf, int size, const char *fmt, ...);
#if defined(__cplusplus)
}
#endif
#endif

/***** These values are no longer required to be hardcoded – mask and shift values are available from Class Address. *****/

/* While changing these ensure that values are consistent with tc/lib/ns-default.tcl */
/* #define NODEMASK 0xfffff */
/* #define NODESHIFT 8 */
/* #define PORTMASK 0xff */

#endif

```

Este código de *config.h* incluye todas las configuraciones de parámetros fundamentales para el funcionamiento interno del simulador. Sólo fue necesario adicionar 2 estructuras para el manejo de algunas variables en el nuevo clasificador: La primera estructura de tipo *puntstruct* (*campos*), para crear una matriz con el Número Hash, colisiones si existen, anchos de banda y por último la Dirección IP Origen y Dirección IP destino, que se utilizan para comparar en caso de que haya colisión, y la segunda estructura de tipo *rescol* (*colisiones*), que se utiliza para crear una matriz con los Número Hash Colisionados con sus respectivos anchos de banda y las direcciones IP de origen y destino. Cabe resaltar que los campos de *origen_* y *destino_* en la primera estructura se utilizan en casos de colisión con el fin de compararlos con los mismos campos de *origen1_* y *destino1_* de los Números Hash que se encuentran en la Tabla de Resolución de Colisiones. Sólo se utilizan estas dos variables como Quintupla debido a que en el diseño se declararon las variables de los puertos (origen y destino) y la Identificación de Protocolo iguales para todos los flujos, por lo tanto no es necesario que estas variables sean guardadas en las matrices declaradas anteriormente. La explicación completa del funcionamiento de cada una de estas estructuras se encuentra detallada en el código del *Classifier-IntServ.h* y *Classifier-IntServ.cc* (Ver anexo A).

Uno de los pasos más importantes en la creación de un nuevo objeto en el simulador, es la adición de éste al archivo *makefile.in*, para que el compilador lo tenga en cuenta y se puedan corregir errores en caso de que los halla. El proceso es bastante sencillo; fácilmente se busca el archivo *makefile.in* con la ruta especificada en la tabla 9 y se añade el Clasificador *IntServ* en la lista de objetos como se muestra en el pequeño código a continuación. La línea resaltada en negrita es la adicionada al programa:

```
OBJ_CC = \
tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
common/scheduler.o common/object.o common/packet.o \
common/ip.o routing/route.o common/connector.o common/ttl.o \
trace/trace.o trace/trace-ip.o \
classifier/classifier.o classifier/classifier-addr.o \
classifier/classifier-hash.o \
classifier/classifier-IntServ.o \
classifier/classifier-virtual.o \
classifier/classifier-mcast.o \
classifier/classifier-bst.o \
classifier/classifier-mpath.o mcast/replicator.o \
classifier/classifier-mac.o \
classifier/classifier-qs.o \
classifier/classifier-port.o src_rtg/classifier-sr.o \
```

Al finalizar este proceso se abre un terminal, ubicándose en la carpeta *ns-allinone-2.31/ns-2.31* y ejecutando el comando *./configure*. De esta forma queda incluido el nuevo objeto para ser compilado. El código completo del archivo *makefile.in* se encuentra en el Anexo A de este documento.

Teniendo en cuenta la creación del anterior módulo de C++, se crean los nuevos archivos *classifier-IntServ.h* y *classifier-IntServ.cc*, incluyendo en éstos las mismas líneas de código descritas anteriormente para el Clasificador y añadiendo algunos métodos nuevos necesarios para el acoplamiento de los dos lenguajes que utiliza el simulador NS-2 (C++ y Tcl). A continuación se presenta la descripción detallada del código de cada uno de estos archivos:

CLASSIFIER-INTSERV.H

// Programa Clasificador Hash con la Quintupla (IntServ)

```
#include "classifier.h"
#include "ip.h"
#include "config.h"
#include "packet.h"
#include <fstream.h>
```

// Herramientas utilizadas para almacenar e imprimir archivos con variables calculadas en el proceso

```
ofstream nflujo("PruebaHash.ods"); // constructor de ofstream
ofstream archivo1("PruebaHash2.ods"); // constructor de ofstream
ofstream archivo2("Contador.ods"); // constructor de ofstream
ofstream archivo3("Colision.ods"); // constructor de ofstream
```

*//******

//CONSTRUCTOR

```
class IntServClassifier : public Classifier {
public:
IntServClassifier() : flujo_(-1), destino_(-1) {
bind("flujo_", &flujo_);
bind("destino_", &destino_);
// Creación de variables necesarios para el enlace entre C++ y Tcl
}
```

*//******

//DESTRUCTOR

```
~IntServClassifier() {
};
```

*//******

// CLASSIFY: Es un método puro virtual indicando que la clase Classifier es solo usada como una clase base. Su función es clasificar cada paquete entrante al router y regresar un valor. El cuerpo de este método se encuentra en el archivo classifier-IntServ.cc.

```
protected:
int classify(Packet *p);
```

*//******

// BUSQUEDA: Accede a la cabecera IP del paquete (quintupla) y lo manda a buscar a la tabla hash.

```
Public:
int busqueda(Packet *p) {
    hdr_ip* h = hdr_ip::access(p);
    return get_hash_1(mshift(h->saddr()),mshift(h->sport()),mshift(h->daddr()),mshift(h->dport()),h->protid());
}
```

```
protected:
```

*//******

// SET-HASH-1: Método para calcular número hash y guardarlos en la tabla hash, para luego verificar si este número hash presenta alguna colisión para almacenarlo en la tabla de resolución de colisiones con su respectiva quintupla y reserva.

```
Int set_hash_1(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {
```

```

    int valor;
    int indice;
    int col;
    indice = 0;
    f = 0;
    cont = 0;
    col = destino_;

    while (src<flujo_) {
        valor = hashkey_1(src,sport,dst,dport,pid);
        // Comentarios utilizados durante las pruebas del programa.
        //printf("Origen: %d \n",src);
        //printf("Destino: %d \n",dst);
        //printf("Hash: %d \n",valor);
        archivo << valor << "\n";
        campos.numhash_ = valor;
        campos.BW_ = col;
        campos.colision_ = 0;
        campos.origen_ = src;
        campos.destino_ = dst;
        registrar(campos, 0, indice); //Llena los datos en la tabla hash
        registrar(campos, 2, indice); //Compara si hay algún hash igual anteriormente
        ++src;
        ++dst;
        ++indice;
        ++col;
    }
    return (1); // Retorno de un número en específico para comprobar que realizó el método correctamente.
}

//*****
// GET-HASH: Hace la Búsqueda de un número hash en la tabla de reservas y retorna un valor.

Int get_hash_1(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

    int valorh;
    int vfid;

    //Comandos de líneas de impresión utilizadas como pruebas para comprobación de valores de variables
    en ciertos puntos del método.
    //printf("src: %d \n",src);
    //printf("dst: %d \n",dst);

    ++cont;
    //printf("Contador: %d \n",cont);
    valorh = hashkey_1(src,sport,dst,dport,pid);
    campos.numhash_ = valorh;
    //printf("Busqueda: %d \n",valorh);
    archivo1 << valorh << "\n";
    vfid = registrar(campos, 1, 0);

    return (vfid);
}

//*****
// COMMAND: Método principal del .cc

virtual int command(int argc, const char*const* argv);
//*****
// VARIABLES UTILIZADAS EN LA CLASE INTSERVCLASSIFIER:

int        flujo_;
int        destino_;
int        f;
int        cont;
int        hash_;
int        mask1_;

```

```

int            temp1_;
int            temp2_;
int            tabla[6];
puntstruct     campos;
rescol         colisiones;
hdr_ip         quintupla;
//*****
// REGISTRAR: Método de acceso a la tabla de BW y Colisiones.

Int registrar(puntstruct& tab, int a, int indice) {

    int         x;
    int         t;
    int         cmp1;
    int         cmp2;
    int         nhash;
    int         rtdocol;
    int         vbuscar;
    int         y;
    int         rtdoflow;
    int         bit;
    static int htabla [16] [5]; // [Filas] [Columnas]
    //a = 0 → Llenar tabla
    //a = 1 → Buscar Datos
    //a = 2 → Comparar datos

    if (a == 0) {
        x = indice;
        htabla [x] [1] = tab.numhash_;
        htabla [x] [2] = tab.BW_;
        htabla [x] [3] = tab.colision_;
        htabla [x] [4] = tab.origen_;
        htabla [x] [5] = tab.destino_;

    return (int) 1;
    }
    else if (a == 1) {
        x = indice;
        vbuscar = tab.numhash_;
        //printf("Vbuscar: %d \n",vbuscar);
        for (x=0;x<fluj0_;x++) {
            nhash = htabla [x] [1];
            //printf("nhash: %d \n",nhash);
            if (vbuscar == nhash) {
                //printf("Indice: %d \n",vbuscar);
                rtdocol = htabla [x] [3];
                if (rtdocol == 1) {
                    colisiones.numhash1_ = nhash;
                    colisiones.origen1_ = htabla [x] [4];
                    colisiones.destino1_ = htabla [x] [5];
                    y = resolver(colisiones, 1);
                    rtdoflow = y;
                    x = 1300; // Para que salga del For
                } else {
                    rtdoflow = htabla [x] [2];
                    //printf("rtdoflow: %d \n",rtdoflow);
                    x = 1300; // Para que salga del For
                }
            }
        }

    }
    return (int) rtdoflow;
}
else if (a == 2) {
    x = indice;
    for (t=0;t<x;t++) {
        cmp1 = htabla [t] [1];

```

```

        cmp2 = htabla [x] [1];
        if (cmp1 == cmp2) {
            bit = htabla [t] [3];
            if (bit == 1) { // Si ese numero ya tiene colision solo se guarda el nuevo hash
                colisionado
                colisiones.numhash1_ = htabla [x] [1]; // Se guarda el numero hash en la
                estructura de colisiones
                colisiones.origen1_ = htabla [x] [4]; // Se guarda el Origen en la estructura
                de colisiones
                colisiones.destino1_ = htabla [x] [5]; // Se guarda el Destino en la
                estructura de colisiones
                colisiones.BW1_ = htabla [x] [2];
                resolver(colisiones, 0);
                htabla [x] [1] = 0;
                htabla [x] [2] = 0;
                htabla [x] [3] = 0;
                htabla [x] [4] = 0;
                htabla [x] [5] = 0; //Borra el numero hash
                t = 1300; //Para que se salga del For
            }
            else if (bit == 0) {
                colisiones.numhash1_ = htabla [t] [1];
                colisiones.origen1_ = htabla [t] [4];
                colisiones.destino1_ = htabla [t] [5];
                colisiones.BW1_ = htabla [t] [2];
                htabla [t] [3] = 1; //Bandera de colision en 1
                y = resolver(colisiones, 0);
                colisiones.numhash1_ = htabla [x] [1];
                colisiones.origen1_ = htabla [x] [4];
                colisiones.destino1_ = htabla [x] [5];
                colisiones.BW1_ = htabla [x] [2];
                y = resolver(colisiones, 0);
                htabla [x] [1] = 0;
                htabla [x] [2] = 0;
                htabla [x] [3] = 0;
                htabla [x] [4] = 0;
                htabla [x] [5] = 0; //Borra el numero hash
                t = 1300; //Para que se salga del For
            }
        }
    }
    return 0;
}
return (rtdoflow);
}

//*****
// RESOLVER:

int resolver(rescol& punt, int k) {

    int vcomp;
    int vcomp2;
    int maxf;
    int vcolis;
    int vquint;
    int reserva;
    int hashcolision;
    static int tablacol [16] [4]; // [Filas] [Columnas]
    //k = 0 → Llenar tabla
    //k = 1 → Buscar Datos

    if (k == 0) {
        f = f + 1;
        //printf("contador: %d \n",f);
        tablacol [f] [1] = punt.numhash1_;
        hashcolision = tablacol [f] [1];
    }
}

```

```

        archivo3 << hashcolision << "\n";
        tablacol [f] [2] = punt.origen1_;
        tablacol [f] [3] = punt.destino1_;
        tablacol [f] [4] = punt.BW1_;

        return (int) 1;
    } else if (k == 1) {
        maxf = flujo_; // Valor máximo de filas en la tabla de colisiones
        for (f=0;f<maxf;f++) {
            vcomp = punt.numhash1_;
            vcomp2 = punt.origen1_;
            vcolis = tablacol [f] [1];
            if (vcomp == vcolis) {
                vquint = tablacol [f] [2];
                if (vcomp2 == vquint) {
                    nflujo = tablacol [f] [4];
                    f = 1300;
                }
            }
        }
        return (int) nflujo;
    }
}

return (reserva);
}

//*****
//HASHKEY: Método que se utiliza para calcular el numero hash con la quintupla.

Int hashkey_1(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

    int i;

    mask1_ = 0;
    temp1_ = 0;
    temp2_ = 0;
    hash_ = 0;

    //Cálculo hash
    mask1_ = 0xffff000; //Mask 1
    temp1_ = src;
    temp1_ &= mask1_;
    temp1_ >>= 3;
    tabla[0] = temp1_; //Variable 1
    mask1_ = 0x00000fff; //Mask 2
    temp1_ = src;
    temp1_ &= mask1_;
    temp1_ <<= 2;
    mask1_ = 0xff000000; //Mask 3
    temp2_ = dst;
    temp2_ &= mask1_;
    temp2_ >>= 6;
    temp1_ |= temp2_;
    tabla[1] = temp1_; //Variable 2
    mask1_ = 0x00ffff0; //Mask 4
    temp1_ = dst;
    temp1_ &= mask1_;
    temp1_ >>= 1;
    tabla[2] = temp1_; //Variable 3
    mask1_ = 0x0000000f; //Mask 5
    temp1_ = dst;
    temp1_ &= mask1_;
    temp1_ <<= 4;
    temp2_ = sport;
    temp1_ |= temp2_;
    tabla[3] = temp1_; //Variable 4
    temp1_ = dport;
    temp1_ <<= 1;

```

```

mask1_ = 0x00000f0; //Mask 6
temp2_ = pid;
temp2_ &= mask1_;
temp2_ >>= 1;
temp1_ |= temp2_;
tabla[4] = temp1_; //Variable 5
mask1_ = 0x000000f; //Mask 7
temp1_ = pid;
temp1_ &= mask1_;
tabla[5] = temp1_; //Variable 6

for (i=0;i<6;i++) {
    hash_ ^= tabla[i];
    //printf("hashkey: %d \n",hash_);
}

return (int) hash_;
}
};
//*****

```

CLASSIFIER-INTSERV.CC

```

#include <stdlib.h>
#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include "classifier-IntServ.h"

/***** IntServClassifier Methods *****/

int IntServClassifier::classify(Packet *p) {

    int nflujo = busqueda(p);
    //printf(" nflujo: %d \n", nflujo);
    archivo2 << nflujo << "\n";
    quintupla.fid_ = nflujo;
    return (1);
} // IntServClassifier::classify

int IntServClassifier::command(int argc, const char*const* argv)
{
    if (argc == 4) {
        /*$classifier add-num src dst */
        if (strcmp(argv[1], "add-num") == 0) {
            nsaddr_t src = atoi(argv[2]);
            int16_t sport = 0;
            nsaddr_t dst = atoi(argv[3]);
            int16_t dport = 0;
            int pid = 0;
            //printf("Dest: %d \n",dst);
            set_hash_1(src,sport,dst,dport,pid);
            //printf("Resultado: %d \n",n);
            return TCL_OK;
        }
    }
    return (Classifier::command(argc, argv));
}

/***** TCL linkage *****/
static class IntServClassifierClass : public TclClass {
public:
    IntServClassifierClass() : TclClass("Classifier/IntServ") {}
    TclObject* create(int, const char*const*) {
        return (new IntServClassifier());
    }
}

```

```

    }
} class_intserv_classifier;

```

3.5.2.2. DISEÑO DEL CLASIFICADOR INTSERV6

Para la creación del nuevo Clasificador *IntServ6* se sigue el mismo procedimiento utilizado para la creación del Clasificador *IntServ*. Para esto es necesario crear las nuevas clases y objetos teniendo en cuenta la Jerarquías de Clases del simulador *NS-2*.

En la Figura 50 se pueden observar las antiguas clases de clasificadores existentes en el simulador y las nuevas clases incluyendo el clasificador *IntServ*, anteriormente explicado, y el Clasificador *IntServ6*, con su respectivo módulo *HostOrigen6*.

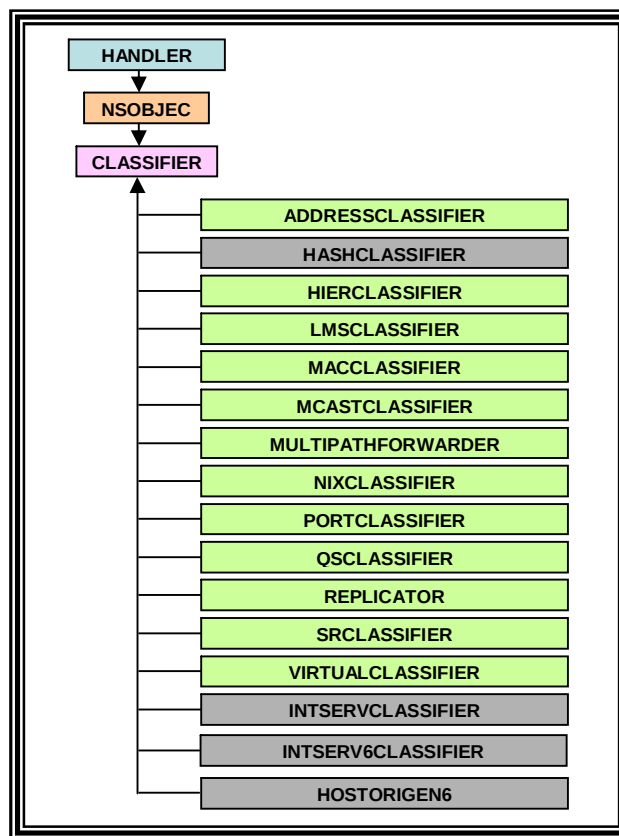


Figura 50. Clase *Classifier*

En la Figura 50 se muestran nuevamente los 13 tipos diferentes de Clasificadores del simulador *NS-2*, y los dos nuevos clasificadores recientemente creados para cumplir con los objetivos del presente proyecto. El nuevo Clasificador *IntServ6*

también hereda las características del Clasificador general del simulador, lo que representa la reutilización de algunos de sus métodos.

En la Figura 51 se presenta la ubicación de los nuevos archivos creados dentro del simulador (*IntServ* e *IntServ6*), el módulo del Cálculo del Número Hash (*HostOrigen*) y cuales son las clases más altas dentro de la Jerarquía interna de la herramienta de simulación.

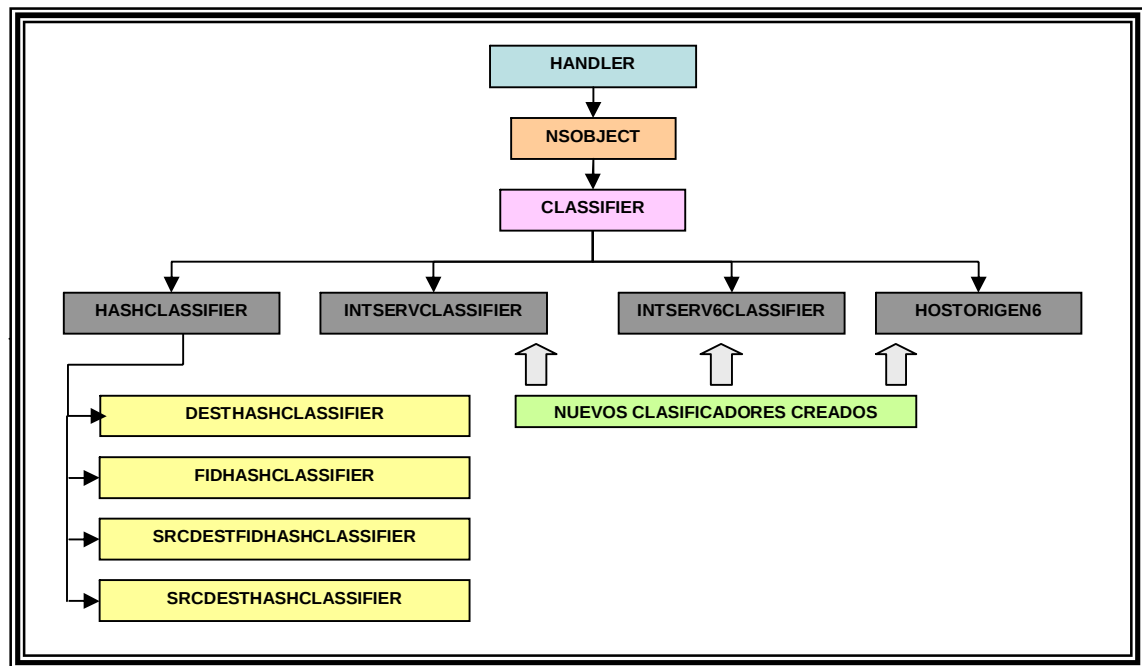


Figura 51. Las nuevas Clases IntServ6Classifier y HostOrigen6

Al culminar con el diseño y la construcción del Clasificador *IntServ*, se realizó un análisis de los métodos utilizados en este clasificador, y se definieron cuales de éstos podían ser reutilizados en la nueva propuesta *IntServ6*. Este nuevo clasificador se crea con el fin de mejorar y agilizar el comportamiento interno de un router, y así disminuir el tiempo del procesamiento de cada uno de los paquetes. La principal característica de esta propuesta es la utilización del nuevo campo de Etiqueta de Flujo del Protocolo de Internet Version 6 (*IPv6*) para almacenar un Número Hash, el cual es calculado por el Host Origen del flujo. Esto conlleva a un cambio radical del clasificador *IntServ*, ya que no se realizaría el cálculo del Número Hash en cada router, sino sólo en cada uno de los Host Origen.

En este nuevo clasificador se tiene en cuenta el tamaño de la dirección IP para *IPv6*, es decir, la dirección IP cambia de 32 bits a 128 bits, lo cual significa que el cálculo del Número Hash es un poco más extenso que el del Clasificador *IntServ*, pero en general presentan la misma estructura. Para facilitar el diseño de este clasificador se creó nuevamente un módulo de C++ independiente del simulador

para analizar el comportamiento del Clasificador *IntServ6* y realizar pruebas hasta verificar que su desempeño es el ideal. Para tener mayor claridad en el momento de diseñar este clasificador, se tomó en cuenta el Diagrama de Transición de estados del clasificador *IntServ6* [54] el cual se muestra en la figura 52.

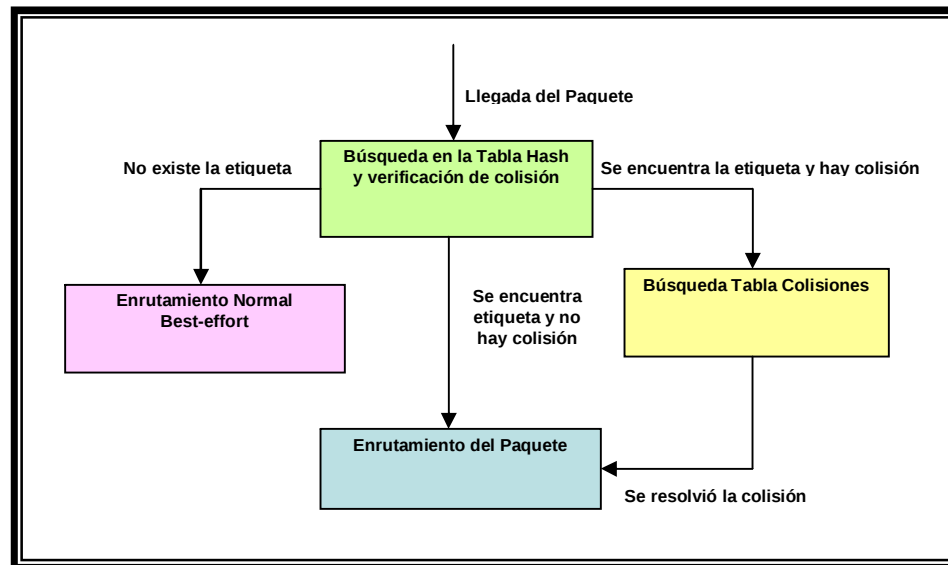


Figura 52. Diagrama de Transición de estados del Clasificador *IntServ6*.

Este diagrama presenta como primer paso la llegada del paquete al router con su Número Hash interno, para seguidamente realizar la búsqueda en la Tabla Hash y verificar si existe colisión. Esta búsqueda es de tipo *Tagging*, es decir, utiliza el Número Hash como índice para apuntar a la Tabla [54]. En caso de no existir la etiqueta en la Tabla Hash, se le da al paquete un enrutamiento normal *Best-Effort*, el cual se enfoca en una cola *FIFO* y enruta cada paquete cuando la red este disponible y tenga capacidad de hacerlo.

También existe la posibilidad de que el Número Hash que trae la Etiqueta de Flujo se encuentre en la Tabla de Reservas y no tenga ninguna colisión con algún otro flujo, lo cual representa que los paquetes de este flujo sean enrutados al Planificador *WFQ* para que les asigne los recursos pactados en el Control de Admisión. Y finalmente, en caso de que se encuentre el Numero Hash y exista colisión con otro flujo se hará una búsqueda en la Tabla de Colisiones para encontrar su respectiva reserva y será enrutado cada paquete de ese flujo.

En el diseño de una red *IntServ6* se requiere ejecutar un módulo llamado *HostOrigen6* ubicado en el Host Origen de cada flujo, el cual realiza el cálculo del Número Hash y lo almacena de nuevo en el campo de Etiqueta de Flujo del paquete, ahorrando así más tiempo en el Clasificador *IntServ6*. La descripción de este módulo se encuentra en la sección A del Diseño del Clasificador *IntServ6*.

En esta sección se crean 4 archivos para el nuevo clasificador (*classifier-IntServ6.h*, *classifier-IntServ6.cc*, *HostOrigen6.h* e *HostOrigen6.cc*), los cuales se encuentran ubicados en las rutas demarcadas por la Tabla N°9 en los ítems 5, 6, 7 y 8. Cabe resaltar que la descripción completa del código de cada uno de estos archivos se encuentra en la parte de los cambios realizados al simulador para el Clasificador *IntServ6* o en el Anexo A.

A. MÓDULO HOSTORIGEN

El objetivo de este módulo es mejorar el desempeño de una red *IntServ6*, ya que se ahorraría el cálculo del Número Hash en cada uno de los routers de la red, y se disminuiría el tiempo de clasificación de cada paquete.

El proceso de este módulo comienza con la generación del paquete por parte del generador de Tráfico acoplado al Host, de modo que inicialmente el paquete no tiene internamente el Número Hash. Al llegar al módulo *HostOrigen*, se toma el paquete y se accede a su cabecera IP, extrayendo cada uno de los valores de la Quintupla. Luego con estos valores se llama al método *hashkey_6()* para realizar el cálculo y obtener el nuevo Número Hash. Finalmente se accede nuevamente a la cabecera IP y en un nuevo campo creado para este fin, se almacena el Número Hash. La creación de este nuevo campo esta detallada más adelante en el código de *ip.h* e *ip.cc*.

Al llegar este paquete al Clasificador *IntServ6* se accede nuevamente a la cabecera IP, y mediante la función de acceso del nuevo campo, se extrae sólo el Número Hash para la clasificación y sus direcciones IP de Origen y Destino, utilizadas en caso de que el flujo presente colisión.

El cálculo del Número Hash de este módulo es el mismo que se realiza más adelante dentro del Clasificador *IntServ6* para el Control de Admisión de los flujos, por lo tanto, su explicación se encuentra detallada en la Figura 53.

B. MODULO DEL CLASIFICADOR INTSERV6 EN C++

En esta etapa del proyecto la idea es simular el comportamiento interno del clasificador *IntServ6* (con el compilador de C++ de *Linux*) y con respecto a él se realizan modificaciones y pruebas que puedan ayudar a mejorar el desempeño de programa. Además, como se había mencionado anteriormente, de esta manera el módulo tiene la ventaja de ser independiente del simulador y es bastante sencillo de crear y manejar, pues no se necesita compilar todos los archivos del simulador *NS-2* para añadir los cambios. La simulación se crea en un archivo de texto con extensión *.cpp* o *.cc*. Para compilar este archivo sencillamente se abre un terminal en *Linux* y se accede al fichero en donde se encuentra el programa. Estando ubicado en la ruta correcta se escribe el siguiente comando para crear su ejecutable:

g++ -o nombre_ejecutable nombre_programa.cpp

De esta forma se creará el ejecutable en el mismo lugar donde se encuentra el programa y al correr el programa se crean los archivos resultantes donde están los valores almacenados durante la ejecución del código.

El primer proceso que se realiza antes de empezar la transmisión de paquetes en una red *IntServ6* es el Control de Admisión. El objetivo de este Control es establecer las reservas de los flujos antes de que inicie el tráfico de paquetes en la red, es decir, se llena la Tabla Hash y la Tabla de Colisiones con todos los Números Hash que identifican a cada flujo, con sus respectivas Direcciones IP y reservas. En la Figura 53 se hace un resumen detallado de cada uno de los pasos que realiza el Control de Admisión para organizar estas reservas antes de iniciar el envío de paquetes desde los Host origen.

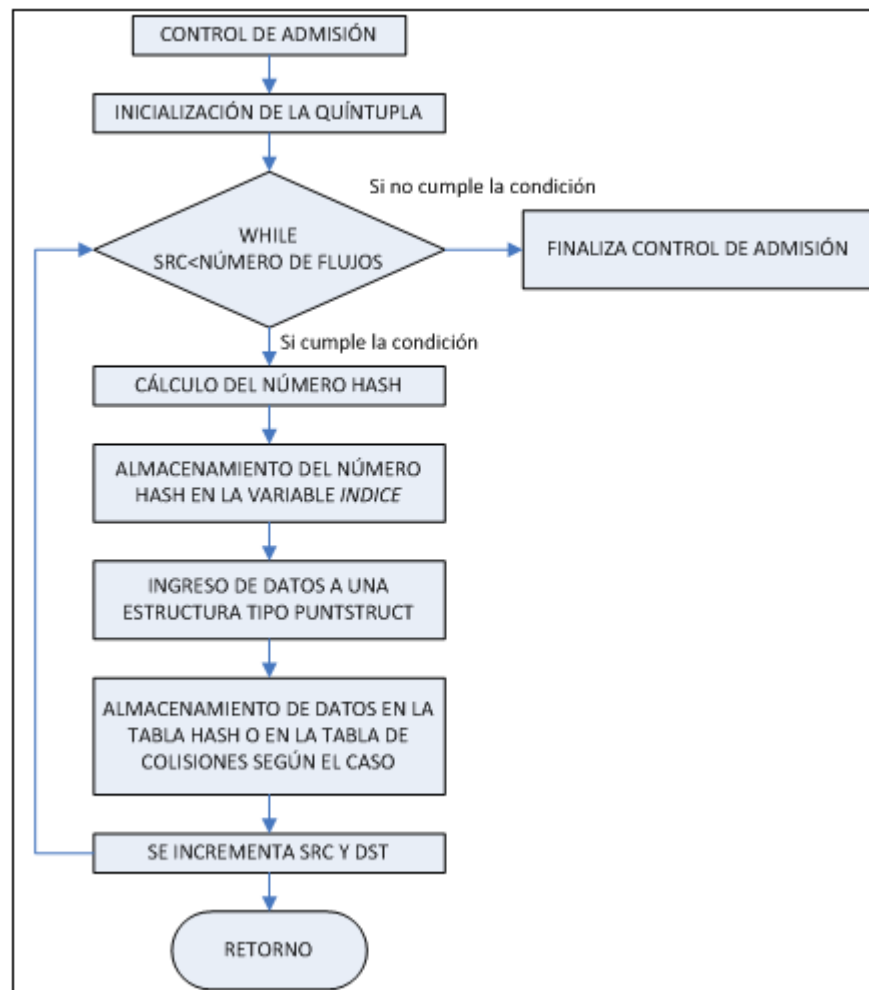


Figura 53. Control de Admisión del Clasificador IntServ6.

El primer paso del Control de Admisión es la inicialización de la Quíntupla. Para esto es indispensable fijar el tipo de topología que se va a utilizar y analizar de qué forma varían los valores de la Quíntupla para el Cálculo del Número Hash. En nuestro caso para *IntServ6* se utilizó el mismo tipo de topología que *IntServ*, esto con el fin de reutilizar el algoritmo de Control de Admisión en C++ y poder comparar el comportamiento en cada uno de los clasificadores diseñados en este proyecto (Ver Figura 54).

En la Figura 54, los nodos *n0* a *n3* representan los computadores Origen, en los cuales se realiza el cálculo del Número Hash y el ingreso de éste en la Cabecera IP de cada paquete. Los nodos *n6* a *n9* representan los computadores Destino a los cuales llega cada uno de los flujos (*f0*, *f1*, *f2* y *f3*). En el nodo *n4* queda ubicado el Clasificador *IntServ6* y por último el nodo *n5* representa un Router normal del simulador *NS-2*.

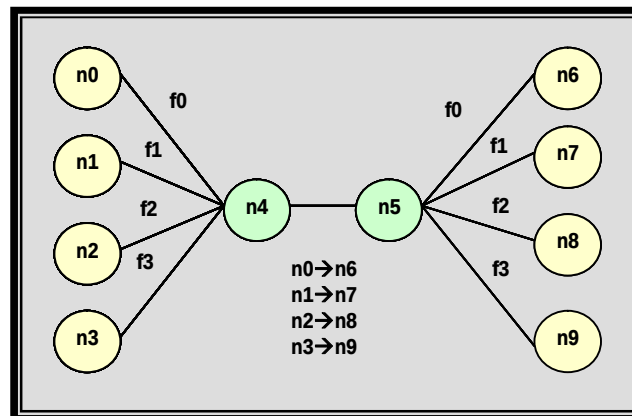


Figura 54. Topología Ejemplo para *IntServ6*

El proceso de inicialización de la Quíntupla anteriormente mencionado, se basa en el algoritmo usado en el Clasificador *IntServ* (Ver ítem A de la sección 3.5.2.1, pag 108 de este documento). La Identificación del Protocolo y los Puertos de Origen y Destino, se fijan en cero y además son iguales para cada uno de los flujos de la topología. De modo que las variables que van cambiando en cada flujo son las Direcciones IP de Origen y Destino.

Este tipo de topología se utilizó con el fin de facilitar la programación en C++ de los archivos internos del simulador, y su diseño puede variar dependiendo de lo que desee el usuario de la herramienta. El código implementado en la inicialización de la Quíntupla es el siguiente:

```

void inicializar() {
    src = 0;
    sport = 0;
    dst = 6;
    dport = 0;
}

```

```

        pid = 0;

    return;
}

```

Con base en la Figura 54, los valores iniciales las direcciones IP Origen y Destino son cero y seis respectivamente, debido a que son los nodos que representan la trayectoria del primer flujo (f_0). Con respecto a estas variables iniciales, las direcciones IP Origen y Destino se van incrementando de uno en uno para ir definiendo la trayectoria de cada uno de los flujos de la simulación.

El siguiente paso del Control de Admisión del Clasificador *IntServ6*, es un bucle *while* que tiene como condición que la Dirección IP Origen sea menor al Número de Flujos. A continuación se detalla el cuerpo de instrucciones de este ciclo *while* y el código utilizado:

```

while (src < flujoint_) {
    valor1 = hashkey_6(src, sport, dst, dport, pid);
    indice1 = valor1;
    campos6.numhash_ = valor1;
    campos6.BW_ = coll;
    campos6.colision_ = 0;
    campos6.origen_ = src;
    campos6.destino_ = dst;
    registrar_6(campos6, 0, indice1); //Llena los datos en la tabla hash
    ++src;
    ++dst;
    ++coll;
}

```

Los pasos del ciclo *while* son los siguientes:

✓ Cálculo del Número Hash

El cálculo del Número Hash se realiza teniendo en cuenta que las direcciones IP de Origen y Destino tienen un tamaño de 128 bits y como resultado de la misma función *XOR Folding* del Clasificador *IntServ*, se obtiene el nuevo Número Hash con un tamaño de 20 bits. La idea de calcular este Número Hash en el Host Origen es que pueda ser almacenado en la Etiqueta de Flujo de la cabecera IP del paquete, y sea transportado a través de la red [54]. El proceso del cálculo del Número Hash se realiza de la misma forma utilizada en el Clasificador *IntServ*, por lo que el ejemplo utilizado en esta etapa para la explicación del corrimiento de bits y almacenamiento de las palabras de 20 bits en el arreglo, es el mismo del Clasificador *IntServ* (Ver ítem A de la sección 3.5.2.1, pag 110 de este documento). Cabe resaltar que este cálculo se realiza con el objetivo de hacer el Control de Admisión y almacenar estos datos en la Tabla Hash o en la Tabla de Colisiones según el caso. Con el fin de realizar un resumen de este cálculo, se presenta un diagrama de flujo del Cálculo del Número Hash en el Clasificador *IntServ6* en la Figura 55.

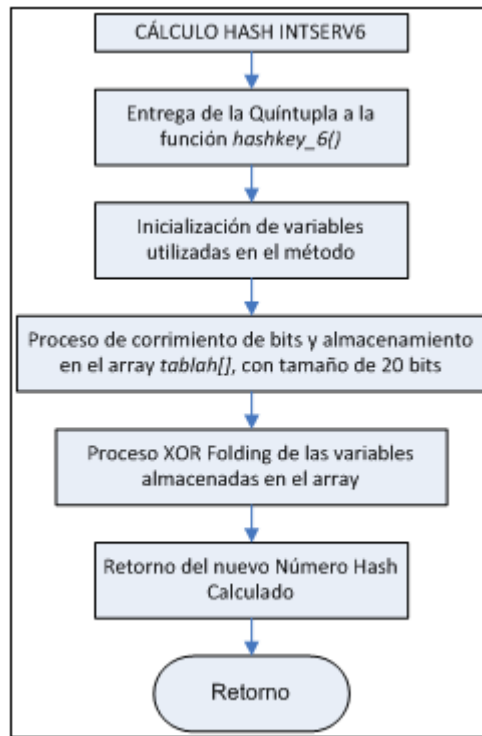


Figura 55. Cálculo del Número Hash en el Clasificador IntServ6.

El primer proceso del algoritmo del cálculo, el cual se realiza mediante el llamado al método *hashkey_6()*, es la entrega de los valores de la Quintupla a esta función. Un aspecto muy importante en este cálculo es la declaración de 3 variables de 32 bits cada una y de tipo entero (*dir1*, *dir2* y *dir3*), utilizadas para rellenar las Direcciones IP de 128 bits. Esto se realiza debido a que en el simulador no existen direcciones IPv6 reales; los nodos se caracterizan por tener una Identificación de Nodo, la cual equivale a la dirección IP en esta simulación. Luego se inicializan las variables utilizadas en el método como son: *dir1*, *dir2*, *dir3*, *w*, *x1_*, *x2_*, *masc1* (descritas más adelante en el código) y *hash2_*. Este Número Hash obtenido es almacenado en una variable *indice1*, que es utilizada más adelante en el método *registrar()*. A continuación se lista el código de la función *hashkey_6*.

