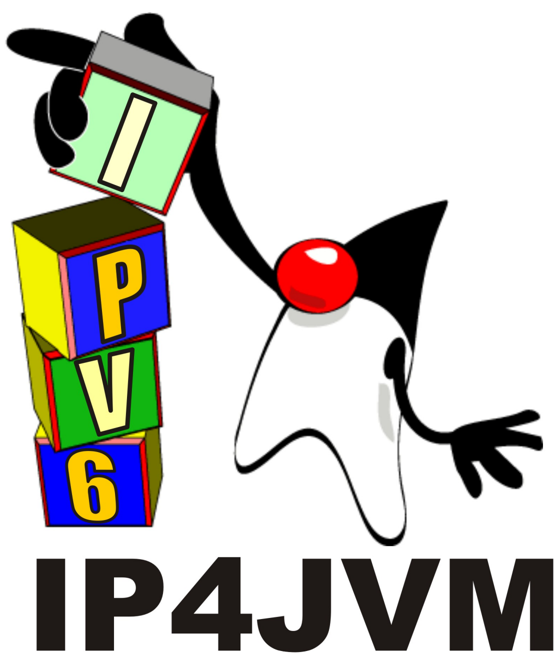


Proyecto de Grado 2007

IP4JVM

Informe Final

Autores: Roger Abelenda
Ignacio Corrales

Tutor: Ariel Sabiguero Yawelak

Instituto de Computación

Facultad de Ingeniería

Universidad de la República

Setiembre de 2008

Índice de contenido

1.RESUMEN **7**
1.1 Palabras Claves 7
1.2 Público Objetivo 8
1.3 Organización del trabajo 8
2.INTRODUCCIÓN **11**
2.1 Motivación 12
2.1.1 Networking en Java 12
2.1.2 Stack de Protocolos 13
2.1.3 Diferencias entre IPv6 e IPv4 15
2.1.4 Implementación en Java 17
3.ESTADO DEL ARTE **19**
3.1 Datos Generales de Proyecto anterior 19
3.2 Performance 19
3.3 Diseño 20
3.4 Portabilidad 20
3.5 Completitud 21
4.OBJETIVOS **23**
4.1 Performance 23
4.2 Diseño 23
4.3 Portabilidad 24
4.4 Completitud 24
4.5 Protocolo TCP 25
4.6 Integración de una aplicación con nuestro stack 25
4.7 Documentación Online 25
5.DESARROLLO DEL PROYECTO **27**
5.1 Planificación Inicial 27
5.2 Selección de entorno de trabajo e instalación 30
5.3 Cambios. Performance y Diseño 32
5.4 Migración a otra VM 41
5.5 Diseño e implementación de TCP 43
5.6 Ejecución de Tomcat en IP4JVM 71
5.7 Plan de pruebas ejecutado 72
6.RESUMEN Y PERSPECTIVAS **75**
6.1 Planificación Inicial 75
6.2 Performance y Diseño 78

6.3 Portabilidad....79
6.4 Completitud / Correctitud....79

7.CONCLUSIONES DEL TRABAJO....81

8.GLOSARIO....83

9.REFERENCIAS....89

10. APÉNDICES....95

10.1 Preparación de entorno (svn, eclipse, etc)....95
10.1.1 Introducción....95
10.1.2 Instalación....96
10.2 Instalación de stack en SableVM y OpenJDK....103
10.2.1 Introducción....103
10.2.2 Objetivos....103
10.2.3 Instalación en SableVM....103
10.2.4 Instalación en OpenJDK....105
10.2.5 Conclusiones....108
10.2.6 Trabajo Futuro....109
10.3 Mejoras realizadas sobre código existente y planificación de ejecución....111
10.3.1 Introducción:....111
10.3.2 Criterios generales:....113
10.3.2.1 Colecciones tipadas....113
10.3.2.2 Uso consistente de colecciones de objetos....113
10.3.2.3 Extensión de colecciones....114
10.3.2.4 Iteraciones....114
10.3.2.5 Constructores....115
10.3.2.6 Conocimiento del Nivel....115
10.3.3 Diseño....117
10.3.3.1 ip4jvm.javafwrk.NetLevel.ItemNetLevelLayer / ip4jvm.javafwrk.NetLevel.ItemNetLevelApplication....117
10.3.3.2 Manejo de Jobs....118
10.3.3.3 Métodos de la Clase ip4jvm.javafwrk.NetLevel....118
10.3.3.4 ip4jvm.javafwrk.TimerServer....118
10.3.4 Clases....121
10.3.4.1 ip4jvm.javafwrk.NetStackState....121
10.3.4.2 ip4jvm.javafwrk.Application....121
10.3.4.3 ip4jvm.javafwrk.NetManagerRead....122
10.3.4.4 ip4jvm.javafwrk.NetParameters....122
10.3.4.5 ip4jvm.javafwrk.NetStack....123
10.3.4.6 ip4jvm.javafwrk.NetLevel....124
10.3.4.7 ip4jvm.javafwrk.NetConfigs....125
10.3.4.8 ip4jvm.net.applications.SocketManager....126

10.3.4.9 ip4jvm.net.DatagramSocket....127
10.3.4.10 ip4jvm.net.Network....127
10.3.4.11 ip4jvm.net.applications.Ping6....128
10.3.4.12 ip4jvm.net.applications.NeighborDiscovery....129
10.3.4.13 ip4jvm.net.protocols.headers.EthernetIIHeader....130
10.3.4.14 ip4jvm.net.protocols.ICMPv6....130
10.3.4.15 ip4jvm.net.protocols.headers. UDPProtocolHeader....130
10.3.4.16 ip4jvm.net.protocols.extras.FragmentIPv6....131
10.3.4.17 ip4jvm.net.protocols.extras.FragmentsIPv6....131
10.3.4.18 ip4jvm.net.applications.neighDisco. NeighborCache....132
10.3.4.19 ip4jvm.net.protocols.headers.icmps. ICMPv6RouterSolicitation....133
10.3.4.20 ip4jvm.net.protocols.headers.icmps.ICMPv6NDOptions....133
10.3.4.21 ip4jvm.net.protocols.ipv6.routingHeaders. RoutingHeader....134
10.3.4.22 ip4jvm.net.applications.neighDisco.RouterList....134
10.3.4.23 ip4jvm.net.applications.neighDisco. PrefixInfoList....135
10.3.4.24 ip4jvm.net.applications.neighDisco. DestinationCache....135
10.3.4.25 ip4jvm.net.applications.neighDisco. RoutingTableIPv6....135
10.3.4.26 ip4jvm.net.protocols.Ipv6Protocol....136
10.3.5 Conclusión:....139
10.3.5.1 Uso de Threads....139
10.3.5.2 Tabla de Cambios....140
10.3.5.3 Ejecución de Cambios....144
10.4 Implementación TCP....149
10.4.1 Introducción....149
10.4.2 Motivación....149
10.4.3 Objetivos....149
10.4.4 Conceptos Básicos....149
10.4.5 RFC 793....150
10.4.5.1 Formato de Datos....150
10.4.5.2 Estados....156
10.4.5.3 Interfaces estándares de TCP....158
10.4.5.4 Timers....166
10.4.6 Diseño e Implementación....167
10.4.6.1 Características Generales....167
10.4.6.2 Clases Implicadas....169
10.4.6.3 Atención de Eventos....187
10.4.6.4 Algoritmia adicional....194
10.4.7 Conclusión del trabajo realizado....200
10.4.8 Trabajo futuro....200
10.5 Configuración y ejecución de Apache Tomcat con IP4JVM....201
10.5.1 Introducción....201
10.5.2 Pasos a Seguir....201

1. Resumen

El presente informe contiene el detalle del desarrollo de nuestro proyecto de grado, el cual consistió en mejorar y completar un prototipo ya existente que agrega el manejo de stacks de protocolos a la máquina virtual Java.

La herramienta consiste en un framework enteramente programado en lenguaje Java que implementa un stack de protocolos y al cual se le puede agregar un protocolo cualquiera programado, éste también, enteramente en Java. La comunicación con el dispositivo físico de red está implementada con el uso de JNI.

Mediante la integración de este stack a una máquina virtual Java obtuvimos una plataforma capaz de interpretar y ejecutar código Java pero que a su vez prescinde de las funcionalidades de red brindadas por el sistema operativo y utiliza las implementadas por nuestro stack.

Además del framework comentado anteriormente, se encuentran implementados también algunos protocolos de distintas capas que permiten una comunicación entre dos nodos a nivel de capa de aplicaciones. El protocolo de enlace de datos implementado en la herramienta es Ethernet II, IPv6 para capa de Internet y para capa de transporte se encuentran implementados tanto TCP como UDP. Sin embargo, y si bien con estos protocolos ya se obtiene una capacidad de conectividad amplia a nivel de cualquiera de las capas mencionadas, cualquier usuario de la herramienta podría agregar un protocolo a elección al framework sin mayores dificultades, siempre y cuando obtenga una implementación de dicho protocolo programada en Java. Por ejemplo, un usuario podría querer usar su propia implementación de TCP que tenga ciertos aspectos diferentes a la implementación realizada por nosotros y podría hacerlo sin mayores dificultades. Otra posibilidad en este sentido sería la de usar el framework para desarrollar en Java y probar un nuevo protocolo de cualquier capa.

El proyecto consistió en varias fases como ser estudio previo, instalación del ambiente de desarrollo, análisis del prototipo existente, identificación de los cambios a realizar a dicho prototipo, ejecución de estos cambios, migrar el stack a una máquina virtual que considerásemos mejor en base a ciertos criterios, diseño y desarrollo del protocolo TCP, ejecución de una herramienta Java que utilice TCP sobre nuestro stack. Cada una de estas fases conjuntamente con el desarrollo y resultados obtenidos de las mismas se detallan más adelante en el documento.

1.1 Palabras Claves

Networking en Java. JNI. Framework. Modelos de Capas de Red. Stack de Protocolos. SableVM. OpenJDK.

1.2 Público Objetivo

Este documento está orientado a personas del perfil informático con cierto conocimiento de redes de computadoras y su organización jerárquica de capas; en particular es deseable que el lector tenga conocimiento del funcionamiento del stack IPv6 o IPv4. También es necesario para el entendimiento integral del presente informe haber adquirido previamente conceptos básicos acerca de la metodología Java y el funcionamiento de su máquina virtual.

Adicionalmente el lector podría desear, previamente, interiorizarse con mayor profundidad en cuanto al contenido de los distintos RFCs que detallan los protocolos de IPv6 y TCP y sus complementos. Esto último no es un requisito excluyente ya que en este documento se aportan los conceptos básicos e imprescindibles necesarios para el entendimiento del trabajo realizado, sin embargo un mayor conocimiento en el área pueden brindarle al lector una mayor capacidad de abstracción y objetividad al respecto.

1.3 Organización del trabajo

Este documento se organiza en nueve secciones. La primera sección contiene un breve resumen del contenido del informe, describe el público que puede verse interesado en leer el mismo y detalla la organización del trabajo.

La segunda sección consta de una breve introducción a los conceptos básicos manejados por el proyecto. Explica un poco más en detalle en que consistió el mismo y cuales fueron las motivaciones para su realización.

La tercera sección detalla como se encontraba la herramienta previamente al inicio de nuestro proyecto y presenta datos generales sobre ésta así como un breve análisis de la misma basado en distintos aspectos de importancia, como ser la performance, el diseño, la portabilidad y completitud de este prototipo.

En la cuarta sección se mencionan y detallan los objetivos trazados a inicios del proyecto entre los cuales se encuentran la mejora de la herramienta en cuanto a diseño, performance, portabilidad y completitud, el desarrollo del protocolo TCP, la integración de un aplicativo Java que utilizase servicios TCP con nuestro stack y la generación de documentación online de nuestro trabajo.

En la quinta sección se describe el desarrollo del proyecto en cada una de sus fases desde la planificación inicial, la selección e instalación del ambiente de desarrollo, la identificación y ejecución de cambios con sus respectivos resultados, la migración de máquina virtual, el diseño y desarrollo de TCP y la ejecución de un aplicativo sobre nuestro stack.

En la sección número seis se resume y analiza cada una de las fases ejecutadas en el proyecto y se esbozan conclusiones generales de las mismas.

En la séptima sección se presentan las conclusiones generales sobre el proyecto.

La sección número ocho corresponde al glosario, donde se detallan los términos utilizados en el informe que pueden ser específicos del dominio y por ende podrían resultar desconocidos para algún lector.

En la novena sección detallamos las referencias de las distintas publicaciones citadas por el informe.

La décima sección corresponde a los distintos apéndices, en los cuales se tratan distintos temas del proyecto con mayor profundidad.

2. Introducción

El proyecto que realizamos y al cuál llamamos IP4JVM, se trata de la continuación de un proyecto ya existente, realizado por Laura Rodríguez a modo de pasantía entre marzo y noviembre del año 2006 [1].

El objetivo del mismo es la implementación en Java [2] de un stack de protocolos, de forma de proveer a la máquina virtual de Java la capacidad de manejar esta funcionalidad no incluida en la misma. Esto se haría mediante la implementación de un framework, con ayuda del cuál se iría desarrollando un stack más especifico (en este caso IPv6) e ir incorporando incrementalmente los diferentes protocolos necesarios por este último.

El protocolo en el que se basa el proyecto es IPv6 (especificado por el RFC 2640 [3]), el cuál se creó con la idea de sustituir IPv4 [4] (actual versión usada del protocolo de Internet) para solucionar varios problemas que éste presenta. Igualmente el objetivo no es sólo implementar este protocolo, sino que es tener un stack completo que abarque los protocolos necesarios para la comunicación desde capa de red hasta capa de transporte, todo esto, íntegramente programado en Java.

El tutor del proyecto es Ariel Sabiguero, quien también tuvo el rol de tutor en el proyecto anterior.

El contexto de este trabajo es el proyecto de grado, de la carrera Ingeniería de Computación y como tal cuenta con la intervención del Instituto de Computación de la Facultad de Ingeniería [5] al cuál pertenece nuestro tutor.

2.1 Motivación

Las motivaciones que impulsaron la realización de este proyecto incluyen innovaciones que el mismo introduciría, como ser el poder manejar un stack de protocolos enteramente programado en Java. También en esta sección se explicarán las razones que impulsaron la decisión de implementar el protocolo IPv6 como protocolo de Internet, cuando se pudo haber elegido otro, como ser el protocolo actualmente usado, o sea IPv4.

A continuación ahondaremos en las temáticas abordadas por el proyecto y que pensamos que representan los puntos innovadores del mismo.

2.1.1 Networking en Java

Como es sabido, el lenguaje Java no brinda servicios de networking de bajo nivel de forma nativa. Para resolver esto, la máquina virtual de Java [8] utiliza los servicios que provee el sistema operativo huésped. De esta manera, cuando se programa en Java, en general se está programando en la capa de aplicaciones y se podrán usar las clases provistas (del paquete *java.net*) para comunicarse con la capa de transporte. Es posible abrir sockets, tanto TCP [9] como UDP [10], y enviar y recibir datos a través de estos. La capa de transporte es totalmente transparente para el usuario, ya que la máquina virtual de Java se encarga de hacer los llamados al sistema operativo necesarios para establecer la comunicación.

El problema surge cuando se desea, a través de código Java, poder acceder directamente a los paquetes que llegan desde la capa física o poder tomar decisiones sobre cómo fragmentar estos paquetes y cómo y cuando enviarlos, o sea, cuando se desea programar en la capa de red. También, a más alto nivel, podría desearse por parte del programador, modificar aspectos de la capa de transporte y que fuese su código el que interactúe con la capa inferior y provea servicios a la capa de aplicaciones. Es aquí cuando se halla en el lenguaje Java una limitación en ese aspecto, ya que como hemos marcado, Java usa los servicios del sistema operativo y no provee herramientas para poder modificar esta situación.

Esto último, a su vez, también hace que el código que implementamos, dependa en su comportamiento del manejo de comunicaciones que hace el sistema operativo, y si bien en la actualidad se puede asegurar que todos los sistemas operativos usados comúnmente tienen un comportamiento similar y estandarizado que hacen que no existan diferencias en este sentido, el prescindir del sistema operativo y proveer a la máquina virtual de Java de esta capacidad, agrega un nivel más de portabilidad a los programas escritos en lenguaje Java.

Otra posibilidad que agrega nuestro proyecto, es el de desarrollar un protocolo nuevo de cualquier capa enteramente en lenguaje Java y poder agregarlo al stack desarrollado sin mayores problemas.

2.1.2 Stack de Protocolos

Para reducir la complejidad del diseño de redes, éstas se encuentran organizadas en capas. La cantidad de capas, las funcionalidades de las mismas y los nombres difieren según la red. El propósito de las capas es ofrecer servicios a las capas superiores de modo de simplificar la implementación ocultando los detalles inherentes de la lógica necesaria para implementar los servicios.

Cada capa de una máquina conectada lleva a cabo una conversación con su par de la otra máquina implicada en la comunicación. Se conoce con el nombre de protocolo de la capa dada, al conjunto de reglas y convenciones que gobiernan estas comunicaciones. Las capas no se comunican directamente unas con otras, sino que transfieren los datos a sus capas inmediatas inferiores hasta llegar así al medio físico que realmente transmitirá los datos. Al recibir en el otro extremo el medio físico los datos, éste los entregará a la capa más baja y se irá subiendo a las capas superiores hasta llegar a la capa destinataria.

Entre cada par de capas se define una interfaz, la que especifica las operaciones y servicios que ofrece la capa inferior a la superior. Además cada capa define una estructura de datos a ser adjuntada a los datos que se quieren enviar a través de la misma para ser utilizada como información de control entre las capas.

Se denomina pila de Protocolos (o stack de protocolos) a la lista de protocolos empleados por cierto sistema con un protocolo en cada capa.

Existen en la actualidad dos modelos de referencia principales en el ámbito de Internet: el modelo de referencia OSI [11] y el modelo de referencia TCP/IP [12].

El modelo de referencia OSI fue desarrollado por la ISO (Organización Internacional de Normas) [13]. El modelo consta de 7 capas las cuales se muestran en la siguiente figura:

Ilustración 1: Capas del modelo OSI

Este modelo, a pesar de estar bien concebido (modular, con roles específicos para cada capa, etc.) y ser completo, nunca fue implementado por ninguna red comercial, y los intentos por establecer este modelo como norma de comunicación de red fallaron. Debido, tal vez, al éxito del modelo TCP/IP.

El modelo TCP/IP patrocinado en sus inicios por el Departamento de Defensa de los Estados Unidos [14] es un derivado de la red ARPANET [15]. A diferencia de OSI este modelo surgió a partir de una implementación (lo cuál no parece muy adecuado) pero rápidamente se popularizó. La red fue adquiriendo nuevas capas y protocolos con el correr del tiempo, evolucionando hasta lo que conocemos hoy como la red de Internet. El modelo se basa en la distribución en 5 capas:

Ilustración 2: Capas del modelo TCP/IP

2.1.3 Diferencias entre IPv6 e IPv4

Como ha sido denunciado en sucesivos trabajos de investigadores informáticos, entre los cuáles podemos encontrar el trabajo titulado *"A Pragmatic Report on IPv4 Address Space Consumption"* escrito por Tony Hain [16] e *"IPv4 Address Consumption"* de Iljitsch van Beijnum [17] (ambos publicados en el *"Internet Protocol Journal"* [18] de la empresa *Cisco Systems* [19]), el principal problema de IPv4 es el poco espacio de direccionamiento ofrecido por las direcciones de 32 bits (2^{32} = 4.294.967.296). Aunque éste parezca un número abultado, claramente no lo es tanto si consideramos que en el mundo moderno existen múltiples dispositivos que necesitan conexión a Internet y que en un futuro cercano habrán aún más. Hoy por hoy cada persona puede contar con su P.C., un teléfono móvil con acceso a Internet, un automóvil que use servicios en línea para auto navegación, una heladera que, a partir de las estadísticas de consumo de productos, pueda advertir cuando se debe hacer un pedido y automáticamente conectarse mediante servicios web a un supermercado virtual y efectivizarlo, etc. Y si todas estas cosas son posibles en la actualidad, imaginemos cuantas cosas más precisarán acceso a Internet en un futuro no tan lejano. Es por esto que no es absurdo pensar que se podrán llegar a precisar 50 o 100 direcciones IP's para una persona que utilice estas tecnologías.