```
int hashkey_6(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {
```

```
    int    w;
    int    dir1;
    int    dir2;
    int    dir3;
```

```
    dir1 = 0;
    dir2 = 0;
    dir3 = 0;
    w = 0;
    x1_ = 0;
    x2_ = 0;
    masc1_ = 0;
    hash2_ = 0;
```

//Cálculo hash

```

masc1_ = 0xffff000; //Mask 1
x1_ = dir1;
x1_ &= masc1_;
x1_ >>= 3;
tablah[0] = x1_; //Variable 1
masc1_ = 0x00000fff; //Mask 2
x1_ = dir1;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0xff000000; //Mask 3
x2_ = dir2;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[1] = x1_; //Variable 2
masc1_ = 0x00ffff0; //Mask 4
x1_ = dir2;
x1_ &= masc1_;
x1_ >>= 1;
tablah[2] = x1_; //Variable 3
masc1_ = 0x0000000f; //Mask 5
x1_ = dir2;
x1_ &= masc1_;
x1_ <<= 4;
masc1_ = 0xffff0000; //Mask 6
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[3] = x1_; //Variable 4
masc1_ = 0x0000ffff; //Mask 7
x1_ = dir3;
x1_ &= masc1_;
x1_ <<= 1;
masc1_ = 0xf0000000; //Mask 8
x2_ = src;
x2_ &= masc1_;
x2_ >>= 7;
x1_ |= x2_;
tablah[4] = x1_; //Variable 5
masc1_ = 0x0fffff00; //Mask 9
x1_ = src;
x1_ &= masc1_;
x1_ >>= 2;
tablah[5] = x1_; //Variable 6
masc1_ = 0x000000ff; //Mask 10
x1_ = src;
x1_ &= masc1_;
x1_ <<= 3;
masc1_ = 0xffff0000; //Mask 11
x2_ = dir1;
x2_ &= masc1_;
x2_ >>= 5;
x1_ |= x2_;
tablah[6] = x1_; //Variable 7
masc1_ = 0x000fffff; //Mask 12
x1_ = dir1;
x1_ &= masc1_;
tablah[7] = x1_; //Variable 8
masc1_ = 0xffff0000; //Mask 13
x1_ = dir2;
x1_ &= masc1_;
x1_ >>= 3;
tablah[8] = x1_; //Variable 9

```

```

    masc1_ = 0x00000fff; //Mask 14
    x1_ = dir2;
    x1_ &= masc1_;
    x1_ <<= 2;
    masc1_ = 0xff000000; //Mask 15
    x2_ = dir3;
    x2_ &= masc1_;
    x2_ >>= 6;
    x1_ |= x2_;
    tablah[9] = x1_; //Variable 10
    masc1_ = 0x00ffff0; //Mask 16
    x1_ = dir3;
    x1_ &= masc1_;
    x1_ >>= 1;
    tablah[10] = x1_; //Variable 11
    masc1_ = 0x0000000f; //Mask 17
    x1_ = dir3;
    x1_ &= masc1_;
    x1_ <<= 4;
    masc1_ = 0xffff0000; //Mask 18
    x2_ = dst;
    x2_ &= masc1_;
    x2_ >>= 4;
    x1_ |= x2_;
    tablah[11] = x1_; //Variable 12
    masc1_ = 0x0000ffff; //Mask 19
    x1_ = dst;
    x1_ &= masc1_;
    x1_ <<= 1;
    masc1_ = 0x0000f000; //Mask 20
    x2_ = sport;
    x2_ &= masc1_;
    x2_ >>= 3;
    x1_ |= x2_;
    tablah[12] = x1_; //Variable 13
    masc1_ = 0x00000fff; //Mask 21
    x1_ = sport;
    x1_ &= masc1_;
    x1_ <<= 2;
    masc1_ = 0x0000ff00; //Mask 22
    x2_ = dport;
    x2_ &= masc1_;
    x2_ >>= 2;
    x1_ |= x2_;
    tablah[13] = x1_; //Variable 14
    masc1_ = 0x000000ff; //Mask 23
    x1_ = dport;
    x1_ &= masc1_;
    x1_ <<= 2;
    x2_ = pid;
    x2_ &= masc1_;
    x1_ |= x2_;
    tablah[14] = x1_; //Variable 15

    for (w=0;w<15;w++) {
        hash2_ ^= tablah[w];
    }

    return (int) hash2_;
}
};

```

La notación utilizada en este cálculo es la siguiente: *src* es la Dirección IP Origen, *dst* es la Dirección IP Destino, *sport* es el Puerto Origen, *dport* es el Puerto

Destino y *pid* es la Identificación del Protocolo. Además están las variables utilizadas para el relleno de las direcciones IP: *dir1*, *dir2* y *dir3*.

✓ Ingreso de Datos a la Estructura

Al igual que en el Clasificador *IntServ*, este proceso se utiliza con el objetivo de organizar y optimizar el código de C++. El método que realiza este proceso es llamado *estructura()*, y su tarea es ingresar los siguientes campos en una estructura denominada *campos6*:

- Numero Hash de 20 bits de tipo entero.
- Direcciones IP Origen y Destino (128 bits c/u y de tipo entero). Estos dos campos son necesarios para ayudar a resolver una colisión y determinar el flujo correcto, dependiendo del caso
- Un campo de colisión, que es utilizado en la Tabla Hash para comunicarle a la red cuando existe una colisión de Números Hash.
- Un campo de Ancho de Banda, que corresponde a la variable que se retorna al realizar una búsqueda de un Número Hash específico y que da paso a clasificar cada uno de los paquetes.

Las variables *campos6* y *colisiones6* son de tipo *puntstruct* y *rescol* respectivamente, y se crean con las siguientes líneas de programación:

```
//Estructuras

puntstruct campos6;           // Nombre estructura Tabla Hash
rescol colisiones6           // Nombre estructura Tabla de Colisiones

struct puntstruct { // Estructura Utilizada en los métodos de la Tabla Hash
    int    numhash_;        // Numero hash
    int    BW_;             // Ancho de Banda
    int    colision_;       // Colision
    int    origen_;         // Dirección IP Origen
    int    destino_;        // Dirección IP Destino
};

struct rescol { // Estructura Utilizada en los métodos de la Tabla de Colisiones
    int    numhash1_;       // Numero Hash
    int    origen1_;        // Dirección IP Origen
    int    destino1_;       // Dirección IP Destino
    int    BW1_;            // Ancho de Banda
};
```

Estos tipos de estructuras son creadas para este proyecto y siguen siendo los mismos del Clasificador *IntServ*, con la única diferencia que las nuevas variables son llamadas *campos6* y *colisiones6*.

El código utilizado para inicializar esta estructura es:

```
void estructura() { // Método para llenar la estructura campos6
    campos6.numhash_ = hash2_; // Número Hash
    campos6.BW_ = 0;           // Ancho de Banda
}
```

```

campos6.colision_ = 0;           // Colisión
campos6.origen_ = src;          // Dirección IP Origen
campos6.destino_ = dst;         // Dirección IP Destino
return;
} // Los Anchos de Banda se definen más adelante cuando se hace el acople del módulo
de C++ con el simulador NS-2

```

Cabe resaltar que en este proceso no se retorna ningún valor, por lo cual el método es de tipo *void*, lo que significa que realiza el anterior proceso y retorna al punto donde fue llamado.

✓ Gestión de la Tabla Hash en *IntServ6*

Al realizar el proceso de ingreso de los datos a la estructura *campos6*, se llama al método *registrar_6()*, el cual presenta varias opciones de proceso según sus datos de entrada. A continuación se puede observar la declaración del método y su respectiva matriz de datos, que equivale a la Tabla Hash.

```

int registrar_6(puntstruct& tab1, int a1, int indice1): // Método para acceder a la Tabla Hash o matriz
hstable[]

```

Donde *tab1* es un apuntador a una estructura de tipo *puntstruct*, *a1* es la variable con la cual se decide que proceso se va a seguir y la variable *indice1* representa el índice dentro de la Tabla Hash, que en este caso es el mismo Número Hash.

La creación de la matriz equivalente a la Tabla Hash se muestra en la siguiente línea de programación:

```

static int hstable [y] [5]; // [Filas] [Columnas]

```

Donde *y* equivale al Número Hash máximo calculado, pues en este Clasificador no existe un número constante de filas en la Tabla Hash, siempre depende del Número Hash más grande de todos los flujos utilizados.

Al ser invocado este método, la primera tarea consiste en comparar el valor que trae la variable de entrada *a1*, y dependiendo de este valor se decide el proceso a seguir. Si *a1=0*, se ingresa un nuevo Número Hash a la Tabla Hash y se verifica si existe alguna colisión con otro flujo ingresado anteriormente. En caso de que exista una colisión, se hace el ingreso de los Números Hash Colisionados en la Tabla de Resolución de Colisiones. Y el otro caso se presenta cuando *a1=1*, en el cual se realiza una búsqueda de un Número Hash en la Tabla Hash o en la Tabla de colisiones según el caso. En la Figura 56 se puede observar el Diagrama de Flujo de este procedimiento y las 2 opciones de proceso que más adelante se explican más a fondo.

En el primer caso (Figura 57), cuando *a1=0*, se toma el Número Hash que trae almacenado la variable *indice1* y se guarda en la variable *x*. Esta variable sirve de índice en la matriz para acceder al campo de colisión y verificar si ese Número

Hash ya ha sido ingresado. Si no se ha utilizado ese Número Hash para otro Flujo y no existe colisión, se ingresa este Número Hash con sus respectivas Direcciones IP de Origen y Detino, su reserva y el campo de colisión que se mantiene en 0.



Figura 56. Diagrama de Flujo del método *registrar_6()*

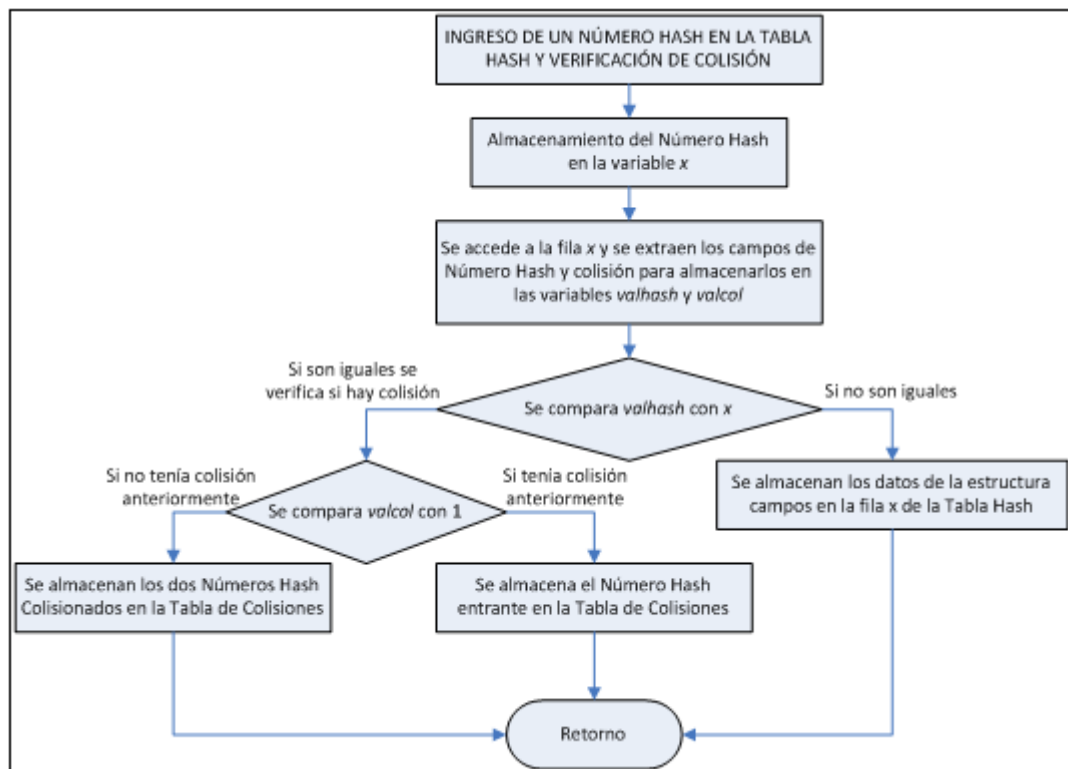


Figura 57. Ingreso de un Número Hash en la Tabla Hash y Verificación de Colisión

Si existe colisión de Números Hash, se compara nuevamente el campo de colisión con el fin de saber si existe más de una colisión (*colisión=1*), o es la primera colisión de ese Número Hash (*colisión=0*). En el primer caso solo se ingresan los

datos del último Número Hash Colisionado en la Tabla de Colisiones, y en el segundo caso, se ingresan los dos Número Hash Colisionados con sus respectivos datos en la Tabla de Colisiones. Para un mejor entendimiento de este proceso remitirse a la Figura 57 anteriormente mencionada.

Y el segundo caso es cuando se realiza una búsqueda de un Número Hash específico en la Tabla Hash primero, y en caso de tener colisión lo envía a buscar en la Tabla de Colisiones mediante el método *resolver()*. Para entender más fácilmente el algoritmo de este proceso remítase a la Figura 58.

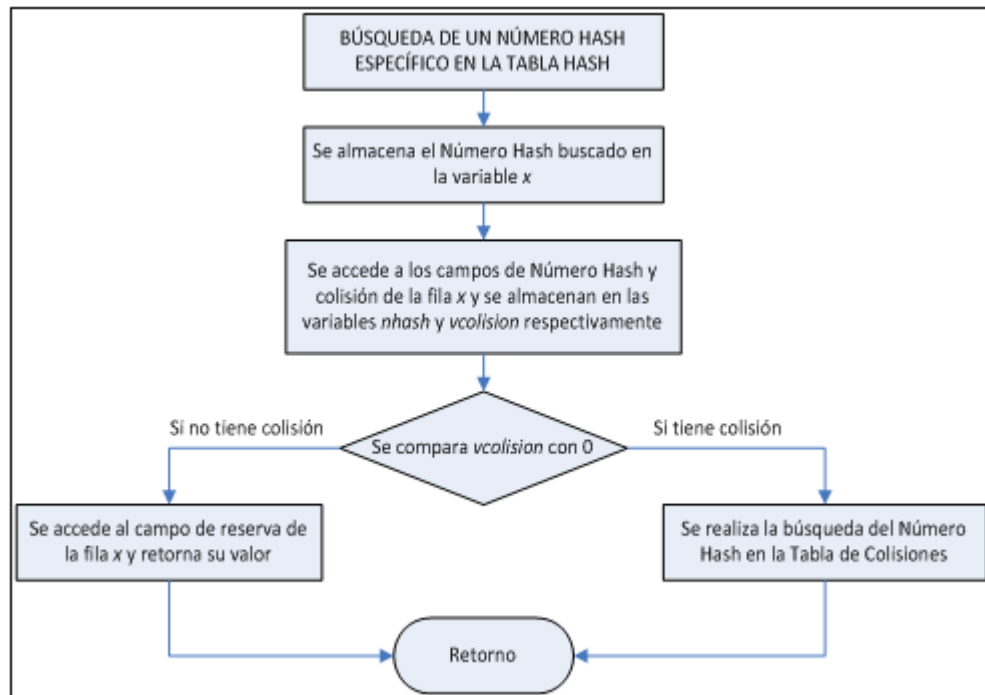


Figura 58. Búsqueda de un Número Hash Específico en la Tabla Hash

A continuación se muestran las líneas de código necesarias para cumplir con el anterior procedimiento:

```

if (a1 == 0) {
    a = indice1;
    valcol = hstabla[a][3];
    valhash = hstabla[a][1];
    if (valhash == indice1) {
        if (valcol == 0) { //Existe un Numero hash igual, pero no tenia colisión hasta esta entrada.
            colisiones6.numhash1_ = hstabla[a][1];
            colisiones6.origen1_ = hstabla[a][4];
            colisiones6.destino1_ = hstabla[a][5];
            colisiones6.BW1_ = hstabla[a][2];
            hstabla[a][3] = 1;
            resolver_6(colisiones6, 0);
            colisiones6.numhash1_ = tab1.numhash_;
        }
    }
}
  
```

```

        colisiones6.origen1_ = tab1.origen_;
        colisiones6.destino1_ = tab1.destino_;
        colisiones6.BW1_ = tab1.BW_;
        resolver_6(colisiones6, 0);
        return 0;

    } else if (valcol == 1) { //Existe un Número Hash igual, con una colisión anterior.
        colisiones6.numhash1_ = tab1.numhash_;
        colisiones6.origen1_ = tab1.origen_;
        colisiones6.destino1_ = tab1.destino_;
        colisiones6.BW1_ = tab1.BW_;
        resolver_6(colisiones6, 0);
        return 0;
    }
} else {
    hstaba [a] [1] = tab1.numhash_; // Primera entrada de este Número Hash
    hstaba [a] [2] = tab1.BW_;
    hstaba [a] [3] = tab1.colision_;
    hstaba [a] [4] = tab1.origen_;
    hstaba [a] [5] = tab1.destino_;
    return 0;

}return 0;

} else if (a1 == 1) {
    a = tab1.numhash_;
    rtdocol1 = hstaba [a] [3];
    nhash1 = hstaba [a] [1];
    if (rtdocol1 == 0) {
        rtdoflow1 = hstaba [a][2];
        return (rtdoflow1);
    } else {
        colisiones6.numhash1_ = tab1.numhash_;
        colisiones6.origen1_ = tab1.origen_;
        colisiones6.destino1_ = tab1.destino_;
        y = resolver_6(colisiones6, 1);
        rtdoflow1 = y;
        return (rtdoflow1);
    }
}

return (int) rtdoflow1;
}

```

Dentro del procedimiento *registrar_6()* se invoca al método *resolver_6()*, el cual es utilizado para acceder a otra matriz que en este caso es llamada *tablacoli[]*, la cual equivale a la Tabla de Resolución de Colisiones. Este método a su vez, utiliza una estructura anteriormente creada que se llama *colisiones6* y que es de tipo *rescol* (Ver página 147).

✓ Gestión de la Tabla de Colisiones en *IntServ6*

Este método sirve para acceder a la Tabla de Colisiones ya sea para ingresar un nuevo dato o para realizar la búsqueda de un Número Hash Colisionado. Por lo tanto, tiene 2 casos de procesos (Ver Figura 59):

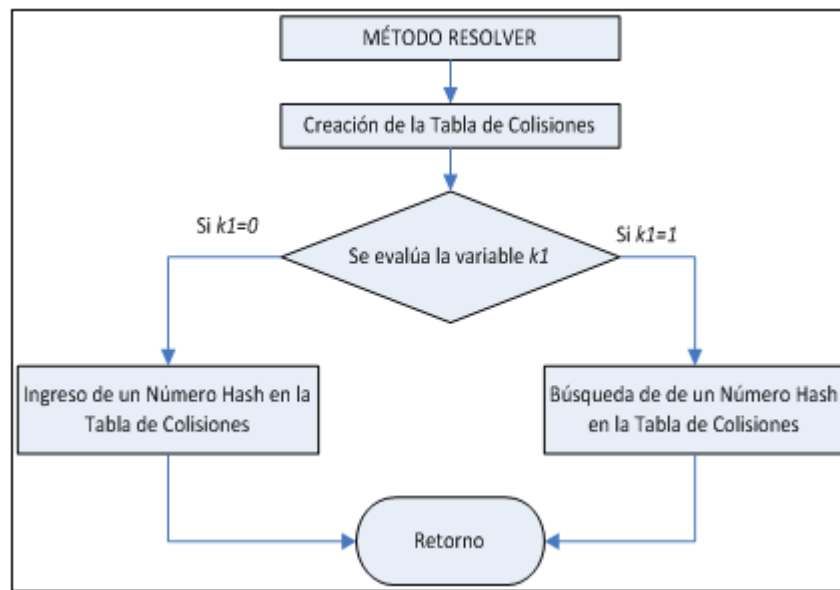


Figura 59. Diagrama de Flujo del método *resolver_6()*

La variable que se evalúa para decidir el siguiente proceso es *k1*. Si *k1=0*, se ingresa un nuevo Número Hash Colisionado en la Tabla de Colisiones, con sus respectivos campos de Direcciones IP y reserva. Y en caso de que *k1=1*, se realiza la búsqueda de un Número Hash específico y se retorna la reserva de dicho flujo. En las Figuras 60 y 61 se realiza una explicación detallada de estos casos mediante Diagramas de Flujo.

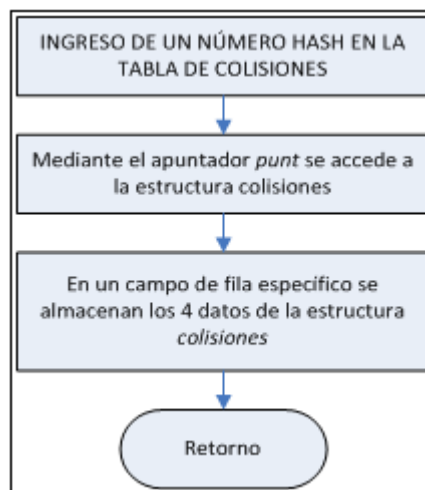


Figura 60. Ingreso un Número Hash en la Tabla de Colisiones

En la Figura 60 se puede observar los dos pasos que se realizan cuando se va a ingresar un nuevo Número Hash en la Tabla de Colisiones. Simplemente se accede a la estructura *colisiones6* y se extraen los datos para luego ser almacenados en una fila específica de la matriz *tablacoli[]*.

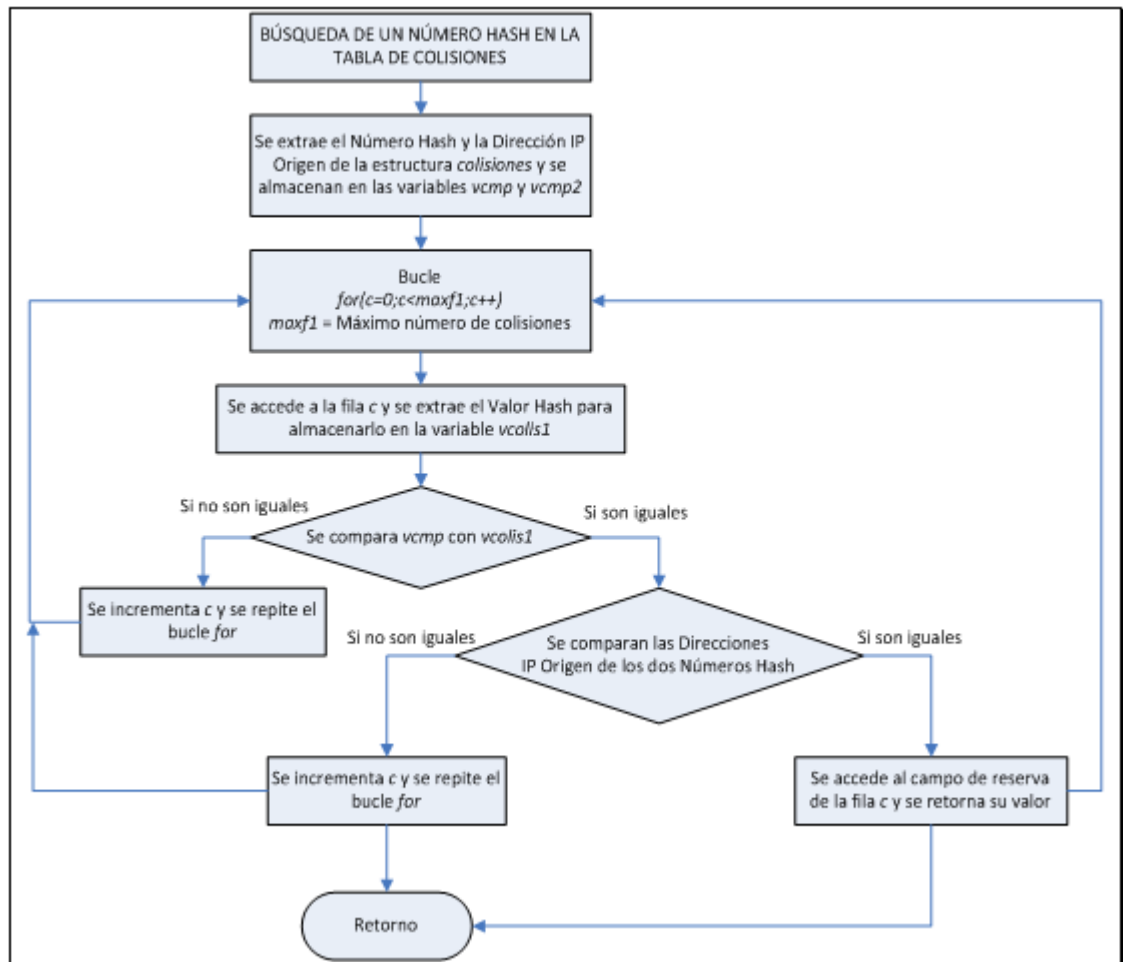


Figura 61. Búsqueda un Número Hash en la Tabla de Colisiones

En la Figura 61 se realiza un bucle *for*, para realizar la búsqueda de un Número Hash específico, el cual viene almacenado en una estructura llamada *colisiones6*, con sus respectivas direcciones IP para comprobar los Número Hash encontrados en la Tabla de Colisiones. Finalmente al encontrar el Número Hash correcto se retorna la variable *reserva1*. El código utilizado para programar el método *resolver_6()*, es el siguiente:

```

if (k1 == 0) {
    c = c + 1;
    tablacoli[c][1] = punt1.numhash1_;
    tablacoli[c][2] = punt1.origen1_;
    tablacoli[c][3] = punt1.destino1_;
    tablacoli[c][4] = punt1.BW1_;
    return (int) 0;
} else if (k1 == 1) {
    maxf1 = flujoint_;
    for (c=0; c<maxf1; c++) {
        vcmp = punt1.numhash1_;
        vcmp2 = punt1.origen1_;
        vcolis1 = tablacoli[c][1];
        if (vcmp == vcolis1) {

```

```

                                vquint1 = tablacoli [c] [2];
                                if (vcmp2 == vquint1) {
                                    reserva1 = tablacoli [c] [4];
                                    c = 1300;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return (int) reserva1;
}

```

C. CAMBIOS REALIZADOS AL SIMULADOR NS-2

A continuación se explica cuales fueron los nuevos archivos creados y los cambios realizados al simulador *NS-2* para la creación del Clasificador *IntServ6*.

El primer aspecto que se tiene en cuenta es hacer las modificaciones a los archivos necesarios para cumplir con los objetivos del Clasificador *IntServ6*. La ubicación de estos archivos se encuentra en la Tabla N°9 en los ítems 5, 6, 7 y 8.

El primer archivo que es indispensable modificar es *ip.h*, que es el encargado de configurar la cabecera IP que utilizan cada uno de los paquetes. En este código se le debe adicionar un nuevo campo de Identificación de Protocolo, y un campo para almacenar el Número Hash del paquete, equivalente a la Etiqueta de Flujo de *IPv6*. Además también se incluyen las funciones de acceso a las nuevos campos creados. A continuación se puede observar las líneas de código adicionadas resaltadas en negrita:

IP.H

```

#ifndef ns_ip_h
#define ns_ip_h

#include "config.h"
#include "packet.h"

#define IP_HDR_LEN    40 // Se le cambió el tamaño de la cabecera IPv6
#define IP_DEF_TTL    32

// The following undef is to suppress warnings on systems where
// IP_BROADCAST is defined.
#ifndef IP_BROADCAST
#undef IP_BROADCAST
#endif

// #define IP_BROADCAST ((u_int32_t) 0xffffffff)
static const u_int32_t IP_BROADCAST = ((u_int32_t) 0xffffffff);

struct hdr_ip {
    /* common to IPv{4,6} */
    ns_addr_t    src_;
    ns_addr_t    dst_;
    int          ttl_;

    // Declaración de los campos de Identificación de Flujo y Número Hash (Etiqueta de Flujo – IPv6)
    int          idprot_;
    int          xhash_;

    /* IPv6 */

```

```

int          fid_;    /* flow id */
int          prio_;

static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_ip* access(const Packet* p) {
    return (hdr_ip*) p->access(offset_);
}

/* per-field member acces functions */
ns_addr_t& src() { return (src_); }
nsaddr_t& saddr() { return (src_.addr_); }
int32_t& sport() { return src_.port_; }

ns_addr_t& dst() { return (dst_); }
nsaddr_t& daddr() { return (dst_.addr_); }
int32_t& dport() { return dst_.port_; }
int& ttl() { return (ttl_); }
/* ipv6 fields */
int& flowid() { return (fid_); }
int& prio() { return (prio_); }

int& protid() { return (idprot_); } // Identificación del Protocolo
int& nhash() { return (xhash_); } // Número Hash

};

#endif

```

Este archivo *ip.h* trae por defecto dos estructuras de tipo *ns_addr_t* llamadas *src_* (Origen) y *dst_* (Destino), cada una con dos campos (de 32 bits y 16 bits respectivamente) llamados *addr_* (Dirección) y *port_* (Puerto). Cabe resaltar que el campo *port_* se le modificó el tipo de *int32_t* a *int16_t* para cumplir con el tamaño de 16 bit. Este cambio se realiza en el archivo *config.h* descrito más adelante. Al hacerle esta modificación a este archivo, también se le debe hacer el mismo cambio al archivo *ip.cc*, es por esto que se añade el código con las nuevas líneas agregadas:

IP.CC:

```

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /cvsroot/nsnam/ns-2/common/ip.cc,v 1.8 1998/08/12 23:41:05 gnguyen Exp $";
#endif

#include "packet.h"
#include "ip.h"

int hdr_ip::offset_;

static class IPHeaderClass : public PacketHeaderClass {
public:
    IPHeaderClass() : PacketHeaderClass("PacketHeader/IP",
        sizeof(hdr_ip)) {
        bind_offset(&hdr_ip::offset_);
    }
    void export_offsets() {
        field_offset("src_", OFFSET(hdr_ip, src_));
        field_offset("dst_", OFFSET(hdr_ip, dst_));
        field_offset("ttl_", OFFSET(hdr_ip, ttl_));
        field_offset("fid_", OFFSET(hdr_ip, fid_));
    }
}

```

```

        field_offset("prio_", OFFSET(hdr_ip, prio_));
        field_offset("idprot_", OFFSET(hdr_ip, idprot_)); // Identificación de Protocolo
        field_offset("xhash_", OFFSET(hdr_ip, xhash_)); // Número Hash
    }
} class_iphdr;

```

Los archivos *ip.h* e *ip.cc* se encuentran ubicados en la ruta de acceso */home/usuario/ns-allinone-2.31/ns-2.31/common*. Esta ruta puede variar dependiendo de la ubicación donde se halla instalado el simulador, en nuestro caso se instaló en la carpeta del usuario en *Linux (Fedora Core 8)*.

Como se puede observar en los anteriores programas, se utiliza una estructura de tipo *ns_addr_t* que se encuentra declarada y configurada en el archivo *config.h*, además en este documento se pueden encontrar las dos clases de estructuras utilizadas para acceder a la Tabla Hash y Tabla de Colisiones mencionadas en el Modulo de C++. Este código se puede observar en el Anexo A de este documento.

Finalmente se muestran los códigos de los nuevos objetos creados para la simulación de *IntServ6*. Para esto primero se deben agregar al archivo *makefile.in*, para que sean añadidos en el momento de la compilación de todos los objetos del simulador. A continuación se muestran estas líneas resaltadas en negrita:

```

OBJ_CC = \
    tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
    common/scheduler.o common/object.o common/packet.o \
    common/ip.o routing/route.o common/connector.o common/ttl.o \
    trace/trace.o trace/trace-ip.o \
    classifier/classifier.o classifier/classifier-addr.o \
    classifier/classifier-hash.o \
    classifier/classifier-IntServ.o \ // Clasificador IntServ
    classifier/classifier-IntServ6.o \ // Clasificador IntServ6
    classifier/IntServ6.o \ // Módulo que calcula el Número Hash para IntServ6 en el Host
    classifier/classifier-virtual.o \
    classifier/classifier-mcast.o \
    classifier/classifier-bst.o \
    classifier/classifier-mpath.o mcast/replicator.o \
    classifier/classifier-mac.o \
    classifier/classifier-qs.o \
    classifier/classifier-port.o src_rtg/classifier-sr.o \

```

CLASSIFIER-INTSERV6.H

// Programa Clasificador IntServ6