Viendo esto, fue que a comienzo de los 90 la *Internet Engineering Task Force* (IETF) [20] comenzó sus esfuerzos para desarrollar el sucesor de IPv4. Dicho sucesor, antes de adoptar el nombre IPv6, fue llamado IPng, lo cuál se traduce a *Internet Protocol of the Next Generation*, o en español, Protocolo de Internet de la Próxima Generación [21]. Varios esfuerzos paralelos comenzaron en simultáneo, todos intentando resolver el eminente problema de limitación de espacio de direccionamiento de IPv4, así como también tratando de brindar funcionalidades adicionales. La IETF fundó el área IPng en 1993 con la finalidad de investigar diferentes propuestas y hacer recomendaciones para los futuros trabajos que se realizasen al respecto.

Los directores de la IETF recomendaron la creación de IPv6 en una convención realizada en Toronto en el año 1994. Esta recomendación está especificada en el *RFC 1752, "Recomendación para el Protocolo IP de la Próxima Generación"* [22]. Se formó un grupo de trabajo llamado ALE (*Address Lifetime Expectation*) [23] cuyo trabajo era determinar si el tiempo estimado de viabilidad del protocolo IPv4 permitiría el desarrollo de un nuevo protocolo con nuevas funcionalidades, o si en cambio, se contaba con el tiempo justo para, simplemente, solucionar el problema de direccionamiento. En 1994 el grupo ALE proyectó que las direcciones IPv4 dejarían de dar abasto en algún momento entre 2005 y 2011, basándose en datos estadísticos disponible en aquella época.

El resultado de todos estos esfuerzos mancomunados por solucionar los aspectos mencionados es IPv6, el cuál soluciona el problema de limitación de direccionamiento mediante el uso de direcciones de 128 bits (2^128 = 340.282.366.920.938.463.463.374.607.431.768.211.456) un número sustancialmente mayor que las 4.294.967.296 combinaciones posibles con 32 bits. Tal vez esta diferencia quede más gráficamente representada al considerar que si las posibles conexiones de IPv4 ocuparan 1 milímetro, las posibles conexiones de IPv6 ocuparían aproximadamente 240.000 veces la distancia entre el Sol y la Tierra.

Sin embargo, a pesar de ser el principal problema del protocolo, y el que más urge solucionar, el espacio de direccionamiento no es el único problema que tiene la versión 4 de IP. Al ser un protocolo diseñado en los 70's y principios de los 80's, resulta ser bastante anacrónico en comparación a las necesidades que presenta la Internet hoy en día. Por ejemplo, la transmisión de multimedia en tiempo real (video conferencias), o la voz sobre IP (telefonía VoIP) son cosas que dejan en evidencia el cuello de botella que presenta el protocolo, ya que los paquetes deben "competir en igualdad de condiciones" con el resto para ser procesados por cada nodo de la transmisión, sin importar el contenido de los datos de dicho paquete. Esto provoca que no haya distinción entre los paquetes que deben ser transmitidos en tiempo real y los que no, por lo cuál en redes lentas o ante problemas de congestión, fácilmente se incurre en la discontinuidad de imagen o voz según el caso.

Hacemos hincapié en estos dos problemas (de los tantos que tiene IPv4) por tratarse de dos dignos exponentes de las dos grandes clases de problemas que se presentan; por un lado están los problemas físicos, atados a tamaños de paquetes y direcciones, y por otro lado los problemas más abstractos y de uso lógico de los bits de encabezados, que son introducidos a partir de las nuevas utilidades que desearía darse al protocolo, siendo que no habían sido previstas al momento de su diseño inicial.

A continuación se listará una serie de diferencias que tiene IPv6 frente a IPv4, las cuáles buscan dar solución a todos estos problemas y que motivaron la creación de IPv6:

- *Capacidad expandida de direccionamiento y mecanismo de auto configuración:*

  Las direcciones en IPv6 han sido aumentadas a 128 bits. Esto resuelve el problema del limitado espacio de direccionamiento de IPv4 y ofrece una jerarquía de direcciones más profunda y una configuración más simple. El mecanismo de auto configuración embebido en el protocolo simplifica la configuración de una red. El ruteo de multicast ha sido mejorado, habiendo sido la dirección de multicast expandida mediante el campo *scope.* Otro agregado del protocolo es la creación de un nuevo tipo de dirección, llamado dirección Anycast, mediante la cuál se puede mandar un mensaje al miembro más cercano de un grupo (una aplicación de esta dirección se puede ver en el algoritmo de DHAAD utilizado en MIPv6 [60]).

- *Simplificación del formato del cabezal:*

  El cabezal de IPv6 tiene un largo fijo de 40 bytes, lo cuál se traduce en un cabezal de 8 bytes más dos direcciones de 16 bytes cada uno (remitente y destino). Algunos campos de IPv4 se han quitado o vuelto opcionales. De esta manera los paquetes pueden ser tratados con mayor rapidez debido al menor costo de procesamiento del cabezal.

- *Soporte mejorado para las opciones y extensiones:*

  En IPv4, las opciones estaba integradas en el cabezal básico; sin embargo, en IPv6, estas opciones son manejadas como Cabezales de Extensión. Estos cabezales de extensión son opcionales, y en caso de ser necesarios, se insertan entre el cabezal de IPv6 y los datos. De esta manera el paquete IPv6 resulta más simple de armar. El reenvío de paquetes resulta más eficiente y nuevas opciones que puedan surgir en el futuro pueden ser integradas de manera trivial.

- *Extensiones para autenticidad y privacidad:*

  Se ha especificado soporte para verificar autenticidad y extensiones para garantizar la integridad y confidencialidad de datos.

- *Capacidad de etiquetado de flujos:*

  Los paquetes que pertenecen al mismo flujo de transferencia, que puedan precisar un tratamiento especial o cierta calidad de servicio, pueden ser etiquetados por el emisor. Los servicios de tiempo real son un ejemplo de dónde esta característica puede ser usada.

2.1.4 Implementación en Java

La motivación de realizarlo en Java tiene mucho que ver con lo detallado anteriormente en la sección 2.1.1, la cuál explica la falta de servicios de networking de bajo nivel que padece este lenguaje. Por lo tanto, a diferencia de otros lenguajes, el llevar a cabo este proyecto en Java deriva también en la adición de una nueva funcionalidad a la máquina virtual Java, la cuál en su versión pura, delega este trabajo al sistema operativo huésped.

Otra de las razones es el hecho de que Java es muy práctico, versátil y tiene en la portabilidad un gran fuerte frente a otros lenguajes. De esta manera, gracias a su metodología de uso de una máquina virtual para ejecutar el código, un producto implementado enteramente en Java supone una migración trivial a cualquier sistema operativo que pueda correr la máquina virtual de Java. La herramienta desarrollada por este producto no escapa a esta bondad de Java, con la salvedad que a parte de usarse la máquina virtual compilada con nuestras modificaciones, deberá también instalarse la librería Pcap [24], utilizada para la interacción mediante JNI [25] de nuestro código con el dispositivo de red.

3. Estado del Arte

En esta sección se brindarán datos acerca del proyecto que precedió al nuestro y el estado en el que lo encontramos. Esta información no es irrelevante, ya que el trabajo realizado anteriormente es el punto de partida de nuestro proyecto, y por lo tanto, el estado de la herramienta y su evolución marcarían las pautas de nuestro trabajo y objetivos.

3.1 Datos Generales de Proyecto anterior

Como se mencionó brevemente en la introducción, nuestro proyecto se dio como continuación de otro proyecto realizado con anterioridad. Dicho trabajo fue desarrollado por Laura Rodríguez, enmarcado en el contexto de una pasantía realizada en Francia en el instituto IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires) [6] que a su vez forma parte del la INRIA (l'Institut National de Recherche en Informatique et en Automatique) [7], instituto francés que se dedica a la investigación dentro de las ciencias de la computación. Laura efectuó su pasantía entre marzo y noviembre del año 2006, logrando en este tiempo los objetivos que se plantearon en sus inicios. Estos objetivos consistían, entre otros, en tener un framework programado en Java mediante el cuál se puede implementar un stack de protocolos y agregar distintas implementaciones de protocolos que se deseasen, éstos también programados en Java.

Además de este framework se implementó parte de los protocolos Ethernet II [26], IPv6 y de ICMPv6 [27], con lo cuál se podía comprobar el correcto funcionamiento mediante el uso del comando ping, el cuál está especificado en el RFC 4443 referente al protocolo ICMPv6. También llegó a implementarse algo de UDP [10] y fue probado a través de una aplicación sencilla que consistía en el envío de mensajes desde un nodo a otro.

Es necesario, además, resaltar que los tutores de este proyecto fueron Ariel Sabiguero (quién a su vez es nuestro tutor en esta segunda iteración de IP4JVM), perteneciente al Instituto de Computación de la Facultad de Ingeniería (InCo) y Cesar Viho, quien pertenece al INRIA.

3.2 Performance

La performance era uno de los puntos bajos del prototipo inicial, esta situación era conocida desde el inicio de nuestro proyecto ya que el objetivo del proyecto anterior era obtener un stack funcional sin hacer demasiado hincapié en requerimientos no funcionales de este estilo. Por ello, en esta segunda iteración de IP4JVM, debíamos atacar este problema como objetivo primordial.

Al comienzo del proyecto, no teníamos las armas para mesurar la magnitud de este problema, pero sabíamos por la propia documentación generada en el proyecto anterior, que la performance no era la deseada y que sería uno de los puntos a atacar. Los principales aspectos a cuidar en este sentido eran el consumo de ciclos de CPU y el tiempo de respuesta, por ser ambos demasiado elevados.

3.3 Diseño

El diseño existente de la herramienta era básicamente correcto, ya que cumplía los requerimientos que se habían planteado en el proyecto anterior. Sin embargo habían detalles en los cuáles deberíamos trabajar.

La principal modificación que desde un comienzo sabíamos que deberíamos hacer era agregar al diseño un *TimerServer* que se encargue de tratar todo lo referente a tiempos, por ejemplo el vencimiento de los fragmentos de IP, entradas del Neighbor Caché, o más tarde, si se implementaba TCP, los tiempos de retransmisión, etc. Otra modificación que nos parecía necesaria en los inicios del proyecto era la de tratar los Jobs (clase que representa a los fragmentos en IP, a los segmentos en TCP, etc. Léase cualquier porción de bits sobre la cuál el stack deba trabajar, ya sea que deba mandarse a un nivel inferior o por el contrario subir a una capa superior) mediante el uso de varios threads, de manera de darle un paralelismo que redunde en una mejor performance. En la herramienta existente hasta ese entonces, cuando un Job llegaba al Stack, éste lo tomaba y se avocaba a su procesamiento hasta su finalización, de esta manera, si llegaba otro Job antes de que se finalizara con el anterior, éste quedaba en espera, hasta que el stack se liberaba y lo atendía. Con el nuevo cambio que nos planteábamos, esto cambiaría, ya que al haber varios hilos, se podrían atender varios Jobs en simultaneo y ser más justos en la atención a todos.

Otros detalles menores como nombres de clases y funciones también fueron enumerados entre las correcciones que se debían hacer a la herramienta.

3.4 Portabilidad

En los inicios del proyecto la máquina virtual con la cuál se encontraba integrada la herramienta, era SableVM en su versión 1.13 [28] compilada en Microsoft Windows [29] y Linux [68].

El hecho de que no se recibiese un ambiente de desarrollo en funcionamiento, hizo que se debiera invertir tiempo en la recompilación de la máquina virtual, así como la correcta configuración del ambiente en su conjunto, lo cuál no fue una tarea trivial dada nuestra inexperiencia hasta ese momento y la complejidad que conlleva la compilación de una máquina virtual modificada, sobre todo cuando la documentación existente y el soporte online de la misma es casi nulo, por tratarse de una máquina virtual no muy usada.

Como veremos más adelante, la idea inicial del proyecto incluía cambiar la máquina virtual

que soportaba a nuestra herramienta, de modo de tener como base un producto más conocido, que brindase así un mejor soporte, a partir del conocimiento compartido en Internet de una mayor masa de usuarios. Cabe resaltar que las máquinas virtuales a las cuáles se deseaba migrar nuestra herramienta, y en las cuáles ahondaremos más adelante, no se encontraban disponibles al momento de realización del proyecto anterior, lo cuál minimizó la posibilidad de elección en aquel momento.

3.5 Completitud

La implementación existente, si bien era funcional, distaba mucho de cubrir un stack de protocolos que fuese útil a un programador Java. Primeramente porque se implementaron los aspectos de los protocolos IPv6, ICMPv6 y UDP imprescindibles para tener una comunicación de datos entre dos nodos. Pero esto hizo que los RFCs de dichos protocolos no se cubrieran en su totalidad, por el contrario, el cubrimiento de los mismos era limitado y suponía un gran trabajo para completarlos.

A parte de completar los protocolos ya implementados, otro de los objetivos inicialmente propuestos, era el de agregarle una implementación del protocolo TCP, a modo de completar definitivamente un stack útil y funcional completamente implementado en Java.

4. Objetivos

Las primeras reuniones con nuestro tutor, tuvieron como propósito discutir y convenir los objetivos que nos plantearíamos para el decorrer del proyecto. Estos objetivos intentaron dar completitud a la idea original que el proyecto tenía en su concepción inicial y que había sido recortada por razones lógicas de tiempos y esfuerzos en el proyecto anterior, en el cuál únicamente trabajó una persona. Aún así intentamos ser pragmáticos con el fin de que los objetivos planteados para nuestro proyecto fueran perfectamente realizables en el tiempo dispuesto para dicha actividad académica.

Los principales objetivos que nos planteamos fueron mejorar la performance, mejorar y evolucionar el diseño de la herramienta, aumentar la portabilidad y solucionar algunos problemas existentes al respecto, implementar partes faltantes en protocolos del stack, incluyendo una implementación propia del protocolo TCP e integrar una aplicación que utilizase dicho protocolo, como ser, por ejemplo, un servidor web. Por último se planteó el objetivo de realizar una página web documentando la herramienta.

4.1 Performance

Los objetivos planteados al comienzo del proyecto fueron varios. Para comenzar, se deseaba mejorar los aspectos negativos de la herramienta, entre los cuáles estaba la performance. No se establecieron medidas que cuantificaran los logros deseados al respecto, pero en general se quería dar solución al exceso de procesador que consumía la herramienta al ejecutarse y para esto serían necesario modificar aspectos del diseño, revisar y mejorar algoritmos, así como eliminar el uso de *busy waits,* que de hecho existían en ese entonces.

El objetivo en sí suponía un gran esfuerzo ya que sería necesario, para su cumplimiento, un exhaustivo análisis del código, de manera de tener un entendimiento integral del funcionamiento de la herramienta, identificar cuellos de botella en el código que pudieran afectar negativamente a la performance del producto final y que a su vez fueran corregibles o perfectibles, para luego si proceder a realizar las modificaciones previstas.

4.2 Diseño

Otro de los objetivos que nos planteamos, y que debíamos atacar tempranamente en el proyecto, era el de mejorar aspectos de diseños que no conformaban. Por ejemplo, existían clases que tenían demasiado procesamiento que, a nuestro entender, debería trasladarse a clases más pequeñas y especializadas. Otro detalle que comentamos fue el de crear un TimerServer que manejara temporizadores para todo lo referente a tiempos de espera y afines. Esto parecía una mejora importante, no solo de diseño, sino que ayudaría a completar el primer objetivo, ya que manejar, por ejemplo, los vencimientos de los fragmentos mediante

temporizadores, ayudaría en pos de un uso más responsable del procesador. Cabe destacar que este tipo de vencimientos se hacía mediante un ciclo que en cada iteración verificaba si dicho tiempo ya se había cumplido, lo cuál, notoriamente, implica un peor desempeño.

Un último detalle marcado al respecto del diseño, fue el de ciertas clases cuyos nombres resultaban poco mnemotécnicos o que podían dar lugar a confusiones. Un claro ejemplo era el de la clase Application, ya que en el contexto de redes, este término puede hacer pensar de que se trata de una aplicación, que usa los servicios de la capa de transporte, sin embargo no era este concepto el que la clase modelaba. Este caso y otros, que incluyen nombres de atributos y funciones, deseaban ser corregidos para facilitar el entendimiento y la lectura del código para programadores que quisieran mantener y/o mejorar la herramienta en un futuro.

4.3 Portabilidad

Otro de los objetivos definidos era el de darle a la herramienta una mayor portabilidad mediante el cambio de máquina virtual sobre la cuál corría.

SableVM es una máquina virtual correcta en su funcionamiento y que presenta varias ventajas, sin embargo únicamente implementa la máquina virtual Java 1.4, versión que prácticamente ha caído en desuso por los desarrolladores Java debido a las nuevas bondades que ofrecen las nuevas versiones a partir de la 5.0, entre las cuales podemos nombrar, por ejemplo, el manejo de "colecciones tipadas".

Otra desventaja de SableVM es que se trata de un producto poco usado, poco conocido y por ende con muy poco apoyo online que pueda dar soporte a problemas que surgieran. En este sentido, se planteó como objetivo la migración de la herramienta hacia otra máquina virtual que tuviese mayor apoyo debido a un uso más globalizado de la misma. Las opciones propuestas, en ese entonces, fueron Harmony [30], desarrollado por Apache [31], ni más ni menos que el creador de Apache Tomcat [32], y OpenJDK [33] que es la máquina virtual de Sun Microsystems [34], siendo esta última la más popular.

Al momento de realizarse el proyecto anterior al nuestro, ambas máquinas enumeradas en el párrafo anterior no se encontraban disponibles, lo cual explica la elección de SableVM.

4.4 Completitud

Un tercer objetivo planteado, fue el de aumentar la cobertura de los RFCs que se encontraban parcialmente implementados. Esto se debía a que el código existente, si bien era funcional, no llegaba a cubrir en su totalidad los RFCs de IPv6, así como tampoco era completa la implementación de ICMPv6 ni de UDP. El objetivo entonces podía traducirse en finalizar la implementación de dichos protocolos, de manera que nuestra herramienta cubriese los RFCs en su totalidad. Especial hincapié se hizo sobre la parte de ruteo de IPv6, sobre la cual no había implementado nada y resultaba importante tener esto integrado a la herramienta

resultante, así como también se deseaba contar con una implementación completa de Stateless Address Autoconfiguration, especificado por el RFC 2462 [53].

4.5 Protocolo TCP

Una vez completa la implementación de IPv6 y los restantes protocolos ya existentes en el stack de la herramienta, el siguiente objetivo planteado era el de agregarle al mismo el protocolo TCP. Como es sabido, este protocolo es el más usado por las aplicaciones de red a nivel de capa de transporte. Si bien hay varios aplicativos que corren sobre UDP, la mayoría sacan provecho del compromiso de orden de datos que propone TCP. Por esto, una vez que se le agregase este protocolo a nuestro stack, podríamos decir que se cuenta con un stack relativamente completo y que ofrece muchas de las funcionalidades que pudiese desear un usuario de nuestra herramienta. Para llevar a cabo este objetivo, debería entonces hacerse un diseño, orientado al funcionamiento de framework de la propia herramienta y también del mismo lenguaje Java. Una vez diseñado debería implementarse y probarse adecuadamente.

4.6 Integración de una aplicación con nuestro stack

El siguiente objetivo sería el de integrar un aplicativo de modo tal que se ejecute usando nuestro stack. La idea consistía en hacer que una aplicación, por ejemplo un servidor Apache Tomcat, demande los servicios TCP implementados por nosotros, y de esta forma utilice en forma integra el stack implementado en el proyecto pasando por toda la jerarquía de capas.

4.7 Documentación Online

Un último objetivo planteado era la necesidad de crear una página web, de ser posible en los idiomas español e inglés, donde se documentase la herramienta, con información de la instalación y del producto en sí, así como también una ordenada organización del SVN del proyecto [35].

5. Desarrollo del Proyecto

En esta sección detallaremos la evolución del proyecto, intentando detallar las fases en las que se dividió el mismo, como se afrontaron dichas fases, los cambios surgidos sobre la marcha, los tiempos insumidos y los resultados obtenidos.

5.1 Planificación Inicial

Al comienzo del proyecto se identificaron las distintas fases que sería necesario llevar a cabo para poder alcanzar los objetivos planteados. Dichas fases constaban de:

- **Estudio previo**

  Al inicio del proyecto se debió leer la documentación existente del proyecto anterior, así como también debimos interiorizarnos en los temas afines al proyecto en sí, como ser IPv6, JNI, Networking en Java, etc.

  Basando nuestros cálculos en una dedicación semanal de 30 a 40 horas hombre (dos desarrolladores con una dedicación de entre 15 y 20 horas), estimábamos que esta tarea conllevaría unas dos semanas de esfuerzo.

- **Instalación del ambiente**

  Una vez instruidos acerca de lo que trataba el proyecto, debíamos comenzar la instalación del entorno de desarrollo, así como del producto en sí.

  El tiempo que se estimó que sería necesario para la instalación era de 2 semanas, ya que se sabía que la documentación existente del proyecto anterior era débil en este punto y que los productos a instalarse, como ser SableVM, no tenían un soporte muy fuerte que pudiese servir de apoyo para hacer funcionar el ambiente en su completitud.

- **Análisis del producto**

  Esta fase implicaría el análisis del código existente, conjuntamente con lecturas de soporte orientadas a la temática, que permitieran una comprensión del funcionamiento integral de la herramienta.

  Planificamos una semana de trabajo para esta tarea.

- **Identificación de cambios**

  La próxima fase implicaría analizar el código más en detalle, intentando identificar posibles mejoras, fuesen estas de diseño o de algoritmia, para intentar darle al

producto mejores prestaciones y tener una base más sólida para futuras fases donde debiesen desarrollarse funcionalidades que tengan como cimiento lo existente.

En esta misma fase, una vez analizados los posibles cambios, debería realizarse un informe con los mismos, indicando para cada uno su impacto en el producto, el esfuerzo que su realización conllevaría y el grado de dependencias que esta porción del código tiene con el resto del producto. En base a estos datos los cambios serían priorizados y reordenados para su posterior ejecución.

Como el cumplimiento de la fase anterior (análisis del producto) implicaba el conocimiento del funcionamiento y del código, previmos que esta fase sería amortizada, en cuanto a tiempos refiere, por esta situación. Es por esto que la planificación inicial tenía una asignación de dos semanas para esta tarea, la cuál parecía compleja pero realizable.

- **Realización de cambios**

  Una vez recavados los cambios a realizar y ordenados éstos según prioridades, en esta fase, dichas modificaciones deberían ser efectuadas, siempre asegurándonos que las funcionalidades de la herramienta no se viesen recortadas ni afectadas negativamente.

  Paralelamente a los cambios, deberíamos ir completando las partes faltantes de los RFCs de IPv6 e ICMPv6, ya que éstos no estaban completamente implementados, y así completar el stack en cuanto a capa de Internet se refiere.

  También en esta fase previmos la posibilidad de realizar algún tipo de prueba comparativa entre la versión inicial de la herramienta, y la obtenida mediante la realización de las modificaciones sugeridas, de modo de poder tener datos tangibles acerca de la evolución del producto gracias a los cambios. A priori no se sabía cuantos cambios deberían realizarse ni cuanto faltaba por implementar para completar los RFCs mencionados. Esto último, a su vez, conllevaría un estudio en profundidad de estos documentos de especificación de protocolos para discernir si un punto dado del documento había sido implementado o no. Por esto, el cuantificar este esfuerzo en términos de tiempo, era complejo, sin embargo, dado a que comenzamos el proyecto a fines de abril, y deseábamos tener los cambios enteramente realizados para fines de setiembre, de manera de llegar a cumplir todos los objetivos en diciembre, se planificaron 14 semanas para esta fase.

- **Migración de MV**

  La máquina virtual que usaba la herramienta al comienzo del proyecto, SableVM, tiene muchas desventajas. Por ejemplo, no es un producto muy conocido, y por lo tanto es poco usado, lo cuál implica poco soporte online sobre posibles problemas que puedan surgir en su uso. Por esto vimos como importante el invertir tiempo en intentar migrar el producto a otra máquina virtual que fuese más usada y que tenga mayor soporte y *background,* como ser Harmony (de Apache) u OpenJDK (con el

soporte de Sun Microsystems). Previmos que esta etapa también tendría mucho de lectura de documentaciones y estudio de las máquinas virtuales seleccionadas para, mediante el análisis de código, visualizar como integrarla con nuestra herramienta. Por esto último fue que los tiempos estipulados para esta tarea fueron mayores de lo que uno podría suponer de antemano, con el fin de poder mitigar riesgos que pudiesen aparecer. El tiempo asignado, fue entonces, de 4 semanas.

- **Diseño y desarrollo de TCP**

  La penúltima fase del proyecto correspondería a diseñar e implementar TCP de manera tal que pueda ser usado en el stack implementado. De esta manera, podríamos decir que la herramienta ofrece una cobertura relativamente amplia de la capa de transporte, ya que la mayoría de las aplicaciones de red corren sobre este protocolo. Con esto, el camino desde la aplicación que se ejecute sobre nuestra herramienta hasta el dispositivo de red, estaría enteramente cubierto por el producto, garantizando que todas las capas están programadas en Java. La única salvedad a esta apreciación, es el uso de JNI para la obtención y envío de paquetes desde y hacia el dispositivo de red (hardware).

  En general, y sobretodo en lo que nuestra experiencia indicaba, el diseñar e implementar algo de cero, puede tener varias ventajas frente a tener que solucionar problemas existentes en un código ya escrito sobre un diseño que no es propio. Este factor estaría presente en esta fase y sobre todo motivando positivamente al grupo de desarrollo, por lo cuál supusimos que 8 semanas serían suficientes para lograr el propósito.

- **Integración de una aplicación con nuestro stack y ajustes finales de TCP.**

  Otro punto a tener en cuenta era lograr ejecutar un aplicativo haciendo que éste utilizase TCP sobre nuestro stack. Con esto no sólo daríamos un marco práctico a la herramienta, sino que podríamos hacer pruebas más orientadas al correcto funcionamiento de la misma y detectar defectos que hubiesen sido pasados por alto en la implementación de TCP. Para esto se planteo como meta que el stack pudiera brindar los servicios requeridos por Apache Tomcat para que éste funcionara adecuadamente. Con esto se lograba tener la herramienta brindando los servicios TCP a una aplicación de alta popularidad y que hace uso exhaustivo de TCP. Un requerimiento adicional fue brindar la posibilidad de utilizar el stack brindado por la herramienta o el del sistema operativo (el por defecto) mediante la definición y uso de una variable de la máquina virtual.

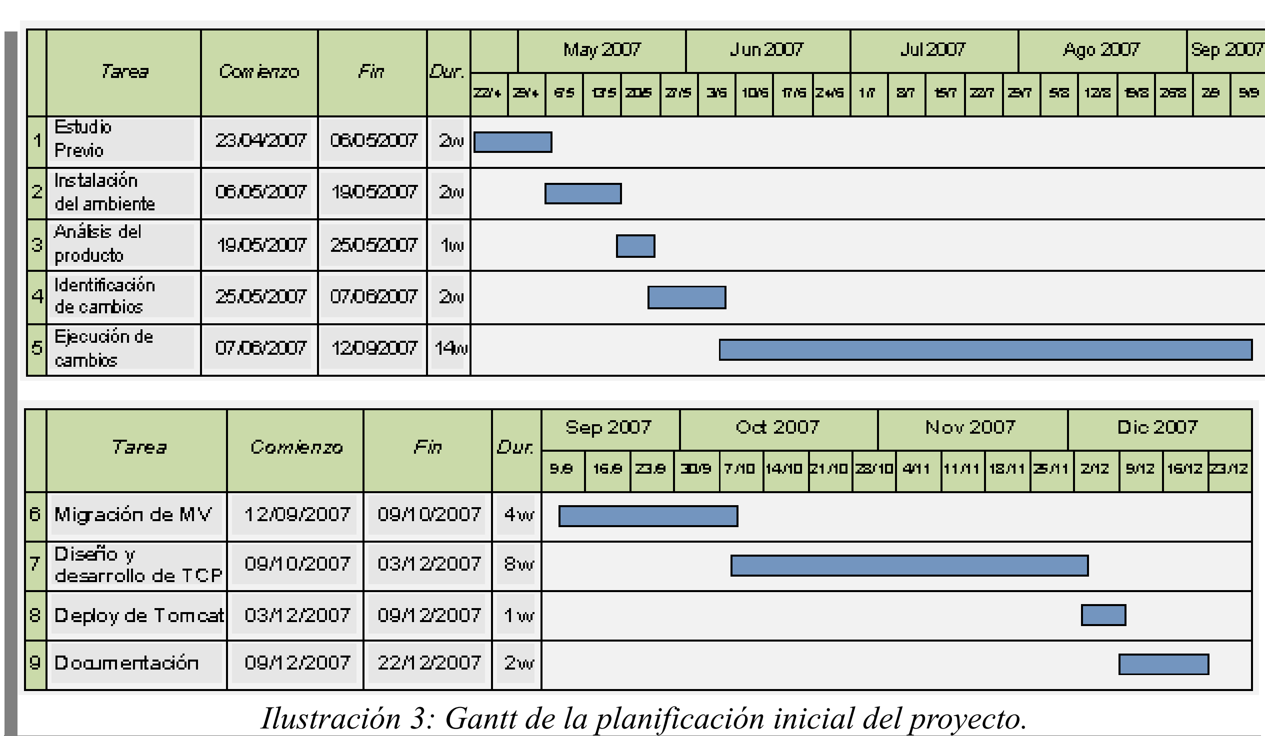
  En un inicio se estimo que el tiempo que debiera ser insumido por esta tarea no debería ser demasiado en caso de que la implementación de TCP fuera correcta. Teniendo en cuenta de que dicha implementación seguramente no cumpliera con el comportamiento esperado ya sea por carencias o por errores de programación se estimó una duración de una semana.

- **Documentación**

  Por último era necesario afinar las documentaciones realizadas y crear el informe

que aquí se presenta, integrando las documentaciones de cada una de las partes del proyecto y completando las mismas. Además un punto adicional que se planteó al inicio del proyecto fue la realización de una página web para el acceso público al proyecto. Para estas tareas se habían estimado 2 semanas.

A continuación se muestra el diagrama de Gantt que representa el desarrollo estimado del proyecto a modo de ilustrar los tiempos que pensamos en un inicio invertir en cada tarea. Veremos en dicho diagrama que el comienzo de una tarea se pensaba dar al finalizar la inmediatamente anterior, esto se debía a la interdependencia que supusimos entre las actividades, las cuáles brindaban su resultado final como entrada a la tarea siguiente.



	Tarea	Comienzo	Fin	Dur.
1	Estudio Previo	23/04/2007	06/05/2007	2w
2	Instalación del ambiente	06/05/2007	19/05/2007	2w
3	Análisis del producto	19/05/2007	25/05/2007	1w
4	Identificación de cambios	25/05/2007	07/06/2007	2w
5	Ejecución de cambios	07/06/2007	12/09/2007	14w

	Tarea	Comienzo	Fin	Dur.
6	Migración de MV	12/09/2007	09/10/2007	4w
7	Diseño y desarrollo de TCP	09/10/2007	03/12/2007	8w
8	Deploy de Tomcat	03/12/2007	09/12/2007	1w
9	Documentación	09/12/2007	22/12/2007	2w

Ilustración 3: Gantt de la planificación inicial del proyecto.

5.2 Selección de entorno de trabajo e instalación

El primer entorno de desarrollo que configuramos al inicio del proyecto, consistía en una máquina virtual con un sistema operativo Linux Fedora Core 6 [36]. La decisión de usar un sistema operativo Linux tuvo que ver con la desconfianza que nos generaba usar IPv6 con Windows, ya que éste no lo traía nativo en su versión XP. Igualmente por tratarse de una herramienta en Java, por más que sea desarrollado en Linux, mantiene su condición de portable a otros sistemas operativos casi de forma trivial, con la única salvedad de tener que instalarse las librerías indicadas y compilar los archivos escritos en lenguaje C que permiten la comunicación vía JNI de la máquina virtual con el dispositivo de red (.dll en Windows, .so en Linux, etc) y adicionalmente compilar la máquina virtual Java (SableVM).

Se utilizó Eclipse [37] como IDE de desarrollo debido a que ambos integrantes del grupo estábamos muy familiarizados con esta herramienta. A su vez Eclipse fue configurado con el plugin de SVN (subeclipse) para tener acceso al sourceforge de INRIA, donde se encuentra alojado el código y la documentación del proyecto. La versión utilizada de Eclipse fue la 3.3.1.1. La URL del SVN utilizado es https://scm.gforge.inria.fr/svn/ip4jvm donde creamos dos usuarios con acceso total al mismo (rabelenda e icorrales). En dicho SVN que ya estaba creado para el proyecto anterior, se hicieron modificaciones en cuanto a la estructura de directorios que nos permitieran trabajar más cómodamente. En este sentido se crearon cuatro directorios:

- src: donde se encuentra todo el código (Java y C) de la herramienta.
- lib: donde dejamos las .dll y los .so ya compilados
- vmachine: donde están las máquinas virtuales utilizadas (SableVM y OpenJDK) en las versiones utilizadas.
- doc: directorio donde se mantiene la documentación relacionada con el proyecto ya sea que la hayamos usado como input o que fuese un documento generado por nosotros.

Al comienzo también se debía tener instalado SableVM (versión 1.13) y la librería Pcap. Nótese que la manera de instalar y configurar el ambiente, así como las direcciones donde conseguir los archivos, se detallará más adelante en el Apéndice 10.1.

Es necesario remarcar que al tratarse de una herramienta cuya funcionalidad hace imprescindible la conectividad a una red, debía contarse con una a la hora de realizar pruebas. Sin embargo, al principio tuvimos algunos problemas con nuestra herramienta si la red estaba compuesta por una computadora Linux y una Windows; bajo estas premisas obteníamos resultados inesperados y desconocíamos el motivo, sin embargo el funcionamiento era correcto cuando la red la componíamos entre terminales con sistemas operativos iguales. Esto que pudiésemos hacer pruebas en una red configurada entre la máquina virtual de desarrollo (Linux) y la máquina host (Windows). Por esto es que necesitábamos dos computadoras físicas con una red bien configurada entre ambas para realizar las pruebas.

Luego, cuando comenzamos a efectivizar cambios en el código, debimos agregar a nuestro entorno un producto llamado Wireshark [38] (utilizamos su versión 0.99.6), el cuál permite analizar los paquetes que llegan o salen por la red; ayudándonos así a verificar el correcto comportamiento de la herramienta y a detectar problemas con la misma.

Más tarde, se decidió cambiar el sistema operativo de Fedora Core 6 a un Linux OpenSuse 10.3 [39], debido a que este sistema operativo era el que usaba nuestro tutor y pensamos sería acertado estar alineados en este sentido.

Por último, en una de las fases detalladas anteriormente, se migró la herramienta de SableVM a OpenJDK, conformando así lo que sería el estado final de nuestro entorno de trabajo. La versión de OpenJDK utilizada fue la 1.7 b22.

Adicionalmente, luego de tener algunos de los RFCs implementados completamente (como Stateless Address Autoconfiguration [53]), se agregaron al ambiente routers, los cuales

consistieron en el demonio radvd [61] corriendo en diferentes distribuciones Linux (OpenSuse 10.3 , Knoppix MIPL [62]). Con esto se puede ver el correcto funcionamiento del stack con computadores en otras redes, pues en el caso básico que se probaba desde el inicio el stack era probado siempre contra implementaciones Linux que estuvieran en la misma Lan que el stack (pues no era posible, en un inicio, otro tipo de comunicación).

5.3 Cambios. Performance y Diseño.

Como se explicó en la sección 5.1, una vez instalado el ambiente de trabajo y entendido el funcionamiento del producto, se procedió a hacer un análisis detallado del código y se confeccionó una lista con todos aquellos aspectos que, según nuestro juicio, ameritaban una modificación con el fin de mejorar el software, ya sea en el aspecto de la performance, como otros aspectos, como ser la escalabilidad o la prolijidad del código en sí.

El documento generado al cabo de este trabajo proponía 4 cambios de diseño y 56 cambios de funcionamiento en 26 clases distintas. Todo estos datos se brindarán de manera más detallada y profunda en el Apéndice 10.3, pero a modo de resumen se puede comentar que los cambios de desarrollo tenían que ver con la creación de un “servidor de Timers” que tratara los vencimientos de los paquetes y los tiempos de espera, ya que en la versión del producto que recibimos, se trataban estos conflictos mediante el continuo cuestionamiento, en un ciclo, de si el tiempo estipulado ya había vencido, incurriendo en *busy waiting*. Otro de los cambios de diseño proponía que no fuera la clase NetStack que procesara los distintos paquetes (tanto de salida como de entrada) de principio a fin, y que se creara un pool de threads para manejarlos.

Los restantes cambios tienen que ver con cosas modificables dentro de una clase, ya sea la algoritmia de una función para optimizarla o corregirla, o por ejemplo, modificar el tipo de un atributo. Un caso común es el de en un atributo que era una lista, proponer su sustitución por varias tablas de mapeo u otras multiestructuras para mejorar tiempos de acceso en las búsquedas por distintos campos.

Sin embargo la tarea no consistió únicamente en identificar lo cambios. Además de listarlos, debimos estimar el impacto que cada cambio tendría sobre la performance del producto en una escala de 1 a 3, siendo 1 un impacto casi nulo, y 3 un impacto importante para la mejora de la herramienta. Así mismo debimos calificar de igual manera cada cambio, según la complejidad de la solución propuesta, 1 querría decir que el cambio no demanda demasiado tiempo y 3 corresponde a un cambio extremadamente complejo y, por ende, que insumiría mucho tiempo. Una tercera y última calificación de los cambios tiene que ver con la dependencia que esa porción del código tiene con el resto del producto; para esta clasificación nos pareció apropiado contar cuantas referencias había en todo el código del proyecto a la función o atributo en cuestión, de esta manera, según la cantidad de referencias, calificamos a cada uno de los cambios de 1 a 3, siendo 1 para aquellos cambios sobre código invocado en pocas partes, y 3 para aquellos muy usados a lo largo del producto.