```

#include "classifier.h"
#include "ip.h"
#include "config.h"
#include "packet.h"
#include <fstream.h>

```

ofstream tabla("HashIntServ6.ods"); // constructor de ofstream

```

//*****
//CONSTRUCTOR

```

```

class IntServ6Classifier : public Classifier {
public:
    IntServ6Classifier() : flujoint_(-1), destinoint_(-1) {

```

```

        bind("flujoint_", &flujoint_);
        bind("destinojoint_", &destinojoint_);
    } // Creación de variables necesarias en el enlace de Tcl con C++

//*****
//DESTRUCTOR

    ~IntServ6Classifier() {
    };
//*****
// CLASSIFY: Es un método puro virtual indicando que la clase Classifier es solo usada como una clase base
protected:
    int classify(Packet *p);

//*****
// BUSQUEDA: Accede a la cabecera ip del paquete(quintupla) y lo manda a buscar a la tabla hash.
public:
    int busqueda_6(Packet *p) {
        hdr_ip* h = hdr_ip::access(p);
        return get_hash_6(h->nhash(),h->saddr(),h->daddr());
    }

//*****
protected:

//*****
// SET-HASH-1: Método para calcular número hash y guardarlos en la tabla hash.

    int set_hash_6(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

        int valor1;
        int indice1;
        int coll;
        coll = destinojoint_;
        c = 0;
        indice1 = 0;

        while (src<flujoint_) {
            valor1 = hashkey_6(src,sport,dst,dport,pid);
            tabla << valor1 << "\n";
            indice1 = valor1;
            campos6.numhash_ = valor1;
            campos6.BW_ = coll;
            campos6.colision_ = 0;
            campos6.origen_ = src;
            campos6.destino_ = dst;
            registrar_6(campos6, 0, indice1); //Llena los datos en la Tabla Hash y compara
            ++src;
            ++dst;
            ++coll;
        }
        return 0; // Retorna un número específico indicando que ya terminó el proceso
    }

//*****
// GET-HASH: Realiza una Búsqueda de un Número Hash específico en la Tabla Hash y retorna un valor de reserva

    int get_hash_6(int valorhash, nsaddr_t src, nsaddr_t dst) {
        int vfid1;
        campos6.numhash_ = valorhash;
        campos6.origen_ = src;
        campos6.destino_ = dst;
        vfid1 = registrar_6(campos6, 1, 0);
        return (vfid1);
    }
}

```

```

//*****
// COMMAND: Método principal del .cc

virtual int command(int argc, const char*const* argv);

//*****
// VARIABLES UTILIZADAS EN LA CLASE INTSERV6CLASSIFIER:

int          flujoint_;
int          destinoint_;
int          keylen2_;
int          hash2_;
int          masc1_;
int          x1_;
int          x2_;
int          tablah[15];
int          idprot_;
puntstruct   campos6;
rescol       colisiones6;
int          c;

//*****
// REGISTRAR: Método de acceso a la Tabla Hash (Llenar y Comparar, o Buscar)

int registrar_6(puntstruct& tab1, int a1, int indice1) {

int          a;
int          nhash1;
int          rtdocol1;
int          rtdoflow1;
int          valcol;
int          valhash;
int          y;
a = 0;
nhash1 = 0;
rtdocol1 = 0;
rtdoflow1 = 0;
valcol = 0;
valhash = 0;
y = 0;

static int hstaba [2500] [5]; // [Filas] [Columnas]
//a = 0 --> Llenar y comparar datos
//a = 1 --> Buscar Datos

if (a1 == 0) {
a = indice1;
valcol = hstaba [a] [3];
valhash = hstaba [a] [1];
if (valhash == indice1) {
if (valcol == 0) { // Existe un Numero Hash igual, pero no tenia colisión hasta esta
entrada.

colisiones6.numhash1_ = hstaba [a] [1];
colisiones6.origen1_ = hstaba [a] [4];
colisiones6.destino1_ = hstaba [a] [5];
colisiones6.BW1_ = hstaba [a] [2];
hstaba [a] [3] = 1;
resolver_6(colisiones6, 0);
colisiones6.numhash1_ = tab1.numhash_;
colisiones6.origen1_ = tab1.origen_;
colisiones6.destino1_ = tab1.destino_;
colisiones6.BW1_ = tab1.BW_;
resolver_6(colisiones6, 0);
return 0;
} else if (valcol == 1) { // Existe un Numero hash igual, con una colisión anterior
colisiones6.numhash1_ = tab1.numhash_;

```

```

        colisiones6.origen1_ = tab1.origen_;
        colisiones6.destino1_ = tab1.destino_;
        colisiones6.BW1_ = tab1.BW_;
        resolver_6(colisiones6, 0);
        return 0;
    }
} else {
    hstaba [a] [1] = tab1.numhash_; // Primera entrada de este Número Hash
    hstaba [a] [2] = tab1.BW_;
    hstaba [a] [3] = tab1.colision_;
    hstaba [a] [4] = tab1.origen_;
    hstaba [a] [5] = tab1.destino_;
    return 0;
}
return 0;
} else if (a1 == 1) {
    a = tab1.numhash_;
    rtdocol1 = hstaba [a] [3];
    nhash1 = hstaba [a] [1];
    if (rtdocol1 == 0) {
        rtdoflow1 = hstaba [a][2];
        return (rtdoflow1);
    } else {
        colisiones6.numhash1_ = tab1.numhash_;
        colisiones6.origen1_ = tab1.origen_;
        colisiones6.destino1_ = tab1.destino_;
        y = resolver_6(colisiones6, 1);
        rtdoflow1 = y;
        return (rtdoflow1);
    }
}
return (int) rtdoflow1;
}

return (rtdoflow1);
}

//*****
// RESOLVER:

int resolver_6(rescol& punt1, int k1) {

    int    vcmp;
    int    vcmp2;
    int    maxf1;
    int    vcolis1;
    int    vquint1;
    int    reserva1;
    vcmp = 0;
    vcmp2 = 0;
    maxf1 = 0;
    vcolis1 = 0;
    vquint1 = 0;
    reserva1 = 0;
    static int tablacoli [2500] [4]; // [Filas] [Columnas]
    //k = 0 --> Llenar tabla
    //k = 1 --> Buscar Datos

    if (k1 == 0) {
        c = c + 1;
        tablacoli [c] [1] = punt1.numhash1_;
        tablacoli [c] [2] = punt1.origen1_;
        tablacoli [c] [3] = punt1.destino1_;
        tablacoli [c] [4] = punt1.BW1_;

        return (int) 0;
    }
}

```

```

    } else if (k1 == 1) {
        maxf1 = flujoint_; // Máximo Número de colisiones
        for (c=0;c<maxf1;c++) {
            vcmp = punt1.numhash1_;
            vcmp2 = punt1.origen1_;
            vcolis1 = tablacoli [c] [1];
            if (vcmp == vcolis1) {
                vquint1 = tablacoli [c] [2];
                if (vcmp2 == vquint1) {
                    reserva1 = tablacoli [c] [4];
                    c = 1300;
                }
            }
        }
        return (int) reserva1;
    }
    return (reserva1);
}

//*****
//HASHKEY: Metodo que se utiliza para calcular el Numero Hash con la Quíntupla.

int hashkey_6(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {
    int w;
    int dir1;
    int dir2;
    int dir3;

    dir1 = 0;
    dir2 = 0;
    dir3 = 0;
    w = 0;
    x1_ = 0;
    x2_ = 0;
    masc1_ = 0;
    hash2_ = 0;

    //Cálculo hash

    masc1_ = 0xffff000; //Mask 1
    x1_ = dir1;
    x1_ &= masc1_;
    x1_ >>= 3;
    tablah[0] = x1_; //Variable 1
    masc1_ = 0x00000fff; //Mask 2
    x1_ = dir1;
    x1_ &= masc1_;
    x1_ <<= 2;
    masc1_ = 0xff000000; //Mask 3
    x2_ = dir2;
    x2_ &= masc1_;
    x2_ >>= 6;
    x1_ |= x2_;
    tablah[1] = x1_; //Variable 2
    masc1_ = 0x00ffff0; //Mask 4
    x1_ = dir2;
    x1_ &= masc1_;
    x1_ >>= 1;
    tablah[2] = x1_; //Variable 3
    masc1_ = 0x0000000f; //Mask 5
    x1_ = dir2;
    x1_ &= masc1_;
    x1_ <<= 4;
    masc1_ = 0xffff0000; //Mask 6
    x2_ = dir3;
    x2_ &= masc1_;
    x2_ >>= 4;

```

x1_ = x2_;		
tablah[3] = x1_;	//Variable 4	
masc1_ = 0x0000ffff;		//Mask 7
x1_ = dir3;		
x1_ &= masc1_;		
x1_ <<= 1;		
masc1_ = 0xf0000000;		//Mask 8
x2_ = src;		
x2_ &= masc1_;		
x2_ >>= 7;		
x1_ = x2_;		
tablah[4] = x1_;	//Variable 5	
masc1_ = 0x0ffff00;		//Mask 9
x1_ = src;		
x1_ &= masc1_;		
x1_ >>= 2;		
tablah[5] = x1_;	//Variable 6	
masc1_ = 0x000000ff;		//Mask 10
x1_ = src;		
x1_ &= masc1_;		
x1_ <<= 3;		
masc1_ = 0xffff0000;		//Mask 11
x2_ = dir1;		
x2_ &= masc1_;		
x2_ >>= 5;		
x1_ = x2_;		
tablah[6] = x1_;	//Variable 7	
masc1_ = 0x000ffff;		//Mask 12
x1_ = dir1;		
x1_ &= masc1_;		
tablah[7] = x1_;	//Variable 8	
masc1_ = 0xffff000;		//Mask 13
x1_ = dir2;		
x1_ &= masc1_;		
x1_ >>= 3;		
tablah[8] = x1_;	//Variable 9	
masc1_ = 0x00000fff;		//Mask 14
x1_ = dir2;		
x1_ &= masc1_;		
x1_ <<= 2;		
masc1_ = 0xff000000;		//Mask 15
x2_ = dir3;		
x2_ &= masc1_;		
x2_ >>= 6;		
x1_ = x2_;		
tablah[9] = x1_;	//Variable 10	
masc1_ = 0x00ffff0;		//Mask 16
x1_ = dir3;		
x1_ &= masc1_;		
x1_ >>= 1;		
tablah[10] = x1_;	//Variable 11	
masc1_ = 0x0000000f;		//Mask 17
x1_ = dir3;		
x1_ &= masc1_;		
x1_ <<= 4;		
masc1_ = 0xffff0000;		//Mask 18
x2_ = dst;		
x2_ &= masc1_;		
x2_ >>= 4;		
x1_ = x2_;		
tablah[11] = x1_;	//Variable 12	
masc1_ = 0x0000ffff;		//Mask 19
x1_ = dst;		
x1_ &= masc1_;		
x1_ <<= 1;		
masc1_ = 0x0000f000;		//Mask 20
x2_ = sport;		

```

x2_ &= masc1_;
x2_ >>= 3;
x1_ |= x2_;
tablah[12] = x1_;           //Variable 13
masc1_ = 0x0000fff;         //Mask 21
x1_ = sport;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0x0000ff00;        //Mask 22
x2_ = dport;
x2_ &= masc1_;
x2_ >>= 2;
x1_ |= x2_;
tablah[13] = x1_;           //Variable 14
masc1_ = 0x000000ff;        //Mask 23
x1_ = dport;
x1_ &= masc1_;
x1_ <<= 2;
x2_ = pid;
x2_ &= masc1_;
x1_ |= x2_;
tablah[14] = x1_;           //Variable 15

for (w=0;w<15;w++) {
    hash2_ ^= tablah[w];
}

return (int) hash2_;
}
};

```

//*****

CLASSIFIER-INTSERV6.CC

```

#include <stdlib.h>
#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include "classifier-IntServ6.h"

/***** IntServ6Classifier Methods *****/

int IntServ6Classifier::classify(Packet *p) {

    busqueda_6(p);

    return (2);

} // IntServ6Classifier::classify

int IntServ6Classifier::command(int argc, const char*const* argv)
{
    if (argc == 4) {
        /*$classifier adc-num src dst */
        if (strcmp(argv[1], "adc-num") == 0) {
            nsaddr_t src = atoi(argv[2]);
            int16_t sport = 0;
            nsaddr_t dst = atoi(argv[3]);
            int16_t dport = 0;
            int pid = 0;
            int n = set_hash_6(src,sport,dst,dport,pid);
            return TCL_OK;
        }
    }
}

```

```

    }
    } return (Classifier::command(argc, argv));
}

/***** TCL linkage *****/
static class IntServ6ClassifierClass : public TclClass {
public:
    IntServ6ClassifierClass() : TclClass("Classifier/IntServ6") {}
    TclObject* create(int, const char*const*) {
        return (new IntServ6Classifier());
    }
} class_intserv6_classifier;

```

HOSTORIGEN6.H:

```

#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include <fstream.h>

ofstream tabla1("nhashintserv6.ods"); // constructor de ofstream

class StartClassifier : public Classifier {
public:

    int classify(Packet *p);

    int      masc1_;
    int      x1_;
    int      x2_;
    int      hash2_;
    int      tablah[15];

    int set_num(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

        int a;

        a = hashkey(src,sport,dst,dport,pid);
        tabla1 << a << "\n";

        return (a) ;
    }

    int calculo(Packet *p) {
        hdr_ip* h = hdr_ip::access(p);
        return set_num(mshift(h->saddr()),mshift(h->sport()),mshift(h->daddr()),mshift(h->dport()),h->protid());
    }

    int hashkey_6(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

        int      w;
        int      dir1;
        int      dir2;
        int      dir3;

        dir1 = 0;
        dir2 = 0;
        dir3 = 0;
        w = 0;
        x1_ = 0;
        x2_ = 0;
        masc1_ = 0;
        hash2_ = 0;
    }

```

//Cálculo hash

```

masc1_ = 0xffff000; //Mask 1
x1_ = dir1;
x1_ &= masc1_;
x1_ >>= 3;
tablah[0] = x1_; //Variable 1
masc1_ = 0x00000fff; //Mask 2
x1_ = dir1;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0xff000000; //Mask 3
x2_ = dir2;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[1] = x1_; //Variable 2
masc1_ = 0x00ffff0; //Mask 4
x1_ = dir2;
x1_ &= masc1_;
x1_ >>= 1;
tablah[2] = x1_; //Variable 3
masc1_ = 0x0000000f; //Mask 5
x1_ = dir2;
x1_ &= masc1_;
x1_ <<= 4;
masc1_ = 0xffff0000; //Mask 6
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[3] = x1_; //Variable 4
masc1_ = 0x0000ffff; //Mask 7
x1_ = dir3;
x1_ &= masc1_;
x1_ <<= 1;
masc1_ = 0xf0000000; //Mask 8
x2_ = src;
x2_ &= masc1_;
x2_ >>= 7;
x1_ |= x2_;
tablah[4] = x1_; //Variable 5
masc1_ = 0x0ffff00; //Mask 9
x1_ = src;
x1_ &= masc1_;
x1_ >>= 2;
tablah[5] = x1_; //Variable 6
masc1_ = 0x000000ff; //Mask 10
x1_ = src;
x1_ &= masc1_;
x1_ <<= 3;
masc1_ = 0xff000000; //Mask 11
x2_ = dir1;
x2_ &= masc1_;
x2_ >>= 5;
x1_ |= x2_;
tablah[6] = x1_; //Variable 7
masc1_ = 0x000ffff; //Mask 12
x1_ = dir1;
x1_ &= masc1_;
tablah[7] = x1_; //Variable 8
masc1_ = 0xffff000; //Mask 13
x1_ = dir2;
x1_ &= masc1_;
x1_ >>= 3;
tablah[8] = x1_; //Variable 9
masc1_ = 0x00000fff; //Mask 14

```

```

x1_ = dir2;
x1_ &= masc1_;
x1_ <=<= 2;
masc1_ = 0xff000000; //Mask 15
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[9] = x1_; //Variable 10
masc1_ = 0x00ffff0; //Mask 16
x1_ = dir3;
x1_ &= masc1_;
x1_ >>= 1;
tablah[10] = x1_; //Variable 11
masc1_ = 0x0000000f; //Mask 17
x1_ = dir3;
x1_ &= masc1_;
x1_ <=<= 4;
masc1_ = 0xffff0000; //Mask 18
x2_ = dst;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[11] = x1_; //Variable 12
masc1_ = 0x0000ffff; //Mask 19
x1_ = dst;
x1_ &= masc1_;
x1_ <=<= 1;
masc1_ = 0x0000f000; //Mask 20
x2_ = sport;
x2_ &= masc1_;
x2_ >>= 3;
x1_ |= x2_;
tablah[12] = x1_; //Variable 13
masc1_ = 0x00000fff; //Mask 21
x1_ = sport;
x1_ &= masc1_;
x1_ <=<= 2;
masc1_ = 0x0000ff00; //Mask 22
x2_ = dport;
x2_ &= masc1_;
x2_ >>= 2;
x1_ |= x2_;
tablah[13] = x1_; //Variable 14
masc1_ = 0x000000ff; //Mask 23
x1_ = dport;
x1_ &= masc1_;
x1_ <=<= 2;
x2_ = pid;
x2_ &= masc1_;
x1_ |= x2_;
tablah[14] = x1_; //Variable 15

for (w=0;w<15;w++) {
    hash2_ ^= tablah[w];
}

return (int) hash2_;
}
};

```

HOSTORIGEN6.CC:

```
#include "IntServ6.h"
#include "ip.h"

int StartClassifier::classify(Packet *p)
{
    int hs = calculo(p);
    hdr_ip* h = hdr_ip::access(p);
    h->nhash() = hs;

    return (1);
}

/***** TCL linkage *****/
static class StartClassifierClass : public TclClass {
public:
    StartClassifierClass() : TclClass("Classifier/Start") {}
    TclObject* create(int, const char*const*) {
        return (new StartClassifier());
    }
} class_start_classifier;
```

3.5.3. DESARROLLO DEL PROTOCOLO EN OTCL

En esta etapa del proyecto se realiza el Desarrollo de los Protocolos *IntServ* e *IntServ6* en *Tcl*; esta fase permitirá la creación de los scripts con extensión *.tcl*, necesarios para configurar el tipo de topología que se desea utilizar en la simulación, y sus respectivos parámetros indispensables para cumplir con el objetivo del presente proyecto.

Para realizar el enlace entre *Tcl* y C++ se necesita primero declarar el nombre del objeto en C++, para luego ser llamado desde *Tcl*. A continuación se muestra un ejemplo de cómo se declara inicialmente el objeto y como es llamado desde *Tcl*:

```
// Declaración en C++
static class StartClassifierClass : public TclClass {
public:
    StartClassifierClass() : TclClass("Classifier/Start") {}
    TclObject* create(int, const char*const*) {
        return (new StartClassifier());
    }
} class_start_classifier;

// Invocación desde Tcl
set cls [new Classifier/Start]
```

Cada objeto de C++ tiene su objeto equivalente en *Tcl*, A continuación se muestran estos dos objetos:

Classifier/Start → Objeto en *Tcl*
StartClassifier → Objeto en C++

3.5.3.1. PLANIFICADOR WFQ

Esta es una contribución al código de *NS-2* que se encuentra disponible en la página oficial del Simulador *NS-2* [62] en la sección que hace referencia al manejador de colas y al Planificador *WFQ*. La herramienta es creada por Paolo Losi [72].

Con el siguiente código se quiere dar un ejemplo de cómo se debe utilizar la herramienta *WFQ* para su correcto funcionamiento.

```
set ns [new Simulator]

# Creación de los archivos para el trazado
set f [open out.tr w]
$ns trace-all $f
set nam [open out.nam w]
$ns namtrace-all $nam

# Creando los nodos
set n0 [$ns node]
set n1 [$ns node]
set r [$ns node]
set d [$ns node]

$ns color 0 blue
$ns color 1 red

$ns duplex-link $n0 $r 5Mb 1ms DropTail
$ns duplex-link $n1 $r 5Mb 1ms DropTail
```

En esta parte del programa se realiza la instalación del Planificador *WFQ*, el cual se encuentra ubicado en el enlace entre los nodos *r* y *d*. Este enlace presenta dos sentidos; si el flujo de paquetes va desde el nodo *r* al nodo *d*, presenta las características de un enlace *WFQ*. Si el sentido del flujo va desde el nodo *d* al nodo *r*, el enlace presenta una cola *FIFO* (*First In First Out*). Esta configuración se debe a que la reserva de un flujo solo se hace en un solo sentido, en nuestro caso desde el nodo *r* al nodo *d*. Las líneas de código que se utilizaron para el anterior procedimiento son resaltadas en negrita a continuación:

```
$ns simplex-link $r $d 2Mb 2ms WFQ
$ns simplex-link $d $r 1Mb 2ms DropTail
$ns duplex-link-op $n0 $r orient right-down
$ns duplex-link-op $n1 $r orient right-up
$ns simplex-link-op $r $d orient right
$ns simplex-link-op $d $r orient left
```

Al finalizar la instalación del Planificador *WFQ* en el enlace, es necesario crear un objeto en *Tcl* que hace referencia al Clasificador interno del *WFQ*. El objetivo de este Clasificador es monitorear cada uno de los paquetes verificando la Identificación de flujo de cada uno de ellos, para finalmente clasificarlos en sus respectivas colas. La creación de este Clasificador y la instalación de éste en el enlace son programas en *Tcl* con las siguientes instrucciones:

```
set cl [new WFQAggregClassifier]
$ns wfqclassifier-install $r $d $cl
```

Cada cola del Planificador *WFQ* tiene un valor de peso asignado por el usuario, que equivale a un porcentaje del Ancho de Banda total del enlace *WFQ*. La asignación de estos pesos se realiza mediante las siguientes líneas de código:

```
$cl setqueue 0 3; # fid 0 -> cola 3
$cl setqueue 1 4; # fid 1 -> cola 4

$cl setweight 3 0.5; # La cola 3 tiene un peso de 0.5
$cl setlength 3 5; # La longitud del paquete en la cola 3 es de 5
$cl setweight 4 0.5; # La cola 5 tiene un peso de 0.5
$cl setlength 4 5; # La longitud del paquete de la cola 4 es de 5
```

Cabe resaltar que el peso de cada cola puede oscilar desde 0.001 hasta 1 y además en cada cola del Planificador *WFQ* el usuario pueden ingresar la cantidad flujos que desee, pero en este caso se le asigno un flujo a cada cola.

```
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
$udp0 set fid_ 0; # this is the info used by WFQ classifier
set cbr0 [new Application/Traffic/CBR]
$cbr0 set rate_ 1.1Mb
$cbr0 set packetSize_ 100
$cbr0 attach-agent $udp0
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
$udp1 set fid_ 1; # esta informacion es utilizada por el clasificador de WFQ.
set cbr1 [new Application/Traffic/CBR]
$cbr1 set rate_ 1.1Mb
$cbr1 set packetSize_ 200
$cbr1 attach-agent $udp1
set null [new Agent/Null]
$ns attach-agent $d $null
$ns connect $udp0 $null
$ns connect $udp1 $null
```

```
$ns at 0.0 "$cbr0 start"
$ns at 0.1 "$cbr1 start"
```

Para realizarle una prueba al Planificador *WFQ*, se le hizo una variación a los pesos de los colas al primer segundo de la simulación, para analizar la respuesta del enlace y el ancho de banda entregado a cada uno de los flujos. La línea utilizada para este proceso se muestra resaltada en negrita a continuación:

```
$ns at 1 "$cl setweight 3 0.8; $cl setweight 4 0.2"
$ns at 2.0 "finish"
proc finish {} {
    global ns f nam
    $ns flush-trace
    close $f
    close $nam
    exit 0
}
$ns run
```

Con la información de la Identificación de flujo que se le fija al protocolo de transporte *UDP*, el planificador *WFQ* clasifica sus paquetes en diferentes colas.

La herramienta utiliza un tiempo de finalización para cada paquete, el cual depende del número de colas activas, del peso relativo asignado a cada cola y de la longitud de cada uno de los paquetes que llega a las colas. El planificador selecciona y desencola el paquete que tenga el menor tiempo de finalización de todos los paquetes almacenados en la cola. El tiempo de finalización no es el tiempo que tarda en transmitirse el paquete, sino un número asignado por la disciplina de planificación a cada paquete para saber el orden en el que los paquetes deberían ser desencolados.

3.5.3.2. DISEÑO DE LA RED INTSERV

Los programas en *Tcl* se organizaron de forma que hay un programa principal y un subprograma, llamado por éste. El primer *script* presenta el código del programa principal en *Tcl* e incluye la configuración de los nodos que utilizan los módulos de C++ descritos anteriormente (*classifier-IntServ*), la configuración de los parámetros utilizados por el Planificador *WFQ*, la creación de agentes generadores de tráfico encargados de enviar los paquetes a través de la red, la declaración de variables globales utilizadas en la ejecución del programa, entre otros procesos necesarios para la simulación de la red.

El segundo *script* describe un subprograma en *Tcl*, que define el tipo de topología que se va a simular, incluyendo el acople entre el modulo C++ y el nodo que se desea configurar como *IntServ*. Este subprograma es llamado por el programa

principal. Para que esto se pueda hacer, se debe declarar en el *script* principal de la simulación el nombre del archivo que contiene el subprograma, y simplemente llamar al método cuando sea necesario. Más adelante mediante el código ejemplo, se hace una explicación más clara acerca de estas configuraciones.

A continuación se realiza una explicación minuciosa de cómo se crean estos tipos de archivo y como se realiza el acople con los nuevos objetos creados en C++ (Ver los archivos *cbr.tcl* y *topologia.tcl* en el Anexo A). Como una ayuda para el entendimiento de este proceso, se creó una sección (2.5) acerca de los principales pasos para el Desarrollo de un Protocolo en *Tcl*.

En esta etapa se desarrolla el *script* de *Tcl* para una red *IntServ*, de una forma bastante sencilla de modo que estos pasos puedan ser útiles más adelante en el Desarrollo de un Protocolo en el simulador *NS-2*. El *script* de *Tcl* puede ser creado en cualquier editor de texto, dándole el nombre deseado por el usuario y guardándolo con extensión *.tcl*.

A. CREACIÓN DE LA TOPOLOGÍA

Para una mayor comprensión se crea una topología ejemplo en la cual se describen cada uno de los módulos ingresados. (Ver Figura 62)

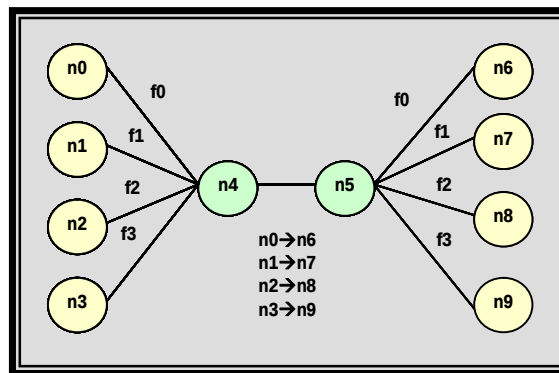


Figura 62. Topología Ejemplo para *IntServ*

La Programación en C++ de los nuevos objetos creados en el Simulador *NS-2*, se realiza dependiendo del tipo de topología que se desea usar. Es por esto que los nuevos módulos solo se deben utilizar con este mismo tipo de topología. En caso de querer cambiarla, se deben modificar los algoritmos de C++ se encuentran ubicados en la Tabla 9 del presente Proyecto.

De la anterior topología el nodo *n4* equivale al Nuevo Clasificador *IntServ* ingresado para esta simulación. El resto de nodos son los que trae el simulador por defecto sin ninguna modificación. El enlace entre los nodos *n4* y *n5* (*r* y *d*) equivale al Planificador *WFQ*.

El subprograma *topologia.tcl* invocado por el programa principal *cbr.tcl*, es utilizado para crear la topología de la simulación y configurar todos sus parámetros. El proceso comienza con la declaración del nombre del proceso y las variables globales utilizadas en el subprograma. A continuación aparece cada uno de los comandos utilizados para esto:

```
proc crear_topo {} {
    global ns n r d cl a b c
```

El segundo paso consiste en la inicialización de las variables globales utilizadas en el subprograma:

```
set a 1
set b 1
set c 0
```

Los valores de las anteriores variables dependen del Número de Flujos utilizados en la simulación, y el número de nodos de la topología.

La topología utilizada en esta simulación presenta dos nodos intermedios que siempre son constantes cuando se varían la cantidad de nodos y flujos del programa. Estos nodos fueron llamados *r* y *d*, donde *r* representa el nodo *IntServ*, y *d* representa un nodo normal del simulador. Estos dos nodos están enlazados por el Planificador *WFQ*, que es creado más adelante. A continuación se muestran las instrucciones que se utilizan para crear estos dos nodos intermedios y los nodos de origen y destino de la simulación:

```
// Creación de nodo r para IntServ y d como nodo IP normal
set r [$ns node]
set d [$ns node]
Creación de nodos de Origen y Destino
for {set i 0} {$i <= $a} {incr i} {
    set n($i) [$ns node]
}
```

El siguiente paso es el más importante en la configuración de la topología, pues es el acople del modulo de *IntServ* creado con el nodo *r*. Primero que todo se inicializa la variable *flujo_*, necesaria en el código de C++ (*ns-allinone-2.31/ns-2.31/tcl/lib/ns-default.tcl*). Luego se crea el nuevo Clasificador *IntServ* y se inserta al nodo *r* mediante el comando *insert-entry*. Cabe resaltar que este nuevo Clasificador hace parte del modulo de enrutamiento de tipo Base del Simulador *NS-2* y es ingresado en el slot 1. Finalmente se ingresa el comando de Control de Admisión del Clasificador *IntServ* con sus valores iniciales para llenar la Tabla Hash y la Tabla de Colisiones. A continuación se presenta el código del anterior procedimiento:

```

Classifier/IntServ set flujo_ $b // Fija Parámetro de Flujo
set cls [new Classifier/IntServ] // Creación del Clasificador IntServ
$r insert-entry RtModule/Base $cls "1" // Inserta Clasificador IntServ en el
nodo r
$cls add-num 0 $b // Ejecuta Control de Admisión e inicia Tabla Hash y
Tabla de Colisiones

```

Luego de configurar los nodos, se crean los enlaces con sus respectivos parámetros de retardo, ancho de banda y tipo de cola. En esta parte se hace uso de los bucles *for*, debido a que son demasiados y se optimizan más con estos ciclos. A continuación se muestra el código:

```

// Crea enlaces entre los nodos Origen y el nodo r
for {set i 0} {$i <= $c} {incr i} {
    $ns duplex-link $n($i) $r 5Mb 1ms DropTail
}
// Crea enlaces entre el nodo d y los nodos de Destino
for {set i $b} {$i <= $a} {incr i} {
    $ns duplex-link $n($i) $d 5Mb 1ms DropTail
}

```

Estos enlaces son creados entre los Host Origen y el nodo *r*, y los Host Destino y el nodo *d*. Además, con las siguientes instrucciones también se crea el enlace *WFQ* entre los nodos *r* y *d*, configurando también sus respectivos parámetros:

```

$ns simplex-link $r $d 2Mb 2ms WFQ
$ns simplex-link $d $r 2Mb 2ms DropTail

```

La configuración del enlace *WFQ* se realiza mediante dos enlaces simplex, debido a que las reservas solo se hacen en un solo sentido.

Por último se realiza la creación de un objeto tipo Clasificador que necesita el Planificador *WFQ* para clasificar los paquetes; este es instalado en el enlace entre *r* y *d*. A continuación se muestran las últimas líneas utilizadas para la configuración de la topología:

```

set cl [new WFQAggregClassifier]
$ns wfqclassifier-install $r $d $cl
}

```

B. CREACIÓN DEL PROGRAMA PRINCIPAL

Ahora se describirá el Programa principal en *Tcl*. Primero, debe declararse el subprograma que crea la topología y configura los parámetros de la misma. Esto se hace con la siguiente instrucción:

```
source topologia.tcl
```

En este caso, el subprograma que crea la topología fue llamado *topologia.tcl*; el comando *source* declara que este archivo puede ser llamado en cualquier momento del programa para crear la topología y configurar cada uno de los routers y enlaces de la misma. Además mediante este subprograma se fijan otros parámetros de la topología como son: tiempos, retardos, anchos de banda, tipos de colas, etc.

A continuación se declaran las variables globales a utilizar en el programa y en los subprogramas. Esto se hace mediante el comando *global* de la siguiente manera:

```
global ns n r d cl a b c reg delaywfq
```

El siguiente paso es fijar los valores iniciales a las variables utilizadas dentro del programa principal mediante el comando *set*. De esta forma es más fácil el cambio al iniciar el código en caso de querer cambiar los parámetros de simulación. El código utilizado es el siguiente:

```
set a 1  
set b 1  
set c 0
```

En las anteriores líneas, la variable *a* representa el número de nodos necesarios en la topología. Para simular esta red bajo diferentes parámetros fue necesario cambiar esta variable aumentándola exponencialmente desde 1 flujo hasta 800 flujos, mediante diferentes simulaciones. Esta variable también representa el último nodo de destino (equivalente al *n9* de la Figura 62) de la topología.

La variable *b* representa el Número de Flujos que tiene la red, y el primer nodo de destino (*n6* de la Figura 62). Y por último la variable *c* equivale al último nodo de origen (equivalente al *n3* de la Figura 62).

Las siguientes líneas son las utilizadas para borrar todas las cabeceras que tiene el simulador por defecto, y activar sólo las necesarias para nuestra simulación. En este caso sólo se añaden las cabeceras *IP* y *TCP*. A continuación se muestra las instrucciones necesarias para este proceso:

```
remove-all-packet-headers  
add-packet-header IP TCP
```

Al finalizar estos pasos, es necesario crear un Objeto Simulador, mediante el siguiente comando:

```
set ns [new Simulator]
```

Luego, se abre un archivo de escritura que se utiliza más adelante para trazar los datos en el *.nam*. Esto se realiza mediante las siguientes instrucciones:

```
set nam [ open out.nam w ]  
$ns namtrace-all $nam
```

La primera línea abre el archivo de escritura "*out.nam*", y se le da internamente el nombre de "*nam*" con el comando *set*. Y mediante la segunda línea se le dice al Objeto Simulador creado anteriormente que almacene todos los datos de la simulación en el archivo *nam*, para luego ser utilizados en la herramienta *nam*.

Las siguientes líneas son utilizadas para crear un archivo de escritura utilizado para monitorear el ancho de banda del enlace *WFQ*. Este proceso se realiza mediante la siguiente instrucción:

```
set reg [ open reg.tr w ]
```

A continuación se crean los archivos de salida donde se almacenan los datos de la simulación, para más adelante ser graficados mediante la herramienta *Xgraph* en forma individual. En estos archivos se guarda el ancho de banda de cada uno de los flujos y se crean mediante las siguientes instrucciones:

```
for {set i 0} {$i <= $b} {incr i} {  
    set f($i) [open fl($i).tr w]  
    $ns trace-all $f($i)  
}
```

En las anteriores instrucciones se utiliza un bucle *for* con la finalidad de optimizar el código y ahorrar instrucciones en el *script* de *Tcl*. Esto se realiza debido a que fue necesario simular la red para un número grande de flujos, y por lo tanto representaba un número grande de líneas de código. La primera línea crea los archivos de escritura y la segunda se encarga de trazar las gráficas. En este caso se crean (*b*+1) archivos para almacenar las trazas de la simulación. En cada archivo *fl(\$i)* se almacenan el tiempo y el ancho de banda de cada flujo y el archivo que se crea de más es utilizado para guardar los eventos de las trazas de la simulación.

El siguiente paso es crear un proceso *record* con el objetivo de leer el número de bytes recibidos en los sumideros de tráfico, calcular los anchos de banda de cada flujo y almacenarlos en los archivos de salida junto con el tiempo actual, para más adelante ser graficados. Luego se resetean los valores de *bytes_* de cada sumidero de tráfico. Cabe resaltar que un sumidero (*sink*) es un objeto que se encuentra ubicado en cada uno de los nodos de destino y se encarga de recibir el tráfico de paquetes y almacenar los anchos de banda de cada flujo para

graficarlos mediante la herramienta de *Xgraph*. A continuación se detallan las líneas de código de este proceso:

```

proc record {} {
    global sink ns f nam c b a reg
    set time 0.5
    for {set i 0} {$i <= $c} {incr i} {
        set bw($i) [$sink([expr $i+$b]) set bytes_ ]
    }
    set now [$ns now]
    for {set i 0} {$i <= $c} {incr i} {
        puts $f($i) "$now [expr $bw($i)/$time*8]"
    }
    for {set i 0} {$i <= $c} {incr i} {
        if {$now == 4.5} {
            puts $reg "[expr $bw($i)/$time*8]"
        } else {}
    }
    for {set i $b} {$i <= $a} {incr i} {
        $sink($i) set bytes_ 0
    }
    $ns at [expr $now+$time] "record"
}

```

Las dos primeras líneas del anterior proceso son la declaración de las variables globales utilizadas dentro del mismo y la fijación de la variable *time* en 0.5 que es el intervalo de tiempo necesario para calcular los anchos de banda y almacenar los resultados de la simulación en los archivos creados para esta tarea. Las siguientes líneas se realizan mediante unos bucles *for*, que ayudan a optimizar el código. El primer bucle *for*, se utiliza para analizar cuantos bytes han sido recibidos por los sumideros de tráfico en cada uno de los nodos de destino. En el segundo bucle se calcula el ancho de banda de cada uno de los *b* flujos y lo almacenan en los archivos creados al iniciar el programa. El tercer bucle representa las instrucciones que se utilizan para monitorear cada 4.5 segundos el ancho de banda en el enlace *WFQ* a la salida del Router. Y finalmente el cuarto bucle es necesario para resetear los bytes en los sumideros de tráfico. Por último, se presenta una línea de código que sirve para re-planificar el proceso que se acaba de describir, que llama nuevamente al proceso "*record*".

El siguiente proceso en el *script* es el *finish*, en el cual se cierran los archivos anteriormente creados y llenados, para finalmente ejecutar el *Xgraph*. Además se configura el tamaño de la grafica resultante y se realizan algunas sumatorias de resultados almacenados, indispensables más adelante para graficar los retardos de los paquetes. A continuación se muestran las líneas de código de este proceso:

```

proc finish {} {
    global ns f nam c fl reg delaywfq
    $ns flush-trace
    for {set i 0} {$i <= $c} {incr i} {
        close $f($i)
    }
    close $nam
    close $reg
    puts stdout "Retardo promedio de la cola wfq: [$delaywfq mean]"

    exec cat reg.tr | awk {{sum+=$1;print sum> "sbw1.tr"}}
    exec cat clintserv.tr | awk {{sum+=$i;print sum>
    "timeintserv1.tr"}}

    for {set i 0} {$i <= $c} {incr i} {
        exec xgraph fl($i).tr -geometry 800x400 -x Tiempo &
    }
    exec nam out.nam &
    exit 0
}

```

En las anteriores líneas resaltadas en negrita se encuentran tres comandos de linux llamados `exec`, `cat` y `awk`. El comando `awk` es un lenguaje de programación diseñado para procesar datos basados en texto, ya sean ficheros o flujos de datos. El comando `cat` sirve para mostrar un archivo y por último el comando `exec` realiza la ejecución de la línea. Para una mayor comprensión de estas líneas se hace una explicación detallada de cada una de ellas:

En la primera línea resaltada se crea un archivo llamado `reg.tr`, en el cual se guarda el ancho de banda de todos los flujos en un tiempo determinado, en este caso a los 4,5 segundos. Cabe resaltar que este proceso se realiza desde el "record". El comando `awk` crea una variable que se llama `sum`, en la cual se va guardando y sumando los datos del campo 1 (\$1), que esta en el archivo `reg.tr`. Finalmente se imprime (`print`) esta sumatoria en el archivo `sbw1.tr`, con el objetivo de que al final de la columna de este archivo se encuentre el ancho de banda total del enlace.

En la segunda línea resaltada, el comando `awk` crea una variable que se llama `sum`, en la cual se va guardando y sumando los datos del campo 1 (\$1), que esta en el archivo `clintserv.tr` (en este archivo se va grabando el retardo de cada paquete que pasa por el clasificador `IntServ`) y despues lo imprime (`print`) en el archivo `timeintserv1.tr`, con el fin de que al final de la columna de este archivo, este guardado la suma de todos los retardos de los paquetes. El archivo `clintserv.tr` se puede encontrar declarado en las líneas de código de C++ de los

archivos *classifier-IntServ.cc* y *classifier-IntServ.h* ubicados en las rutas descritas en la Tabla 9.

El paso siguiente en el programa principal es la creación de la topología, para esto se llama al proceso llamado *crear_topo*, el cual se invoca simplemente digitando su nombre y automáticamente se llama al subprograma *topologia.tcl*. Este código se describe más adelante.

crear_topo

Las siguientes líneas en el programa principal, son las encargadas de configurar los parámetros del Planificador *WFQ*. El comando *setqueue* le asigna una determinada cola a un flujo representado por su Identificación de Flujo. El comando *setweight* le asigna un peso a cada cola. Este peso puede oscilar desde 0.001 a 1, y la suma de todos los pesos de las colas deben sumar 1 que equivale al 100% del Ancho de Banda. Y por último el comando *setlength* le asigna el tamaño del paquete dentro del Planificador *WFQ*. A continuación se presenta la configuración de esta herramienta:

```
for {set i 0} {$i <= $c} {incr i} {  
    $cl setqueue $i [expr $i+$b]  
}  
for {set i 0} {$i <= $c} {incr i} {  
    $cl setweight [expr $i+$b] 0.01  
    $cl setlength [expr $i+$b] 5
```

Cabe resaltar que el objeto *cl* que corresponde al Planificador *WFQ*, es declarado en el subprograma *topologia.tcl*.

Luego se necesita configurar un Objeto para monitorear el comportamiento y retardos de las colas del Planificador *WFQ*. Para esto se necesita decidir que tipo de análisis se le quiere hacer a las colas del Planificador y asignarle un Objeto de muestreo para este monitoreo. Todas estas configuraciones se realizan mediante las siguientes instrucciones:

```
set monitorwfq [$ns monitor-queue $r $d stdout]  
set gpi [$monitorwfq get-pkts-integrator]  
set sam [new Samples]  
$monitorwfq set-delay-samples $sam  
set delaywfq [$monitorwfq get-delay-samples]
```

La herramienta *monitorwfq* es de fácil manejo y de una gran utilidad para monitorear un conjunto de paquetes, bytes, paquetes entrantes, paquetes salientes y paquetes perdidos. Esta también incluye un soporte para agregar estadísticas tales como tamaño promedio de cola, retardo promedio de cola, etc.

Para mayor información acerca de esta herramienta se puede remitir al manual del simulador *NS-2* que se encuentra en la página actual[62].

El siguiente paso es la creación de un agente *UDP* el cual es acoplado a cada uno de los Host origen y es el encargado de transportar los paquetes a través de la red, y luego se crea un agente generador de tráfico de tipo *CBR*, que es acoplado a su vez al agente *UDP*. El agente *CBR* tiene una transmisión de paquetes a una rata constante. A cada flujo se le asigna una Identificación de Flujo, que es utilizada en el Planificador *WFQ* para entregarle a cada paquete su reserva pactada en el Control de Admisión del Clasificador *IntServ*.

```
for {set i 0} {$i <= $c} {incr i} {
    set udp($i) [new Agent/UDP]
    $ns attach-agent $n($i) $udp($i)
    $udp($i) set fid_ $i
}
for {set i 0} {$i <= $c} {incr i} {
    set cbr($i) [new Application/Traffic/CBR]
    $cbr($i) attach-agent $udp($i)
}
```

Al finalizar la creación de los agentes de Tráfico, se crean los sumideros de tráfico (de tipo *LossMonitor*) y se acoplan a los nodos de destino, con la finalidad de monitorear los paquetes que llegan y guardar los resultados para luego graficarlos. La creación de estos sumideros se encuentra a continuación:

```
for {set i $b} {$i <= $a} {incr i} {
    set sink($i) [new Agent/LossMonitor]
    $ns attach-agent $n($i) $sink($i)
}
for {set i 0} {$i <= $c} {incr i} {
    $ns connect $udp($i) $sink([expr $i+$b])
}
```

Finalmente se fijan los tiempos de inicio y finalización de los procesos *record*, *finish* y los generadores de Tráfico:

```
$ns at 0.0 "record"

for {set i 0} {$i <= $c} {incr i} {
    $ns at 0.0 "cbr($i) start"
}$ns at 5.0 "finish"
```

Por ultimo se comienza la simulación con el siguiente comando:

```
$ns run
```

3.5.3.3. DISEÑO DE LA RED INTSERV6

Para las simulaciones de *IntServ6* se crean dos *scripts*. Inicialmente se encuentra un *script* con toda la configuración de la red, sus parámetros y además algunos procesos necesarios para utilizar herramientas como el *Nam* y el *Xgraph*. Y en segundo lugar, se crea el *script* de la topología incluyendo los nuevos objetos creados para una red *IntServ6*, los cuales se detallarán más adelante. En esta red se debe tener en cuenta que se utilizan dos tipos de objetos de C++ creados en el capítulo de Desarrollo del Protocolo en C++. El primer objeto de C++ es el adjuntado a los Host Origen de la topología, el cual es el encargado de acceder a la cabecera IP de cada paquete, extraer su Quíntupla y finalmente calcular el Número Hash para ser ingresado de nuevo en el campo de Etiqueta de Flujo de la cabecera IP.

El segundo objeto de C++, corresponde al Clasificador *IntServ6* ubicado en el router, el cual simplemente extrae el Número Hash de la Cabecera de cada paquete, y dependiendo el valor de este número clasifica el paquete teniendo en cuenta la reserva previamente establecida en el Control de Admisión.

En los programas *cbr6.tcl* y *topologia6.tcl*, se detallan los comandos necesarios para realizar una simulación de una red *IntServ6* con los nuevos objetos de C++ creados anteriormente. Estos archivos se encuentran en el Anexo A de este documento. A continuación se realiza una explicación paso a paso de la creación de los anteriores *scripts*:

A. CREACIÓN DE LA TOPOLOGÍA

Para una mayor comprensión se crea una topología ejemplo en la cual se describen cada uno de los módulos ingresados. (Ver Figura 63)

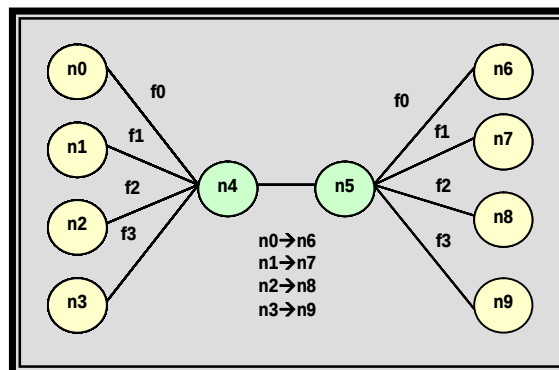


Figura 63. Topología Ejemplo para IntServ6

De la anterior topología el nodo *n4* equivale al Nuevo Clasificador *IntServ6* ingresado para esta simulación. Los nodos *n0* al nodo *n3*, equivalen a los módulos

que calculan el Número Hash del Flujo y lo ingresan a sus paquetes; el nombre utilizado para este modulo es *HostOrigen* y se invoca desde Tcl como un *Classifier/Start*. El resto de nodos son los que trae el simulador por defecto sin ninguna modificación. El enlace entre los nodos *n4* y *n5* (*r* y *d*) equivale al Planificador *WFQ*.

El subprograma *topologia6.tcl* tiene como objetivo crear la topología para una red *IntServ6* y configurar los parámetros de la misma. El primer paso de este *script* es la declaración del proceso que es llamado en el programa principal y de las variables globales utilizadas tanto en el programa principal como en el subprograma. Las líneas utilizadas son las siguientes:

```
proc crear_topo6 {} {  
    global ns n r d cl a b c
```

En las siguientes líneas se hace la inicialización de algunas variables utilizadas en este proceso, como se muestra a continuación:

```
set    a    1  
set    b    1  
set    c    0
```

Cabe resaltar que la inicialización de estas variables depende del Número de Flujos utilizados y del Número de nodos de la topología, por lo que estos valores son los mismos para *IntServ* cuando solo hay un flujo.

A continuación se crean los nodos intermedios de la topología llamados *r* y *d*, los cuales son constantes en el proceso de aumento de flujos y nodos para el análisis del comportamiento de la red. Cabe resaltar que el enlace entre estos dos nodos intermedios es donde se encuentra ubicado el Planificador *WFQ*. Los nodos de origen y los nodos de destino de la red son creados mediante un bucle *for*, debido a que estos varían dependiendo del número de flujos para los que se ejecute la simulación. Las líneas de código utilizadas para crear estos nodos son las siguientes:

```
//Creación de nodos r y d  
set r [$ns node]  
set d [$ns node]  
// Creación de nodos de origen y destino  
for {set i 0} {$i <= $a} {incr i} {  
    set n($i) [$ns node]  
}
```

El siguiente paso a seguir es la configuración de cada uno de los nodos. Los primeros nodos en configurarse son los nodos Host Origen, los cuales deben

calcular el Número Hash de cada flujo e ingresarlo a la cabecera de cada paquete. Para esto se creó un nuevo módulo *HostOrigen*, el cual es invocado como un clasificador de tipo *Start*. Este nuevo clasificador es ingresado a todos los nodos Origen mediante el comando *insert-entry* en el slot 1. Las siguientes líneas describen este proceso:

```
for {set i 0} {$i <= $a} {incr i} {
    set cls($i) [new Classifier/Start]
    $n($i) insert-entry RtModule/Base $cls($i) "1"
}
```

Después de configurar los nodos origen, se accede a crear un Clasificador de tipo *IntServ6* y acoplarlo al nodo *r*, que hace las veces de Router *IntServ6*. Además se inicializan algunas variables necesarias dentro del Clasificador *IntServ6* y se realiza el Control de Admisión y establecimiento de Tablas de Reservas antes de la transmisión de los paquetes. A continuación se describen las instrucciones que realizan este procedimiento:

```
Classifier/IntServ6 set flujoint_ $b //Fija el Número de Flujos del Clasificador
Classifier/IntServ6 set destinoint_ $b // Fija el nodo de destino inicial (nodo
n6 de la Figura 62)
set cl [new Classifier/IntServ6] // Se crea el Clasificador con nombre cl
$r insert-entry RtModule/Base $cl "2" // Se inserta el Clasificador en el nodo r
$cl adc-num 0 $b //Ejecuta el Control de Admisión y establece las Tablas
de Reservas
```

Luego de configurar los nodos, se crean los enlaces con sus respectivos parámetros de retardo, ancho de banda y tipo de cola. En esta parte se hace uso de los bucles *for*, debido a que son demasiados y se optimiza el código con estos ciclos. A continuación se muestra el código:

```
// Creación de enlaces entre el nodo r y los nodos de Origen
for {set i 0} {$i <= $c} {incr i} {
    $ns duplex-link $n($i) $r 5Mb 1ms DropTail
} // Creación de enlaces entre el nodo d y los nodos de Destino
for {set i $b} {$i <= $a} {incr i} {
    $ns duplex-link $n($i) $d 5Mb 1ms DropTail
}
```

Estos enlaces son creados entre los Host Origen y el nodo *r*, y los Host Destino y el nodo *d*. Además también se crea el enlace *WFQ* entre los nodos *r* y *d*, configurando también sus respectivos parámetros:

```
//Creación de enlace WFQ en sentido r→d
$ns simplex-link $r $d 2Mb 2ms WFQ
```

```
// Creación enlace DropTail en sentido d→r
$ns simplex-link $d $r 2Mb 2ms DropTail
```

Por ultimo se realiza la creación de un objeto tipo Clasificador que necesita el Planificador *WFQ* para clasificar los paquetes y es instalado en el enlace creado en el anterior procedimiento. A continuación se muestran las últimas líneas utilizadas para la configuración de la topología:

```
set cl [new WFQAggregClassifier]
$ns wfqclassifier-install $r $d $cl
}
```

B. CREACIÓN PROGRAMA PRINCIPAL

El programa principal se llama *cbr6.tcl*, y comienza con la declaración del subprograma y sus variables globales utilizadas en los dos programas:

```
source topologia6.tcl
global ns n r d cl a b c reg delaywfq
```

El siguiente paso es la inicialización de algunas variables utilizadas tanto en el programa principal, como en el subprograma de la topología.

```
set a 1
set b 1
set c 0
```

En las anteriores líneas, la variable *a* representa el número de nodos necesarios en la topología. Para simular esta red bajo diferentes parámetros fue necesario cambiar esta variable aumentándola exponencialmente desde 1 flujo hasta 1000 flujos, mediante diferentes simulaciones. Esta variable también representa el último nodo de destino (equivalente al *n9* de la Figura 63) de la topología.

La variable *b* representa el Número de Flujos que tiene la red, y el primer nodo de destino (*n6* de la Figura 63). Y por último la variable *c* equivale al último nodo de origen (equivalente al *n3* de la Figura 63).

Al igual que en *IntServ*, se remueven las cabeceras de los paquetes que tiene el simulador, y se activan solo las utilizadas en *IntServ6*. Esto se realiza mediante los siguientes comandos:

```
remove-all-packet-headers
add-packet-header IP TCP
```

Luego se crea un Objeto Simulador mediante el siguiente comando:

```
set ns [new Simulator]
```

Se crean los archivos de escritura que son utilizados más adelante para almacenar datos y finalmente ser graficados, ya sea por la herramienta *Nam* o *Xgraph*. A continuación se muestran las líneas de programación utilizadas:

```
set nam [open out.nam w]
$ns namtrace-all $nam
set reg [open reg.tr w]
for {set i 0} {$i <= $b} {incr i} {
    set f($i) [open fl($i).tr w]
    $ns trace-all $f($i)
}
```

El siguiente paso es la configuración del proceso *record*, el cual tiene como objetivo leer el número de bytes recibidos de los sumideros de tráfico. Además calcula los anchos de banda de cada uno de los flujos y los almacena en los archivos de salida junto con el tiempo actual, antes de resetear los valores de *bytes_* de los sumideros de tráfico. Por último se re-planifica el proceso *record*. Este procedimiento se realiza mediante las siguientes líneas de código:

```
proc record {} {
    global sink ns f nam c b a reg
    set time 0.5
    for {set i 0} {$i <= $c} {incr i} {
        set bw($i) [sink([expr $i+$b]) set bytes_]
    }
    set now [$ns now]
    for {set i 0} {$i <= $c} {incr i} {
        puts $f($i) "$now [expr $bw($i)/$time*8]"
    }
    for {set i 0} {$i <= $c} {incr i} {
        if {$now == 4.5} {
            puts $reg "[expr $bw($i)/$time*8]"
        } else { }
    }
    for {set i $b} {$i <= $a} {incr i} {
        $sink($i) set bytes_ 0
    }
    $ns at [expr $now+$time] "record"
}
```

El siguiente proceso que se programa es el *finish*, en este se cierran los archivos anteriormente creados y se ejecuta el *Xgraph*. Además se configura el tamaño de la grafica resultante y se realizan algunas sumatorias de resultados almacenados,

indispensables más adelante para graficar los retardos de los paquetes. A continuación se muestran las líneas de código de este proceso:

```

proc finish {} {
    global ns f nam c fl reg delaywfq
    $ns flush-trace
    for {set i 0} {$i <= $c} {incr i} {
        close $f($i)
    }
    close $nam
    puts stdout "Retardo Promedio de la cola WFQ: [$delaywfq mean]"

    exec cat reg.tr | awk {{sum+=$1;print sum}> "sbw1.tr"}}
    exec cat clintserv.tr | awk {{sum+=$1;print sum}>
    "timeintserv1.tr"}}

    for {set i 0} {$i <= $c} {incr i} {
        exec xgraph fl($i).tr -geometry 800x400 -x Tiempo &
    }
    exec nam out.nam &
    exit 0
}

```

En las anteriores líneas resaltadas en negrita se encuentran tres comandos de linux llamados `exec`, `cat` y `awk`. El comando `awk` es un lenguaje de programación diseñado para procesar datos basados en texto, ya sean ficheros o flujos de datos. El comando `cat` sirve para mostrar un archivo y por último el comando `exec` realiza la ejecución de la línea. Para una mayor comprensión de estas líneas se hace una explicación detallada de cada una de ellas:

En la primera línea resaltada se crea un archivo llamado `reg.tr`, en el cual se guarda el ancho de banda de todos los flujos en un tiempo determinado, en este caso a los 4,5 segundos. Cabe resaltar que este proceso se realiza desde el "record". El comando `awk` crea una variable que se llama `sum`, en la cual se va guardando y sumando los datos del campo 1 (`$1`), que esta en el archivo `reg.tr`. Finalmente se imprime (`print`) esta sumatoria en el archivo `sbw1.tr`, con el objetivo de que al final de la columna de este archivo se encuentre el ancho de banda total del enlace.

En la segunda línea resaltada, el comando `awk` crea una variable que se llama `sum`, en la cual se va guardando y sumando los datos del campo 1 (`$1`), que esta en el archivo `clintserv.tr` (en este archivo se va grabando el retardo de cada paquete que pasa por el Clasificador `IntServ6`) y despues lo imprime (`print`) en el archivo `timeintserv1.tr`, con el fin de que al final de la columna de este archivo, este guardado la suma de todos los retardos de los paquetes. El archivo

clintserv.tr se puede encontrar declarado en las líneas de código de C++ de los archivos *classifier-IntServ6.cc* y *classifier-IntServ6.h* ubicados en las rutas descritas en la Tabla 9.

La siguiente línea del programa principal, corresponde a la invocación del subprograma que se encarga de crear la topología y configura sus parámetros. Para realizar esto se digita la siguiente línea:

```
crear_topo6
```

Luego se configura el Planificador *WFQ* mediante 3 comandos diferentes. El primer comando es *setqueue*, el cual le asigna una cola a un determinado flujo. El segundo comando es *setweight* y le asigna un peso determinado a cada cola. Este peso puede oscilar entre 0.001 y 1, teniendo en cuenta que la suma de todos los flujos sea igual a 1 que equivale al 100% del ancho de banda del enlace *WFQ*. Y el tercer comando es *setlength* y es el encargado de asignarle el tamaño a cada paquete dentro del planificador *WFQ*. A continuación se muestran las líneas de código de esta configuración:

```
for {set i 0} {$i <= $c} {incr i} {  
    $cl setqueue $i [expr $i+$b]  
}  
for {set i 0} {$i <= $c} {incr i} {  
    $cl setweight [expr $i+$b] 0.01  
    $cl setlength [expr $i+$b] 5  
}
```

Después de configurar el Planificador *WFQ* se crea un objeto para monitorear las colas dentro de él, de forma que se pueda hacer un análisis de todos los paquetes entrantes y el retardo promedio de las colas. Para este análisis se utiliza un objeto de muestreo llamado *monitorqueue*, mediante el cual se calcula el retardo de la cola. Este procedimiento se realiza mediante las siguientes instrucciones:

```
set monitorwfq [$ns monitor-queue $r $d stdout]  
set gpi [$monitorwfq get-pkts-integrator]  
set sam [new Samples]  
$monitorwfq set-delay-samples $sam  
set delaywfq [$monitorwfq get-delay-samples]
```

El siguiente paso es la creación de un agente *UDP*, el cual es acoplado a cada uno de los Host origen, y luego se crea un agente generador de tráfico *CBR* que es acoplado a su vez al agente *UDP*. El agente *CBR* tiene una transmisión de paquetes a una rata constante. Cada flujo tiene una identificación de flujo que es utilizada en el Planificador *WFQ* para entregarle a cada paquete su reserva pactada en el Control de Admisión del Clasificador *IntServ6*.

```

for {set i 0} {$i <= $c} {incr i} {
    set udp($i) [new Agent/UDP]
    $ns attach-agent $n($i) $udp($i)
    $udp($i) set fid_ $i
}
for {set i 0} {$i <= $c} {incr i} {
    set cbr($i) [new Application/Traffic/CBR]
    $cbr($i) attach-agent $udp($i)
}

```

Cabe resaltar que el Agente *UDP* es el encargado de controlar el transporte de los datos a través de la red, y un sumidero los recibe en el Host Destino.

Al finalizar la creación de los agentes de tráfico, se procede a crear los sumideros en los nodos de destino encargados de monitorear los paquetes recibidos, para luego graficarlos:

```

for {set i $b} {$i <= $a} {incr i} {
    set sink($i) [new Agent/LossMonitor]
    $ns attach-agent $n($i) $sink($i)
}

```

Y al finalizar la creación de estos sumideros se acoplan a los agentes *UDP* generadores de tráfico mediante las siguientes instrucciones:

```

for {set i 0} {$i <= $c} {incr i} {
    $ns connect $udp($i) $sink([expr $i+$b])
}

```

Finalmente se fijan los tiempos de inicio y finalización de los procesos *"record"*, *"finish"* y los generadores de tráfico *CBR*. Las líneas utilizadas son las siguientes.

```

$ns at 0.0 "record"
for {set i 0} {$i <= $c} {incr i} {
    $ns at 0.0 "$cbr($i) start"
}
$ns at 5.0 "finish"

```

Y por último se digita el comando para correr la simulación:

```

$ns run

```

4. PRUEBAS REALIZADAS Y RESULTADOS

En esta etapa del proyecto se realizaron las pruebas a cada módulo creado tanto en C++ como en *Tcl*. De esta forma se verifica el funcionamiento de cada módulo por aparte antes de acoplarlos para la simulación final y el análisis del comportamiento de cada red (*IntServ*, *IntServ6*).

4.1. PRUEBAS REALIZADAS A LOS MÓDULOS DE C++

El proceso de pruebas en C++ comienza desde que se empieza a diseñar la estructura del código hasta el acople de la programación con el simulador *NS-2*. Es por esto que en este capítulo se muestran paso a paso el procedimiento utilizado para realizar estas pruebas.

En primer lugar, como se había mencionado en el Desarrollo del Protocolo en C++, se creó un módulo de C++ independiente del simulador, con el cual se pudo simular el proceso interno tanto del Clasificador *IntServ* como del Clasificador *IntServ6*. Al finalizar con la programación de cada módulo se procedió a compilarlos cada uno por aparte y detectar así los errores de sintaxis que presentaba el código.

En caso de que se detectaran errores se revisó nuevamente la programación para verificar y corregir estos errores, que por lo general son mostrados por el compilador dando la línea en donde se encuentra ubicado el error. Al comprobar finalmente que no tuviera ningún error de sintaxis, se procedió a ingresar algunas herramientas utilizadas en C++ para almacenar resultados de la programación en archivos de salida y para imprimir resultados en pantalla.

La primera herramienta utilizada fue la función *printf*, la cual permite visualizar datos formateados en pantalla, o en caso de tener algún error de salida imprime un valor negativo. La sintaxis que tiene esta función es la siguiente:

```
printf([OBJETO] FORMATO, VALORES);  
printf("La dirección IP Origen es: %d \n",src);
```

Donde la letra siguiente al símbolo % es el formato de la función *printf*. El formato es una cadena que tiene ciertas "expresiones" que describe como deberá ordenar y desplegar los valores que se ponen en *VALORES(src)*. En nuestro caso la cadena de formato es %d, la cual imprime un valor decimal. Existen otros tipos de formatos, pero solo fue necesario utilizar éste. La variable *src* equivale al valor o número decimal que se va a imprimir en pantalla.

Con esta herramienta se realizaron seguimientos a variables importantes para el programa, como los valores de la Quintupla, y el Número Hash calculado. Cabe

resaltar que para hacer uso de esta herramienta se deben incluir las librerías de C necesarias, que en este caso es *iostream.h*. Este archivo de cabecera contiene las facilidades standard de entrada y salida de C++. La línea de código indispensable para incluir estas librerías en el programa de C++ es la siguiente:

```
#include <iostream.h>
```

Esta línea es ingresada al principio de cada programa, en donde se desea utilizar la función *printf*.

La siguiente herramienta utilizada en la etapa de pruebas del proyecto fue el almacenamiento de datos obtenidos durante el programa en archivos *I/O*. En este caso se utilizaron archivos de escritura para imprimir resultados de cálculos y procesos realizados en la ejecución del programa.

Así como se incluye la librería para la función *printf*, de igual forma, cuando se desea usar archivos *I/O*, es necesario incluir la librería *fstream.h* al programa como se muestra a continuación:

```
#include <fstream.h>
```

La forma de utilizar esta herramienta es primero declarar un archivo como tipo *ofstream* dándole un nombre cualquiera y la extensión del archivo que se desea crear, y después utilizar este archivo un comando *cout*, pero saca los valores al archivo creado anteriormente. Las líneas utilizadas para este proceso son las siguientes:

```
ofstream sumatoria("Hash.xls"); // constructor de ofstream
```

Con el archivo anterior creado, se procede a utilizarlo para almacenar las variables que el usuario desee. Con un breve ejemplo se muestra la utilidad de esta herramienta:

```
archivo << hash_ << "\n";
int numero1, numero2, suma;
numero1 = 2;
numero2 = 5;

while (numero1<15) {
    suma = numero1 + numero2;
    sumatoria << suma << "\n";
    ++numero1;
}
```

Como resultado del ejemplo anterior se crea un archivo llamado *sumatoria*, el cual almacena la suma de las variables *numero1* y *numero2*. La cantidad de veces que

se realiza la suma, es la cantidad de números resultantes que son almacenados en el archivo *sumatoria*.

Esta función es utilizada en este proyecto para obtener los Números Hash de cada Clasificador en un archivo de salida, y verificar la cantidad de colisiones presentes en la red. Esto equivale a una ventaja en el momento de acoplar el código con el simulador *NS-2*, ya que se pueden comparar los resultados del módulo de C++ y los programas del *NS-2*, y deben coincidir para la verificación de la simulación de *NS-2*.

Durante el acoplamiento del módulo de C++ y los nuevos objetos creados en el simulador de *NS-2* se reutilizan las anteriores funciones para realizar las mismas pruebas del módulo, y verificar si los resultados de los Números Hash y las colisiones coinciden. De esta forma se comprueba que se llenan las Tabla Hash y la Tabla de Colisiones para cumplir con el Control de Admisión de cada uno de los clasificadores (*IntServ* e *IntServ6*).

4.2. PRUEBAS REALIZADAS A LOS SCRIPTS DE TCL

Esta etapa de pruebas se inicio con el estudio de los tutoriales de la herramienta que recomienda la página oficial de *NS-2* [62]. Durante el proceso de capacitación del Simulador *NS-2*, se encontro una herramienta que se utiliza para ver la configuración interna de cada uno de los objetos de la topología y que consiste en la digitación de un comando al final del *script* de *Tcl*. A continuación se muestra resaltada en negrita la línea de codigo utilizada para esta prueba en *Tcl*:

```
set ns [new Simulator]
set nf [open out.nam w]
ns namtrace-all $nf
proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exec nam out.nam &
    exit 0
}
set n0 [$ns node]
set n1 [$ns node]
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
$ns at 5.0 "finish"
$ns gen-map
$ns run
```

Finalmente al ejecutar el *script* desde un terminal se obtiene el siguiente resultado:

```
Node_o10(id 0)
  rtsize_0()
  ptnotif_o11(RtModule/Base)
  dmux_(NULL_OBJECT)
  rtnotif_o11(RtModule/Base)
  mod_assoc_o12_o11
  nodetype_(NULL_OBJECT)
  reg_module_Base_o11
```

```

classifier__o12(Classifier/Hash/Dest)
address_0()
multiPath_0()
Link _o17, fromNode__o10(id 0) -> toNode__o13(id 1)
Components (in order) head first
_o19 Connector
_o22 Trace/Enque
_o16 Queue/DropTail
_o23 Trace/Deque
_o20 DelayLink
_o21 TTLChecker
_o25 Trace/Recv
---
Node _o13(id 1)
rsize_0()
ptnotif__o14(RtModule/Base)
dmux_(NULL_OBJECT)
rtnotif__o14(RtModule/Base)
mod_assoc__ _o15_o14
nodetype_(NULL_OBJECT)
reg_module__ Base_o14
classifier__o15(Classifier/Hash/Dest)
address_1()
multiPath_0()
Link _o27, fromNode__o13(id 1) -> toNode__o10(id 0)
Components (in order) head first
_o29 Connector
_o32 Trace/Enque
_o26 Queue/DropTail
_o33 Trace/Deque
_o30 DelayLink
_o31 TTLChecker
_o35 Trace/Recv
---

```

De esta manera vemos como se van creando los objetos dentro de la simulación, como son: nodos, enlaces, colas, módulos de enrutamiento, retardos de enlaces, clasificadores, entre otros.

Además, con esta herramienta se pudo observar que el simulador *NS-2* tiene por defecto un tipo de Clasificador en los nodos (*Classifier/Hash/Dest*) y un Planificador de Colas (*Queue/DropTail*) en los enlaces, lo que nos impulsó a encontrar la forma de modificar estos parámetros del simulador. Mediante la capacitación del Simulador *NS-2* se encontró una herramienta, la cual inserta un nuevo módulo Clasificador y lo enlaza al Clasificador que tiene por defecto el nodo. Las líneas de código utilizadas para realizar este proceso se encuentra en el siguiente código resaltadas en negrita:

```

set ns [new Simulator]
set nf [open out.nam w]
$ns namtrace-all $nf
proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exec nam out.nam &
    exit 0
}
set n0 [$ns node]
set n1 [$ns node]
Classifier/Hash set default 1
set cls [new Classifier/Hash/SrcDestFid 33]

```

```

$n0 insert-entry RtModule/Base $cls "1"
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
$ns at 5.0 "finish"
$ns gen-map
$ns run

```

La primera línea resaltada en negrita hace referencia a la inicialización de la variable *default_* del Clasificador Hash en 1, ya que ésta por defecto está declarada en -1 en el archivo *ns.default.tcl*, el cual se encuentra ubicado en la ruta *ns-allinone-2.31/ns-2.31/tcl/lib/*. La segunda línea resaltada crea un objeto Clasificador de tipo *Hash/SrcDestFid*, el cual tiene el Simulador NS-2 y utiliza la Dirección IP Origen, la Dirección IP Destino y la Identificación de Flujo para clasificar cada paquete. Finalmente, la última línea resaltada inserta el nuevo clasificador Hash declarado en la anterior línea, en el nodo 0 mediante el comando *insert-entry*, y es ingresado como un módulo de enrutamiento de tipo Base *RtModule/Base* en el slot "1".

Luego se ejecuta el *script* desde un terminal y el obtiene el siguiente resultado:

```

Node _o10(id 0)
#El objeto _o16 es Classifier/Hash/SrcDestFid y el _o12 Classifier/Hash/Dest
  hook_assoc_ _o16 _o12
  rtsize_0()
  ptnotif__o11(RtModule/Base)
  dmux_(NULL_OBJECT)
  rtnotif__o11(RtModule/Base)
  mod_assoc_ _o16 RtModule/Base _o12 _o11
  nodetype_(NULL_OBJECT)
  reg_module_ Base _o11
  classifier__o16(Classifier/Hash/SrcDestFid)
  address_0()
  multiPath_0()
Link _o18, fromNode__o10(id 0) -> toNode__o13(id 1)
Components (in order) head first
  _o20 Connector
  _o23 Trace/Enque
  _o17 Queue/DropTail
  _o24 Trace/Deque
  _o21 DelayLink
  _o22 TTLChecker
  _o26 Trace/Recv
---
Node _o13(id 1)
  rtsize_0()
  ptnotif__o14(RtModule/Base)
  dmux_(NULL_OBJECT)
  rtnotif__o14(RtModule/Base)
  mod_assoc_ _o15 _o14
  nodetype_(NULL_OBJECT)
  reg_module_ Base _o14
  classifier__o15(Classifier/Hash/Dest)
  address_1()
  multiPath_0()
Link _o28, fromNode__o13(id 1) -> toNode__o10(id 0)
Components (in order) head first
  _o30 Connector
  _o33 Trace/Enque
  _o27 Queue/DropTail
  _o34 Trace/Deque
  _o31 DelayLink

```

```

_o32  TTLChecker
_o36  Trace/Recv
---
```

De esta manera se verifica que el Clasificador *Hash/SrcDestFid* se ha instalado correctamente en el nodo 0. El nodo donde se inserta el nuevo Clasificador puede variar dependiendo del diseño de topología que desea el usuario. Cabe resaltar que el nuevo Clasificador ingresado queda enlazado con el Clasificador que tiene por defecto el nodo, de modo que no se modifica la estructura interna del nodo, simplemente se añade el nuevo módulo creado.

Teniendo en cuenta la herramienta anterior se realizó la misma prueba pero ingresando el nuevo Clasificador *IntServ* diseñado en este proyecto. El siguiente *script* muestra las líneas de código necesarias para este proceso:

```

set ns [new Simulator]
set nf [open out.nam w]
$ns namtrace-all $nf
proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exec nam out.nam &
    exit 0
}
set n0 [$ns node]
set n1 [$ns node]
#Se inicializa la variable flujo_ con el número de flujos que tiene la simulación.
Classifier/IntServ set flujo_ 1
#Se crea el objeto cls que representa el nuevo Clasificador IntServ
set cls [new Classifier/IntServ]
#Se inserta el Clasificador IntServ en el nodo 0, en el slot "1".
$n0 insert-entry RtModule/Base $cls "1"
#Mediante la siguiente línea se realiza el control de admisión de la topología creada.
$cls add-num 0 1
# La Dirección IP Origen es 0 y la Dirección IP Destino es 1.
$ns duplex-link $n0 $n1 1Mb 10ms DropTail
$ns at 5.0 "finish"
$ns gen-map
$ns run
```

Se ejecuta el *script* desde un terminal y se obtiene el siguiente resultado:

```

Node _o10(id 0)
#El objeto _o16 es Classifier/IntServ y el _o12 Classifier/Hash/Dest
hook_assoc_ _o16 _o12
rtsize_0()
ptnotif__o11(RtModule/Base)
dmux_(NULL_OBJECT)
rtnotif__o11(RtModule/Base)
mod_assoc_ _o16 RtModule/Base _o12 _o11
nodetype_(NULL_OBJECT)
reg_module_ Base _o11
classifier__o16(Classifier/IntServ)
address_0()
multiPath_0()
Link _o18, fromNode__o10(id 0) -> toNode__o13(id 1)
Components (in order) head first
_o20 Connector
_o23 Trace/Enque
```

```

_o17 Queue/DropTail
_o24 Trace/Dequeue
_o21 DelayLink
_o22 TTLChecker
_o26 Trace/Recv
---
Node _o13(id 1)
  rsize_0()
  ptnotif__o14(RtModule/Base)
  dmux_(NULL_OBJECT)
  rtnotif__o14(RtModule/Base)
  mod_assoc__ _o15 _o14
  nodetype_(NULL_OBJECT)
  reg_module_ Base_o14
  classifier__o15(Classifier/Hash/Dest)
  address_1()
  multiPath_0()
Link _o28, fromNode__o13(id 1) -> toNode__o10(id 0)
Components (in order) head first
_o30 Connector
_o33 Trace/Enqueue
_o27 Queue/DropTail
_o34 Trace/Dequeue
_o31 DelayLink
_o32 TTLChecker
_o36 Trace/Recv

```

De esta forma queda instalado el nuevo Clasificador *IntServ* en el nodo que se destinó para esta tarea.

Ya instalado el Clasificador en el nodo, el siguiente paso consistió en el estudio e implementación del Planificador *WFQ* mencionado anteriormente en la topología utilizada para estas pruebas. En el siguiente *script* se utiliza esta contribución hecha al Simulador *NS-2* para la planificación y gestión de colas, con el objetivo de analizar el Planificador por separado y verificar si la asignación de los pesos a las colas coinciden en los resultados finales:

```

set ns [new Simulator]
set f [open out.tr w]
$ns trace-all $f
set nam [open out.nam w]
$ns namtrace-all $nam
# Creación de los nodos
set n0 [$ns node]
set n1 [$ns node]
set r [$ns node]
set d [$ns node]
$ns color 0 blue
$ns color 1 red
# Creación de los enlaces
$ns duplex-link $n0 $r 5Mb 1ms DropTail
$ns duplex-link $n1 $r 5Mb 1ms DropTail
# Instalación de un enlace simple con el Planificador WFQ entre los nodos r y d. En caso de que se devuelvan
paquetes es usada una cola FIFO
$ns simplex-link $r $d 2Mb 2ms WFQ
$ns simplex-link $d $r 1Mb 2ms DropTail
$ns duplex-link-op $n0 $r orient right-down
$ns duplex-link-op $n1 $r orient right-up
$ns simplex-link-op $r $d orient right
$ns simplex-link-op $d $r orient left
#Se crea el nuevo objeto Clasificador del Planificador WFQ
set cl [new WFQAggregClassifier]

```

```

#Se instala el nuevo objeto creado en el enlace entre los nodos r y d.
$ns wfqclassifier-install $r $d $cl
#Se le asigna una cola a cada Identificación de Flujo
$cl setqueue 0 3;
$cl setqueue 1 4;
#Se le asignan los pesos y el tamaño de los paquetes a cada una de las colas
$cl setweight 3 0.5;
$cl setlength 3 5;
$cl setweight 4 0.5;
$cl setlength 4 5;
#Creación de los Agentes de Tráfico
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
$udp0 set fid_ 0; # this is the info used by WFQ classifier
set cbr0 [new Application/Traffic/CBR]
$cbr0 set rate_ 1.1Mb
$cbr0 set packetSize_ 100
$cbr0 attach-agent $udp0
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
$udp1 set fid_ 1; # this is the info used by WFQ classifier
set cbr1 [new Application/Traffic/CBR]
$cbr1 set rate_ 1.1Mb
$cbr1 set packetSize_ 200
$cbr1 attach-agent $udp1
set null [new Agent/Null]
$ns attach-agent $d $null
$ns connect $udp0 $null
$ns connect $udp1 $null
$ns at 0.0 "$cbr0 start"
$ns at 0.1 "$cbr1 start"
#Se modifican los pesos de las colas en el primer segundo de la simulación
$ns at 1 "$cl setweight 3 0.8; $cl setweight 4 0.2"
$ns at 2.0 "finish"
proc finish {} {
    global ns f nam
    $ns flush-trace
    close $f
    close $nam
    exit 0
}
$ns gen-map
$ns run

```

Se ejecuta el script desde un terminal y se obtiene el siguiente resultado:

```

Node _o10(id 0)
  rsize_1()
  ptnotif__o11(RtModule/Base)
  dmux__o84(Classifier/Port)
  rtnotif__o11(RtModule/Base)
  mod_assoc__o12_o11
  nodetype_(NULL_OBJECT)
  reg_module_ Base_o11
  classifier__o12(Classifier/Hash/Dest)
  address_0()
  multiPath_0()
  Agents at node (possibly in order of creation):
    _o83 Agent/UDP dst-addr/port: 3/0
  Link _o23, fromNode__o10(id 0) -> toNode__o16(id 2)
  Components (in order) head first
    _o25 Connector
    _o28 Trace/Enque
    _o22 Queue/DropTail
    _o29 Trace/Deque
    _o26 DelayLink

```

```

        _o27  TTLChecker
        _o31  Trace/Recv
    ---
Node _o13(id 1)
    rsize_1()
    ptnotif__o14(RtModule/Base)
    dmux__o87(Classifier/Port)
    rtnotif__o14(RtModule/Base)
    mod_assoc__ _o15_o14
    nodetype_(NULL_OBJECT)
    reg_module_ Base_o14
    classifier__o15(Classifier/Hash/Dest)
    address_1()
    multiPath_0()
    Agents at node (possibly in order of creation):
        _o86  Agent/UDP          dst-addr/port: 3/0
    Link _o43, fromNode__o13(id 1) -> toNode__o16(id 2)
    Components (in order) head first
        _o45  Connector
        _o48  Trace/Enque
        _o42  Queue/DropTail
        _o49  Trace/Deque
        _o46  DelayLink
        _o47  TTLChecker
        _o51  Trace/Recv
    ---
Node _o16(id 2)
    rsize_0()
    ptnotif__o17(RtModule/Base)
    dmux_(NULL_OBJECT)
    rtnotif__o17(RtModule/Base)
    mod_assoc__ _o18_o17
    nodetype_(NULL_OBJECT)
    reg_module_ Base_o17
    classifier__o18(Classifier/Hash/Dest)
    address_2()
    multiPath_0()
    Link _o33, fromNode__o16(id 2) -> toNode__o10(id 0)
    Components (in order) head first
        _o35  Connector
        _o38  Trace/Enque
        _o32  Queue/DropTail
        _o39  Trace/Deque
        _o36  DelayLink
        _o37  TTLChecker
        _o41  Trace/Recv
    Link _o53, fromNode__o16(id 2) -> toNode__o13(id 1)
    Components (in order) head first
        _o55  Connector
        _o58  Trace/Enque
        _o52  Queue/DropTail
        _o59  Trace/Deque
        _o56  DelayLink
        _o57  TTLChecker
        _o61  Trace/Recv
    Link _o63, fromNode__o16(id 2) -> toNode__o19(id 3)
    Components (in order) head first
        _o65  Connector
        _o68  Trace/Enque
    #El objeto se instalo en el enlace satisfactoriamente.
        _o62  Queue/WFQ
        _o69  Trace/Deque
        _o66  DelayLink
        _o67  TTLChecker
        _o71  Trace/Recv
    ---
Node _o19(id 3)

```

```

    rtsize_1()
    ptnotif__o20(RtModule/Base)
    dmux__o90(Classifier/Port)
    rtnotif__o20(RtModule/Base)
    mod_assoc__o21_o20
    nodetype_(NULL_OBJECT)
    reg_module__Base_o20
    classifier__o21(Classifier/Hash/Dest)
    address_3()
    multiPath_0()
    Agents at node (possibly in order of creation):
    _o89 Agent/Null dst-addr/port: 1/0
    Link _o73, fromNode__o19(id 3) -> toNode__o16(id 2)
    Components (in order) head first
    _o75 Connector
    _o78 Trace/Enque
    _o72 Queue/DropTail
    _o79 Trace/Deque
    _o76 DelayLink
    _o77 TTLChecker
    _o81 Trace/Recv
    ---

```

Instalado el Planificador *WFQ*, se hacen pruebas individuales para ver su correcto funcionamiento.

Con el siguiente *script* se grafican con la herramienta *xgraph* el ancho de banda de cada uno de los flujos en el enlace *WFQ*. Las líneas resaltadas en **negrita** corresponden a las nuevas líneas de código utilizadas para graficar estos anchos de banda con el *Xgraph*:

```

set ns [new Simulator]
set nam [open out.nam w]
$ns namtrace-all $nam

set f0 [open out0.tr w]
set f1 [open out1.tr w]
set f2 [open out2.tr w]

set n0 [$ns node]
set n1 [$ns node]
set r [$ns node]
set d [$ns node]
set u1 [$ns node]
set u2 [$ns node]

$ns duplex-link $n0 $r 5Mb 1ms DropTail
$ns duplex-link $n1 $r 5Mb 1ms DropTail
$ns simplex-link $r $d 2Mb 2ms WFQ
$ns simplex-link $d $r 2Mb 2ms DropTail
$ns duplex-link $d $u1 5Mb 1ms DropTail
$ns duplex-link $d $u2 5Mb 1ms DropTail

set cl [new WFQAggregClassifier]
$ns wfqclassifier-install $r $d $cl

$cl setqueue 0 3; # fid 0
$cl setqueue 1 4; # fid 1

$cl setweight 3 0.7;
$cl setlength 3 5;
$cl setweight 4 0.3;
$cl setlength 4 5;

```

```

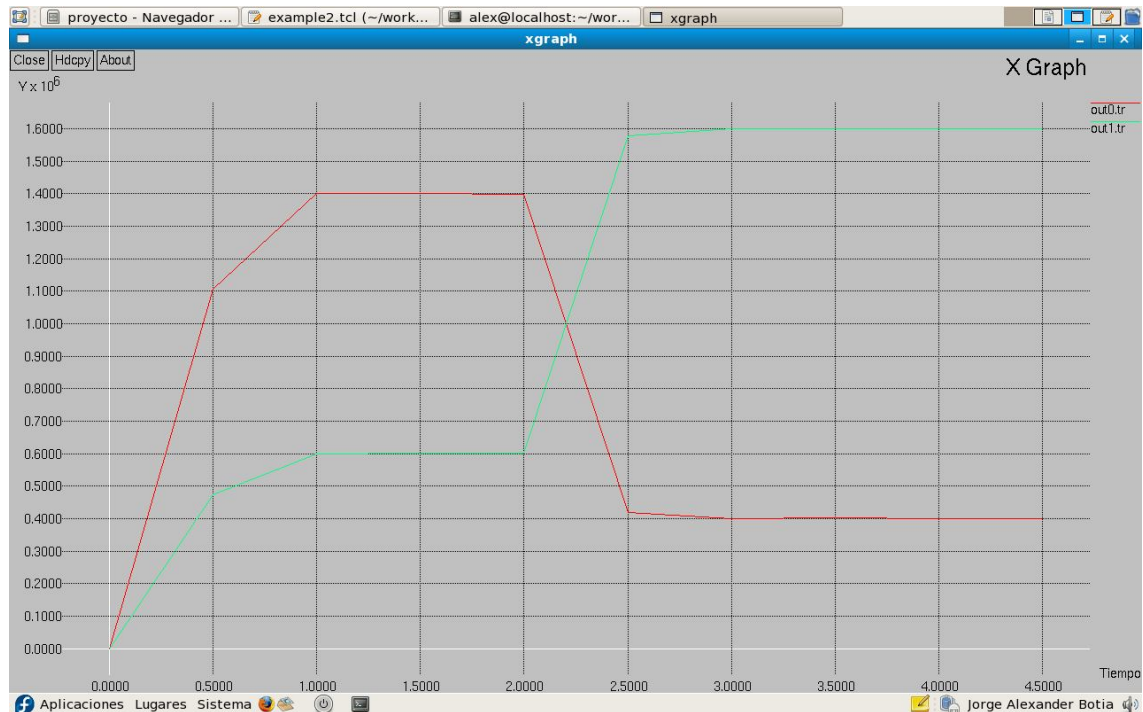
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
$udp0 set fid_0 # this is the info used by WFQ classifier
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set rate_ 10Mb
$cbr0 set PacketSize_ 100
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
$udp1 set fid_1 # this is the info used by WFQ classifier
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
$cbr1 set rate_ 5Mb
$cbr1 set PacketSize_ 100

#Crea los agentes de monitoreo en los nodos destino.
set sink0 [new Agent/LossMonitor]
$ns attach-agent $u1 $sink0
set sink1 [new Agent/LossMonitor]
$ns attach-agent $u2 $sink1
set sink2 [new Agent/LossMonitor]
$ns attach-agent $d $sink2
$ns connect $udp0 $sink0
$ns connect $udp1 $sink1

$ns at 0.0 "record"
$ns at 0.1 "$cbr0 start"
$ns at 0.1 "$cbr1 start"
$ns at 2 "$cl setweight 3 0.2; $cl setweight 4 0.8"
$ns at 5 "finish"
#Procedimiento para grabar y calcular el ancho de banda del enlace.
proc record {} {
    global sink0 sink1 ns f0 f1 f2 nam
    set time 0.5
    set bw0 [$sink0 set bytes_]
    set bw1 [$sink1 set bytes_]
    set now [$ns now]
    puts $f0 "$now [expr $bw0/$time*8]"
    puts $f1 "$now [expr $bw1/$time*8]"
    $sink0 set bytes_ 0
    $sink1 set bytes_ 0
    $ns at [expr $now+$time] "record"
}
proc finish {} {
    global ns f0 f1 f2 nam
    $ns flush-trace
    close $f0
    close $f1
    close $f2
    close $nam
    puts " Ejecutando simulación y grafica..."
    exec xgraph out0.tr out1.tr out2.tr -geometry 800x400 -x Tiempo &
    exit 0
}
$ns run

```

Se ejecuta el script desde un terminal y se obtiene la siguiente gráfica de anchos de banda:



El ancho de banda total del enlace *WFQ* es de 2Mb. Al flujo 0 se le asignó un peso de 0,7 y al flujo 1 se le asignó 0,3

En la grafica se puede apreciar que después de 2 segundos de simulación se cambian los pesos de las colas quedando el flujo 0 con un peso de 0,2 y el flujo 1 con un peso de 0.8. También se puede apreciar gráficamente que la suma de los dos anchos de banda no excede los 2Mb, comprobando así que la herramienta *WFQ* funciona correctamente y puede ser implementado en este proyecto.

Para seguir verificando el correcto funcionamiento del Planificador *WFQ* se realizó una segunda prueba la cual se explica a continuación:

Con el siguiente *script* se analizó el comportamiento de una red *IntServ* con un Planificador *WFQ* y tráfico *CBR*. Durante la ejecución de este script se le modificó la velocidad de transmisión de paquetes al generador de tráfico *CBR*, con el objetivo de observar y analizar la respuesta del Planificador *WFQ* al cambio de velocidad. Las líneas de código que se utilizaron para este proceso se encuentran resaltadas en amarillo:

```
set ns [new Simulator]
set nam [open out.nam w]
$ns namtrace-all $nam

set f0 [open out0.tr w]
set f1 [open out1.tr w]
```

```

set n0 [$ns node]
set n1 [$ns node]
set r [$ns node]
set d [$ns node]
set n2 [$ns node]
set n3 [$ns node]

Classifier/IntServ set flujo_2
set cls [new Classifier/IntServ]
$r insert-entry RtModule/Base $cls "1"
$cls add-num 0 2

$ns duplex-link $n0 $r 5Mb 1ms DropTail
$ns duplex-link $n1 $r 5Mb 1ms DropTail
$ns simplex-link $r $d 2Mb 2ms WFQ
$ns simplex-link $d $r 2Mb 2ms DropTail
$ns duplex-link $d $n2 5Mb 1ms DropTail
$ns duplex-link $d $n3 5Mb 1ms DropTail

set cl [new WFQAggregClassifier]
$ns wfqclassifier-install $r $d $cl

$cl setqueue 0 3;
$cl setqueue 1 4;
$cl setweight 3 0.8;
$cl setlength 3 5;
$cl setweight 4 0.2;
$cl setlength 4 5;

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
$udp0 set fid_ 0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set rate_ 0.7Mb
$cbr0 set PacketSize_ 100

set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
$udp1 set fid_ 1
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
$cbr1 set rate_ 0.3Mb
$cbr1 set PacketSize_ 100

set sink0 [new Agent/LossMonitor]
$ns attach-agent $n2 $sink0

set sink1 [new Agent/LossMonitor]
$ns attach-agent $n3 $sink1

$ns connect $udp0 $sink0
$ns connect $udp1 $sink1

$ns at 0.0 "record"
$ns at 0.1 "$cbr0 start"
$ns at 0.3 "$cbr1 start"
$ns at 2 "$cbr0 set rate_ 1.2Mb; $cbr1 set rate_ 0.5Mb"
$ns at 5 "finish"

proc record {} {
    global sink0 sink1 ns f0 f1 nam
    set time 0.1
    set bw0 [$sink0 set bytes_]
    set bw1 [$sink1 set bytes_]

```

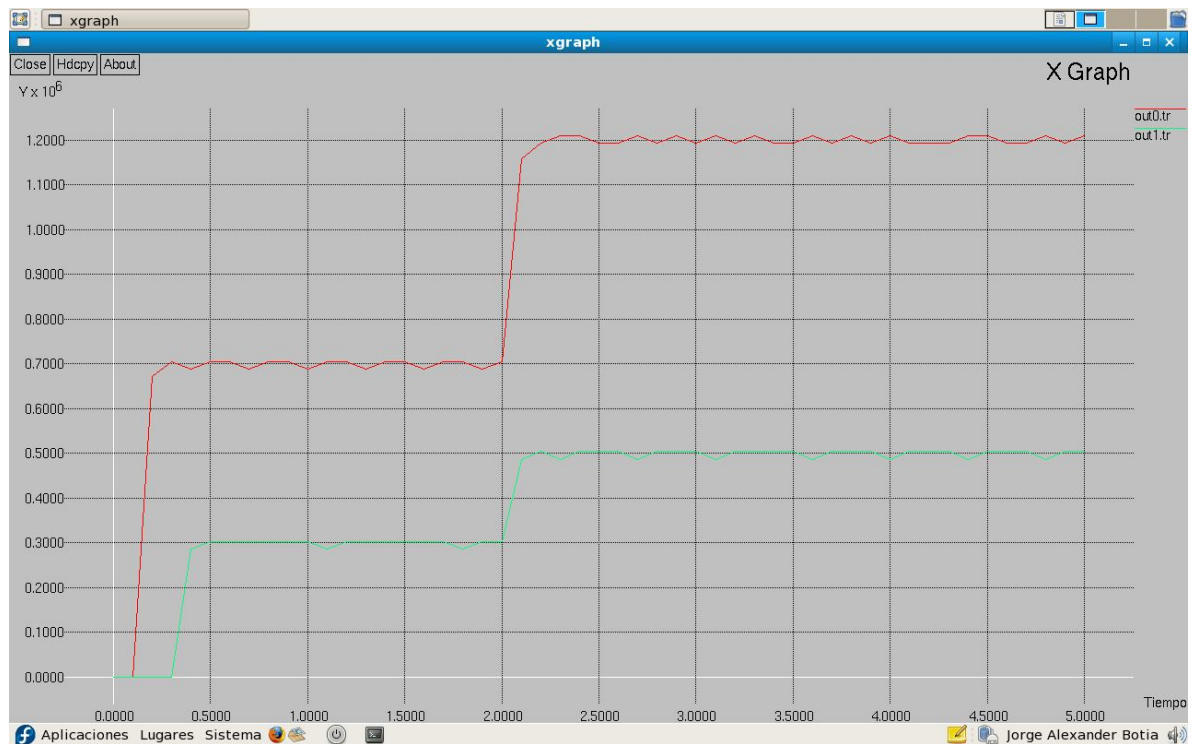
```

set now [$ns now]
puts $f0 "$now [expr $bw0/$time*8]"
puts $f1 "$now [expr $bw1/$time*8]"
$sink0 set bytes_ 0
$sink1 set bytes_ 0
$ns at [expr $now+$time] "record"
}

proc finish {} {
    global ns f0 f1 nam
    $ns flush-trace
    close $f0
    close $f1
    close $nam
    puts " Ejecutando simulación y grafica..."
    exec xgraph out0.tr out1.tr -geometry 800x400 -x Tiempo &
    exit 0
}
$ns run

```

Al ejecutar el script anterior se obtuvo la siguiente gráfica:



Como se puede apreciar, el flujo 0 esta de color rojo y el flujo 1 esta de color verde.

Al flujo 0, desde el inicio de la simulación hasta los dos segundos, se le asigna un ancho de banda que corresponde a la rata 0,7Mb (si esta rata excede el ancho de banda del enlace, que en este caso es de 2Mb, la asignación de la reserva se la

hace el Planificador *WFQ*, entregandole el porcentaje equivalente al peso de esa cola, que es de 0,8).

Al flujo 1, desde el inicio de la simulación hasta los dos segundos, se le asigna un ancho de banda que corresponde a la rata 0,3Mb (si esta rata excede el ancho de banda del enlace, que en este caso es de 2Mb, la asignación de la reserva se la hace el Planificador *WFQ*, entregandole el porcentaje equivalente al peso de esa cola, que es de 0,2).

Al llegar al cambio de velocidades de los flujos, al flujo 0 se le asigna un ancho de banda de 1,2Mb (si esta rata excede el ancho de banda del enlace, que en este caso es de 2Mb, la asignación de la reserva se la hace el Planificador *WFQ*, entregandole el porcentaje equivalente al peso de esa cola, que es de 0,8).

Y por último al realizar el cambio de velocidades de transmisión, al flujo 1 se le asigna un ancho de banda de 0,5Mb (si esta rata excede el ancho de banda del enlace, que en este caso es de 2Mb, la asignación de la reserva se la hace el Planificador *WFQ*, entregandole el porcentaje equivalente al peso de esa cola, que es de 0,2).

Esta última prueba se le realizó también para *IntServ6*, quedando el siguiente script.

#Las líneas resaltadas en amarillo corresponden a los cambios hechos para IntServ6.

```
set ns [new Simulator]

set nam [open out.nam w]
$ns namtrace-all $nam

set f2 [open out2.tr w]
set f3 [open out3.tr w]

set n0 [$ns node]
set n1 [$ns node]
set r [$ns node]
set d [$ns node]
set n2 [$ns node]
set n3 [$ns node]

set cls0 [new Classifier/Start]
$ns0 insert-entry RtModule/Base $cls0 "1"
set cls1 [new Classifier/Start]
$ns1 insert-entry RtModule/Base $cls1 "1"

Classifier/IntServ6 set flujoint_ 2
Classifier/IntServ6 set destinoint_ 2
set cl [new Classifier/IntServ6]
$ns insert-entry RtModule/Base $cl "2"
$cl adc-num 0 2

$ns duplex-link $ns0 $r 5Mb 1ms DropTail
$ns duplex-link $ns1 $r 5Mb 1ms DropTail
$ns simplex-link $ns0 $d 2Mb 2ms WFQ
$ns simplex-link $ns1 $d 2Mb 2ms DropTail
$ns duplex-link $ns2 $n2 5Mb 1ms DropTail
```

```

$ns duplex-link $d $n3 5Mb 1ms DropTail

set cl [new WFQAggregClassifier]
$ns wfqclassifier-install $r $d $cl
$cl setqueue 0 3;
$cl setqueue 1 4;
$cl setweight 3 0.8;
$cl setlength 3 5;
$cl setweight 4 0.2;
$cl setlength 4 5;

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
$udp0 set fid_ 0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set rate_ 0.7Mb
$cbr0 set PacketSize_ 100

set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
$udp1 set fid_ 1
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
$cbr1 set rate_ 0.3Mb
$cbr1 set PacketSize_ 100

set sink0 [new Agent/LossMonitor]
$ns attach-agent $n2 $sink0

set sink1 [new Agent/LossMonitor]
$ns attach-agent $n3 $sink1

$ns connect $udp0 $sink0
$ns connect $udp1 $sink1

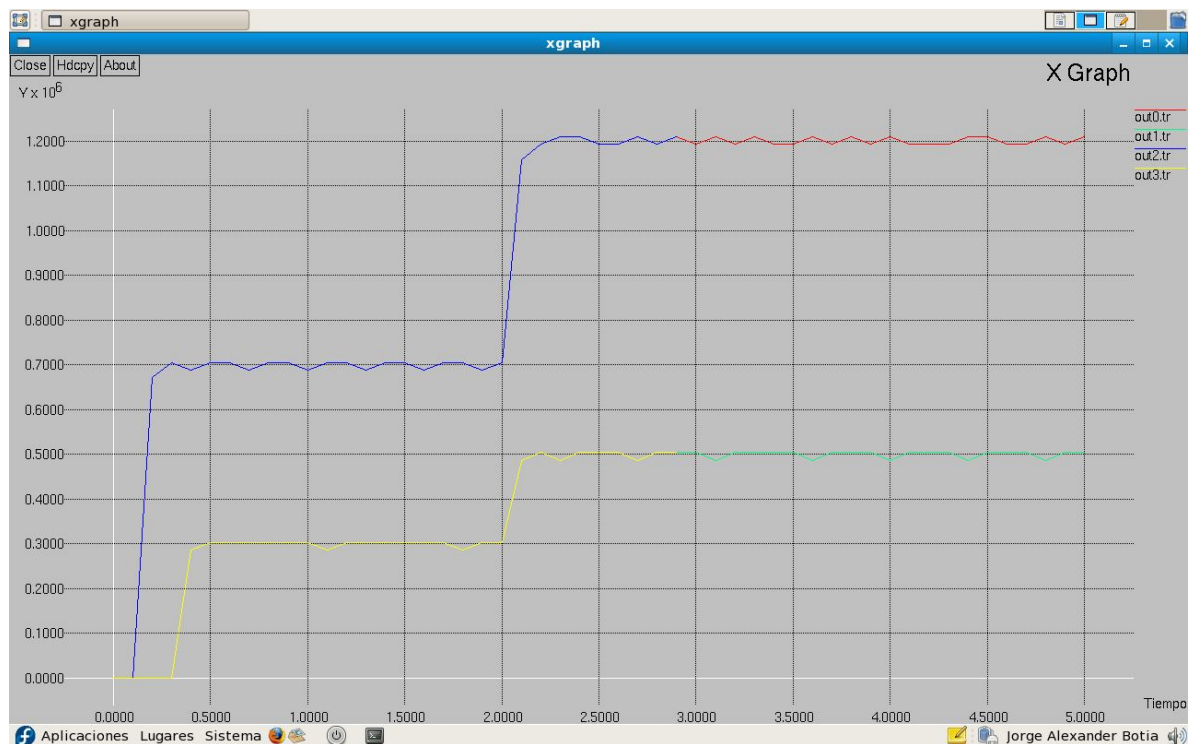
$ns at 0.0 "record"
$ns at 0.1 "$cbr0 start"
$ns at 0.3 "$cbr1 start"
$ns at 2 "$cbr0 set rate_ 1.2Mb; $cbr1 set rate_ 0.5Mb"
$ns at 3 "finish"

proc record {} {
    global sink0 sink1 ns f2 f3 nam
    set time 0.1
    set bw0 [$sink0 set bytes_]
    set bw1 [$sink1 set bytes_]
    set now [$ns now]
    puts $f2 "$now [expr $bw0/$time*8]"
    puts $f3 "$now [expr $bw1/$time*8]"
    $sink0 set bytes_ 0
    $sink1 set bytes_ 0
    $ns at [expr $now+$time] "record"
}