Al clasificar los 58 cambios identificados según las tres categorías antes citadas se obtuvieron los números que se muestran en la siguiente ilustración:

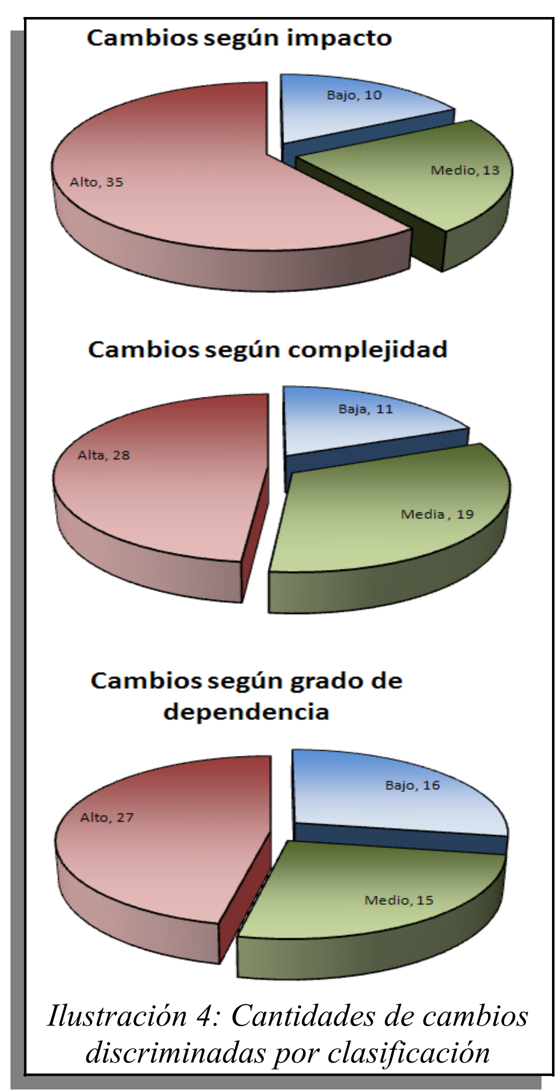


Ilustración 4: Cantidades de cambios discriminadas por clasificación

Los números ilustrados, hablan a las claras de que la mayoría de los cambios propuestos, tendrían tras su realización, un impacto medio o alto en la performance del aplicativo (48 de 58). Posiblemente de la mano con este último dato, va el hecho de que 47 de los 58 cambios identificados fueron identificados como de complejidad media o alta.

Una vez hecho este relevamiento, ayudados por las distintas calificaciones, se dio una priorización a cada cambio intentando contemplar la necesidad de tener alto impacto al comienzo e intentando realizar cambios más complejos y con bajo impacto por último. En este sentido se dividieron los cambios en 7 fases de cambios, las cuáles contaban con los siguientes encabezados:

- *Fase I (Cambios Iniciales de Diseño)*
- *Fase II (Cambios Iniciales en clases con impacto alto)*

- *Fase III (Cambios Iniciales en clases con impacto medio o bajo)*
- *Fase IV (Cambios Generales de alto impacto)*
- *Fase V (Cambios Generales de impacto medio)*
- *Fase VI (Cambios de complejidad baja, dependencias bajas e impacto bajo)*
- *Fase VII (Cambios con impacto bajo y no triviales)*

El contenido de cada fase en cuanto a porcentajes de cambios según complejidad, impacto y dependencias se detalla más en el Apéndice 10.3, pero a modo ilustrativo mostramos las siguientes gráficas que ilustran dichos porcentajes en función de las 7 fases.

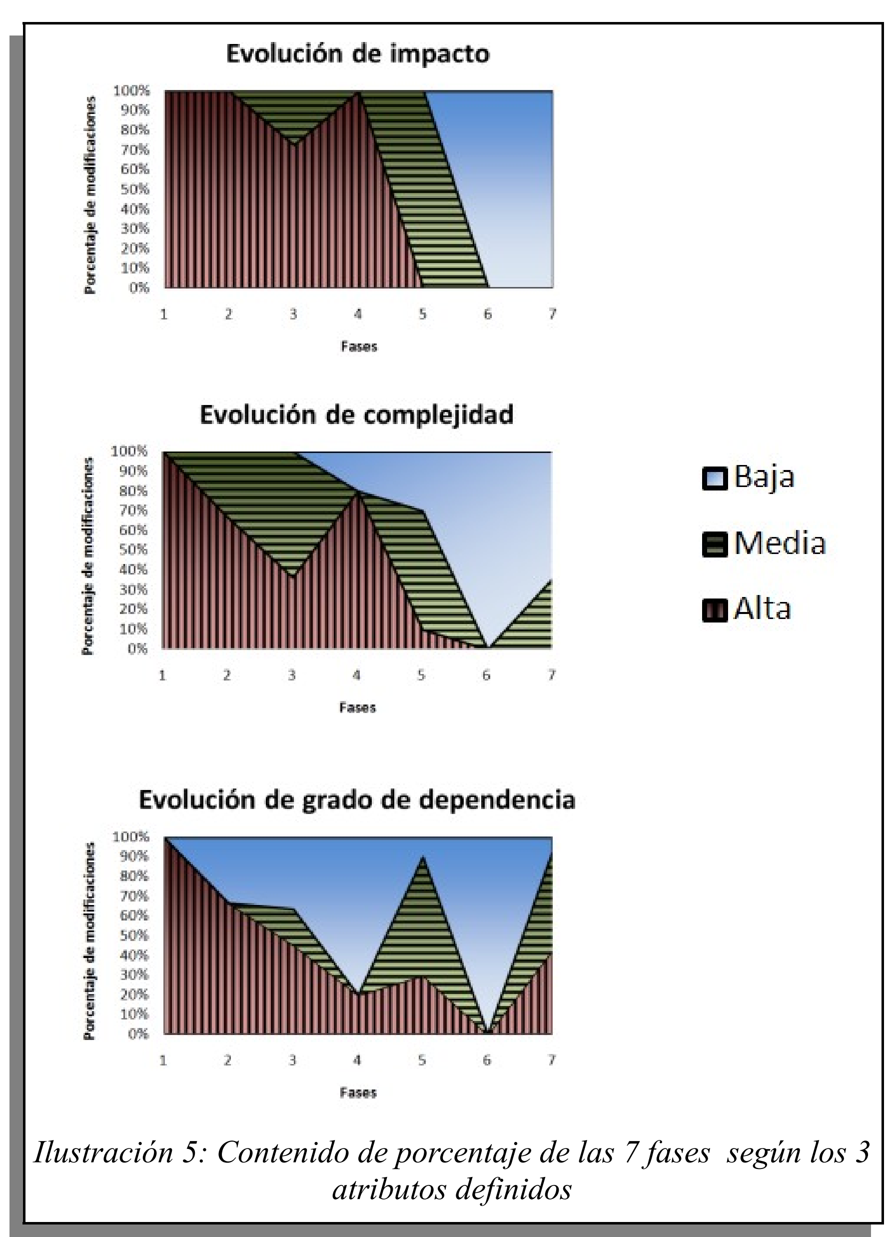


Ilustración 5: Contenido de porcentaje de las 7 fases según los 3 atributos definidos

Se ilustra en dichas gráficas que en las fases iniciales el porcentaje mayor de cambios se correspondía con modificaciones de alto impacto, lo cual se debe a que intentamos realizar los cambios que mejorasen en mayor medida el producto en las primeras fases de cambios. Así mismo este impacto alto viene de la mano de un alto esfuerzo, el cual se ve en la segunda gráfica.

Trazado el plan, comenzaron a ejecutarse los cambios según lo planificado hasta dar por finalizada cada una de las 7 fases.

Resultados:

Los resultados obtenidos luego de los cambios hablan de un buen trabajo y de cierto grado de acierto en los cambios planteados y realizados.

En la siguiente imagen se detalla el diseño obtenido en lo referente a la parte del stack:

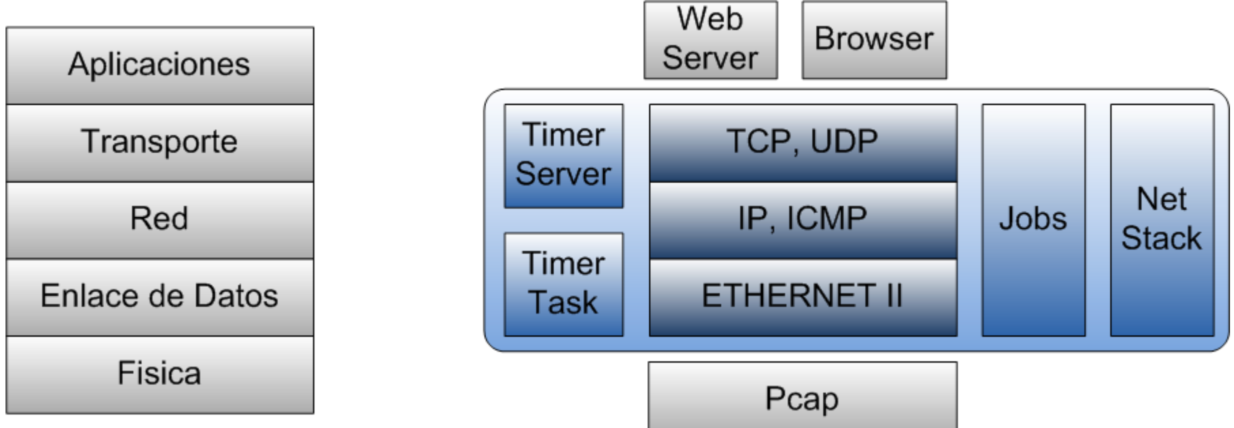


Ilustración 6: Diseño general del stack

En la parte derecha de esta ilustración, se muestra un uso, a modo de ejemplo, de nuestro stack, siendo este último la parte encerrada en el rectángulo azul. Se aprecia que el stack tiene los protocolos TCP y UDP en capa de transporte (los cuáles ofrecen servicios a las aplicaciones que corren por encima y que son externas a nuestra herramienta), IP e ICMP brindan las funcionalidades de capa de red mientras que Ethernet II implementa la capa de enlace de datos, la cual hace uso de la librería Pcap para comunicarse con la capa física. Como diferencia principal frente al diseño anterior, aparecen las entidades TimerServer y TimerTask. TimerServer refiere a un servidor encargado de manejar los tiempos de vencimientos y otros temporizadores inherentes a todos los protocolos implementados independientemente de la capa en la cual se encuentra el mismo. Por cada control de tiempo que deba llevar dicho servidor, se asigna un TimerTask, que es el que sabe que tarea debe realizarse una vez que el tiempo estipulado se agote, de esta manera se puede extender a esta clase con las distintas medidas a tomarse según el temporizador. Por ejemplo puede extenderse con una clase que contenga el accionar necesario al vencerse el tiempo de un fragmento, otra que atienda los

vencimientos de las entradas de las tablas de neighbors (vecinos de red), y así sucesivamente con cada uno de los temporizadores de todo el stack. Vemos que estas extensiones logran una transparencia hacia el servidor principal (TimerServer) quien desconoce como manejar los vencimientos particulares. Además, otra bondad de este diseño es el comportamiento modular del mismo, el cuál permite agregar nuevos temporizadores y acciones a tomar extendiendo a la clase TimerTask, sin la necesidad de cambiar nada de lo ya implementado, así como también cambiar completamente el comportamiento del vencimiento de un temporizador únicamente cambiando una función en la clase deseada, siendo también completamente transparente para el servidor.

En cuanto a performance se hicieron dos tipos de pruebas con el fin de comparar el uso tanto del CPU como de la memoria RAM. Las primera prueba se realizó con el programa en ejecución pero de forma ociosa, o sea, sin procesar ningún paquete ni de entrada ni de salida, simplemente esperando que alguna transferencia de datos, entrante o saliente, llegase al stack para ser procesada. Dicha prueba se ejecutó repetidas veces durante el lapso de un minuto por vez y se pudo apreciar que el uso del CPU por parte de la versión inicial de la herramienta (Prototipo 1) oscilaba siempre entorno al 100% (el máximo fue 99,6% y el mínimo 89,2%, mínimo este que se dio únicamente en el comienzo de la ejecución, para luego aumentar paulatinamente y nunca regresar este valor). Por otra parte, el uso del CPU de la versión resultante a partir de los cambios efectuados (Prototipo 2), no se despega del 0%, esto se debe a que en la nueva versión de la herramienta, el stack está "dormido" hasta que no le llegue algún paquete sobre el cuál trabajar, mientras que en la versión inicial se incurría en *busy waiting* consultando en cada iteración si había llegado algo en lo que trabajar.

A continuación se muestra una gráfica que ilustra el uso del procesador en función del tiempo, construida a partir de los promedios de los datos obtenidos en las pruebas realizadas:

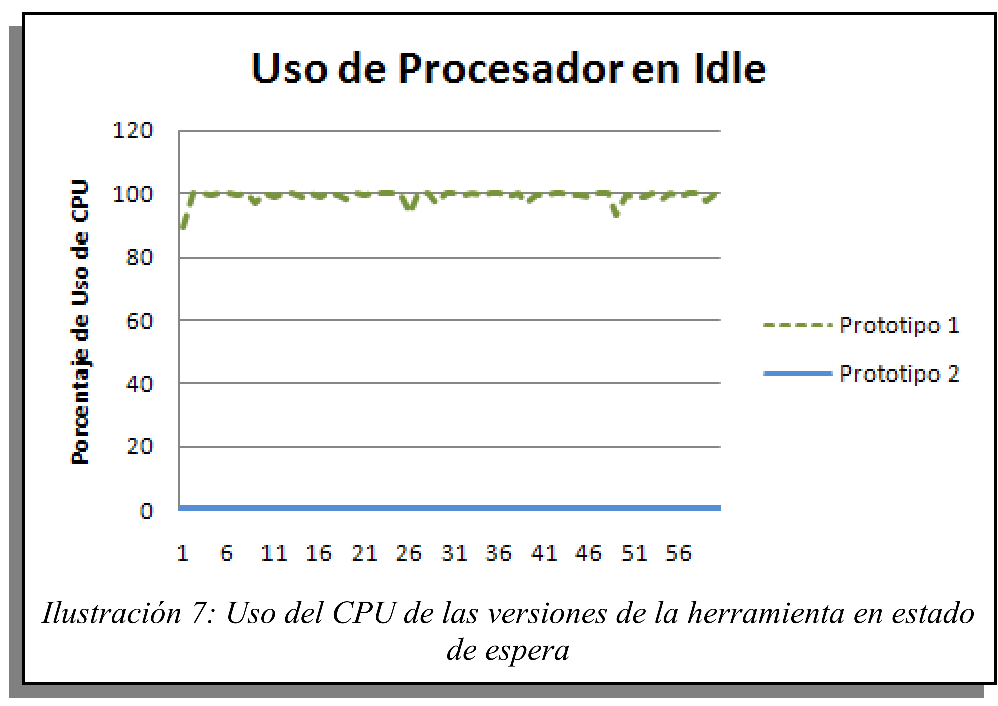


Ilustración 7: Uso del CPU de las versiones de la herramienta en estado de espera

El uso de memoria de las distintas versiones también fue bastante dispar, y habla de un mejor uso por parte la versión post-modificaciones. Esto en parte fue una sorpresa, ya que al agregar el uso de threads y de timers, así como de estructuras más complejas para almacenamiento de datos (por ejemplo se cambiaron algunos atributos que eran listas por varios hashes con distintos campos como clave), pensábamos que podría darse exactamente al contrario. Sin embargo, mientras que la versión obtenida a partir de los cambios se mantuvo invariante en un 1,9% de uso de memoria (porcentaje de memoria usada por el programa en comparación con la memoria total de la computadora), la versión inicial osciló de forma vertiginosa entre 1,8% y 3,4% y manteniéndose casi siempre por encima de la franja del 2,0%. Lo antedicho queda ilustrado por la siguiente gráfica:

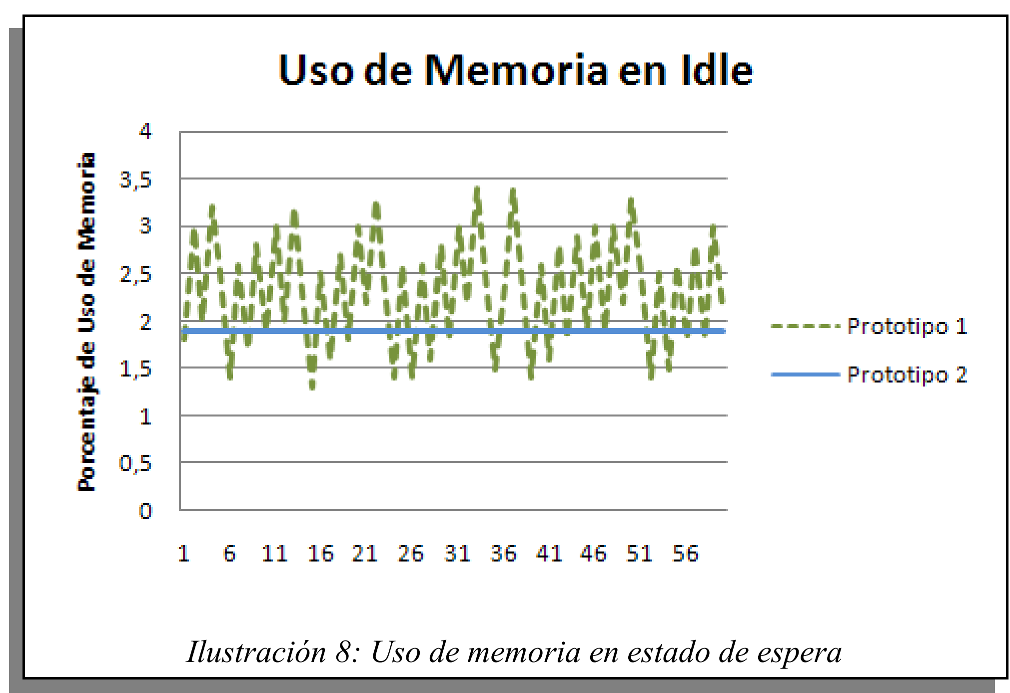


Ilustración 8: Uso de memoria en estado de espera

La segunda prueba consistió en medir nuevamente el uso de CPU y memoria en ambas versiones, pero esta vez en ejecución de un ping cada cinco segundos. De esta manera se ejecutaron 12 pings en un lapso de un minuto en cada versión y se recabaron los datos de ambos programas mientras estos procesaban paquetes de salida y de entrada (los pings y sus respuestas).

El uso de CPU nuevamente demostró las carencias que tenía la versión inicial al respecto. Se observó que el uso estaba cercano al 100% excepto cuando se procesaba un paquete de salida, donde habían pequeños picos de baja que llegaron a un mínimo de 82%. Además se observa que al comenzar la ejecución, o sea cuando se mandan los paquetes de Neighbor Solicitation y Router Solicitation, el consumo del procesador baja hasta un 72%. Esto ha de deberse a que el programa encuentra otros cuellos de botella, como ser la escritura de los datos en el dispositivo de red, lo cuál puede demorar el tiempo suficiente como para que se le

expropie algo del procesador, cosa que no puede ocurrir mientras está en el loop constantemente verificando si llega algo.