proc finish {} {
    global ns f2 f3 nam
    $ns flush-trace
    close $f2
    close $f3
    close $nam
    puts " Ejecutando simulación y grafica..."
    exec xgraph out0.tr out1.tr out2.tr out3.tr -geometry 800x400 -x Tiempo &
    exit 0
}
$ns run

```

Ejecutando el *script* anterior se obtuvo la siguiente gráfica de anchos de banda:



Este ejemplo se realiza con un Clasificador *Intserv6* y al simularlo se comporto de la misma manera que con un Clasificador *Intserv*. Las condiciones son las mismas que en el ejemplo anterior, solo que en este graficamos los dos ejemplos para poder apreciar la grafica. Mediante esta prueba podemos observar las distintas aplicaciones que tiene el Simulador *NS-2*.

Con los resultados obtenidos de las anteriores pruebas en *Otcl*, se puede procedió a diseñar los *scripts* de *Tcl* tanto de *IntServ* como de *IntServ6*. Estos procedimientos de diseño se encuentran detallados en la sección (3.5.3) de Desarrollo del Protocolo en *Otcl*.

4.3. RESULTADOS OBTENIDOS

Los siguientes resultados se obtuvieron con los siguientes parámetros de simulación:

CBR IntServ y CBR IntServ6:

- Ancho de Banda del enlace WFQ = 2Mb
- Ancho de Banda de los enlaces de Origen y Destino = 5Mb
- Retardo del enlace WFQ = 1ms
- Retardo de los enlaces de Origen y Destino = 2ms
- Tamaño del paquete CBR = 210
- Velocidad de Transmisión (Rate) = 448Kb
- Intervalo de paquetes = 3.75ms

Expo IntServ y Expo IntServ6:

- Ancho de Banda del enlace WFQ = 2Mb
- Ancho de Banda de los enlaces de Origen y Destino = 5Mb
- Retardo del enlace WFQ = 1ms
- Retardo de los enlaces de Origen y Destino = 2ms
- Tamaño del Paquete Expo = 64
- Burst_time = 1 seg
- Idle_time = 0 seg
- Velocidad de Transmisión (Rate) = 200Kb

Equipo de Simulación:

- Distribución de Linux → Fedora Core 8
- Versión del Simulador de Redes → NS-2.31
- Memoria RAM → 1GB
- Capacidad de disco duro de Linux → 10 GB
- Procesador → Intel Celeron

➤ Simulación de una Red IntServ con Tráfico CBR

Para esta simulación se ejecutaron los programas *cbr1.tcl* y *topologia1.tcl* los cuales se encuentran ubicados en el Anexo A del presente documento.

Mediante la herramienta de monitoreo de colas del *WFQ* se calcularon los retardos promedios de las colas del Planificador y además se obtuvieron los retardos promedios de los paquetes en el Clasificador IntServ. Con estos resultados se creó la Tabla 12 con la finalidad de organizar los datos, para después ser graficados.

Columna A: Número de Flujos.

Columna B: Retardo Promedio del Planificador *WFQ* (seg).

Columna C: Suma de retardos de la clasificación de todos los paquetes (Clasificador *IntServ*) (seg).

Columna D: Cantidad de paquetes que ingresaron al Clasificador *Intserv*.

Columna E: Retardo Promedio del Clasificador *IntServ* (seg).

Columna F: Suma del Retardo Promedio del Clasificador *IntServ* y el Retardo Promedio del Planificador *WFQ*.

A	B	C	D	E	F
1	0	0	0	0	0
2	4,20E-04	8,27E-03	2667	3,10E-06	4,20E-04
3	8,40E-04	1,20E-02	4000	3,00E-06	8,40E-04
4	1,26E-03	1,57E-02	5333	2,94E-06	1,26E-03
5	8,98E-03	1,93E-02	6666	2,90E-06	8,98E-02
6	1,32E-02	2,32E-02	7999	2,90E-06	1,32E-02
7	1,73E-02	2,70E-02	9332	2,89E-06	1,73E-02
8	2,15E-02	3,08E-02	10665	2,89E-06	2,15E-02
9	2,56E-02	3,47E-02	11998	2,89E-06	2,56E-02
10	2,97E-02	3,86E-02	13331	2,90E-06	2,97E-02
20	7,04E-02	7,74E-02	26661	2,90E-06	7,04E-02
30	1,10E-01	1,19E-01	39991	3,02E-06	1,10E-01
40	1,50E-01	1,61E-01	53321	3,02E-06	1,50E-01
50	1,90E-01	1,99E-01	65544	3,04E-06	1,90E-01
60	2,31E-01	2,41E-01	79199	3,05E-06	2,31E-01
70	2,70E-01	2,86E-01	92854	3,08E-06	2,70E-01
80	3,08E-01	3,29E-01	106509	3,09E-06	3,08E-01
90	3,45E-01	3,83E-01	118799	3,23E-06	3,45E-01
100	3,82E-01	4,25E-01	132454	3,21E-06	3,82E-01
200	7,43E-01	9,73E-01	266273	3,66E-06	7,43E-01
300	1,07	1,61	398726	4,05E-06	1,07
400	1,34	2,25	532545	4,22E-06	1,34
500	1,59	2,96	666362	4,44E-06	1,59
600	1,82	3,63	798818	4,55E-06	1,82
700	1,99	4,40	932637	4,72E-06	1,99
800	2,14	4,88	1065090	4,59E-06	2,14

Tabla 12. Resultados de Simulación de una Red IntServ con Tráfico CBR

A partir de los anteriores datos se realizó la Figura 64, la cual presenta el retardo Clasificación de cada uno de los paquetes a medida que va aumentando el tiempo:

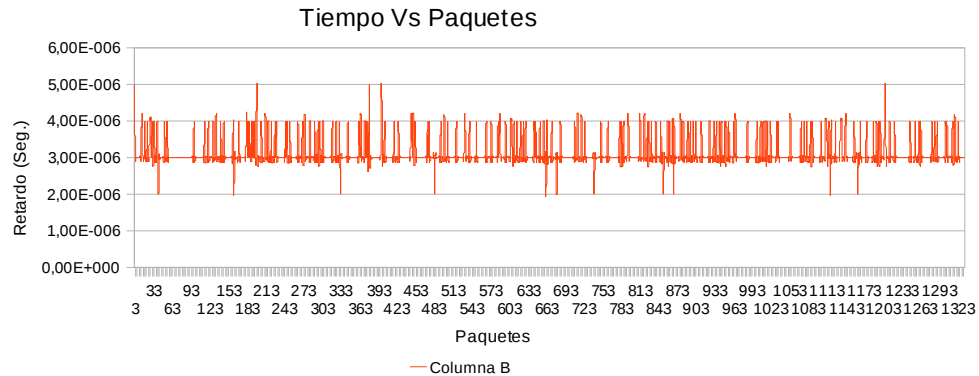


Figura 64. Retardo Individual de los Paquetes Ingresados al Clasificador IntServ

Y finalmente se se obtuvo el Retardo Promedio de la gráfica anterior el cual corresponde a 3,36 μ seg.

Para un mayor análisis del Retardo a través del tiempo se generó la Figura 65. En esta Gráfica se puede observar el Retardo Promedio del Clasificador *IntServ* a medida que se aumentan el número de flujos:

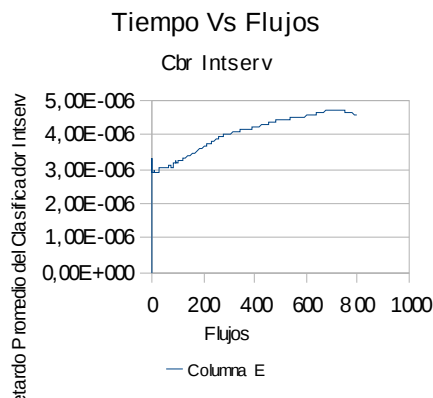


Figura 65. Retardo Promedio del Clasificador IntServ con Tráfico CBR

Como se puede observar el retardo empieza a aumentar a partir del segundo flujo y permanece constante alrededor de los 3 μ seg, y finalmente cuando el número de flujos aumenta considerablemente el Retardo Promedio de Clasificación comienza a ascender.

Y para finalizar con este caso de simulación se realizó la suma de los Retardos Promedios del Clasificación de paquetes y los Retardos Promedios de Planificación (*WFQ*) de paquetes en una red *IntServ*. Esto se hace con el objetivo

de analizar mediante una gráfica los Retardos Promedios Totales de una Red *IntServ*. A continuación se presenta la Figura 66:

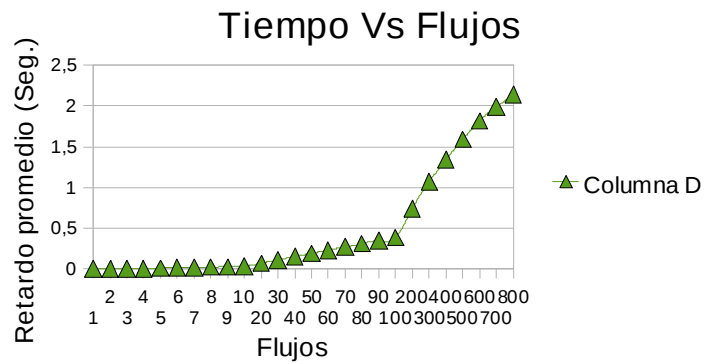


Figura 66. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red *IntServ* con Tráfico CBR.

➤ **Simulación de una Red IntServ6 con un Tráfico CBR**

En esta etapa de la simulación se ejecutaron los programas *cbr6.tcl* y *topologia6.tcl*, en los cuales se fue aumentando la cantidad de flujos para analizar el comportamiento de cada módulo. A continuación se detallan en la Tabla 13 los datos obtenidos a partir de esta simulación:

A	B	C	D	E	F
1	0	0	0	0	0
2	4,20E-04	2,77E-03	1366	2,03E-06	4,22E-04
3	8,40E-04	5,54E-03	2731	2,03E-06	8,42E-04
4	1,26E-03	8,09E-03	4097	1,97E-06	1,26E-03
5	8,98E-03	1,06E-02	5462	1,94E-06	8,98E-03
6	1,32E-02	1,33E-02	6828	1,95E-06	1,32E-02
7	1,73E-02	1,55E-02	8193	1,89E-06	1,73E-02
8	2,15E-02	1,81E-02	9559	1,90E-06	2,15E-02
9	2,56E-02	2,08E-02	10924	1,90E-06	2,56E-02
10	2,97E-02	2,31E-02	12290	1,88E-06	2,97E-02
20	7,04E-02	4,79E-02	25945	1,85E-06	7,04E-02
30	1,10E-01	7,30E-02	39600	1,84E-06	1,10E-01
40	1,50E-01	9,83E-02	53255	1,88E-06	1,50E-01
50	1,90E-01	1,23E-01	65544	1,88E-06	1,90E-01
60	2,31E-01	1,47E-01	79199	1,85E-06	2,31E-01
70	2,70E-01	1,70E-01	92854	1,84E-06	2,70E-01
80	3,08E-01	1,96E-01	106509	1,84E-06	3,08E-01
90	3,45E-01	2,20E-01	118799	1,86E-06	3,45E-01
100	3,82E-01	2,49E-01	132454	1,88E-06	3,82E-01
200	7,43E-01	4,98E-01	266273	1,87E-06	7,43E-01
300	1,07	7,54E-01	398726	1,89E-06	1,07
400	1,34	1,01	532545	1,89E-06	1,34
500	1,59	1,27	666364	1,90E-06	1,59
600	1,82	1,62	798818	2,02E-06	1,82
700	1,99	1,94	932637	2,08E-06	1,99
800	2,14	2,06	1065090	1,93E-06	2,14
900	2,21	4,28	1332728	3,21E-06	2,21
1000	2,43	4,28	1332728	3,21E-06	2,43

Tabla 13. Resultados de Simulación de una Red IntServ6 con Tráfico CBR

Explicación de las columnas:

Columna A: Número de Flujos.

Columna B: Retardo Promedio del Planificador *WFQ* (seg).

Columna C: Sumatoria de los Retardos de clasificación de todos los paquetes (Clasificador *IntServ6*) (seg).

Columna D: Cantidad de paquetes que ingresaron al Clasificador *Intserv6*.

Columna E: Retardo Promedio del Clasificador *IntServ6* (seg).

Columna F: Suma del Retardo Promedio del Clasificador *IntServ* y el Retardo Promedio del Planificador *WFQ*.

Por medio del uso de la herramienta de archivos de salida se pudieron obtener los retardos de clasificación de cada uno de los paquetes, y así poder graficarlos como se muestra en la Figura 67.

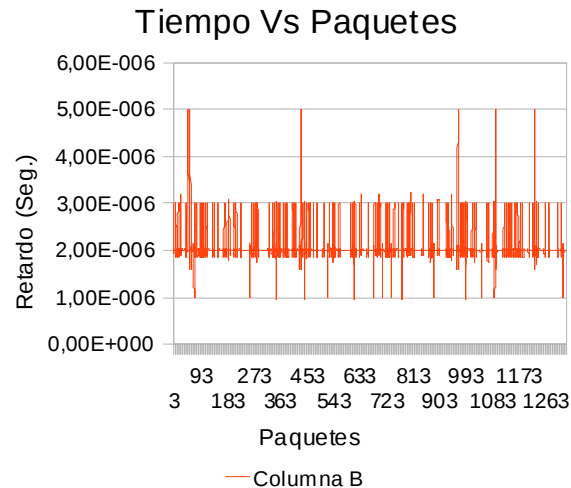


Figura 67. Retardo Individual de los Paquetes que ingresaron al Clasificador IntServ6.

Por medio de los resultados de la Tabla 13 se pudo calcular el Tiempo Promedio de Clasificación de los paquetes en una Red *IntServ6*, el cual corresponde aproximadamente a 2 μ seg. Con la anterior Figura 67 se puede verificar el Retardo Promedio calculado, ya que oscila entre 2 μ seg y 3 μ seg. También se graficaron los Retardos Promedios de Clasificación (*IntServ6*) con respecto al Número de Flujos, lo que nos ayudó a verificar la efectividad de una red *IntServ6*, debido a que los retardos de clasificación son mucho menores que los del Clasificador *IntServ*. Estos resultados se pueden observar en la Figura 68.

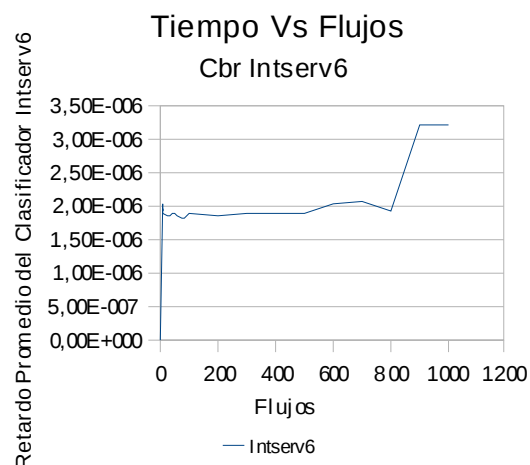


Figura 68. Retardo Promedio del Clasificador IntServ6 con Tráfico CBR

Además para obtener un resultado final de la Red *IntServ6* se realizó la sumatoria de los Retardos Promedios de Clasificación y los Retardos Promedios de la Planificación (*WFQ*) de los paquetes, con la finalidad de tener un punto de vista general del comportamiento de la red. Este comportamiento se puede detallar en la Figura 69:

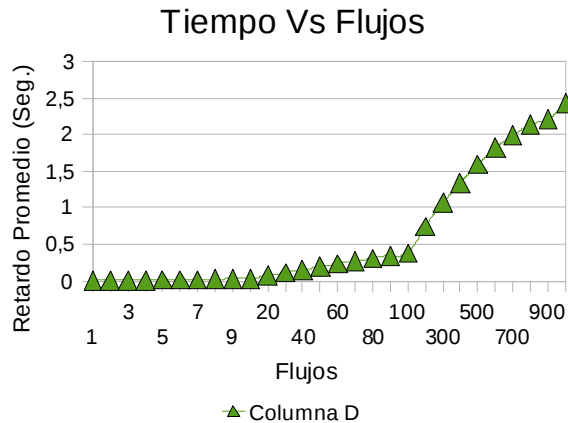


Figura 69. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red *IntServ6* con Tráfico CBR.

Finalmente se hace una comparación de las simulaciones de *IntServ* e *IntServ6* con Tráfico *CBR* con la finalidad de analizar el comportamiento del Clasificador en las dos redes y verificar cual tiene menor retardo y por consiguiente es más eficaz. Esta comparación se puede observar en la Figura 70:

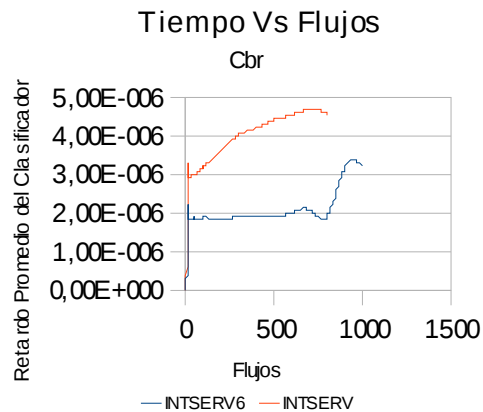


Figura 70. Comparación de Retardos Promedios de Clasificación en las redes *IntServ* e *IntServ6* con Tráfico CBR

➤ Simulación de una Red IntServ con Tráfico Exponencial

La presente simulación se realizó con la ejecución de los scripts *cbr1.tcl* y *topologia1.tcl* con la modificación de los Generadores de Tráfico de CBR a Exponencial.

Al finalizar la ejecución de los *scripts* y calcular los retardos mediante las herramientas anteriormente mencionadas, se creó la siguiente tabla con el objetivo de organizar y graficar los resultados obtenidos para un mejor análisis.

A	B	C	D	E	F
1	0	4,23E-03	1952	2,17E-06	2,17E-06
2	1,28E-04	8,37E-03	3904	2,14E-06	1,30E-04
3	2,56E-04	1,62 E-02	5856	2,77E-06	2,59E-04
4	3,84E-04	2,02 E-02	7808	2,58E-06	3,87E-04
5	5,12E-04	2,80 E-02	9760	2,87E-06	5,15E-04
6	6,40E-04	3,18 E-02	11712	2,71E-06	6,43E-04
7	7,68E-04	3,94 E-02	13664	2,88E-06	7,71E-04
8	8,96E-04	4,38 E-02	15615	2,80E-06	8,99E-04
9	1,02E-03	4,67 E-02	17568	2,66E-06	1,02E-03
10	1,15E-03	5,42 E-02	19520	2,77E-06	1,15E-03
20	2,12E-02	1,08 E-01	39040	2,76E-06	2,12E-02
30	3,39E-02	1,66 E-01	58560	2,84E-06	3,39E-02
40	4,65E-02	2,29 E-01	78080	2,94E-06	4,65E-02
50	5,91E-02	2,89 E-01	97600	2,96E-06	5,91E-02
60	7,16E-02	3,53 E-01	117120	3,01E-06	7,16E-02
70	8,41E-02	4,18 E-01	136640	3,06E-06	8,41E-02
80	9,66E-02	4,79 E-01	156160	3,07E-06	9,66E-02
90	1,09E-01	5,61 E-01	175680	3,19E-06	1,09E-01
100	1,21E-01	6,19 E-01	195200	3,17E-06	1,21E-01
200	0,24	1,41	390400	3,61E-06	2,40E-01
300	0,36	2,35	585600	4,01E-06	3,60E-01
400	0,48	3,28	780800	4,20E-06	4,80E-01
500	0,59	4,29	827253	5,19E-06	5,90E-01
600	0,7	5,3	1171200	4,53E-06	7,00E-01
700	0,8	6,24	1366400	4,57E-06	8,00E-01
800	0,91	7,13	1561600	4,57E-06	9,10E-01

Tabla 14. Resultados de Simulación de una Red IntServ con Tráfico Exponencial.

Explicación de las columnas:

Columna A: Número de Flujos.

Columna B: Retardo Promedio del Planificador *WFQ* (seg).

Columna C: Sumatoria de los Retardos de clasificación de todos los paquetes (Clasificador *IntServ*) (seg).

Columna D: Cantidad de paquetes que ingresaron al Clasificador *IntServ*.

Columna E: Retardo Promedio del Clasificador *IntServ* (seg).

Columna F: Suma del Retardo Promedio del Clasificador *IntServ* y el Retardo Promedio del Planificador *WFQ*.

Teniendo en cuenta los resultados anteriores se realizaron gráficas para una mayor comprensión del comportamiento de una Red *IntServ* con un Tráfico Exponencial. La primera gráfica se observa en la Figura 71, la cual representa el Retardo individual de cada uno de los paquetes en el Clasificador *IntServ*. El Retardo Promedio de estos paquetes es 3.23μseg.

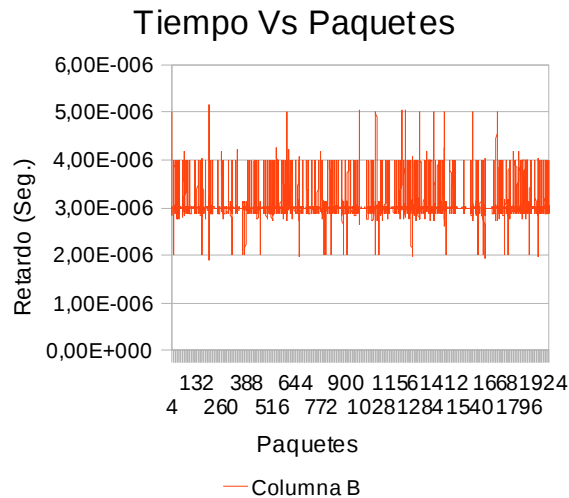


Figura 71. Retardo Individual de los Paquetes ingresados al Clasificador IntServ

La segunda gráfica se observa en la Figura 72, en donde se detalla el Retardo Promedio del Clasificador *IntServ* con respecto al Número de Flujos de la Red (*IntServ*). En esta Figura se puede observar que a medida que aumenta el número de Flujos el retardo promedio de clasificación también aumenta.

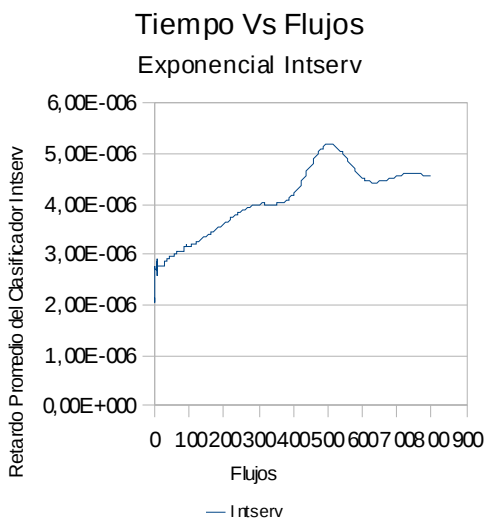


Figura 72. Retardo Promedio del Clasificador IntServ con Tráfico Exponencial.

Finalmente se realiza la grafica de las sumatorias de los Retardos Promedios del Clasificador y el Planificador (*WFQ*) en una red *IntServ*. Esta gráfica se puede observar en la Figura 73 que se observa a continuación:

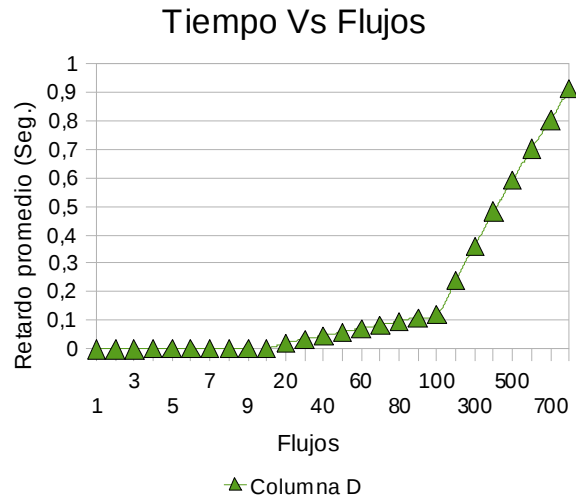


Figura 73. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red *IntServ* con Tráfico Exponencial.

➤ Simulación de una Red *IntServ* con Tráfico Exponencial.

En esta simulación se utilizaron los scripts *cbr6.tcl* y *topologia6.tcl*, incluyendoles las modificaciones necesarias para el Tráfico Exponencial. Además se fueron incrementando el Número de Flujos exponencialmente como se observa en la columna A de la Tabla 15.

Explicación de las columnas:

Columna A: Número de Flujos.

Columna B: Retardo Promedio del Planificador *WFQ* (seg).

Columna C: Sumatoria de los Retardos de clasificación de todos los paquetes (Clasificador *IntServ6*) (seg).

Columna D: Cantidad de paquetes que ingresaron al Clasificador *Intserv6*.

Columna E: Retardo Promedio del Clasificador *IntServ6* (seg).

Columna F: Suma del Retardo Promedio del Clasificador *IntServ* y el Retardo Promedio del Planificador *WFQ*.

A	B	C	D	E	F
1	0	2,81E-03	1952	1,44E-06	1,44E-06
2	1,28E-04	5,52E-03	3904	1,42E-06	1,29E-04
3	2,56E-04	1,08E-02	5856	1,84E-06	2,58E-04
4	3,84E-04	1,33E-02	7808	1,71E-06	3,86E-04
5	5,12E-04	1,85E-02	9760	1,90E-06	5,14E-04
6	6,40E-04	2,13E-02	11712	1,82E-06	6,42E-04
7	7,68E-04	2,60E-02	13664	1,90E-06	7,70E-04
8	8,96E-04	2,87E-02	15615	1,84E-06	8,98E-04
9	1,02E-03	3,16E-02	17568	1,80E-06	1,02E-03
10	1,15E-03	3,62E-02	19520	1,85E-06	1,15E-03
20	2,12E-02	7,23E-02	39040	1,85E-06	2,12E-02
30	3,39E-02	1,06E-01	58560	1,80E-06	3,39E-02
40	4,65E-02	1,43E-01	78080	1,84E-06	4,65E-02
50	5,91E-02	1,80E-01	97600	1,85E-06	5,91E-02
60	7,16E-02	2,16E-01	117120	1,84E-06	7,16E-02
70	8,41E-02	2,56E-01	136640	1,87E-06	8,41E-02
80	9,66E-02	2,87E-01	156160	1,84E-06	9,66E-02
90	1,09E-01	3,21E-01	175680	1,83E-06	1,09E-01
100	1,21E-01	3,56E-01	195200	1,83E-06	1,21E-01
200	0,24	7,41E-01	390400	1,90E-06	2,40E-01
300	0,36	1,11	585600	1,90E-06	3,60E-01
400	0,48	1,47	780800	1,88E-06	4,80E-01
500	0,59	1,84	827253	1,88E-06	5,90E-01
600	0,7	2,30	1171200	1,97E-06	7,00E-01
700	0,8	2,8	1366400	2,05E-06	8,00E-01
800	0,91	3	1561600	1,92E-06	9,10E-01
900	1,01	3,38	1756800	1,92E-06	1,01

Tabla 15. Resultados de Simulación de una Red IntServ6 con Tráfico Exponencial.

La primera Figura 74 de esta simulación corresponde al Retardo Promedio de Clasificación de cada uno de los paquetes en una Red *IntServ6* con un Tráfico Exponencial. Calculando manualmente el retardo Promedio de Clasificación de un Paquete, mediante la columna E de la anterior tabla, se obtiene un valor de 1.83μseg.

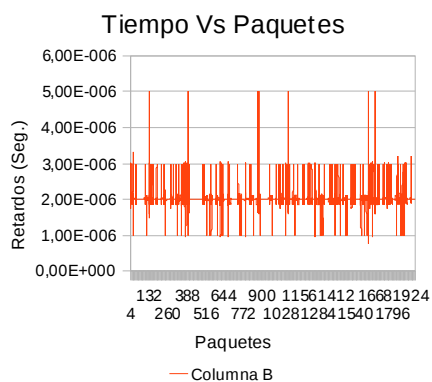


Figura 74. Retardo Individual de los Paquetes ingresados al Clasificador IntServ6

La siguiente gráfica corresponde al Retardo Promedio de Clasificación en una red *IntServ6* con un Tráfico Exponencial, con respecto al Número de Flujos. (Ver Figura 75):

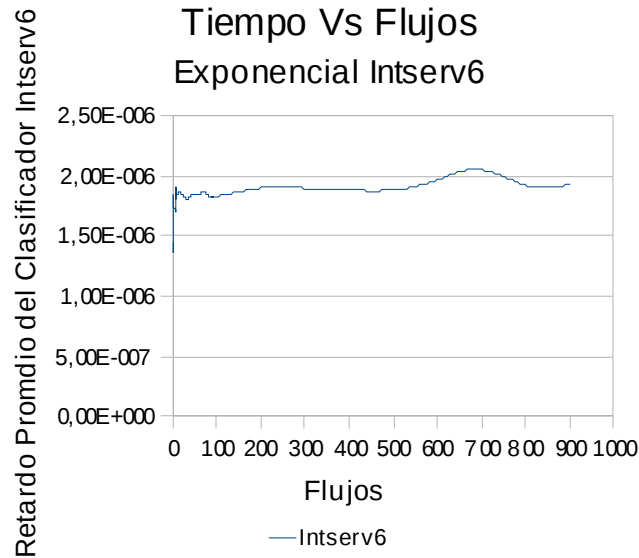


Figura 75. Retardo Promedio del Clasificador IntServ6 con Tráfico Exponencial.

Como una ayuda más para el análisis de esta simulación se creó la gráfica de la sumatoria de los Retardos de Clasificación y Planificación (*WFQ*) en una red *IntServ6* (Columna F), con respecto al Número de Flujos.

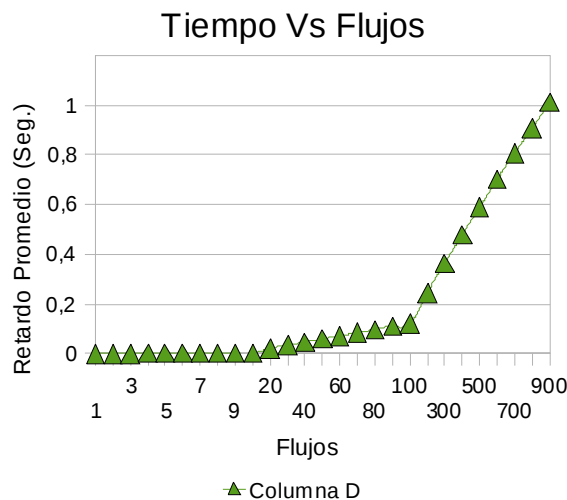


Figura 76. Sumatoria de Retardos Promedios de Clasificación y Planificación en una Red IntServ6 con Tráfico Exponencial.

Y finalmente se realiza una comparación de los Retardos Promedios de Clasificación en las redes Intserv e IntServ6, con el objetivo de analizar cual de las dos redes tiene menor retardo y por consiguiente es más eficaz.

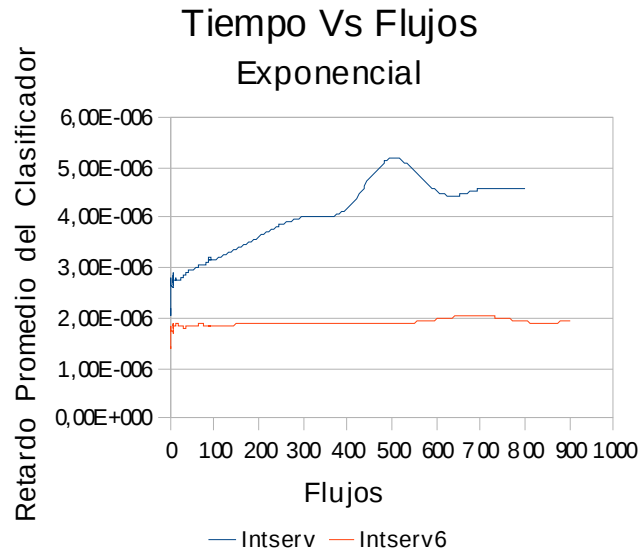


Figura 77. Comparación de Retardos Promedios de Clasificación en las redes IntServ e IntServ6 con Tráfico Exponencial

➤ Simulación del Planificador WFQ

En esta simulación se ejecutan los scripts *cbr.tcl* y *topologia.tcl*, realizandole las modificaciones para dos flujos. Esto se realiza con el fin de poder apreciar mediante la herramienta *Xgraph* como el Planificador *WFQ* le entrega a cada flujo su respectivo Ancho de Banda, tanto para *IntServ*, como para *IntServ6*.

Los *scripts* presentan un enlace *WFQ* con un Ancho de Banda Máximo de 0.5Mb. El primer flujo tiene el 80% del enlace y el segundo flujo tiene el 20% del enlace. Y en el tiempo 1.4 (seg) de la simulación se les modifica el porcentaje entregado a los dos flujos, quedando el primer flujo con el 40% y el segundo flujo con el 60% del Ancho de Banda del enlace.

En la Figura 78 se puede observar el cambio del Ancho de Banda realizado con el objetivo de analizar el comportamiento del Planificador *WFQ*, y verificar que su funcionamiento es el deseado.

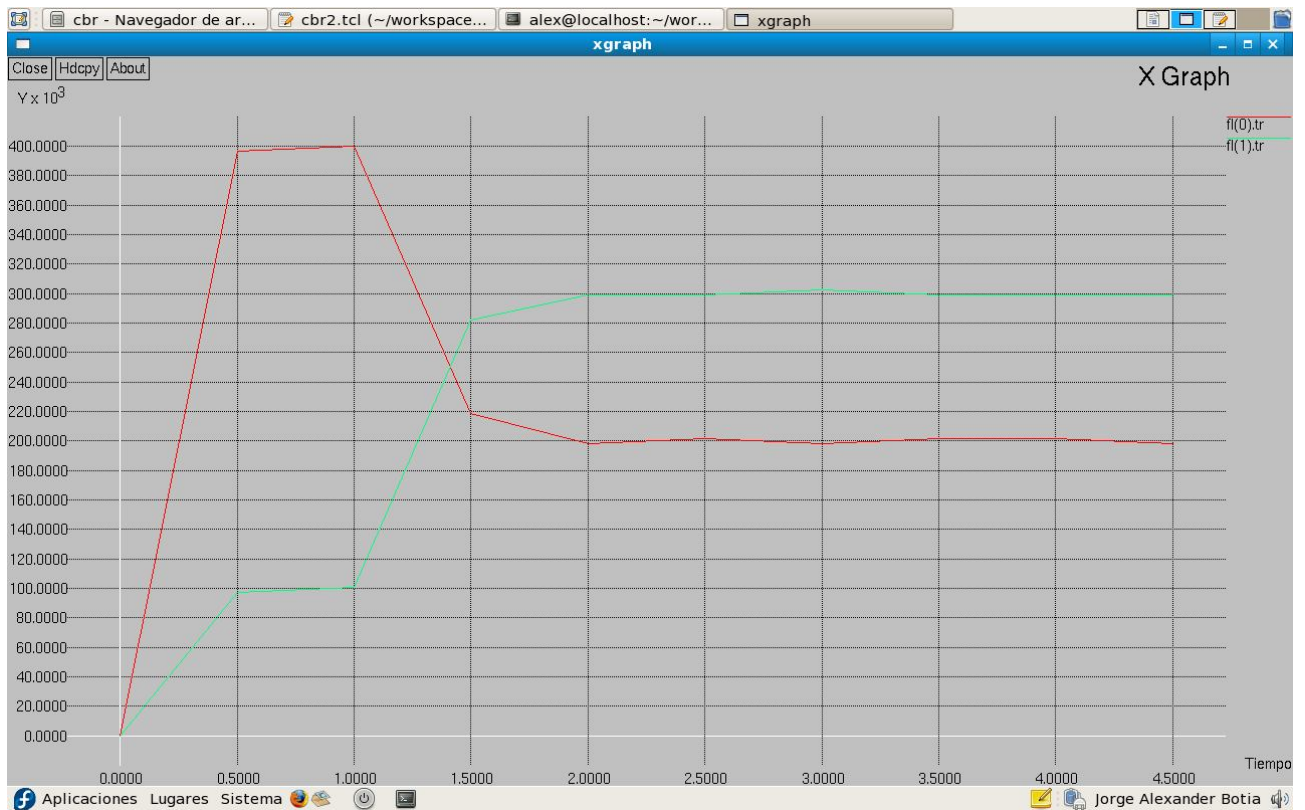


Figura 78. Comportamiento del Planificador WFQ

Finalmente se realizan los análisis pertinentes de las gráficas resultantes creando así una serie de conclusiones, las cuales se describen a continuación:

- Los Retardos promedios de los paquetes en un clasificador *IntServ* están alrededor de los 3useg y los de *IntServ6* están alrededor de los 2useg. Por lo tanto se puede deducir que la Red *IntServ6* tiene un mejor desempeño debido a su Cálculo en el Host Origen y no en todos los routers de la red.
- La rata del tráfico es importante para que el planificador y el clasificador funcionen correctamente, esta debe ser aproximadamente mayor a 100kb/seg.
- Los tiempos promedio del planificador *WFQ* fueron calculados con una herramienta de *NS-2* para monitorear colas y es igual en todas las simulaciones, a excepción de los retardos (*WFQ*) en una red *IntServ6* con Tráfico Exponencial, los cuales son menores a los retardos de las demás simulaciones descritas.
- La cantidad de flujos que se generaron en las diferentes simulaciones dependía del tipo de Clasificador con el que se estuviera trabajando (*IntServ* o *IntServ6*). En *IntServ* solo se pudieron añadir hasta 800 flujos, y en *IntServ6* se añadieron hasta 1000 flujos. Esto se debe a que a partir de esos límites el Clasificador no trabaja correctamente.

- El número de flujos de *IntServ6* debe a que el Planificador solo soporta 1000 Flujos, es decir, el mínimo peso que se le puede asignar a una cola es de 0,001. Si se añaden más flujos a *IntServ6* se obtiene un error en *Tcl* de Violación de Segmento.
- Existen limitaciones en el momento de simular grandes redes mayores a 1000 flujos, ya que los archivos que se generan para grabar datos sobre retardos, anchos de banda y cantidad de paquetes tienen la capacidad de almacenar sólo hasta 1,500,000. En caso de generar más líneas el archivo se bloquea al intentar acceder a él.
- Los tiempos reales de simulaciones de 1000 flujos están alrededor de 10 minutos.
- El espacio para la partición de *Linux* en un usuario por lo mínimo debe ser de 10GB ya que las simulaciones requieren generar una cantidad de archivos para el análisis de la misma.

5. CONCLUSIONES Y RECOMENDACIONES

Con esta tesis se pretendía comparar la Arquitectura de Servicios Integrados (*IntServ*), con la Arquitectura de Servicios Integrados sobre IPv6 (*IntServ6*), propuesta por Jhon Jairo Padilla Aguilar en su Tesis Doctoral, la cual se basaba en el Cálculo del Número Hash en el Host Origen y almacenamiento del mismo en la Etiqueta de Flujo de *IPv6*.

Mediante las simulaciones de las dos Arquitecturas anteriores, se pudo observar que el retardo de Clasificación en una red *IntServ6* fue menor al retardo de Clasificación en una red *IntServ*, por lo que se puede asegurar que la propuesta de *IntServ6* presenta un mejor desempeño que la Arquitectura estándar (*IntServ*).

La simulación de las dos Arquitecturas (*IntServ* e *IntServ6*), difieren además en el límite de flujos que cada red es capaz de soportar; esto debido a que la nueva propuesta (*IntServ6*) soporta más flujos que en *IntServ* por su eficiencia en la Clasificación de los paquetes. Al exceder este límite se generan errores resultados incoherentes. Cabe resaltar que a medida que se aumenta el Número de Flujos en la topología aumenta el tiempo real de simulación y el agota los recursos del sistema.

Los diseños de los Algoritmos de los nuevos módulos en C++, que se agregan al Simulador *NS-2*, dependen básicamente del tipo de topología que el usuario desea utilizar. Por consiguiente, para hacer uso de esta nueva herramienta es indispensable la utilización del mismo diseño de topología utilizado en el presente documento.

En las simulaciones de las redes *IntServ* e *IntServ6* se utilizó un módulo de Gestión de Colas llamado Planificador *WFQ*, el cual se creó como una contribución al código estándar del simulador *NS-2*. Una de las limitaciones de esta herramienta fue el número de flujos, siendo este mismo el máximo para la red *IntServ6*.

Una de las recomendaciones más importantes para futuras investigaciones y avances de esta herramienta es la capacidad de disco duro para el sistema Operativo utilizado (*Linux*), ya que de éste depende el correcto funcionamiento de la herramienta creada y la creación de los archivos para grandes topologías con grandes cantidades de flujos.

6. ANEXOS

CLASSIFIER-INTSERV.H:

```
#include <stdio.h>
#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include <math.h>
#include <time.h>
#include <sys/time.h>
#define FIJAR_TMP() gettimeofday(&temp_1, 0)
#define FIN_TMP() gettimeofday(&temp_2, 0)
#include <iostream>
#include <fstream>

//ofstream archivo("PruebaHash.ods"); // constructor de ofstream
ofstream archivo1("PruebaHash2.ods"); // constructor de ofstream
//ofstream archivo2("Contador.ods"); // constructor de ofstream
ofstream archivo3("clintserv.tr");

//*****
//CONSTRUCTOR

class IntServClassifier : public Classifier {
protected:

    int classify(Packet *p);
    virtual int command(int argc, const char*const* argv);

//*****
// BUSQUEDA: Accede a la cabecera ip del paquete(quintupla) y lo manda a buscar a la tabla hash.

    int busqueda(Packet *p) {
        hdr_ip* h = hdr_ip::access(p);
        printf("h: %d \n",h);
//
        return get_hash_1(mshift(h->saddr()),mshift(h->sport()),mshift(h->daddr()),mshift(h->dport()),h->protid());
    }

//*****
// SET-HASH-1: Método para calcular número hash y guardarlos en la tabla hash.

    int set_hash_1(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {
        int valor;
        int indice;
        int col;
        indice = 0;
        f = 0;
        cont = 0;
        col = destino_;
        while (src<flujo_) {
            valor = hashkey_1(src,sport,dst,dport,pid);
            //archivo << valor << "\n";
            campos.numhash_ = valor;
            campos.BW_ = col;
            campos.colision_ = 0;
            campos.origen_ = src;
            campos.destino_ = dst;
            registrar(campos, 0, indice); //Llena los datos en la tabla hash
            registrar(campos, 2, indice); //Compara si hay algun hash igual anteriormente
            ++src;
        }
    }
};
```

```

        ++dst;
        ++indice;
        ++col;
    }
    return (1);
}

//*****

// GET-HASH: Encuentra una entrada en la tabla hash

int get_hash_1(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {
    int valorh;
    int vfid;
    ++cont;
    valorh = hashkey_1(src,sport,dst,dport,pid);
    campos.numhash_ = valorh;
    archivo1 << valorh << "\n";
    vfid = registrar(campos, 1, 0);
    return (vfid);
}

//*****
// VARIABLES UTILIZADAS EN LA CLASE INTSERVCLASSIFIER:

int          flujo_;
int          destino_;
int          f;
int          cont;
int          hash_;
int          mask1_;
int          temp1_;
int          temp2_;
int          tabla[6];
puntstructcampos;
rescol      colisiones;
hdr_ip      quintupla;

//*****
// REGISTRAR: Metodo de acceso a la tabla de BW y Colisiones.

int registrar(puntstruct& tab, int a, int indice) {
    int x;
    int t;
    int cmp1;
    int cmp2;
    int nhash;
    int rtdocol;
    int vbuscar;
    int y;
    int rtdoflow;
    int bit;
    int variable;
    int var;
    static int htabla [2000] [5]; // [filas] [Columns]
    //a = 0 --> Llenar tabla
    //a = 1 --> Buscar Datos
    //a = 2 --> Comparar datos

    if (a == 0) {
        x = indice;
        htabla [x] [1] = tab.numhash_;
        var = htabla [x] [1];
        htabla [x] [2] = tab.BW_;
        variable = htabla [x] [2];
        htabla [x] [3] = tab.colision_;
        htabla [x] [4] = tab.origen_;
    }
}

```

```

        htabla[x][5] = tab.destino_;

        return (int) 1;
    }
    else if (a == 1) {
        x = indice;
        vbuscar = tab.numhash_;
        for (x=0;x<flujo_;x++) {
            nhash = htabla[x][1];
            if (vbuscar == nhash) {
                rtdocol = htabla[x][3];
                if (rtdocol == 1) {
                    colisiones.numhash1_ = nhash;
                    colisiones.origen1_ = htabla[x][4];
                    colisiones.destino1_ = htabla[x][5];
                    y = resolver(colisiones, 1);
                    rtdoflow = y;
                    x = 2000;
                } else {
                    rtdoflow = htabla[x][2];
                    x = 2000;
                }
            }
        }

        return (int) rtdoflow;
    }
    else if (a == 2) {
        x = indice;
        for (t=0;t<x;t++) {
            cmp1 = htabla[t][1];
            cmp2 = htabla[x][1];
            if (cmp1 == cmp2) {
                bit = htabla[t][3];
                if (bit == 1) { // Si ese numero ya tiene colision solo se guarda el nuevo
                    hash colisionado
                    estructura de colisiones
                    de colisiones
                    estructura de colisiones

                    colisiones.numhash1_ = htabla[x][1]; // Se guarda el numero hash en la
                    colisiones.origen1_ = htabla[x][4]; // Se guarda el Origen en la estructura
                    colisiones.destino1_ = htabla[x][5]; // Se guarda el Destino en la
                    colisiones.BW1_ = htabla[x][2]; // BW = 0
                    resolver(colisiones, 0);
                    htabla[x][1] = 0;
                    htabla[x][2] = 0;
                    htabla[x][3] = 0;
                    htabla[x][4] = 0;
                    htabla[x][5] = 0; //Borra el numero hash
                    t = 700; //Para que se salga del For
                }
            }
            else if (bit == 0) {
                colisiones.numhash1_ = htabla[t][1];
                colisiones.origen1_ = htabla[t][4];
                colisiones.destino1_ = htabla[t][5];
                colisiones.BW1_ = htabla[t][2];
                htabla[t][3] = 1; //Bandera de colision en 1
                y = resolver(colisiones, 0);
                colisiones.numhash1_ = htabla[x][1];
                colisiones.origen1_ = htabla[x][4];
                colisiones.destino1_ = htabla[x][5];
                colisiones.BW1_ = htabla[x][2];
                y = resolver(colisiones, 0);
                htabla[x][1] = 0;
                htabla[x][2] = 0;
                htabla[x][3] = 0;
                htabla[x][4] = 0;
            }
        }
    }
}

```

```

        htabla [x] [5] = 0; //Borra el numero hash
        t = 2000; //Para que se salga del For
    }
}
return 0;
}
return (rtdoflow);
}

//*****
// RESOLVER:

int resolver(rescol& punt, int k) {
int vcomp;
int vcomp2;
int maxf;
int vcolis;
int vquint;
int reserva;
static int tablacol [2000] [4]; // [flujos] [Columnas]
//k = 0 --> Llenar tabla
//k = 1 --> Buscar Datos

if (k == 0) {
    f = f + 1;
    tablacol [f] [1] = punt.numhash1_;
    tablacol [f] [2] = punt.origen1_;
    tablacol [f] [3] = punt.destino1_;
    tablacol [f] [4] = punt.BW1_;

    return (int) 1;

} else if (k == 1) {
    maxf = flujo_; //CAMBIAR POR EL ORIGINAL.
    for (f=0;f<maxf;f++) { // NO OLVIDAR DAR EL VALOR DE MAXF
        vcomp = punt.numhash1_;
        vcomp2 = punt.origen1_;
        vcolis = tablacol [f] [1];
        if (vcomp == vcolis) {
            vquint = tablacol [f] [2];
            if (vcomp2 == vquint) {
                reserva = tablacol [f] [4];
                f = 2000;
            }
        }
    }
    return (int) reserva;
}
return (reserva);
}

//*****
//HASHKEY: Metodo que se utiliza para calcular el numero hash con la quintupla.

int hashkey_1(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

int i;

mask1_ = 0;
temp1_ = 0;
temp2_ = 0;
hash_ = 0;

//Calculo hash

```

```

mask1_ = 0xffff000; //Mask 1
temp1_ = src;
temp1_ &= mask1_;
temp1_ >>= 3;
tabla[0] = temp1_; //Variable 1
mask1_ = 0x00000fff; //Mask 2
temp1_ = src;
temp1_ &= mask1_;
temp1_ <<= 2;
mask1_ = 0xff000000; //Mask 3
temp2_ = dst;
temp2_ &= mask1_;
temp2_ >>= 6;
temp1_ |= temp2_;
tabla[1] = temp1_; //Variable 2
mask1_ = 0x00ffff0; //Mask 4
temp1_ = dst;
temp1_ &= mask1_;
temp1_ >>= 1;
tabla[2] = temp1_; //Variable 3
mask1_ = 0x0000000f; //Mask 5
temp1_ = dst;
temp1_ &= mask1_;
temp1_ <<= 4;
temp2_ = sport;
temp1_ |= temp2_;
tabla[3] = temp1_; // variable 4
temp1_ = dport;
temp1_ <<= 1;
mask1_ = 0x000000f0; //Mask 6
temp2_ = pid;
temp2_ &= mask1_;
temp2_ >>= 1;
temp1_ |= temp2_;
tabla[4] = temp1_; //Variable 5
mask1_ = 0x0000000f; //Mask 7
temp1_ = pid;
temp1_ &= mask1_;
tabla[5] = temp1_; //Variable 6

for (i=0;i<6;i++)
    hash_ ^= tabla[i];
return (int) hash_;
}

public:

    IntServClassifier() : flujo_(1), destino_(1) {
        bind("flujo_", &flujo_);
        bind("destino_", &destino_);
    }

};

```

CLASSIFIER-INTSERV.CC:

```

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /cvsrcroot/nsnam/ns-2/classifier/classifier-IntServ.cc,v 1.30 2005/09/18 23:33:31 tomh Exp $
(LBL)";
#endif

extern "C" {
#include <cl.h>

```

```

}

#include <stdlib.h>
#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include "classifier-IntServ.h"

#include <math.h>
#include <time.h>
#include <sys/time.h>
#define FIJAR_TMP() gettimeofday(&temp_1, 0)
#define FIN_TMP() gettimeofday(&temp_2, 0)
#include <iostream>
#include <fstream>

ofstream archivo3("clintserv.tr");

/***** IntServClassifier Methods *****/

int IntServClassifier::classify(Packet *p) {

    float y=0,z=0;

    struct timeval temp_1, temp_2;
    FIJAR_TMP();

    busqueda(p);

    FIN_TMP();
    y = ((temp_2.tv_sec-temp_1.tv_sec)+(float)(temp_2.tv_usec-temp_1.tv_usec)/1000000);
    z = 0.000009;

    if (y < 0.000009){
        archivo3 << y << "\n";
    } else {
        archivo3 << z << "\n";
    }

    return (1);
}

int IntServClassifier::command(int argc, const char*const* argv)
{
    /*$classifier add-num src dst */
    if (argc == 4 && strcmp(argv[1], "add-num") == 0) {
        nsaddr_t src = atoi(argv[2]);
        int16_t sport = 0;
        nsaddr_t dst = atoi(argv[3]);
        int16_t dport = 0;
        int pid = 0;
        int n = set_hash_1(src,sport,dst,dport,pid);
        printf("Resultado: %d \n",n);
        return(TCL_OK);
    }

    return (Classifier::command(argc, argv));
}

/***** TCL linkage *****/
static class IntServClassifierClass : public TclClass {
public:

```

```

        IntServClassifierClass() : TclClass("Classifier/IntServ") {}
        TclObject* create(int, const char*const*) {
            return (new IntServClassifier());
        }
    } class_intserv_classifier;

```

CLASSIFIER-INTSERV6.H:

```

// Programa Clasificador Hash con la Quintupla (IntServ6)

#include "classifier.h"
#include "ip.h"
#include "config.h"
#include "packet.h"
#include <fstream.h>

ofstream tabla("HashIntServ6.ods"); // constructor de ofstream

//*****
//CONSTRUCTOR

class IntServ6Classifier : public Classifier {
public:
    IntServ6Classifier() : flujoint_(-1), destinoint_(-1) {
        bind("flujoint_", &flujoint_);
        bind("destinoint_", &destinoint_);
    }

//*****
//DESTRUCTOR

    ~IntServ6Classifier() {
    };

//*****
// CLASSIFY: Es un método puro virtual indicando que la clase Classifier es solo usada como una clase base
protected:
    int classify(Packet *p);

//*****
// BUSQUEDA: Accede a la cabecera ip del paquete(quintupla) y lo manda a buscar a la tabla hash.
public:
    int busqueda_6(Packet *p) {
        hdr_ip* h = hdr_ip::access(p);
        return get_hash_6(h->nhash(), h->saddr(), h->daddr());
    }

//*****
protected:

//*****
// SET-HASH-1: Método para calcular número hash y guardarlos en la tabla hash.

    int set_hash_6(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

        int valor1;
        int indice1;
        int coll;
        coll = destinoint_;
        c = 0;
        indice1 = 0;

        while (src < flujoint_) {
            //printf("Origen: %d \n", src);

```

```

        //printf("Destino: %d \n",dst);
        valor1 = hashkey_6(src,sport,dst,dport,pid);
        //printf("Hash: %d \n",valor1);
        tabla << valor1 << "\n";
        indice1 = valor1;
        campos6.numhash_ = valor1;
        campos6.BW_ = coll;
        campos6.colision_ = 0;
        campos6.origen_ = src;
        campos6.destino_ = dst;
        registrar_6(campos6, 0, indice1); //Llena los datos en la tabla hash
        ++src;
        ++dst;
        ++coll;
    }
    return 8;
}

//*****

// GET-HASH: Encuentra una entrada en la tabla hash

int get_hash_6(int valorhash, nsaddr_t src, nsaddr_t dst) {

    //printf("valorhash: %d \n",valorhash);
    int vfid1;

    campos6.numhash_ = valorhash;
    campos6.origen_ = src;
    campos6.destino_ = dst;
    vfid1 = registrar_6(campos6, 1, 0);
    return (vfid1);
}

//*****
// COMMAND: Método principal del .cc

virtual int command(int argc, const char*const* argv);

//*****
// VARIABLES UTILIZADAS EN LA CLASE INTSERVCLASSIFIER:

int          flujoint_;
int          destinoint_;
int          keylen2_;
int          hash2_;
int          masc1_;
int          x1_;
int          x2_;
int          tablah[15];
int          idprot_;
puntstructcampos6;
rescol      colisiones6;
int          c;

//*****
// REGISTRAR: Metodo de acceso a la tabla de BW y Colisiones.

int registrar_6(puntstruct& tab1, int a1, int indice1) {

    int      a;
    int      nhash1;
    int      rtdocol1;
    int      rtdoflow1;
    int      valcol;
    int      valhash;
    int      y;

```

```

a = 0;
nhash1 = 0;
rtdocol1 = 0;
rtdoflow1 = 0;
valcol = 0;
valhash = 0;
y = 0;

static int hstaba [2500] [5]; // [Filas] [Columnas]
//a = 0 --> Llenar tabla
//a = 1 --> Buscar Datos
//a = 2 --> Comparar datos

if (a1 == 0) {
    a = indice1;
    valcol = hstaba [a] [3];
    valhash = hstaba [a] [1];
    if (valhash == indice1) {
        if (valcol == 0) { // Existe un numero hash igual, pero no tenia colisión
            hasta esta entrada.
            colisiones6.numhash1_ = hstaba [a] [1];
            colisiones6.origen1_ = hstaba [a] [4];
            colisiones6.destino1_ = hstaba [a] [5];
            colisiones6.BW1_ = hstaba [a] [2];
            hstaba [a] [3] = 1;
            resolver_6(colisiones6, 0);
            colisiones6.numhash1_ = tab1.numhash_;
            colisiones6.origen1_ = tab1.origen_;
            colisiones6.destino1_ = tab1.destino_;
            colisiones6.BW1_ = tab1.BW_;
            resolver_6(colisiones6, 0);
            return 0;
        } else if (valcol == 1) { // Existe un numero hash igual, con una colision
            anterior
            colisiones6.numhash1_ = tab1.numhash_;
            colisiones6.origen1_ = tab1.origen_;
            colisiones6.destino1_ = tab1.destino_;
            colisiones6.BW1_ = tab1.BW_;
            resolver_6(colisiones6, 0);
            return 0;
        }
    } else {
        hstaba [a] [1] = tab1.numhash_; // Primera entrada de este numero hash
        hstaba [a] [2] = tab1.BW_;
        hstaba [a] [3] = tab1.colision_;
        hstaba [a] [4] = tab1.origen_;
        hstaba [a] [5] = tab1.destino_;
        return 0;
    }
}
return 0;
} else if (a1 == 1) {
    a = tab1.numhash_;
    rtdocol1 = hstaba [a] [3];
    nhash1 = hstaba [a] [1];
    if (rtdocol1 == 0) {
        rtdoflow1 = hstaba [a][2];
        return (rtdoflow1);
    } else {
        colisiones6.numhash1_ = tab1.numhash_;
        colisiones6.origen1_ = tab1.origen_;
        colisiones6.destino1_ = tab1.destino_;
        y = resolver_6(colisiones6, 1);
        rtdoflow1 = y;
        return (rtdoflow1);
    }
}
return (int) rtdoflow1;
}

```

```

        return (rtdoflow1);
    }

//*****
// RESOLVER:

int resolver_6(rescol& punt1, int k1) {

    int        vcmp;
    int        vcmp2;
    int        maxf1;
    int        vcolis1;
    int        vquint1;
    int        reserva1;
    vcmp = 0;
    vcmp2 = 0;
    maxf1 = 0;
    vcolis1 = 0;
    vquint1 = 0;
    reserva1 = 0;
    static int tablacoli [2500] [4]; // [Filas] [Columnas]
    //k = 0 --> Llenar tabla
    //k = 1 --> Buscar Datos

    if (k1 == 0) {
        c = c + 1;
        //printf("colision: %d \n",c);
        tablacoli [c] [1] = punt1.numhash1_;
        tablacoli [c] [2] = punt1.origen1_;
        tablacoli [c] [3] = punt1.destino1_;
        tablacoli [c] [4] = punt1.BW1_;

        return (int) 0;

    } else if (k1 == 1) {
        maxf1 = flujoint_; //CAMBIAR POR EL ORIGINAL.
        for (c=0;c<maxf1;c++) { // NO OLVIDAR DAR EL VALOR DE MAXF
            vcmp = punt1.numhash1_;
            vcmp2 = punt1.origen1_;
            vcolis1 = tablacoli [c] [1];
            if (vcmp == vcolis1) {
                vquint1 = tablacoli [c] [2];
                if (vcmp2 == vquint1) {
                    reserva1 = tablacoli [c] [4];
                    c = 1300;
                }
            }
        }
        return (int) reserva1;
    }
    return (reserva1);
}

//*****
//HASHKEY: Metodo que se utiliza para calcular el numero hash con la quintupla.

int hashkey_6(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

    int        w;
    int        dir1;
    int        dir2;
    int        dir3;

    dir1 = 0;
    dir2 = 0;
    dir3 = 0;
    w = 0;

```

```

x1_ = 0;
x2_ = 0;
masc1_ = 0;
hash2_ = 0;

//Calculo hash
masc1_ = 0xffff000; //Mask 1
x1_ = dir1;
x1_ &= masc1_;
x1_ >>= 3;
tablah[0] = x1_; //Variable 1
masc1_ = 0x00000fff; //Mask 2
x1_ = dir1;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0xff000000; //Mask 3
x2_ = dir2;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[1] = x1_; //Variable 2
masc1_ = 0x00ffff0; //Mask 4
x1_ = dir2;
x1_ &= masc1_;
x1_ >>= 1;
tablah[2] = x1_; //Variable 3
masc1_ = 0x0000000f; //Mask 5
x1_ = dir2;
x1_ &= masc1_;
x1_ <<= 4;
masc1_ = 0xffff0000; //Mask 6
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[3] = x1_; //Variable 4
masc1_ = 0x0000ffff; //Mask 7
x1_ = dir3;
x1_ &= masc1_;
x1_ <<= 1;
masc1_ = 0xf0000000; //Mask 8
x2_ = src;
x2_ &= masc1_;
x2_ >>= 7;
x1_ |= x2_;
tablah[4] = x1_; //Variable 5
masc1_ = 0x0ffff00; //Mask 9
x1_ = src;
x1_ &= masc1_;
x1_ >>= 2;
tablah[5] = x1_; //Variable 6
masc1_ = 0x000000ff; //Mask 10
x1_ = src;
x1_ &= masc1_;
x1_ <<= 3;
masc1_ = 0xff000000; //Mask 11
x2_ = dir1;
x2_ &= masc1_;
x2_ >>= 5;
x1_ |= x2_;
tablah[6] = x1_; //Variable 7
masc1_ = 0x000ffff; //Mask 12
x1_ = dir1;
x1_ &= masc1_;
tablah[7] = x1_; //Variable 8
masc1_ = 0xffff000; //Mask 13
x1_ = dir2;

```

```

x1_ &= masc1_;
x1_ >>= 3;
tablah[8] = x1_; //Variable 9
masc1_ = 0x00000fff; //Mask 14
x1_ = dir2;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0xff000000; //Mask 15
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[9] = x1_; //Variable 10
masc1_ = 0x00ffff0; //Mask 16
x1_ = dir3;
x1_ &= masc1_;
x1_ >>= 1;
tablah[10] = x1_; //Variable 11
masc1_ = 0x0000000f; //Mask 17
x1_ = dir3;
x1_ &= masc1_;
x1_ <<= 4;
masc1_ = 0xffff0000; //Mask 18
x2_ = dst;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[11] = x1_; //Variable 12
masc1_ = 0x0000ffff; //Mask 19
x1_ = dst;
x1_ &= masc1_;
x1_ <<= 1;
masc1_ = 0x0000f000; //Mask 20
x2_ = sport;
x2_ &= masc1_;
x2_ >>= 3;
x1_ |= x2_;
tablah[12] = x1_; //Variable 13
masc1_ = 0x00000fff; //Mask 21
x1_ = sport;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0x0000ff00; //Mask 22
x2_ = dport;
x2_ &= masc1_;
x2_ >>= 2;
x1_ |= x2_;
tablah[13] = x1_; //Variable 14
masc1_ = 0x000000ff; //Mask 23
x1_ = dport;
x1_ &= masc1_;
x1_ <<= 2;
x2_ = pid;
x2_ &= masc1_;
x1_ |= x2_;
tablah[14] = x1_; //Variable 15

for (w=0;w<15;w++) {
    hash2_ ^= tablah[w];
}

//printf("#hash: %d \n",hash2_);

return (int) hash2_;
}
};

```

CLASSIFIER-INTSERV6.CC:

```
#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /cvsroot/nsnam/ns-2/classifier/classifier-IntServ6.cc,v 1.30 2005/09/18 23:33:31 tomh Exp $
(LBL)";
#endif

extern "C" {
#include <ttl.h>
}

#include <stdlib.h>
#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include "classifier-IntServ6.h"

#include <math.h>
#include <time.h>
#include <sys/time.h>
#define FIJAR_TMP() gettimeofday(&temp_1, 0)
#define FIN_TMP() gettimeofday(&temp_2, 0)
#include <iostream>
#include <fstream>

ofstream archivo("clintserv6.tr");

/***** IntServ6Classifier Methods *****/

int IntServ6Classifier::classify(Packet *p) {

    float y=0,z=0;
    struct timeval temp_1, temp_2;
    FIJAR_TMP();

    busqueda_6(p);

    FIN_TMP();
    y = ((temp_2.tv_sec-temp_1.tv_sec)+(float)(temp_2.tv_usec-temp_1.tv_usec)/1000000);
    z = 0.000005;

    if (y < 0.000005){
        archivo << y << "\n";
    } else {
        archivo << z << "\n";
    }

    return (2);

} // IntServ6Classifier::classify

int IntServ6Classifier::command(int argc, const char*const* argv)
{
    if (argc == 4) {
        /*$classifier adc-num src dst */
        if (strcmp(argv[1], "adc-num") == 0) {
            nsaddr_t src = atoi(argv[2]);
            int16_t sport = 0;
            nsaddr_t dst = atoi(argv[3]);
            int16_t dport = 0;
            int pid = 0;
            int n = set_hash_6(src,sport,dst,dport,pid);
            //printf("Resultado: %d \n",n);
            return TCL_OK;
        }
    }
}
```

```

    }
    } return (Classifier::command(argc, argv));
}

/***** TCL linkage *****/
static class IntServ6ClassifierClass : public TclClass {
public:
    IntServ6ClassifierClass() : TclClass("Classifier/IntServ6") {}
    TclObject* create(int, const char*const*) {
        return (new IntServ6Classifier());
    }
} class_intserv6_classifier;

```

HOSTORIGEN.H:

```

#include "config.h"
#include "packet.h"
#include "ip.h"
#include "classifier.h"
#include <fstream.h>

ofstream tabla1("nhashintserv6.ods"); // constructor de ofstream

class StartClassifier : public Classifier {
public:

    int classify(Packet *p);

    int      masc1_;
    int      x1_;
    int      x2_;
    int      hash2_;
    int      tablah[15];

    int set_num(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

        int a;

        a = hashkey(src,sport,dst,dport,pid);
        tabla1 << a << "\n";
        //printf("a = %d \n",a);

        return (a) ;
    }

    int calculo(Packet *p) {
        hdr_ip* h = hdr_ip::access(p);
        return set_num(mshift(h->saddr()),mshift(h->sport()),mshift(h->daddr()),mshift(h->dport()),h-
>protid());
    }

    int hashkey(nsaddr_t src, int16_t sport, nsaddr_t dst, int16_t dport, int pid) {

        int      w;
        int      dir1;
        int      dir2;
        int      dir3;

        dir1 = 0;
        dir2 = 0;
        dir3 = 0;
        w = 0;
        x1_ = 0;

```

```

x2_ = 0;
masc1_ = 0;
hash2_ = 0;

//Calculo hash
masc1_ = 0xffff000; //Mask 1
x1_ = dir1;
x1_ &= masc1_;
x1_ >>= 3;
tablah[0] = x1_; //Variable 1
masc1_ = 0x00000fff; //Mask 2
x1_ = dir1;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0xff000000; //Mask 3
x2_ = dir2;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[1] = x1_; //Variable 2
masc1_ = 0x00ffff0; //Mask 4
x1_ = dir2;
x1_ &= masc1_;
x1_ >>= 1;
tablah[2] = x1_; //Variable 3
masc1_ = 0x0000000f; //Mask 5
x1_ = dir2;
x1_ &= masc1_;
x1_ <<= 4;
masc1_ = 0xffff0000; //Mask 6
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[3] = x1_; //Variable 4
masc1_ = 0x0000ffff; //Mask 7
x1_ = dir3;
x1_ &= masc1_;
x1_ <<= 1;
masc1_ = 0xf0000000; //Mask 8
x2_ = src;
x2_ &= masc1_;
x2_ >>= 7;
x1_ |= x2_;
tablah[4] = x1_; //Variable 5
masc1_ = 0x0ffff00; //Mask 9
x1_ = src;
x1_ &= masc1_;
x1_ >>= 2;
tablah[5] = x1_; //Variable 6
masc1_ = 0x000000ff; //Mask 10
x1_ = src;
x1_ &= masc1_;
x1_ <<= 3;
masc1_ = 0xfff00000; //Mask 11
x2_ = dir1;
x2_ &= masc1_;
x2_ >>= 5;
x1_ |= x2_;
tablah[6] = x1_; //Variable 7
masc1_ = 0x000ffff; //Mask 12
x1_ = dir1;
x1_ &= masc1_;
tablah[7] = x1_; //Variable 8
masc1_ = 0xffff000; //Mask 13
x1_ = dir2;
x1_ &= masc1_;

```

```

x1_ >>= 3;
tablah[8] = x1_; //Variable 9
masc1_ = 0x00000fff; //Mask 14
x1_ = dir2;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0xff000000; //Mask 15
x2_ = dir3;
x2_ &= masc1_;
x2_ >>= 6;
x1_ |= x2_;
tablah[9] = x1_; //Variable 10
masc1_ = 0x00ffff0; //Mask 16
x1_ = dir3;
x1_ &= masc1_;
x1_ >>= 1;
tablah[10] = x1_; //Variable 11
masc1_ = 0x0000000f; //Mask 17
x1_ = dir3;
x1_ &= masc1_;
x1_ <<= 4;
masc1_ = 0xffff0000; //Mask 18
x2_ = dst;
x2_ &= masc1_;
x2_ >>= 4;
x1_ |= x2_;
tablah[11] = x1_; //Variable 12
masc1_ = 0x0000ffff; //Mask 19
x1_ = dst;
x1_ &= masc1_;
x1_ <<= 1;
masc1_ = 0x0000f000; //Mask 20
x2_ = sport;
x2_ &= masc1_;
x2_ >>= 3;
x1_ |= x2_;
tablah[12] = x1_; //Variable 13
masc1_ = 0x00000fff; //Mask 21
x1_ = sport;
x1_ &= masc1_;
x1_ <<= 2;
masc1_ = 0x0000ff00; //Mask 22
x2_ = dport;
x2_ &= masc1_;
x2_ >>= 2;
x1_ |= x2_;
tablah[13] = x1_; //Variable 14
masc1_ = 0x000000ff; //Mask 23
x1_ = dport;
x1_ &= masc1_;
x1_ <<= 2;
x2_ = pid;
x2_ &= masc1_;
x1_ |= x2_;
tablah[14] = x1_; //Variable 15

for (w=0;w<15;w++) {
    hash2_ ^= tablah[w];
}

}

return (hash2_);
}

};

```

HOSTORIGEN.CC:

```
#include "IntServ6.h"
#include "ip.h"

int StartClassifier::classify(Packet *p)
{
    int hs = calculo(p);
    hdr_ip* h = hdr_ip::access(p);
    h->nhash() = hs;

    //printf("xhash = %d\n",h->nhash());

    return (1);
}

/***** TCL linkage *****/
static class StartClassifierClass : public TclClass {
public:
    StartClassifierClass() : TclClass("Classifier/Start") {}
    TclObject* create(int, const char*const*) {
        return (new StartClassifier());
    }
} class_start_classifier;
```

CONFIG.H:

```
* @(#) $Header: /cvsroot/nsnam/ns-2/config.h,v 1.58 2007/01/01 17:38:41 mweigle Exp $ (LBL)
*/

#ifndef ns_config_h
#define ns_config_h

#define MEMDEBUG_SIMULATIONS

/* pick up standard types */
#include <sys/types.h>
#if STDC_HEADERS
#include <stdlib.h>
#include <stddef.h>
#endif

/* get autoconf magic */
#ifdef WIN32
#include "autoconf-win32.h"
#else
#include "autoconf.h"
#endif

/* after autoconf (and HAVE_INT64) we can pick up tclcl.h */
#ifndef stand_alone
#ifdef __cplusplus
#include <tclcl.h>
#endif /* __cplusplus */
#endif

/* handle stl and namespaces */
/*
 * add u_char and u_int
 * Note: do NOT use these expecting them to be 8 and 32 bits long...
 * use {u_int{8,16,32}_t if you care about size.
 */
```

```

/* Removed typedef and included checks in the configure.in
typedef unsigned char u_char;
typedef unsigned int u_int;
*/

typedef int32_t nsaddr_t;
typedef int32_t nsmask_t;

/* 32-bit addressing support */
struct ns_addr_t {
    int32_t addr_;
    int32_t port_;
#ifdef __cplusplus
    bool isEqual (ns_addr_t const &o) {
        return ((addr_ == o.addr_) && (port_ == o.port_))?true:false;
    }
#endif /* __cplusplus */
};

// ESTRUCTURA DE BW Y COLISIONES
struct puntstruct {
    int    numhash_;
    int    BW_;
    int    colision_;
    int    origen_;
    int    destino_;
};

// ESTRUCTURA DE LA TABLA DE COLISIONES
struct rescol {
    int    numhash1_;
    int    origen1_;
    int    destino1_;
    int    BW1_;
};

/* 64-bit integer support */
#ifndef STRTOI64
#ifdef defined(SIZEOF_LONG) && SIZEOF_LONG >= 8
#define STRTOI64 strtol
#define STRTOI64_FMTSTR "%ld"
/* #define STRTOI64(S) strtol((S), NULL, 0) */

#elif defined(HAVE_STRTOQ)
#define STRTOI64 strtol
#define STRTOI64_FMTSTR "%lld"
/* #define STRTOI64(S) strtol((S), NULL, 0) */

#elif defined(HAVE_STRTOI64)
#define STRTOI64 strtol
#define STRTOI64_FMTSTR "%lld"
/* #define STRTOI64(S) strtol((S), NULL, 0) */
#endif
#endif

#define NS_ALIGN      (8)      /* byte alignment for structs (eg packet.cc) */

/* some global definitions */
#define TINY_LEN      8
#define SMALL_LEN     32
#define MID_LEN       256
#define BIG_LEN       4096
#define HUGE_LEN      65536
#define TRUE          1
#define FALSE         0

/* get definitions of bcopy and/or memcpy

```

```

* Different systems put them in string.h or strings.h, so get both
* (with autoconf help).
*/
#ifdef HAVE_STRING_H
#include <string.h>
#endif /* HAVE_STRING_H */
#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif /* HAVE_STRINGS_H */

#ifdef HAVE_BZERO
#define bzero(dest,count) memset(dest,0,count)
#endif
#ifdef HAVE_BCOPY
#define bcopy(src,dest,size) memcpy(dest,src,size)
#endif

#include <stdlib.h>

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif /* HAVE_UNISTD_H */

#ifdef HAVE_TIME_H
#include <time.h>
#endif /* HAVE_TIME_H */

#ifdef HAVE_ARPA_INET_H
#include <arpa/inet.h>
#endif /* HAVE_ARPA_INET_H */

#if (defined(__hpux) || defined(_AIX)) && defined(__cplusplus)
/* these definitions are perhaps vestigial */
extern "C" {
int strcasecmp(const char *, const char *);
clock_t clock(void);
#if !defined(__hpux)
int gethostid(void);
#endif
time_t time(time_t *);
char *ctime(const time_t *);
}
#endif

#if defined(NEED_SUNOS_PROTOS) && defined(__cplusplus)
extern "C" {
struct timeval;
struct timezone;
int gettimeofday(struct timeval*, ...);
int ioctl(int fd, int request, ...);
int close(int);
int strcasecmp(const char*, const char*);
int srandom(int); /* (int) for sunos, (unsigned) for solaris */
int random();
int socket(int, int, int);
int setsockopt(int s, int level, int optname, void* optval, int optlen);
struct sockaddr;
int connect(int s, struct sockaddr*, int);
int bind(int s, struct sockaddr*, int);
struct msghdr;
int send(int s, void*, int len, int flags);
int sendmsg(int, struct msghdr*, int);
int recv(int, void*, int len, int flags);
int recvfrom(int, void*, int len, int flags, struct sockaddr*, int);
int gethostid();
int getpid();
int gethostname(char*, int);

```

```

void abort();
}
#endif

#if defined(SOLARIS_MIN_MAX)
/* Macros for min/max. */
#ifndef MIN
#define MIN(a,b) (((a)<(b))?(a):(b))
#endif /* MIN */
#ifndef MAX
#define MAX(a,b) (((a)>(b))?(a):(b))
#endif /* MAX */
#endif

#if defined(NEED_SUNOS_PROTOS) || defined(solaris)
extern "C" {
#if defined(NEED_SUNOS_PROTOS)
    caddr_t sbrk(int incr);
#endif
    int getrusage(int who, struct rusage* rusage);
}
#endif

#if defined(__SUNPRO_CC)
#include <cmath>

static double log(const int x)
{ return log((double)x); }

static double log10(const int x)
{ return log10((double)x); }

static double pow(const int x, const int y)
{ return std::pow((double)x,(double)y); }

static double pow(const int x, const double y)
{ return std::pow((double)x,y); }
#endif

#ifdef WIN32

#include <windows.h>
#include <winsock.h>
#include <time.h>          /* For clock_t */

#include <minmax.h>
#define NOMINMAX
#undef min
#undef max
#undef abs

#define MAXHOSTNAMELEN      256

#define _SYS_NMLN          9
struct utsname {
    char sysname[_SYS_NMLN];
    char nodename[_SYS_NMLN];
    char release[_SYS_NMLN];
    char version[_SYS_NMLN];
    char machine[_SYS_NMLN];
};

typedef char *caddr_t;

struct iovec {
    caddr_t   iov_base;

```

```

        int         iov_len;
};

#ifndef TIMEZONE_DEFINED_
#define TIMEZONE_DEFINED_
struct timezone {
    int tz_minuteswest;
    int tz_dsttime;
};
#endif

typedef int pid_t;
typedef int uid_t;
typedef int gid_t;

#if defined(__cplusplus)
extern "C" {
#endif

int uname(struct utsname *);
int getopt(int, char * const *, const char *);
int strcasecmp(const char *, const char *);
/* these shouldn't be used/needed, even on windows */
/* #define srand srand */
/* #define random rand */
int gettimeofday(struct timeval *p, struct timezone *z);
int gethostid(void);
int getuid(void);
int getgid(void);
int getpid(void);
int nice(int);
int sendmsg(int, struct msghdr*, int);
/* Why this is here, inside a #ifdef WIN32 ??
#ifdef WIN32
    time_t time(time_t *);
#endif
*/
#define strncasecmp _strnicmp
#if defined(__cplusplus)
}
#endif

#ifdef WSAECONNREFUSED
#define ECONNREFUSED WSAECONNREFUSED
#define ENETUNREACH WSAENETUNREACH
#define EHOSTUNREACH WSAEHOSTUNREACH
#define EWOULDBLOCK WSAEWOULDBLOCK
#endif /* WSAECONNREFUSED */

#ifdef M_PI
#define M_PI 3.14159265358979323846
#endif /* M_PI */

#endif /* WIN32 */

#ifdef sgi
#include <math.h>
#endif

/* Declare our implementation of snprintf() so that ns etc. can use it. */
#ifndef HAVE_SNPRINTF
#if defined(__cplusplus)
extern "C" {
#endif
    extern int snprintf(char *buf, int size, const char *fmt, ...);
#if defined(__cplusplus)
}
#endif

```

```

#endif
#endif

/***** These values are no longer required to be hardcoded -- mask and shift values are
        available from Class Address. *****/

/* While changing these ensure that values are consistent with tcl/lib/ns-default.tcl */
/* #define NODEMASK      0xffff */
/* #define NODESHIFT     8 */
/* #define PORTMASK      0xff */

#endif

```

IP.H:

```

* @(#) $Header: /cvsrcroot/nsnam/ns-2/common/ip.h,v 1.16 2006/02/22 13:32:23 mahrenho Exp $
*/

/* a network layer; basically like IPv6 */
#ifndef ns_ip_h
#define ns_ip_h

#include "config.h"
#include "packet.h"

#define IP_HDR_LEN    20
#define IP_DEF_TTL    32

// The following undef is to suppress warnings on systems where
// IP_BROADCAST is defined.
#ifdef IP_BROADCAST
#undef IP_BROADCAST
#endif

// #define IP_BROADCAST ((u_int32_t) 0xffffffff)
static const u_int32_t IP_BROADCAST = ((u_int32_t) 0xffffffff);

struct hdr_ip {
    /* common to IPv{4,6} */
    ns_addr_t    src_;
    ns_addr_t    dst_;
    int          ttl_;

    int          idprot_;
    int          xhash_;

    /* Monarch extn */
    // u_int16_t sport_;
    // u_int16_t dport_;

    /* IPv6 */
    int          fid_;    /* flow id */
    int          prio_;

    static int offset_;
    inline static int& offset() { return offset_; }
    inline static hdr_ip* access(const Packet* p) {
        return (hdr_ip*) p->access(offset_);
    }

    /* per-field member access functions */
    ns_addr_t& src() { return src_; }
    ns_addr_t& saddr() { return src_.addr_; }
    int32_t& sport() { return src_.port_; }

```

```

        ns_addr_t& dst() { return (dst_); }
        nsaddr_t& daddr() { return (dst_.addr_); }
        int32_t& dport() { return dst_.port_; }
        int& ttl() { return (ttl_); }
        /* ipv6 fields */
        int& flowid() { return (fid_); }
        int& prio() { return (prio_); }

        int& protid() { return (idprot_); }
        int& nhash() { return (xhash_); }
};

#endif

```

IP.CC:

```

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /cvsroot/nsnam/ns-2/common/ip.cc,v 1.8 1998/08/12 23:41:05 gnguyen Exp $";
#endif

#include "packet.h"
#include "ip.h"

int hdr_ip::offset_;

static class IPHeaderClass : public PacketHeaderClass {
public:
    IPHeaderClass() : PacketHeaderClass("PacketHeader/IP",
                                         sizeof(hdr_ip)) {
        bind_offset(&hdr_ip::offset_);
    }
    void export_offsets() {
        field_offset("src_", OFFSET(hdr_ip, src_));
        field_offset("dst_", OFFSET(hdr_ip, dst_));
        field_offset("ttl_", OFFSET(hdr_ip, ttl_));
        field_offset("fid_", OFFSET(hdr_ip, fid_));
        field_offset("prio_", OFFSET(hdr_ip, prio_));
        field_offset("idprot_", OFFSET(hdr_ip, idprot_));
        field_offset("xhash_", OFFSET(hdr_ip, xhash_));
    }
} class_iphdr;

```

CBR.TCL:

```

source topologia.tcl
global ns n r d cl a b c reg delaywfq

set      a      1
set      b      1
set      c      0

remove-all-packet-headers
add-packet-header IP TCP

set ns [new Simulator]

set nam [open out.nam w]
$ns namtrace-all $nam

set      reg      [open reg.tr w]

for {set i 0} {$i <= $b} {incr i} {

```

```

set f($i) [open fl($i).tr w]
$ns trace-all $f($i)}

proc record {} {
global sink ns f nam c b a reg
set time 0.5
    for {set i 0} {$i <= $c} {incr i} {
        set bw($i) [$sink([expr $i+$b]) set bytes_]
        set now [$ns now]
        for {set i 0} {$i <= $c} {incr i} {
            puts $f($i) "$now [expr $bw($i)/$time*8]"
        }
        for {set i 0} {$i <= $c} {incr i} {
            if {$now == 4.5} {
                puts $reg "[expr $bw($i)/$time*8]"
            } else {}
        }
    }
    for {set i $b} {$i <= $a} {incr i} {
        $sink($i) set bytes_ 0
    }
    $ns at [expr $now+$time] "record"
}

proc finish {} {
global ns f nam c fl reg delaywfq
$ns flush-trace
    for {set i 0} {$i <= $c} {incr i} {
        close $f($i)
        close $nam
        close $reg

        puts stdout "retardo promedio de la cola wfq: [$delaywfq mean]"

        exec cat reg.tr | awk {{sum+=$1;print sum>"sbw1.tr"}}
        exec cat clintserv.tr | awk {{sum+=$1;print sum>"timeintserv1.tr"}}

        for {set i 0} {$i <= $c} {incr i} {
            exec xgraph fl($i).tr -geometry 800x400 -x Tiempo &
        }

        exec nam out.nam &
        exit 0
    }

crear_topo

    for {set i 0} {$i <= $c} {incr i} {
        $cl setqueue $i [expr $i+$b]}

    for {set i 0} {$i <= $c} {incr i} {
        $cl setweight [expr $i+$b] 0.01;
        $cl setlength [expr $i+$b] 5;}

    set monitorwfq [$ns monitor-queue $r $d stdout]
    set gpi [$monitorwfq get-pkts-integrator]
    set sam [new Samples]
    $monitorwfq set-delay-samples $sam
    set delaywfq [$monitorwfq get-delay-samples]

#UDP Sources
    for {set i 0} {$i <= $c} {incr i} {
        set udp($i) [new Agent/UDP]
        $ns attach-agent $n($i) $udp($i)
        $udp($i) set fid_ $i }

#CBR sources
    for {set i 0} {$i <= $c} {incr i} {
        set cbr($i) [new Application/Traffic/CBR]
        $cbr($i) attach-agent $udp($i)}

```

```

#UDP Destinations
for {set i $b} {$i <= $a} {incr i} {
    set sink($i) [new Agent/LossMonitor]
    $ns attach-agent $n($i) $sink($i)
}

for {set i 0} {$i <= $c} {incr i} {
    $ns connect $udp($i) $sink([expr $i+$b])
}

$ns at 0.0 "record"

for {set i 0} {$i <= $c} {incr i} {
    $ns at 0.0 "$cbr($i) start"
}
$ns at 5.0 "finish"
$ns run

```

TOPOLOGIA.TCL:

```

proc crear_topo {} {
    global ns n r d cl a b c

    set a 1
    set b 1
    set c 0

    set r [$ns node]
    set d [$ns node]

    for {set i 0} {$i <= $a} {incr i} {
        set n($i) [$ns node]
    }

    Classifier/IntServ set flujo_ $b
    set cls [new Classifier/IntServ]
    $r insert-entry RtModule/Base $cls "1"
    $cls add-num 0 $b

    for {set i 0} {$i <= $c} {incr i} {
        $ns duplex-link $n($i) $r 5Mb 1ms DropTail
    }

    for {set i $b} {$i <= $a} {incr i} {
        $ns duplex-link $n($i) $d 5Mb 1ms DropTail
    }

    $ns simplex-link $r $d 2Mb 2ms WFQ
    $ns simplex-link $d $r 2Mb 2ms DropTail

    set cl [new WFQAggregClassifier]
    $ns wfqclassifier-install $r $d $cl
}

```

CBR6.TCL:

```

source topologia6.tcl
global ns n r d cl a b c reg delaywfq

set a 1
set b 1
set c 0

```

```

remove-all-packet-headers
add-packet-header IP TCP

set ns [new Simulator]

set nam [open out.nam w]
$ns namtrace-all $nam

set      reg      [open reg.tr w]

for {set i 0} {$i <= $b} {incr i} {
    set f($i) [open fl($i).tr w]
    $ns trace-all $f($i)}

proc record {} {
    global sink ns f nam c b a reg
    set time 0.5
    for {set i 0} {$i <= $c} {incr i} {
        set bw($i) [$sink([expr $i+$b]) set bytes_]}
    set now [$ns now]
    for {set i 0} {$i <= $c} {incr i} {
        puts $f($i) "$now [expr $bw($i)/$time*8]" }
    for {set i 0} {$i <= $c} {incr i} {
        if {$now == 4.5} {
            puts $reg "[expr $bw($i)/$time*8]"
        } else {}
    }
    for {set i $b} {$i <= $a} {incr i} {
        $sink($i) set bytes_ 0
        $ns at [expr $now+$time] "record"
    }
}

proc finish {} {
    global ns f nam c fl reg delaywfq
    $ns flush-trace
    for {set i 0} {$i <= $c} {incr i} {
        close $f($i)}
    close $nam
    close $reg

    puts stdout "retardo promedio de la cola wfq: [$delaywfq mean]"

    exec cat reg.tr | awk {{sum+=$1;print sum>"sbw1.tr"}}
    exec cat clintserv.tr | awk {{sum+=$1;print sum>"timeintserv1.tr"}}
    exec cat plwfq.tr | awk {{sum+=$1;print sum>"timewfq1.tr"}}

    for {set i 0} {$i <= $c} {incr i} {
        exec xgraph fl($i).tr -geometry 800x400 -x Tiempo & }

    exec nam out.nam &
    exit 0
}

crear_topo

for {set i 0} {$i <= $c} {incr i} {
    $cl setqueue $i [expr $i+$b]}

for {set i 0} {$i <= $c} {incr i} {
    $cl setweight [expr $i+$b] 0.01;
    $cl setlength [expr $i+$b] 5;}

set monitorwfq [$ns monitor-queue $r $d stdout]
set gpi [$monitorwfq get-pkts-integrator]
set sam [new Samples]
$monitorwfq set-delay-samples $sam
set delaywfq [$monitorwfq get-delay-samples]

```

```

for {set i 0} {$i <= $c} {incr i} {
    set udp($i) [new Agent/UDP]
    $ns attach-agent $n($i) $udp($i)
    $udp($i) set fid_ $i }

for {set i 0} {$i <= $c} {incr i} {
    set cbr($i) [new Application/Traffic/CBR]
    $cbr($i) attach-agent $udp($i)}

for {set i $b} {$i <= $a} {incr i} {
    set sink($i) [new Agent/LossMonitor]
    $ns attach-agent $n($i) $sink($i)
}

for {set i 0} {$i <= $c} {incr i} {
    $ns connect $udp($i) $sink([expr $i+$b])
}

$ns at 0.0 "record"

for {set i 0} {$i <= $c} {incr i} {
    $ns at 0.0 "$cbr($i) start"
}

$ns at 5.0 "finish"
$ns run

```

TOPOLOGIA6.TCL:

```

proc crear_topo {} {
    global ns n r d cl a b c

    set a 1
    set b 1
    set c 0

    set r [$ns node]
    set d [$ns node]

    for {set i 0} {$i <= $a} {incr i} {
        set n($i) [$ns node]
    }

    for {set i 0} {$i <= $a} {incr i} {
        set cls($i) [new Classifier/Start]
        $n($i) insert-entry RtModule/Base $cls($i) "1"
    }

    Classifier/IntServ6 set flujoint_ $
    Classifier/IntServ6 set destinoint_ $b b
    set cl [new Classifier/IntServ6]
    $r insert-entry RtModule/Base $cl "2"
    $cl adc-num 0 $b

    for {set i 0} {$i <= $c} {incr i} {
        $ns duplex-link $n($i) $r 5Mb 1ms DropTail
    }

    for {set i $b} {$i <= $a} {incr i} {
        $ns duplex-link $n($i) $d 5Mb 1ms DropTail
    }

    $ns simplex-link $r $d 2Mb 2ms WFQ
    $ns simplex-link $d $r 2Mb 2ms DropTail

```

```
set ci [new WFQAggregClassifier]  
$ns wfqclassifier-install $r $d $ci  
}
```

7. GLOSARIO

Almacenamiento y Reenvío: Características de las redes que usan conmutadores de paquetes para reenviar paquetes. El nombre proviene de cada conmutador de la trayectoria al destino recibe un paquete y lo almacena temporalmente en memoria. Mientras tanto, el conmutador selecciona de manera continua un paquete de la cola en memoria, lo canaliza y lo transmite a la siguiente parada.

Ancho de banda: Medida de la capacidad de un sistema de transmisión. El ancho de banda se mide en hertz.

ARPA: Advanced Research Project Agency, Agencia de Investigación de proyectos avanzados.

ATM: Asynchronous Transfer Mode, Modo de Transferencia Asíncrona. Tecnología orientada a conexión definida por la ITU y el foro ATM. Al nivel más bajo, el ATM envía todos los datos en células fijas con 48 octetos de datos por célula.

Bits por segundo: Razón a la cual puede transmitir datos por una red. La cantidad de bits por segundo puede diferir de la razón de baudios puesto que es posible codificar más de 1 bit en un solo baudio.

Canal Virtual: Sinónimo de circuito virtual. El término canal virtual se usa en tecnologías como ATM.

Carga: De manera general, datos transportados en un paquete. La carga de un cuadro son los datos del cuadro; La carga de un datagrama es el área de datos del datagrama.

CBQ: Class Based Queuing, Colas basadas en clases.

CBR: Constant Bit Rate, Tasa constante de bits.

Congestionamiento: Condición en la que cada paquete que se envía por la red experimenta un retardo excesivo debido a que esta se encuentra sobrecargada de paquetes. A menos que el software del protocolo detecte el congestionamiento y reduzca la razón de envío de los paquetes, la red puede experimentar un colapso por congestionamiento.

CPU: Unidad Central de Proceso.

DARPA: Defense Advanced Research Project Agency

Datagrama IP: Forma de un paquete enviado por una inter-red TCP/IP. Cada datagrama tiene una cabecera que identifica tanto al transmisor como al receptor, seguida de datos.

Dirección Destino: Dirección en un paquete que especifica el destino último del paquete. En un cuadro de hardware, la dirección destino, debe ser una dirección de hardware. En un datagrama IP, la dirección destino debe ser una dirección IP.

Dirección IP: Dirección de 32 o 128 bits (IPV4 o IPv6) asignada a una computadora que usa protocolos TCP/IP. El transmisor debe conocer la dirección IP de la computadora destino antes de enviar un paquete.

DS: Differentiated Services, Servicios Diferenciados.

DSCP: Differentiated Services Code Point.

DOD: Department of Defense, Departamento de Defensa.

Dúplex: Transferencia simultánea en dos direcciones.

ECN: Explicit Congestion Notification, Notificación de congestión explícita.

Enrutador: Bloque de construcción básico de una inter-red. Un enrutador es una computadora que se conecta a 2 o más redes y reenvía paquetes de acuerdo con la información encontrada en su tabla de enrutamiento.

E/S: Entrada/Salida

FIFO: First In First Out, Primero en entrar es el Primero en salir.

FTP: File Transfer Protocol, Protocolo de transferencia de archivo.

Gateway: En términos de Internet un gateway es un dispositivo que enruta los datagramas. Más recientemente utilizado para hacer referencia a cualquier dispositivo de red que traduce los protocolos de un tipo de red a los de otra red.

GPS: Generalized Processor Sharing

H.323: Estándar de la ITU-T para voz y videoconferencia interactivo en tiempo real en redes de área local LAN e Internet.

Host: Es un ordenador que funciona como el punto de inicio y final de las transferencias de datos. Tiene una dirección de Internet única (dirección IP) y un nombre de dominio único o nombre de host.

HTTP: Hypertext Transfer Protocol, Protocolo de Transferencia de Hipertexto.

IEEE: Institute of Electric and Electronic Engineers, Instituto de Ingenierías Eléctrica y Electrónica.

IGMP: Internet Group Management Protocol, Protocolo de Gestión de Grupos de Internet.

Internet: Conjunto de redes conectadas entre sí que abarca todo el mundo y utiliza la NFSNET como columna vertebral. Internet es el término específico de una inter-red o de un conjunto de redes.

Inter-red: Grupos de redes conectadas mediante enrutadores que se configuran para pasar tráfico entre las computadoras conectadas a las redes del grupo. La mayor parte de las inter-redes usan protocolos TCP/IP.

IntServ: Integrate Services Internet QoS Model, Modelo de Calidad de Servicios Integrados de Internet.

IP: Internet Protocol, Protocolo de Internet. Protocolo que define tanto el formato de los paquetes usados en una Internet TCP/IP como el mecanismo de enrutamiento de un paquete a su destino.

IPng: IP next generation (IPv6)

IP Multicast: Extensión del protocolo Internet para dar soporte a comunicaciones multidifusión.

IPv4: Internet Protocol version 4, Protocolo de Internet version 4. La versión del IP actualmente usada en Internet. El IPv4 usa direcciones de 32 bits.

IPv6: Internet Protocol versión 6, Protocolo de Internet versión 6. Protocolo específico que ha sido propuesto por IETF como sucesor del IPv4. El IPv6 usa direcciones de 128 bits.

IS: Integrated Services, Servicios Integrados

ISDN: Integrated Service Data Network, Red Digital de Servicios Integrados (RDSI).

ISPs: Internet Service Providers, Proveedores del Servicio de Internet.

ITU-T : International Telecommunications Union – Telecommunications, Union Internacional de Telecomunicaciones – Telecomunicaciones.

Jitter: Es un término que se refiere al nivel de variación de retardo que introduce una red.

LAN: Local Area Network, Red de Area Local. Red que usa tecnología diseñada para abarcar un área geográfica pequeña.

MAC: Media Access Control, Control de Acceso al Medio.

MBONE: Multicast BackBone, Red Troncal de Multidifusión.

MG: Media Gateway

Mpbs: Millones de bits por segundo.

MPLS: Multiprotocol Label Switching, Multiprotocolo de Conmutación de paquetes.

MTA: Message Transport Agent, Agente de Transporte de Mensajes.

NCP: Network Control Protocol, Protocolo de Control de Red.

Nodo: Término usado informalmente para hacer referencia a un enrutador o a una computadora conectada a una red. El término se deriva de la teoría de gráficas.

NS-2: Network Simulator 2, Simulador de Redes versión 2.

OSI: Open System Interconnection, Modelo de Referencia de Interconexión de Sistemas Abiertos.

Packet Switching: Conmutación de Paquetes. Técnica de conmutación en la cuál los mensajes se dividen en paquetes antes de su envío. A continuación cada paquete se transmite en forma individual y puede incluso seguir rutas diferentes hasta su destino. Una vez que los paquetes lleguen a éste se agrupan para reconstruir el mensaje original.

Paquete: Fragmento de dato pequeño y auto contenido enviado por una red de computo. Cada paquete contiene una cabecera que identifica al transmisor y al receptor, así como los datos a entregar.

PDU: Protocol Data Unit, Unidad de Datos de Protocolo.

PHB: Per Hop Behavior, Comportamiento por Salto.

PPP: Point to Point Protocol, Protocolo Punto a Punto.

Protocolo: Las reglas que rigen el comportamiento o el método de operación de alguno de los aspectos de una red.

QoS: Quality of Service, Calidad de Servicio.

RDSI: Red Digital de Servicios Integrados.

RFC: Request For Comment.

RIP: Routing Information Protocol, Protocolo de Información.

Router: Encaminador, enrutador. Dispositivo que distribuye tráfico entre las redes. La decisión sobre a donde enviar los datos se realiza basándose en la información de nivel de red y tablas de direccionamiento. Es el nodo básico de una red IP.

RSVP: Resource Reservation Protocol, Protocolo de Reservación de Recursos.

SLA: Service Level Agreement, Acuerdo de Nivel de Servicio.

Subred: Subset. Parte de una red TCP/IP identificada por una parte de la dirección de Internet.

Tabla de Enrutamiento: Tabla usada por el software de enrutamiento para determinar el siguiente salto de un paquete. Se guarda en la memoria del enrutador.

TCP: Protocolo TCP/IP, que proporciona a los programas de aplicación, acceso al servicio de comunicación orientado a conexión. El TCP ofrece una entrega confiable y de flujo controlado. El TCP se ajusta a las condiciones cambiantes de Internet adaptando su esquema de transmisión.

TOS: Tipo de Servicio.

Tráfico: Un término general utilizado para describir la cantidad de datos que se encuentran en la columna vertebral de una red.

TTL: Time to Live, Tiempo de Vida de un paquete en la red.

UDP: User Datagram Protocol.

UMTS: Universal Mobile Telecommunications System.

VoIP: Voice over IP.

Web: Sinónimo de World Wide Web (WWW).

WFQ: Weighted Fair Queueing, Colas Equitativas Ponderadas.

WWW: World Wide Web. Sistemas de hipermedios usados en Internet en el que una página de información puede contener texto, imágenes, fragmentos de audio o video y referencias a otras páginas.

BIBLIOGRAFÍA

1. Ramirez, A.J.R., *Simulación de la propuesta de calidad de Servicio IntServ6*, in *Ingeniería Electrónica*. 2006, Universidad Pontificia Bolivariana: Bucaramanga.
2. Ian F. Akyildiz, W.S., Yogesh Sankarasubramaniam, Erdal Cayirci, *A Survey on Sensor Networks*. IEEE Communicatios Magazine, 2002: p. 13.
3. Stallings, W., *Comunicaciones y redes de computadores*. 6 ed. 2000: Prentice Hall.
4. Quemada, J., *Hacia una Internet de Nueva Generación*. 2004, Universidad Politécnica de Madrid: Madrid. p. 64.
5. Montserrat Tesouro Cid, J.P.A., *Evaluación y Utilización de Internet en la Educación*, in *Pixel-Bit*. 2004.
6. Alvarez, G.V., *Seguridad en Redes IP: Los Protocolos TCP/IP*.
7. Postel, J., *Internet Protocol*, in *RFC 791*. 1981, IETF: California.
8. Ortega, E.J., *Algoritmos de encolamiento en routers para una distribución justa de la capacidad de un enlace compartido en una red TCP/IP*, in *Departamento de Ingeniería de Sistemas e Industrial*. 2006, Universidad Nacional de Colombia: Colombia. p. 40.
9. Gutiérrez, A.T., *Análisis y Propuesta de un Esquema de Calidad de Servicio (QoS) para la Red de la Universidad de Colima*, in *Facultad de Ingeniería Mecánica y Eléctrica*. 2001, Universidad de Colima: Coquimatlán. p. 100.
10. S-B Lee, G.-S.A., X. Zhang, A.T. Campbell. *Evaluation of the INSIGNIA Signaling System*. in *8th IFIP International Conference on High Performance Networking*. 2000. Paris, France.
11. Domingo, A.N.y.J. *Analizador en tiempo real de calidad de servicio en redes IP*. in *Quality of Service Real-time Analyzer for IP Networks*.
12. Sebastián Andrés Alvarez Moraga, A.J.G.V., *Estudio y Configuración de Calidad de Servicio para Protocolos IPv4 e IPv6 en una Red de Fibra Óptica WDM*. Vol. 3. 2005, España: Universidad Técnica Federico Santa María. 104-113.
13. Boix, M.P., *Contribución al Estudio y Diseño de Mecanismos Avanzados de Servicio de Flujos Semi-elásticos en Internet con Garantías de Calidad de Servicio Extremo a Extremo*, in *Telecomunicaciones*. 2003: España. p. 207.
14. Gutiérrez, A.T., *Análisis y Propuesta de un Esquema de Calidad de Servicio (QoS) para la red de la Universidad de Colima*, in *Facultad de Ingeniería Mecánica y Eléctrica*. 2003, Universidad de Colima: Coquimatlán, Colima. p. 100.
15. Montañana, R., *Calidad de Servicio(QoS)*. 2006, Universidad de Valencia, Departamento de Informatica: España. p. 83.
16. www.ietf.org.
17. *Protocolo de Reservación de Recursos : RSVP*. Danysoft, 2006: p. 8.
18. Torres, A.G., *Visualización del monitoreo de tráfico de red*, in *Técnicas de visualización y diseño de interfaces gráficas*.
19. Wang, Z., *Internet QoS: Architectures and Mechanisms for Quality of Service*. 1st edition ed. 2001, March 15: Morgan Kaufmann. 256.

20. Orallo, E.H., *Reserva Eficiente de Recursos en Redes para Transmisión en Tiempo Real*, in *Departamento de Informática de Sistemas y Computadores*. 2001, Universidad Politécnica de Valencia: Valencia, España.
21. Angel Ramón Aguirre Gutierrez, W.O.G., Juan Esteban González, Ingrid Lawler, *Sistemas Multimedia Distribuidos*. 2002: Argentina.
22. Abbas, C.J.B., *Nuevas Soluciones en la Internet: IntServ y DiffServ*, Universidad de Brasilia.
23. S, D., *Internet Protocol Version 6 (IPv6)*, in *RFC 2460*. 1998, IETF.
24. Ibáñez, J.A.G., *Herramientas para Desarrollo de Software Educativo sobre Internet*, in *Facultad de Telemática*. 1999, Universidad de Colima: Colima. p. 31-33.
25. R.Braden, *Integrated Services in the Internet Architecture: an Overview*, in *RFC 1633*. 1994, IETF.
26. Jorge Escribano Salazar, C.G.G., Celia Seldas Alarcón, José Ignacio Moreno Novella, *Diffserv como una solución a la provisión de QoS en Internet*, Universidad Carlos III de Madrid: Madrid. p. 7.
27. García, C.G., *Propuesta de arquitectura de QoS en entorno inalámbrico 802.11e basado en Diffserv con ajuste dinámico de parámetros*, in *Departamento de Ingeniería Telemática*. 2006, Universidad Carlos III de Madrid: Leganés. p. 27-28.
28. L. Zhang, S.D., D.Estrin, S. Shenker, D. Zappala, D., *RSVP: a new resource ReSerVation Protocol*. Network, IEEE, 1993. 7(5): p. 8-18.
29. Martínez, E.M., *IPv6: El protocolo de Internet de la Nueva Generación*. Revista RED, 2004.
30. Blake, S., D. Black, and M. Carlson, *An Architecture for Differentiated Services*, in *RFC 2475*. 1998, IETF.
31. Navarro, D.F., *Controlador de Ancho de Banda*, in *Departamento de Teleinformática*. 2005, Universidad de Mendoza: Argentina. p. 124.
32. Aguilar, J.J.P., *Arquitectura de Servicios Integrados (IntServ)*.
33. García, J.A., *MBone: Arquitectura y Aplicaciones*.
34. Zhang, L., et al., *RSVP: A New Resource Reservation Protocol*. Network, IEEE, 1993. 7(5): p. 8-18.
35. Jhon Jairo Padilla Aguilar, J.P.A., Monica Karel Huerta, Xavier Hesselbach, *Soporte de QoS sobre IPv6 con IntServ6*. Gerencia Tecnologica Informatica, 2005. 4(8): p. 109.
36. J. Padilla, J.P., M. Huerta, X. Hesselbach. *IntServ6: An Approach to Support QoS over IPv6 Networks*. in *10th International Symposium on Computer and Communications*. 2005. Cartagena, España.
37. B, H., T. Mudigonda, M. Dahlin, H. Vin, *Impact of Network Protocols on Programmable Router Architectures*. 2003, Laboratory of Advanced Systems Reserach, Department of Computer Science, University of Texas at Austin: Austin, Texas.
38. Chee Hock, N., *Queueing Modeling Fundamentals*. 1997: Jhon Wiley & Sons Ltd. 222.

39. Yuguang Fang, Y.Z., *Call Admission Control Schemes and Performance Analysis in Wireless Mobile Networks*. Vehicular Technology, IEEE Transactions on, 2002. 51(2): p. 371-382.
40. María E. Villapol, P.L.G., *Usando RSVP para Soportar Servicios Adaptativos en Redes Móviles*, Universidad Central de Venezuela: Caracas. p. 12.
41. Aguilar, J.J.P., *Protocolo RSVP*.
42. González, A.J., *Tablas HASH*. 2002.
43. Lozano, M.Á.R., *Desarrollo de un Nodo Encaminador para Filtrado y Simulación de Tráfico en Subredes IP*, in *Departamento de Tecnología Electrónica*. 2003, Universidad de Málaga: Málaga. p. 76-83.
44. Lozano, M.Á.R., *Desarrollo de un Nodo Encaminador para Filtrado y Simulación Tráfico en subredes IP*, in *Ingeniería de Telecomunicación*. 2003, Universidad de Málaga: Malaga. p. 221.
45. Aguilar, J.J.P., *Modelos de Servicio*. 2003, Universidad Pontificia Bolivariana: Bucaramanga.
46. Orallo, E.H., *Reserva Eficiente de Recursos en Redes para Transmisión en Tiempo Real*, in *Departamento de Informática de Sistemas y Computadores*. 2001, Universidad Politécnica de Valencia: Valencia. p. 36.
47. Ana B. García, D.L., Arturo Azcorra, Luis Bellido, David Fernández, Julio Berrocal, *Soporte de Calidad de Servicio en Internet sobre Redes ATM*, Universidad Politécnica de Madrid: Madrid.
48. Cambroner, D.F., *Introducción al Protocolo IPv6*. 2001, ETSIT-UPM. p. 27.
49. Peralta, L., *IPv6 @UJI -Rev:22*. 2002. p. 36.
50. Pérez, M.A.O., *Protocolo de Encaminamiento en Origen con Identificadores no únicos para Redes Ad-Hoc de Dispositivos con Recursos Limitados*, in *Departamento de Ingeniería Telemática y Tecnología Electrónica*. 2006, Universidad Rey Juan Carlos: Móstoles. p. 50-53.
51. A. Conta, S.D., *IPv6 Flow Label Specification*, in *draft-ietf-ipv6-flow-label-03.txt*. 2002.
52. Flores, C.E.S., *QoS y mecanismos de transición IPv4/IPv6*, Universidad Carlos III de Madrid: Madrid. p. 5.
53. Partridge, C., *Using the Flow Label Field in IPv6*, in *RFC 1809*, IETF, Editor. 1995, IETF.
54. Aguilar, J.J.P., *Contribución al Soporte de Calidad del Servicio en Redes Móviles*. 2007, Universidad Politécnica de Cataluña: Barcelona. p. 226.
55. M., J.M.H., *NS2 - Network Simulator*. 2004: Valparaíso. p. 17.
56. Edward Alberto Jaimes Gonzalez, P.A.B.C., *Simulación de Flujos del Protocolo IPv6 mediante el Software Network Simulator*, in *Ingeniería Electrónica*. 2004, Universidad Pontificia Bolivariana: Bucaramanga.
57. Castellón, I.V., *Simulación de Redes de Computadores aplicado a docencia*. 2005. p. 33.
58. A. Triviño Cabrera, E.C.P., A. Ariza Quintana, *Implementación de Protocolos en el Network Simulator (NS-2)*, in *Departamento de Tecnología Electrónica*, Universidad de Málaga. p. 7.
59. M., J.M.H., *NS2-Network Simulator*. 2004: Valparaíso. p. 17.

- 60. Aladrén, M.C.D., *Diferenciación de Servicios y Mejora de la Supervivencia en Redes Ad-hoc Conectadas a Redes Fijas*, in *Departamento de Ingeniería Telemática*. 2005, Universidad Politécnica de Cataluña: Cataluña. p. 142-163.
- 61. Sosa, V.J.S., *Arquitectura para la Distribución de Documentos en un Sistema Distribuido a Gran Escala*, in *Departamento de Arquitectura de Computadores*. 2002, Universidad Politécnica de Cataluña: Cataluña. p. 62-68.
- 62. www.isi.edu/nsnam/ns/.
- 63. R. Braden, L., Zhang, *Resource Reservation Protocol (RSVP)*, in *RFC 2205*. 1997, IETF.
- 64. Sierra, J.C., *Programación Orientada a Objetos con C++*. Tercera Edición ed. 2004, México: AlfaOmega.
- 65. Harvey M. Deitel, P.J.D., *Como Programar en C++*. Cuarta Edición ed. 2003, México: Prentice Hall. 927.
- 66. Gottfried, B.S., *Programación en C*, ed. McGraw-Hill. 1996, México.
- 67. Alonso, S.H., *Breve Manual de Tcl*. 2004.
- 68. Castellón, I.V., *Manual de Referencia Otcl*. 2006.
- 69. Juan Suardíaz Muro, B.M.A.-H., *Diseño de un Sistema de Búsqueda de Ruta Óptima basada en una Interfaz Visual mediante la Herramienta Tcl/Tk*, in *Tecnología y Desarrollo*. 2006. p. 16.
- 70. Wang, Z., *Internet QoS: Architectures and Mechanisms for Quality of Service*. 1st edition ed. March 15, 2001: Morgan Kaufmann. 256.
- 71. Clerget, A., *C++ Class Hierarchy NS-2*.
- 72. <http://www.cclinf.polito.it/~s77950/>.