En contrapartida, la versión post-cambios del aplicativo tiene picos de alta en cada procesamiento de paquete que hacen que se despegue ocasionalmente del 0% de uso. Al inicio llega hasta al 3,9% de uso de CPU, debido a la mayor cantidad de paquetes que se envían al comienzo, para luego tener picos de 1% o 2% en cada ping. Igualmente se observa en la siguiente gráfica que esta versión permanece la mayoría del tiempo sin usar procesador:

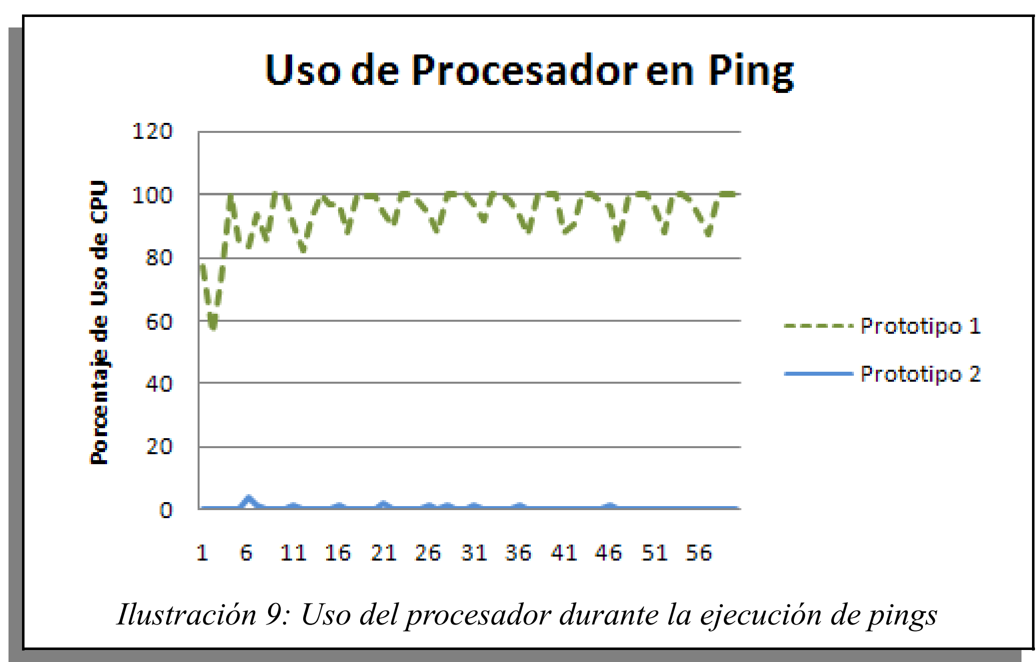


Ilustración 9: Uso del procesador durante la ejecución de pings

El uso de memoria durante la ejecución de los sucesivos pings también resulta favorable a la versión modificada de la herramienta, lo cuál nuevamente era inesperado, ya que el manejo de paquetes implica creación de timers y despertar threads, lo cual supone un uso extra por el cambio de contexto que debe hacerse entre un hilo y otro. Sin embargo el uso se mantuvo casi todo el tiempo en 2,0% apenas teniendo un pico de 2,1%.

Por su parte, el prototipo 1 osciló entre 1,3% y 3,5% pero en promedio supera la línea del 2,0% que tiene la versión post-modificaciones, como puede verse en la siguiente gráfica:

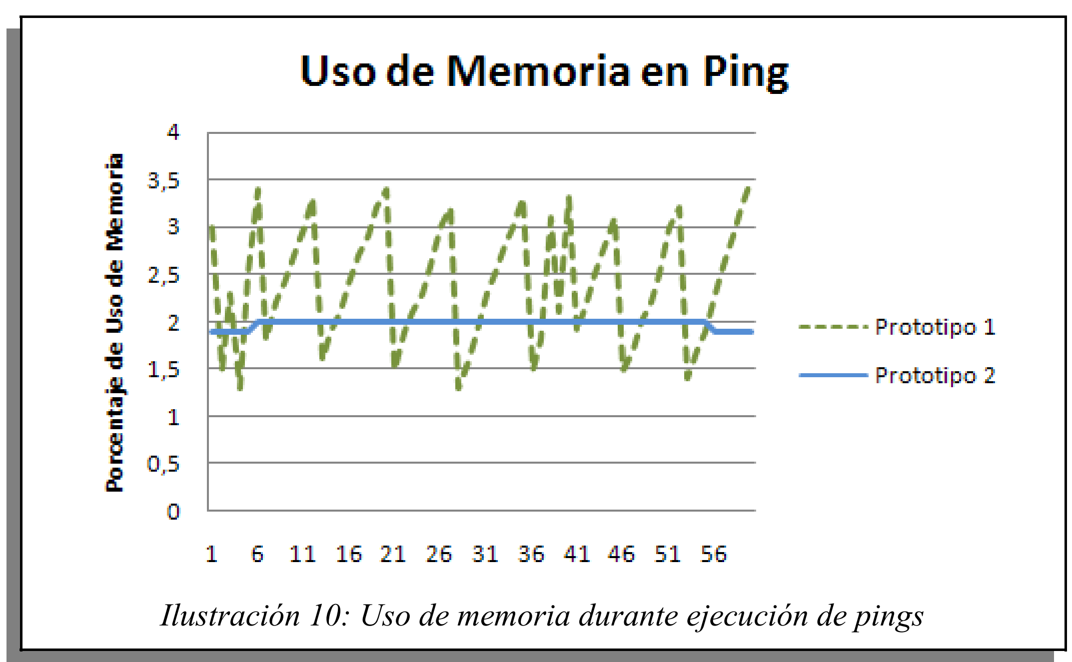


Ilustración 10: Uso de memoria durante ejecución de pings

Por último se compararon los tiempos de respuesta del aplicativo entre que se ejecuta el ping y se despliega la respuesta. Este tiempo, que es el tiempo total resultante de la ejecución del ping, es el tiempo que se demora en enviar el paquete por parte del aplicativo (identifiquémoslo como *e*) más el tiempo de respuesta del nodo, llamémosle *k*, más el tiempo que nuestro aplicativo demora en procesar el paquete de respuesta, identificar que es para él, subirlo a ICMPv6 y que este imprima el mensaje en la consola (llamémosle a este tiempo *r*). Por lo tanto si *t* es el tiempo total al que hacemos referencia, podemos decir que:

$$t = e + k + r.$$

Siendo e y r tiempos que tienen que ver con la performance de la herramienta y con la demoras que esta agrega al proceso de ping, mientras que k está atado al tráfico de red así como a la estructura de la misma entre los nodos involucrados en el ping.

Como para ambas versiones la secuencia de pings se ejecutó bajo las mismas condiciones de conectividad, se puede decir que k, por más que sea variable, en condiciones normales, y en una secuencia de varias ejecuciones, no debería introducir errores en favor de una u otra implementación. Por esto es que tomamos t como una parámetro de ayuda para comparar e + r, o sea los tiempos insumidos por el aplicativo para tratar los paquetes de salida y entrada. También cabe resaltar que la red sobre la cuál se ejecutó la secuencia de pings para ambas versiones, era una red local, conformada por dos computadoras con conexión directa entre ambas, y no existían otros procesos de red corriéndose en simultaneo, por lo cuál el tráfico era casi constante. Diferente hubiese sido si la red usada tuviera mayor carga de tráfico, como por ejemplo, la propia Internet, lo cuál haría que cualquier conclusión obtenida a partir de tiempos, estuviese sujeta a variables para nada despreciables.

Se ejecutó en ambas versiones una secuencia de 15 pings consecutivos, si bien los tiempos son variables, se vio en general que la nueva versión resolvía los pings más rápido. Se debe tener en cuenta para el correcto análisis de los datos, que cada ping no es independiente del anterior, ya que el estado de un stack cuando recibe el n-ésimo ping no precisamente va a ser igual que cuando llegue el siguiente, y este cambio de estado, puede estar dado justamente por el procesamiento de dicho paquete. Por ejemplo cuando dos respuestas llegan en simultaneo, una se procesará mientras la otra deberá esperar a que el stack pueda procesarla.

La diferencia entre ambas versiones se ve en la siguiente gráfica, donde se muestran los tiempos de cada ping en cada versión, y si bien la diferencia no es constante, se puede apreciar que hay entre 50 y 100 milisegundos de diferencia entre una y otra, a favor de la nueva versión, con la única salvedad de la sexta ejecución del comando, donde la versión previa obtuvo un mejor tiempo:

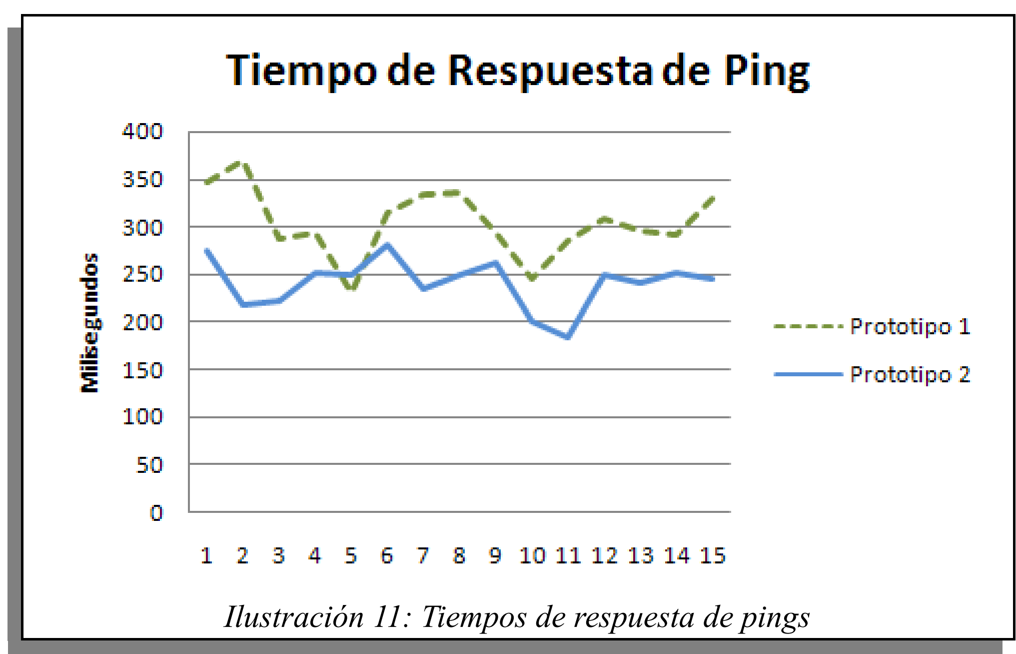


Ilustración 11: Tiempos de respuesta de pings

Vemos que los tiempos de respuesta en la versión previa a los cambios oscilan entorno a los 300 milisegundos (promedio de 304 milisegundos en las 15 ejecuciones), mientras que la versión obtenida a partir de las modificaciones tiene siempre respuestas que rondan los 250 milisegundos (promedio de 241 milisegundos en las 15 ejecuciones). Por otra parte, al calcular la desviación estándar en ambos juegos de datos, vemos que los tiempos del Prototipo 1 tienen una desviación igual a 36,8 mientras que los del nuevo prototipo tienen una desviación de 27,5 por lo cual podemos decir, no sólo que los tiempos del nuevo prototipo son más bajos, sino que también se comportan mejor, en el sentido de que los tiempos de respuesta se alejan menos del tiempo promedio esperado.

En conclusión, como se ve reflejado en las pruebas comparativas realizadas, los resultados arrojados fueron muy positivos, y hasta mejor de lo esperado en algunos puntos, como el uso de memoria. Se obtuvo una herramienta con mucho mejor aprovechamiento de los ciclos de

CPU y con un uso mucho más responsable del mismo, con una demanda de memoria RAM levemente menor y más estable y con tiempos de respuesta que suponen una menor demora agregada por parte del aplicativo al intercambio de datos.

Todo esto habla de una acertada ejecución de esta fase y de la anterior, la cuál sirvió de input para la realización de los cambios.

Por último cabe resaltar que todas las pruebas se realizaron sobre las siguientes condiciones:

- Computador Host del Stack

  El stack se corrió dentro de una máquina virtual con sistema operativo Linux Fedora Core 6 que contaba con 700 MB de memoria RAM. Además la implementación utilizada estaba integrada, como se mencionó anteriormente, a la máquina virtual Java SableVM. El computador huésped se trataba de un Sempron [40] +2500 (1,7 GHz) con 1,5 GB de RAM y con el sistema operativo Windows XP + SP2.

- Computador auxiliar para respuestas

  El computador auxiliar utilizado fue otra máquina virtual, pero esta con 768 MB de memoria RAM y corriendo el sistema operativo OpenSuse 10.3. El host de la máquina virtual fue un computador con procesador AMD Turion 62 X2 TL-56 (1.81 Ghz) [41] y con 2 GB de memoria RAM; su sistema operativo era Windows XP + SP2.

- Conectividad

  La conectividad entre los dos nodos estaba dado por una conexión cableada fast ethernet de 100Mb/s, full duplex mediante cable cruzado.

5.4 Migración a otra VM

Como se vio en la sección 4.3, en un principio se planteó migrar el proyecto a otra máquina virtual con mayor soporte y más usada, como ser Harmony y OpenJDK. Se marcó como de mayor importancia la migración a esta última, tratándose de una máquina virtual altamente usada.

La migración se dividió en los siguientes pasos:

- Estudio de la máquina virtual
- Compilación de la máquina virtual
- Integración del stack a la máquina virtual de manera que quedara por defecto como stack a usar cada vez que se requiriera una instancia de DatagramSocket. Esta clase hasta el momento era la única que utilizaba por defecto el stack ya que era la única brindada por una máquina virtual Java de la cuál se encontraba parte de la lógica

implementada como para soportar su funcionamiento.

- Test de la correctitud de las funcionalidades y que se mantuvieran tal cual se encontraban antes de la migración.

En un inicio se realizó en paralelo el estudio y compilación de ambas máquinas virtuales, lo cuál consumió más tiempo de lo esperado.

Con respecto a OpenJDK, en un principio se probó tratando de compilar el proyecto IcedTea [42] que se trata de un proyecto para compilar OpenJDK en RedHat [43]. No se tuvo éxito al tratar de compilarlo en Fedora Core 6 (que era el entorno que estábamos utilizando en ese entonces). Luego de esto utilizando la versión b22 de OpenJDK 1.7 y con ayuda de la máquina virtual JDK 1.6.0.3 se compiló satisfactoriamente en Fedora Core 6. En este caso fue de extrema importancia la utilización del mail list de OpenJDK [44] debido a la poca documentación existente al respecto. Se requirieron la instalación de varios paquetes y configuraciones de variables de entorno que se detallan en el Apéndice 10.2.

A continuación se procedió con la integración del stack. En base a varias pruebas, tiempo y sentido común se logró integrar tanto las clases del stack así como las librerías que la misma utiliza. Un punto importante a notar en la integración es que en el caso con SableVM se requerían cambios en un makefile, 2 scripts, 3 clases (dentro de las cuales se encontraban la clase ClassLoader y la clase Resolve las cuales estaban programadas en C++) y en la estructura de directorios. Por el contrario en la integración realizada de OpenJDK se modificó un makefile agregando una única linea, se modificó también una linea en el script de compilación y se cambió una sola clase (clase Java DatagramSocket) agregando una única linea a la misma. En base a estos criterios, pensamos que la integración realizada con OpenJDK es más prolija que la que se tenia con SableVM. Por otro lado se encontró, investigando, que la selección de la clase que implementa DatagramSocket se puede realizar en base a parametrizaciones de la máquina virtual a la hora de ejecutar un programa dado. El testing final se realizó satisfactoriamente.

Luego de haber compilado satisfactoriamente en Fedora Core 6, se decidió migrar todo a OpenSuse 10.3 para alinearnos con el tutor en base al sistema operativo que el mismo utilizaba. Esto requirió un nuevo estudio y re parametrizaciones pero únicamente para compilar la máquina virtual (no fueron necesarios cambios para la integración del stack en la máquina virtual).

Para más detalles sobre la migración a OpenJDK en ambos sistemas operativos se puede consultar el Apéndice 10.2.

Ahora bien, con respecto a la máquina virtual Harmony, se estudió durante un tiempo considerable (tres semanas y media) su compilación en Windows, todo esto en paralelo al estudio que se realizó también de OpenJDK. Luego de este tiempo al ver los avances alentadores realizados sobre OpenJDK y al estar la versión de Harmony en Beta y con muy poca documentación y fuentes de donde obtener apoyo, se decidió abandonar la investigación en la misma y dedicar tiempo a otros puntos más importantes a los que restaba prestarle atención.

5.5 Diseño e implementación de TCP

En esta sección se presentarán los principales aspectos sobre el diseño y la implementación inicial de TCP en el stack. Se brindará una introducción donde se explicarán las generalidades de TCP. Se expondrán y comentaran puntos importantes del RFC 793 [9] que especifica el protocolo y finalmente se explicaran los principales aspectos propios de la implementación realizada.

Introducción

Como se mencionó en la sección 4.5 uno de los aspectos importantes del desarrollo del proyecto consta en la implementación de TCP en el stack. Esto permitirá a cualquier aplicación implementada en Java conectarse mediante el uso de sockets a aplicaciones en otros computadores brindando un servicio confiable para la transferencia de datos.

TCP (Transmission Control Protocol) conjuntamente con UDP y otros protocolos forma parte de la capa 4 (de transferencia de datos) de un stack típico de Internet y es uno de los protocolos centrales de los mismos. Brinda una conexión confiable y un despacho ordenado de streams de datos entre 2 nodos de una red, mediante el establecimiento de una conexión donde se especifican las características de la transferencia, por ejemplo tamaños de buffers, importancia de los datos que se envían, tiempos, etc.

Este protocolo es muy usado en aplicaciones de Internet como son el e-mail, transferencia de archivos, HTTP, etc; por lo que su implementación es crucial en cualquier stack que quiera brindar estos servicios.

Existen en la actualidad muchas implementaciones de TCP (Reno [45], new Reno [46], Vegas [47], etc). La realizada en este proyecto se basa en la implementación Net/3 [49], utilizada en la familia de sistemas operativos Linux BSD-Lite 4.4 [48]. Como base para realizar esta implementación se utilizo como referencia el libro de TCP Illustrated, Vol2 - The Implementation de Gary R. Wright y W. Richard Stevens [49] y adicionalmente el RFC 793.

Los requerimientos básicos requeridos para una implementación TCP se encuentran especificados en el RFC 793. Entre los aspectos cubiertos en este RFC están: el formato con el cual se enviarán los datos, estructuras y componentes necesarios para cubrir con los requerimientos básicos de TCP, una máquina de estados que indica como debe comportarse el protocolo frente a distintos eventos y una descripción más detallada del comportamiento del mismo. Aunque al ver el RFC 793 parecería que se indica claramente como debería la implementación proceder en la mayoría de los casos, quedan muchos asuntos por aclarar (esto se ve reflejado en la presencia de RFCs adicionales que complementan a éste), sobre todo cómo proceder para obtener un mejor provecho de los recursos de red existentes evitando las congestiones de la red. Aquí es donde se encuentra en la mayor parte de las implementaciones diferencias que las distingue unas de otras.

Una conexión TCP se establece entre un par de sockets. Estos se identifican mediante la dirección IP de una de las interfaces del nodo de red en el cuál se encuentra y un entero denominado puerto. Una conexión queda identificada entonces por la cuaterna (IP Host1, Puerto Host1, IP Host2, Puerto Host2). Los puertos se asignan ya sea a la hora de establecer una conexión con un servidor o a la hora de esperar conexiones desde un servidor. Para que una aplicación cliente se conecte a otra aplicación servidor, la primera debe conocer el puerto de la segunda. Existen puertos llamados conocidos (well known), en los cuáles se atienden servicios básicos de Internet como DNS [63], Telnet [64], etc. Estos puertos reservados van desde el 1 al 1024.

Formato del Cabezal

TCP al recibir un pedido de envío divide este pedido en segmentos de manera tal de poder gestionar adecuadamente la carga de datos que se envía a la capa de red. Para que los mensajes originales sean adecuadamente entregados a la aplicación destinataria, evidentemente, TCP deberá, del lado del receptor, re-ensamblar los mensajes originales a partir de los segmentos y subir dichos mensajes a las aplicaciones que así lo requieran.

Como todo protocolo de un stack, TCP define un cabezal para el envío de datos, el cuál será parte de los segmentos que TCP genere. En la siguiente imagen se muestra como es el formato del cabezal definido en el RFC 793:

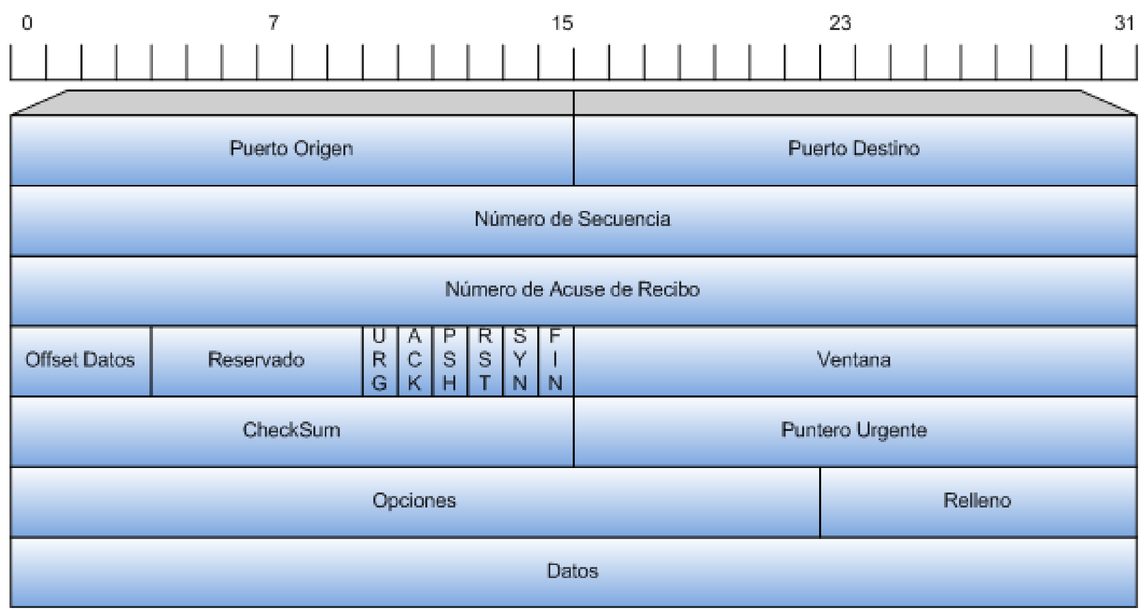


Ilustración 12: Cabezal TCP

- Número de Secuencia

    Este número representa el número de secuencia del primer byte de datos contenido en este segmento. Usualmente se hace referencia al mismo como SN debido a las

siglas de su nombre en inglés "sequence number". En el caso de que la bandera denominada SYN (que será explicada posteriormente) se encuentre seteada a uno, el número representa el número de secuencia inicial de la conexión (ISN: 'initial sequence number' también llamado ISS), en este caso el primer octeto de datos es ISN+1. El ISN se establece al iniciar la conexión, y se obtiene en función del timestamp actual del sistema.

- Número de Acuse de Recibo

  Si el bit de control ACK ("acknowledgement", explicado más adelante) está inicializado en uno, este campo (número de acuse de recibo, o también llamado simplemente ACK) contiene el valor del siguiente número de secuencia que el emisor del segmento espera recibir. Una vez que una conexión queda establecida, este número es correctamente inicializado en cada segmento enviado.

- Offset Datos

  Este número representa la cantidad de palabras de 32 bits que ocupa la cabecera de TCP, lo cuál indica dónde comienzan los datos transferidos por el segmento. La cabecera de TCP (incluso una que lleve opciones) esta compuesta siempre un número entero de palabras de 32 bits.

- URG (bandera de datos urgentes)

  Bandera que indica si los datos que contiene el segmento son urgentes. Estos datos son tratados de manera diferente que los normales y al recibir datos con esta bandera inicializada a uno se debería notificar a la aplicación receptora que los datos son urgentes. Esta funcionalidad no es soportada en Java ya que la interfaz de sockets brindada no dispone de indicador alguno sobre este tipo de datos. Para evitar la perdida de estos datos, la interfaz brindada por Java permite especificar la opción OOBInline que trata a los datos urgentes como datos normales, en el caso de que esta opción no sea especificada los datos urgentes serán descartados sin ninguna acción adicional. El RFC 793 tiene algunos aspectos no especificados con respecto a como se tienen que administrar estos datos.

- ACK (bandera de acuse de recibo)

  Bandera que indica si el ACK (número de acuse de recibo) dado en el cabezal se debe procesar.

- PSH (bandera de función push)

  Bandera que indica la utilización de la función push. Cuando llega un segmento con esta bandera inicializada en uno indica que los datos de este segmento y los que hayan llegado anteriormente para la conexión y que aun no se hayan entregado al usuario deben ser entregados inmediatamente a este último. Esta bandera modifica el comportamiento estándar de la recepción de segmentos. Este define que los segmentos seran entregados a la aplicación de usuario destinataria al llenar el buffer de recepción dado, al momento de solicitar la recepción de datos, por la aplicación receptora, o cuando ocurra algún error en la comunicación.

- RST (bandera de función reset)

  Bandera utilizada para reiniciar la conexión. Esta bandera es utilizada principalmente cuando la conexión se encuentra en un estado indebido al recibir un determinado segmento o cuando ocurre algún error en la conexión que amerite abortar la misma. Al recibir esta bandera inicializada en uno la conexión correspondiente debe ser abortada, descartando todos los datos sobre la misma, por parte del receptor del segmento.

- SYN (bandera de inicio de conexión)

  Bandera utilizada para establecer una conexión. La utilización de esta bandera será explicada en la explicación de la función Open detallada en la sección de interfaces estándares de TCP.

- FIN (bandera de fin de conexión)

  Bandera utilizada para finalizar una conexión. El uso de esta bandera se puede apreciar posteriormente en la explicación de la función Close en la sección de interfaces estándares de TCP.

- Ventana (tamaño de ventana de recepción)

  Este número indica la cantidad de bytes a partir del actual número de ACK que el emisor del segmento está dispuesto a aceptar. Este es un indicador de cuanto espacio tiene el emisor en su buffer de recepción a la hora de enviar el segmento. En base a este campo (conjuntamente con otros datos y estimaciones realizadas) se desarrollan las politicas y algoritmos que permiten evitar congestionamientos.

- Checksum

  Suma de comprobación de integridad de los datos del segmento, o sea, permite establecer al recibir un segmento que el mismo no ha sido modificado intencional o accidentalmente. Se calcula teniendo en base a un pseudo-cabezal el cual se muestra en la siguiente figura:

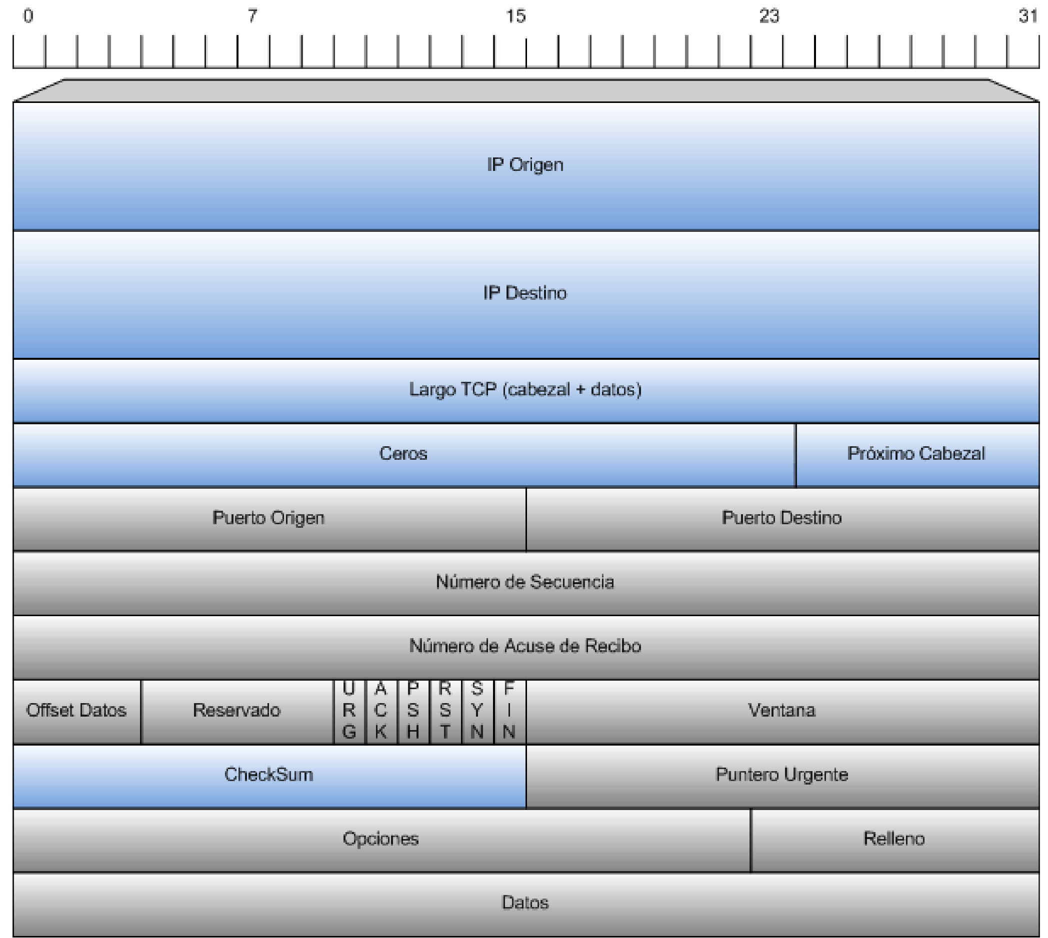


Ilustración 13: Pseudo Cabezal TCP

Los campos marcados de gris son aquellos que se setean con los valores que se obtienen del cabezal TCP. El checksum en este pseudo cabezal es seteado en 0. El campo de próximo cabezal se setea con el valor de TCP (6).

- Puntero Urgente

  Offset con respecto del número de secuencia dado en el cabezal a partir del cual empiezan los datos urgentes del segmento.

- Opciones

  Campo de tamaño variable que contiene parámetros adicionales a ser utilizados por los nodos TCP. Si la lista de opciones ocupa un espacio no múltiplo de 4 bytes se rellena con ceros, denominado campo de relleno.

  Se establecen para las opciones 2 posibles formatos representados por las siguientes figuras:

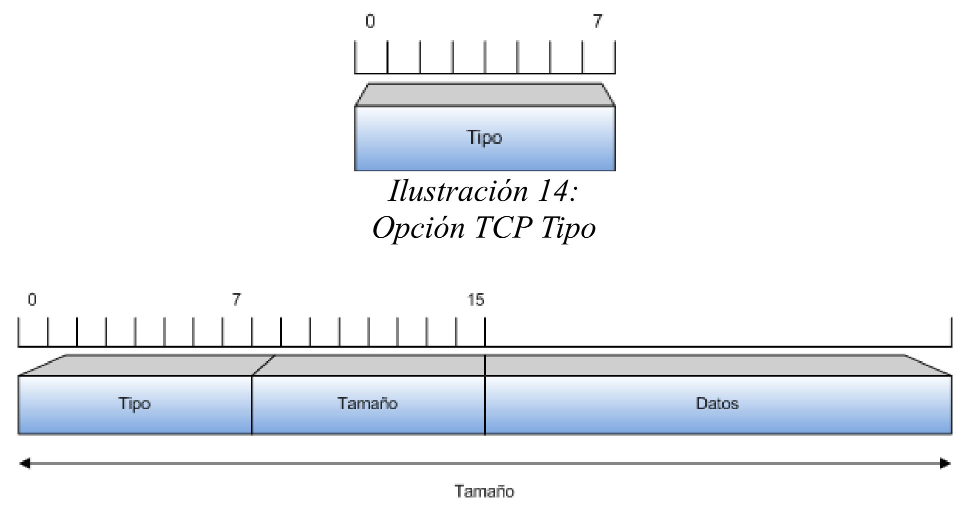


Ilustración 14: Opción TCP Tipo

Ilustración 15: Opción TCP Tipo y Datos

Tipo: identifica a la opción.

Tamaño: Tamaño en bytes de toda la opción, incluyendo los campos de tipo y tamaño.

En el RFC 793 se especifican 3 opciones básicas: NOP, ListEnd y MaximumSegmentSize. Adicionalmente a estas 3 opciones, como se verá más adelante, se implementaron 3 más.

Funcionamiento

El ciclo de vida estándar de una conexión TCP pasa de estar cerrada (o no creada) a abrirse, establecerse la conexión, enviarse y/o recibirse datos, y finalmente cerrarse.

En el siguiente diagrama se muestra una máquina de estados muy similar a la definida en el RFC 793. La misma especifica los principales aspectos y cómo una conexión va pasando de un estado a otro dependiendo de los eventos surgidos. Esta máquina de estados difiere de la definida en el RFC 793 ya que se denomina al Open pasivo como Listen y al activo como Connect, lo cuál se orienta más a lo que se va a implementar finalmente. Otra diferencia es que se especifica el estado Finished, que en realidad es el estado Closed pero se decidió utilizar este nombre (Finished) para diferenciar claramente entre el inicio de una conexión y el fin de la misma, pero se debe considerar que los dos estados (Finished y Closed) en realidad se tratan de uno solo. Además muestra algunas transiciones adicionales no presentadas en el RFC 793.

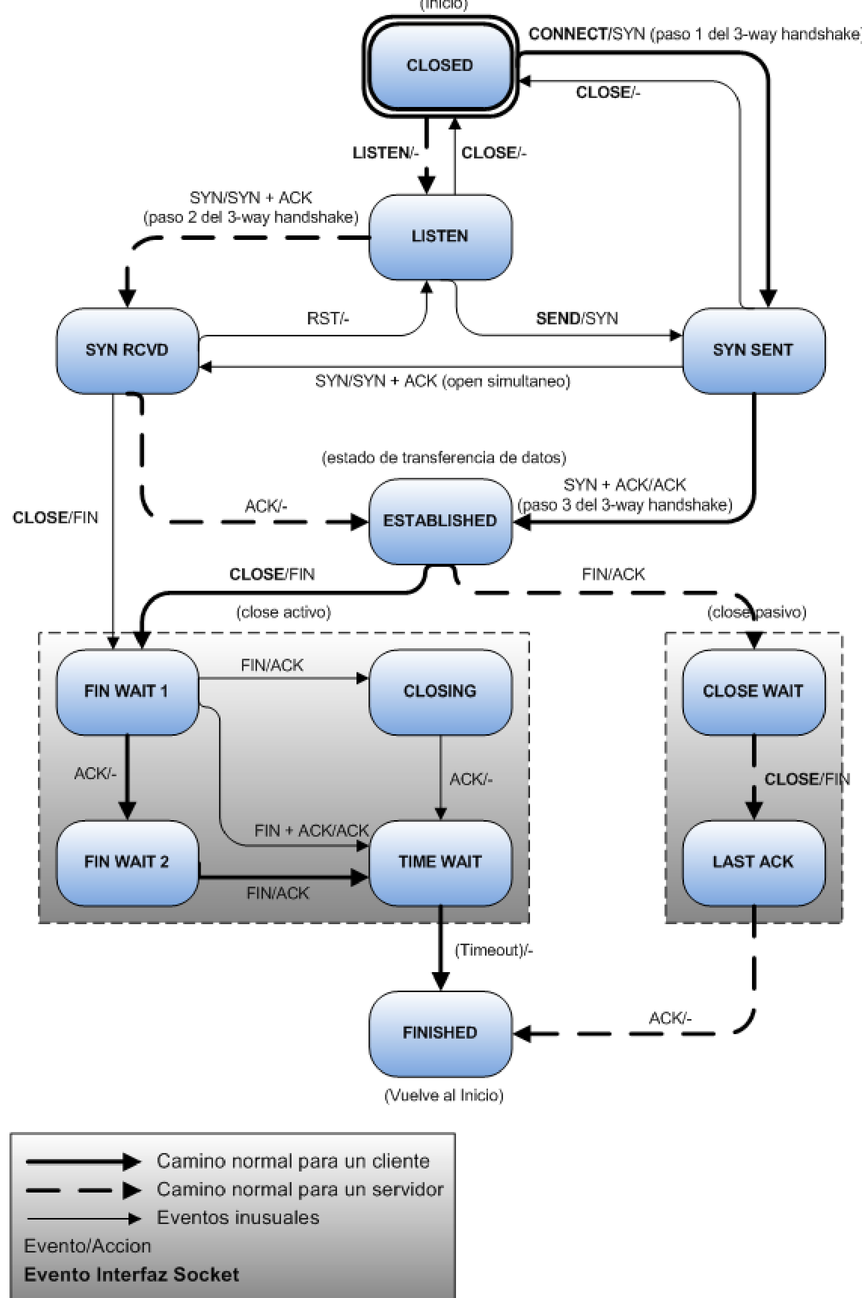


Ilustración 16: Máquina de estados de TCP

Este diagrama de estados se explica en detalle en el Apéndice 10.4 dónde se menciona como es el procesamiento de cada uno de los eventos. Es por esto que aquí nos remitiremos solamente a presentarlo y en la siguiente sección se presentarán las principales características de procesamientos de eventos al explicar las interfaces requeridas por TCP.

Interfaces estándares de TCP

TCP al encontrarse en capa 4, debe brindar una interfaz para la comunicación con la capa de aplicación y debe utilizar la interfaz definida por la capa de red (ésta depende directamente de la implementación de la capa de red).

El RFC 793 define las siguientes primitivas a ser implementadas en la interfaz brindada a la capa de aplicación:

1. *OPEN(puerto local, conector remoto, activo/pasivo [, tiempo de espera] [, prioridad] [,seguridad/compartimentación] [, opciones]) -> id de la conexión local*

   Llamada utilizada para abrir una conexión TCP.

   Existen 2 tipos de open dependiendo del valor que se pase a la llamada en el tercer parámetro:

   - Pasivo

     Al realizar un open pasivo el proceso quedara esperando a que otro nodo quiera establecer una conexión con este proceso (este estado es el representado por el estado LISTEN en la máquina de estados de la ilustración 16). Este tipo de open es el usado por los servidores que esperan en determinado puerto conexiones entrantes.

     Al realizar un open pasivo se debe especificar el puerto local en el cuál se desea escuchar. En el caso donde no se especifique el conector remoto se aceptaran peticiones de conexiones de cualquier nodo, al establecerse una conexión, este conector remoto quedará determinado por la IP y el puerto del otro participante de la conexión.

     Esta función usualmente se representa con la primitiva LISTEN (que es como aparece en la máquina de estados de TCP mostrada).

     En la máquina de estados se puede ver un escenario típico del funcionamiento de esta función siguiendo el camino marcado desde el estado CLOSED al estado ESTABLISHED por las flechas punteadas. En la misma se muestran los estados por los que pasa y los paquetes que se envían.

- Activo

  Al realizar un open activo se inicia el proceso de inicio de conexión que consta en un saludo de 3 vías. En el diagrama de estados presentado, este proceso comprende todos los estados y transiciones que se encuentran por encima del estado ESTABLISHED. Un escenario típico del funcionamiento de este proceso se muestra en la siguiente figura:

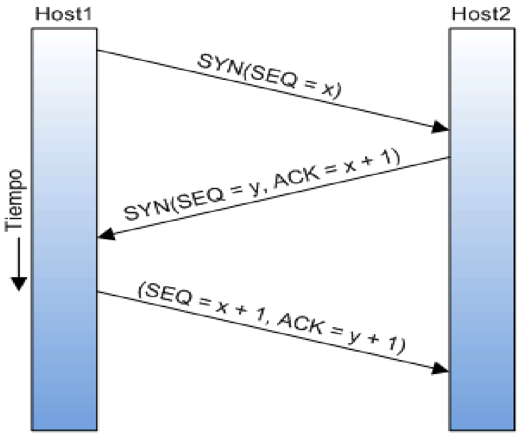


Ilustración 17: Establecimiento de Conexión

En el Apéndice 10.4 se describen con mayor detalle el saludo de 3 vías y se presentan otros escenarios alternativos que explican la importancia de este proceso.

Esta llamada es usualmente la utilizada por los procesos clientes que desean establecer una conexión con un servidor. El puerto local en este caso no es requerido. En el caso de que no se especifique TCP asignará uno efímero (auto generado) para su uso en la conexión. Por otro lado el conector remoto es necesario para saber a que servidor se desea conectarse.

A esta llamada usualmente se le asocia la función denominada Connect (como en el caso de la máquina de estados).

Parámetros:

- Puerto local: puerto que usará el proceso que ejecutó la llamada open en el nodo en el que se encuentra.
- Conector remoto: puerto y dirección IP de la máquina destino a la cuál se quiere establecer la conexión.
- Activo/Pasivo: indica si se quiere realizar un open pasivo o activo.
- Tiempo de espera: tiempo luego de cada envío de datos dentro del cuál los datos tienen que haberse enviado por completo. De lo contrario la conexión se

cierra. El valor por defecto para este parámetro es de 5 minutos.

- Prioridad y Seguridad/Compartimentación: Parámetros utilizados en la conexión para especificar prioridad de los paquetes y también la seguridad, que permite identificar que la información es sensible y debe ser recibida únicamente por usuarios que hayan establecido una conexión con seguridad apropiada. En el caso de la implementación realizada, estos parámetros no se programaron.
- Opciones: opciones a especificar en la conexión. Algunos ejemplos son: especificación de uso de algún algoritmo de la implementación, uso de timer de cierre y especificación del tiempo del mismo, tamaño de los segmentos, etc.

La llamada retorna un identificador de la conexión establecida en caso de que se haya establecido correctamente o, en caso contrario, deberá informar al proceso invocador del error que causó que no se pueda establecer la conexión.

2. *SEND(id de la conexión local, dirección del buffer, contador de bytes, indicador PUSH, indicador URGENT [, tiempo de espera])*

Llamada utilizada para enviar datos al otro extremo de la conexión luego de que una conexión se encuentra establecida. Los datos (como ya se indicó anteriormente) se segmentan y se envían adjuntando el cabezal de TCP. En el cabezal se setean los puertos de origen y destino, el SN, con el número de secuencia que identifica el primer byte del segmento enviado, la cantidad de bytes libres en la ventana de recepción, el ACK con el SN del último paquete recibido más 1 y los restantes valores y opciones como corresponda. Al llegar los datos al receptor éste los almacena en su buffer de recepción para re-ensamblar los datos. Si el emisor no recibe el ACK de los datos enviados luego de un cierto periodo (llamado tiempo de retransmisión) este los reenvía. Este tiempo de retransmisión es calculado dinámicamente por el protocolo midiendo los tiempos de ida y vuelta de los paquetes. El RFC 793 no indica como realizar esto, en el Apéndice 10.4 se explicará con mayor detalle cómo se implemento este proceso.

Parámetros:

- Id de la conexión local: conexión por la cuál se quieren enviar los datos.
- Dirección del buffer: puntero a los datos que se quieren enviar
- Contador de bytes: cantidad de bytes de los datos que se quieren enviar
- Indicador Push: bandera utilizada para indicar que el receptor debe subir los datos que posea de forma inmediata a la capa de aplicación.
- Indicador Urgent: parámetro que indica que los datos que se están enviando son urgentes. Estos datos deberían ser notificados como urgentes del lado del receptor a la capa aplicación al ser entregados.
- Tiempo de espera: parámetro que permite modificar el tiempo de espera establecido en la llamada open.

3. *RECEIVE (id de la conexión local, dirección del buffer, número de bytes) -> número de bytes, indicador 'urgent',indicador 'push'*

Llamada utilizada para recibir en un buffer todos los datos enviados por el otro nodo de red participante en la conexión. Al recibir un segmento, este se almacena en un buffer de reensamblado a partir del cual se reconstruyen los mensajes a ser entregados al buffer de recepción. Cuando se llena el buffer de recepción o cuando se recibe un mensaje con el indicador de push los datos son devueltos al llamador del receive indicando la cantidad de bytes que se recibieron.

Parámetros:

- Id de la conexión local: identificador de la conexión desde la cual se quieren recibir datos.
- Dirección del buffer: puntero al buffer donde se deben poner los datos que lleguen
- Número de bytes: capacidad del buffer de recepción
- Número de bytes retornado: cantidad de bytes que se recibieron
- Indicador 'urgent': indica si los datos que han llegado son urgentes
- Indicador push: indica si el nodo emisor ha realizado un push.

4. *CLOSE (id de la conexión local)*

Función utilizada para iniciar el proceso de cierre de una conexión. En TCP el cierre de conexiones afronta el problema de los dos ejércitos, por lo cuál se utiliza un timer que luego de vencer cierra la conexión forzosamente. El problema de los dos ejércitos se encuentra detallado en el libro Redes de Computadoras, escrito por Andrew S. Tanenbaum [50], en su página 499 de la 3ª edición.

El comportamiento luego de la invocación de esta función se puede visualizar en la máquina de estados presentada. Básicamente al cerrar la conexión se continua mandando los datos de llamadas a sends y receives pendientes, se permite realizar nuevas llamadas a receives pero no de sends. Luego de que el otro nodo también cierre la conexión es cuando entra en juego el timer de cierre de la conexión y efectivamente se cierra.

Un escenario típico luego de la invocación de esta función se muestra en la siguiente figura:

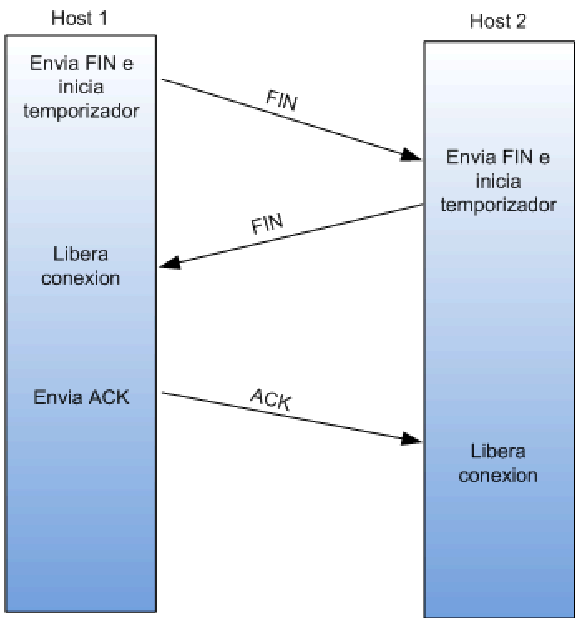


Ilustración 18: Desconexión

Luego de cerrarse una conexión conviene esperar un cierto tiempo antes de establecer una conexión con los mismos sockets para evitar que segmentos ya enviados sean confundidos con segmentos de la nueva conexión.

5. *STATUS (id de la conexión local) -> datos de estado*

Retorna información sobre el estado de la conexión.

Parámetros:

- id de la conexión local: identificador de la conexión de la cuál se quieren obtener los datos de estado.

Datos de estado: conjunto de datos que se retorna luego del llamado de la función, estos pueden variar según la implementación. En el RFC 793 se enumeran los siguientes como posibles datos a mostrar:

conector local,
conector remoto,
nombre de la conexión local,
ventana de recepción,
ventana de envío,
estado de la conexión,
número de buffers en espera del acuse de recibo,
número de buffers pendientes de recepción,

estado urgente,
prioridad,
seguridad/compartimentación,
y tiempo de espera de transmisión

6. *ABORT (id de la conexión local)*

Llamada utilizada para abortar definitivamente una conexión y liberar todos los recursos que ésta tenga. Todos sends pendientes y/o receives serán cancelados. Esta función no notifica de ningún modo al otro nodo acerca del llamado. Luego de llamada esta función, al igual que en el caso de la función close, el stack responde de manera estándar a cualquier segmento que este destinado a una conexión cerrada o inexistente: envía un paquete de reset. Según la implementación puede ser que las aplicaciones que hayan realizado un send y/o receive en el socket reciban un mensaje indicando el aborto de la conexión.

Estas primitivas son brindadas a través de los sockets que son la implementación de la interfaz que brinda TCP a la capa de aplicación. Los sockets por lo general re formulan las primitivas definidas en el RFC 793 por otras, en nuestro caso al tratarse de aplicaciones Java deberemos respetar la interfaz ya especificada para el caso.

Interfaces Java para TCP

Java brinda cuatro clases principales para el manejo de sockets. En la siguiente figura se muestran las principales operaciones de estas clases para implementar los requerimientos del RFC 793:

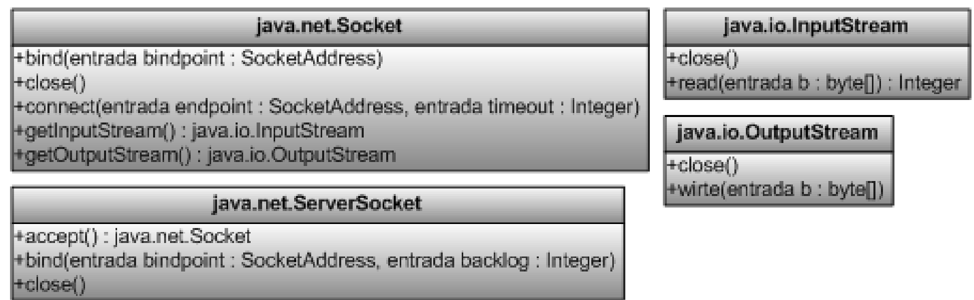


Ilustración 19: Clases Java para Sockets

En la figura anterior se omiten muchas funciones que muestran y permiten setear atributos de la conexión y obtener datos del estado de la conexión. Además se omiten aquellas operaciones que se puedan ser sustituidas por alguna de las presentadas.

La clase Socket permite establecer conexiones a los procesos clientes y además a cualquiera de los dos tipos de procesos (cliente o servidor) cerrar las conexiones y enviar y recibir datos. La función bind permite establecer la dirección IP y puerto local donde se quiere establecer la conexión.

La clase ServerSocket permite crear sockets para procesos que oficiarán de servidores escuchando en un puerto para establecer conexión con algún cliente. La función bind permite establecer la dirección IP y puerto local donde se quiere establecer la conexión y además la cantidad de conexiones en paralelo que puede atender el servidor.

La clase InputStream permite recibir datos desde la conexión desde la cual se haya obtenido la instancia a la clase, y además cerrar la conexión. La clase OutputStream enviar datos y cerrar la conexión.

Un uso típico de estas clases se muestra a continuación con un ejemplo de servidor que envía mensajes y el correspondiente cliente que le recibe los mensajes y los despliega en pantalla:

```
public class Servidor{

public static void main(String[] args){

    ServerSocket server = new
ServerSocket(2004, 10);

    do{
        Socket conn = server.accept();

        OutputStream os=
conn.getOutputStream();

        ObjectOutputStream out = new
ObjectOutputStream();

        try{
            out.flush();
            out.write("HOLA");
            out.flush();
        } catch(IOException e){
            e.printStackTrace();
        } finally{
            try{
                out.close();
                connection.close();
            } catch(IOException e){
                e.printStackTrace();
            }
        }
    }while(true);
}

}
```

```
public class Cliente{

public static void main(String[] args){

    InetAddress add=
InetAddress.getByName("2001::1");

    Socket cli = new Socket(add,2004);
    InputStream is= cli.getInputStream();

    ObjectInputStream in = new
ObjectInputStream(is);

    try{
        String msg= in.readObject();
        System.out.println(msg);
    } catch(IOException e){
        e.printStackTrace();
    } finally{
        try{
            in.close();
            cli.close();
        } catch(IOException e){
            e.printStackTrace();
        }
    }
}

}
```

Diseño e Implementación

En los diagramas presentados en esta sección se utilizan los siguientes símbolos para representar los diferentes tipos de relaciones entre clases, estos símbolos que se derivan de la especificación de UML [67]:

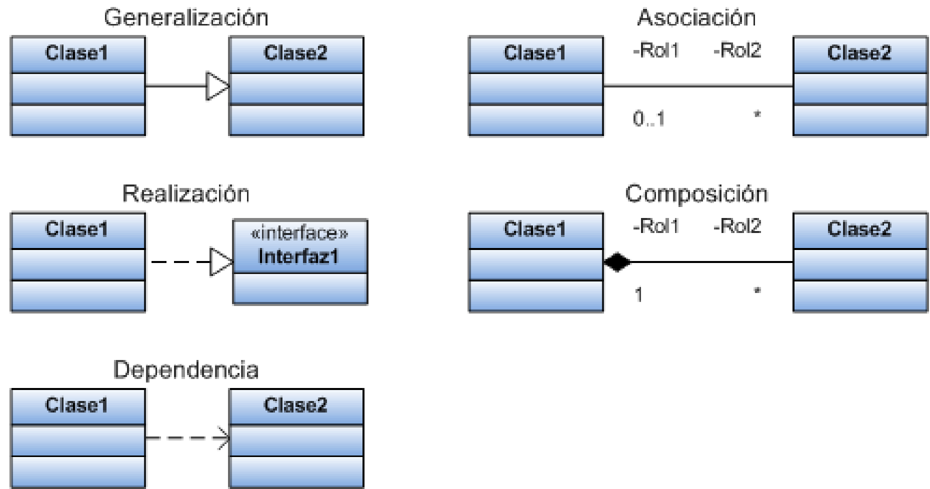


Ilustración 20: Nomenclatura relaciones

En la imagen se muestran los símbolos para los siguientes tipos de relación:

- Generalización

  En la imagen se representa en la generalización que la clase Clase1 es subclase de la clase Clase2.

- Realización

  En la imagen mostrada la realización representa que la clase Clase1 implementa todas las operaciones definidas por la interfaz Interfaz1.

- Composición

  La composición, en la imagen antes mencionada, representa que una instancia de Clase1 esta compuesta por instancias de Clase2. Ademas con los nombres que se encuentran en cada uno de los extremos de la relación se indica el rol de la clase que se encuentra en dicho extremo con respecto a la relación, por ejemplo Clase1 cumple el rol Rol1 para la clase Clase2. Estos roles no son requeridos, por lo que pueden ser omitidos en cualquier composición. Otro punto a detallar aquí es el de las multiplicidades, las mismas se indican con un número mayor o igual a cero, un número (con las mismas condiciones) seguido de dos puntos y otro número, un asterisco (que representa entre 0 e infinito) o un número seguido de 2 puntos y un asterisco (que representa infinito). La multiplicidad indica cuantas instancias de una clase van a estar relacionadas con la misma instancia de la otra clase que forma parte de la relación, bajo la misma relación. Cuando se utilizan los 2 puntos, se

establece que la cantidad de instancias se encuentra acotada por los dos números (o número y asterisco). Por ejemplo en la imagen se representa que una instancia de Clase1 esta compuesta entre una e infinitas instancias de Clase2.

- Asociación

  Una asociación como la mostrada en la imagen anterior representa que las dos clases se encuentran asociadas. Esto, para el uso de este documento, se considera cuando una de las clases (o las dos) tiene algún atributo que es del tipo de la segunda o es una colección de instancias de la segunda. Además se aplican los conceptos explicados anteriormente de roles y multiplicidades. El ejemplo mostrado en la imagen representa que una instancia de Clase1 tiene asociadas entre 0 e infinitas instancias de Clase2, esto se puede dar debido a que las instancias de Clase1 contentan un atributo que sea una colección de instancias de Clase2. Se puede realizar una observación similar mirando la asociación desde el punto de vista de Clase2.

- Dependencia

  Las dependencias en los diagramas que se muestran a continuación en el documento permiten representar cuando una clase hace uso de otra pero no cuenta con otro tipo de relación con esta clase. En el ejemplo dado se muestra que una instancia de Clase1 puede hacer uso de instancias de Clase2.

Luego de un análisis detallado de los requerimientos establecidos por el RFC 793 se realizó un diseño de las clases que se requería implementar, teniendo en cuenta que la implementación se debería integrar con la máquina virtual OpenJDK. De esto se desprendió el siguiente diagrama:

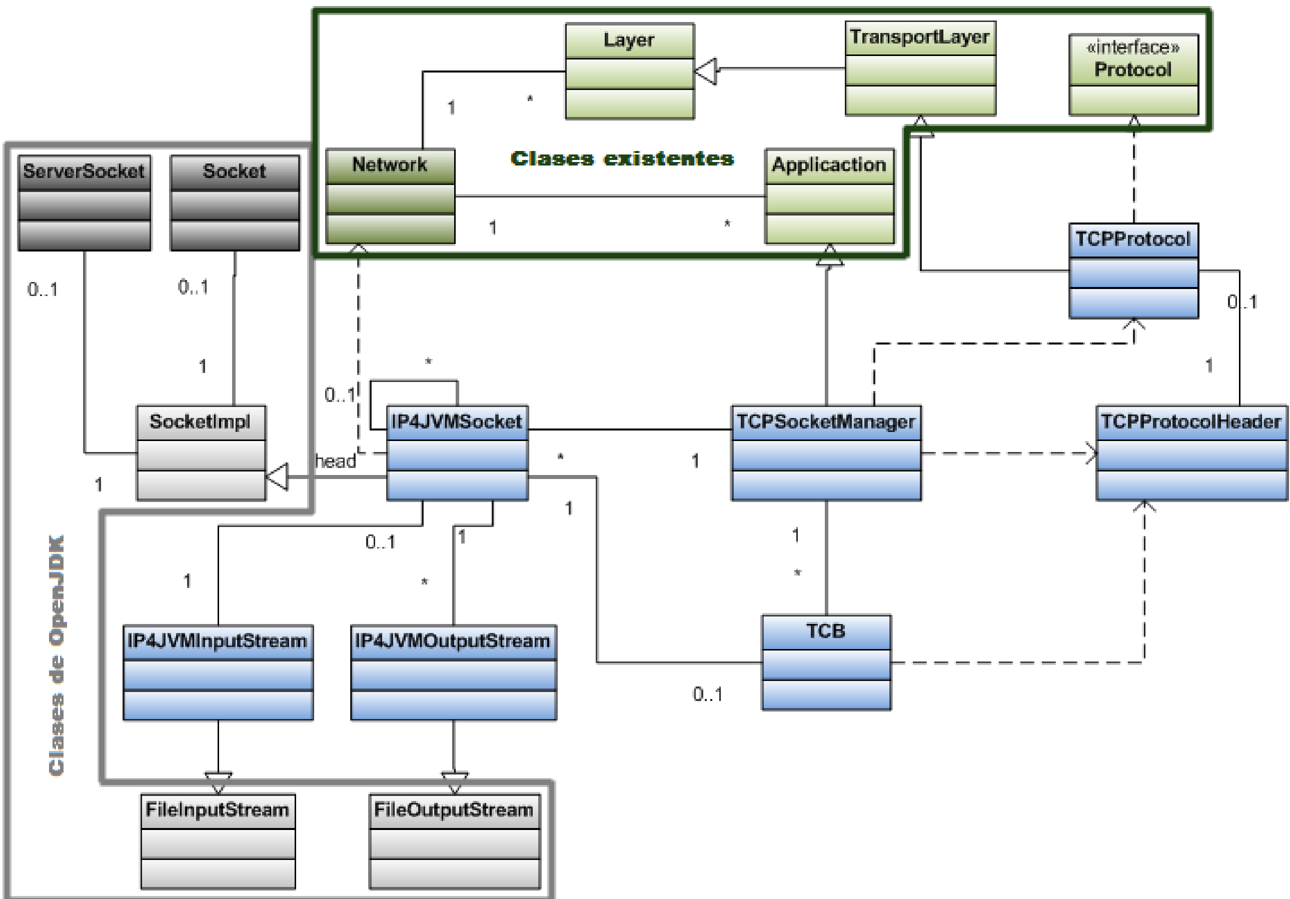


Ilustración 21: Diagrama de las principales clases

El diagrama que se muestra aquí presenta las relaciones entre las clases más importantes diseñadas y posteriormente implementadas y las clases con las cuales se relacionan. Este diagrama no representa fielmente todos los aspectos que en realidad se implementaron ya que no presenta todas las clases implicadas ni las propiedades ni métodos de las mostradas, pero sirve de referencia para explicar los principales aspectos de la implementación realizada, como se organizan las funcionalidades, y como se cumple con lo establecido por el RFC 793. Se podría interpretar con este diagrama que la cantidad de clases y por consiguiente el código realizado no fueron grandes, pero no fue así, ya que se invirtió mucho código en la clase TCB (transfer control block), encargada de mantener toda la información sobre cada conexión y ejecutar toda la lógica necesaria para el correcto de la conexión, y además se implementaron más clases que aquí no se detallan. Para más información de la implementación realizada consultar el Apéndice 10.4.

Leyendas de los colores para este diagrama y los subsiguientes que presentan clases:

- Las clases marcadas en gris claro son las ya provistas por la máquina virtual Java que no se modificaron.
- Las que están en verde claro son clases que ya se habían implementado en etapas

anteriores del desarrollo del stack y que no se modificaron.

- Las clases que están en gris oscuro son las clases que se tuvieron que modificar de la máquina virtual para poder hacer que nuestro stack fuera el utilizado.
- Las clases de verde oscuro son las clases que se modificaron de las que ya se habían implementado en el stack.
- Las clases celestes son las nuevas, que se implementaron de cero.

Principales Clases de la implementación

- *ServerSocket y Socket*

  Se modificaron estas clases para que la máquina virtual Java permita parametrizar la implementación de socket usada y así usar la clase IP4JVMSocket como implementación. Esto se debió realizar de esta manera ya que la máquina virtual no contaba con forma alguna de parametrizar que implementación utilizar. Como se puede ver en la imagen y en el Apéndice de instalación 10.2 la integración con OpenJDK se realiza de una manera mucho más simple que de la que se hacia con SableVM para integrar UDP, no es necesario modificar ningun archivo C++ ni demasiados makefiles (solo uno en el caso de OpenJDK).

- *SocketImpl*

  Clase abstracta que define las operaciones a ser implementadas en cualquier implementación de socket de la máquina virtual.

- *IP4JVMSocket, IP4JVMInputStreams y IP4JVMOutputStreams*

  Clases encargadas de manejar toda la comunicación con el thread (o hilo de ejecución) invocador de las operaciones del socket, y redirigir los pedidos adecuadamente a la conexión/es que corresponda. Estas clases son las encargadas de bloquear los threads que invoquen llamados a nuevas conexiones, receives, accepts y cierres hasta que los mismos se completen o aborten por algún motivo. Además son las clases encargadas de manejar las diferentes opciones que se pueden setear al socket y mapearlas adecuadamente a parámetros para las invocaciones de las funciones brindadas por la conexión.

  Una instancia IP4JVMSocket puede llegar a tener asociada otras instancias. Esto ocurre cuando se crea una instancia de ServerSocket y en cada accept se retorna un Socket, las implementaciones de los Socket resultantes de invocaciones a accept quedaran referenciadas por una lista contenida en la implementación del ServerSocket. Esto se realiza de esta manera para que el ServerSocket pueda seguir el rastro de todas las conexiones que ha creado pudiendo así limitar la cantidad de conexiones y además cerrarlas todas en el caso de que se invoque un close al ServerSocket.

  En la imagen que se muestra a continuación se presentan las clases involucradas en la gestión de los sockets:

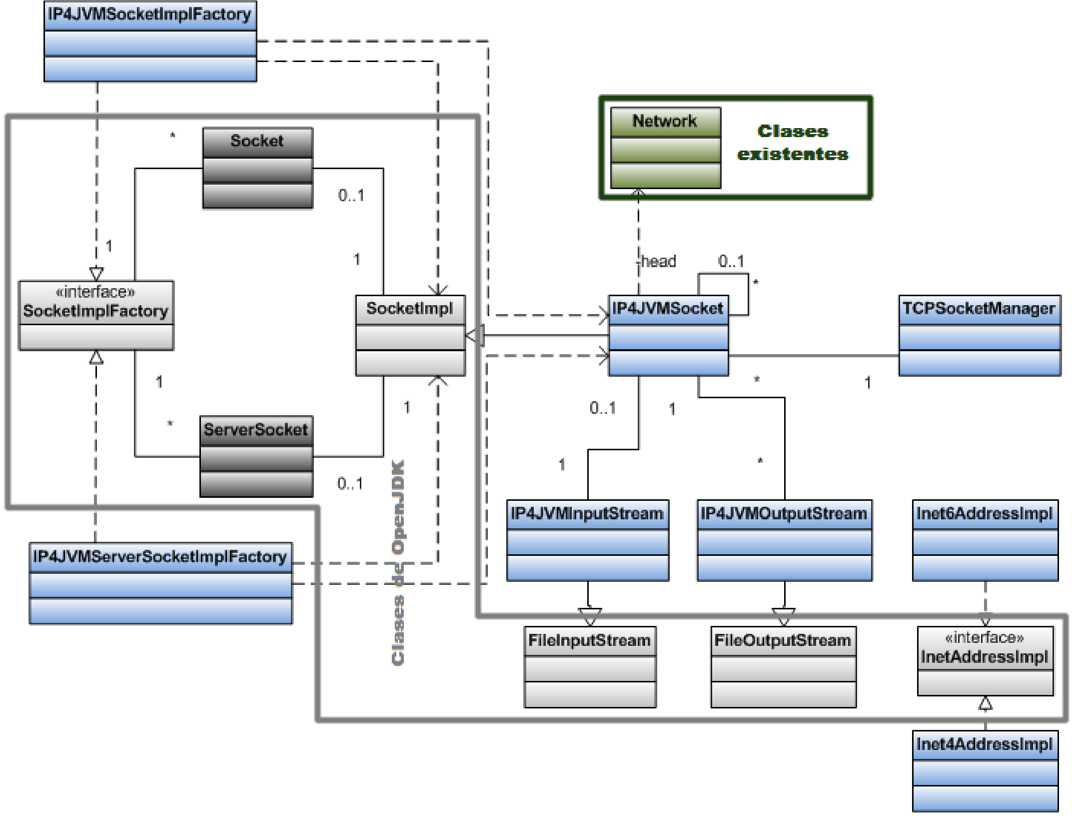


Ilustración 22: Clases para implementación de Sockets

- *Network*

  Se modificó esta clase para incluir la aplicación TCPSocketManager y el protocolo TCPProtocol.

- *TCPSocketManager y TCPProtocol*

  TCPSocketManager se encarga de saber al llegar un segmento a qué conexión corresponde el mismo, gestionar el uso de las conexiones, y pasar a las conexiones los pedidos realizados en los sockets. Además implementa las funciones que serán invocadas por los diferentes timers utilizados en la implementación.

  Las funciones aquí implementadas a simple vista parecería que debieran haber pertenecido a la clase TCPProtocol, pero al tener que ser esta última clase necesariamente un Protocolo (por imposición del modelo del framework) se tuvo que implementar la aplicación TCPSocketManager, quedando la clase TCPProtocol casi desprovista de funcionalidad, se utiliza casi exclusivamente como interfaz para las invocaciones que finalmente realizan al TCPProtocolHeader al que está asociado. Esto se implemento de esta manera pues la implementación de UDP ya era así y se deseaba ser coherente con la misma, además el modelo del framework requiere que un protocolo no sea el eslabón final de procesamiento de un segmento, por lo que

se requería una aplicación que realmente realizara el procesamiento final.

En la implementación realizada se utilizaron únicamente dos threads Java para implementar todos los timers de todas las conexiones. Uno de los timers lo llamamos slowtimo (que interrumpe cada medio segundo) y al otro fasttimo (que interrumpe cada 200 mili segundos). El fasttimo es utilizado únicamente para la realización de piggy backing, mientras que slowtimo es utilizado por todos los timers restantes de TCP. Para que se pudieran implementar todos los timers necesarios para la implementación, cada instancia de TCB tiene un array de enteros que contiene en cada posición la cantidad de ticks (o cantidad de "timeouts") de timer slowtimo que restan para que venza el timer definido por la posición del array. Cuando una posición del array tiene valor igual a cero, significa que el timer está desactivado.

La implementación fue realizada de esta manera pues tener un thread Java por cada timer de cada conexión requería estar creando dichos threads y cambiando el contexto a ellos, lo cual es costoso en términos de procesador y memoria. Esto puede llegar a no ser critico en un cliente, pero de seguro lo sea en un servidor que mantenga muchas conexiones concurrentes.

Para más detalles sobre la utilización de los timers se puede consultar el Apéndice 10.4.

En la siguiente figura se muestran las clases con las cuales interactúa la clase TCPSocketManager:

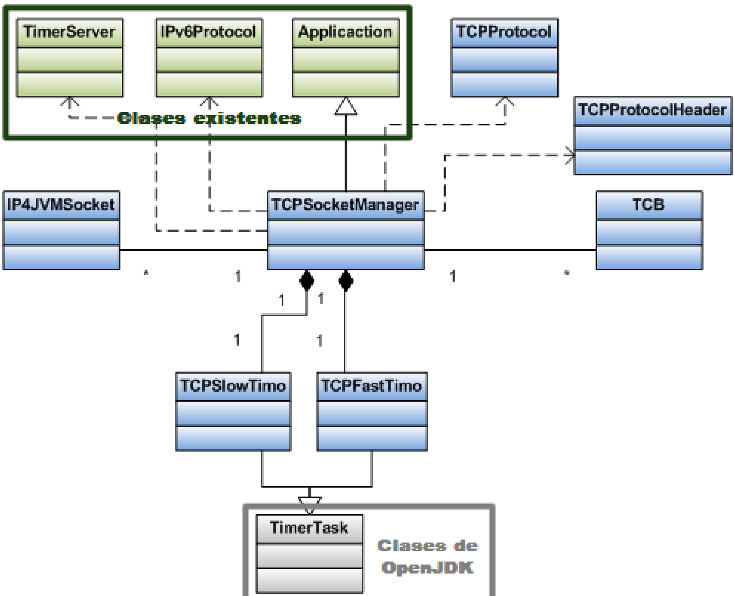


Ilustración 23: Clases auxiliares de TCPSocketManager

- *TCB*

  La clase TCB como ya se dijo contempla todos los aspectos vinculados a una conexión TCP, desde el open hasta el close de la misma. Es la encargada de enviar datos cuando así se requiera, partirlos en segmentos, re-ensamblar los mensajes entrantes y administrar los estados de la conexión según los eventos que se den lugar. Implementa la mayor parte de la lógica correspondiente a atender las solicitudes de la aplicación que esté utilizando el socket, la lógica correspondiente a atender el vencimiento de los timers y además la necesaria para atender los eventos provenientes de la red por medio de los segmentos entrantes.

  Para realizar todas las funcionalidades cuenta con clases de ayuda que en el diagrama presentado anteriormente no se muestran y que permiten simplificar los algoritmos necesarios. Aunque se implementaron estas clases auxiliares TCB resulta ser una clase con gran cantidad de código principalmente en las funciones de envío y recepción de segmentos y muy especialmente en esta última pues gestiona todos los eventos provenientes desde la red.

  En la siguiente imagen se muestran las clases auxiliares utilizadas por TCB:

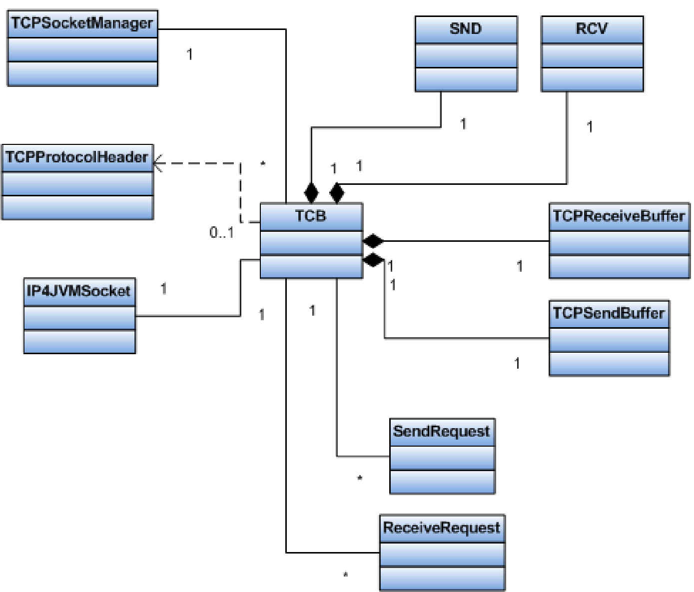


Ilustración 24: Clases Auxiliares de TCB

Se opto por administrar los buffers de envío y recepción en la clase TCB y no en la clase IP4JVMSocket para hacer a esta última clase la total responsable de gestionar el envío y la recepción de los datos y dejar a la implementación del Socket únicamente la responsabilidad de conocer a que clase enviar los pedidos, cómo enviar los mismos y gestionar la comunicación con la capa de aplicación que utiliza los sockets.

Esta clase mantiene una asociación con el socket correspondiente para notificar al mismo de eventos que ocurran (por ejemplo notificarlo que se ha establecido correctamente una conexión).

- *TCPProtocolHeader*

  La clase TCPProtocolHeader es la encargada de a partir de una tira de datos obtener toda la información del cabezal para luego ser procesada adecuadamente por la clase TCPSocketManager o la clase TCB. Además brinda la función que permite, a partir del mensaje a enviar y de los datos especificados para el cabezal, obtener la tira de bytes que se debe enviar a la capa de red. Esta clase con la ayuda de clases auxiliares parsea las opciones posibles del cabezal.

  A continuación se presenta una imagen donde se muestran las clases auxiliares utilizadas por la clase TCPProtocolHeader:

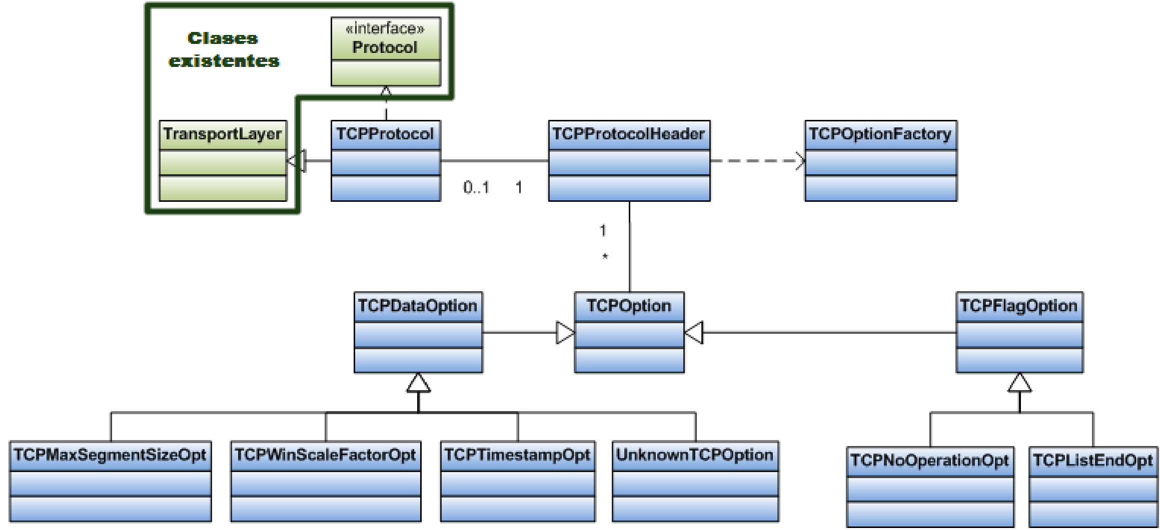


Ilustración 25: Clases Auxiliares TCPProtocolHeader

Se implementó la opción de cabezal TCP Win Scale Factor que permite tener ventanas de tamaños no representables con 16 bits. Este tipo de ventanas es requerido en conexiones de altos anchos de banda o altos tiempos de ida y vuelta. Adicionalmente se implemento una opción utilizada para omitir todas aquellas opciones desconocidas por el stack. Estas dos opciones conjuntamente con la opción de TimeStamp (la cual permite medir los tiempos de ida y vuelta de la conexión) son las tres opciones que se agregaron a las definidas en el RFC 793. En el Apéndice 10.4 se describe con mayor detalle la importancia y funcionalidad de estas opciones.

Procesamiento de diferentes eventos

1. Creación del socket

   Al crearse un socket se inicializan las estructuras y variables que se utilizaran posteriormente para cumplir con las funcionalidades requeridas según el tipo de socket creado. Estas inicializaciones se realizan todas en la clase IP4JVMSocket.

2. Bind

   Al realizarse un Bind la instancia de IP4JVMSocket verifica que no se haya hecho uno anteriormente, verifica las restricciones sobre los puertos y direcciones, y en caso de que no se haya asignado un puerto local pide a la instancia de TCPSocketManager que le de uno libre. Luego de este procesamiento de direcciones se pasan los datos del bind a la instancia de TCPSocketManager que actualiza la información almacenada sobre la conexión correspondiente y le pasa la nueva dirección y puertos locales a la instancia de TCB para que ésta actualice también su información.

   Para la asignación de puertos efímeros la instancia de TCPSocketManager mantiene un entero. Al pedir un nuevo puerto efímero se va incrementando recorriendo todos los puertos desde el último brindado hasta el 5000 (el máximo puerto utilizado para puertos efímeros), si no se encuentra ninguno libre se continua desde el 1025 (el primero no reservado) hasta encontrar uno que no se esté utilizando. En el caso de que todos los puertos se encuentren ocupados entonces se seguira iterando sobre los puertos buscando uno libre hasta que se encuentre uno. Esto se realizo de esta manera ya que nos parece que es muy poco probable que se ocupen todos los puertos.

3. Accept

   Cuando se invoca un accept la instancia de IP4JVMSocket verifica que el mismo puede ser invocado por el tipo de socket al que implementa la instancia. La instancia de IP4JVMSocket recibe en el llamado de accept como parámetro la implementación del socket (dada por la instancia de Socket) que debe retornar el accept original. IP4JVMSocket verifica en su cola de conexiones pendientes de accept si hay alguna, en caso de que no haya se bloquea al thread invocador de la función hasta que exista una. Al llegar un pedido de conexión desde la red, la instancia de TCPSocketManager crea una instancia TCB y otra de IP4JVMSocket temporal para la conexión. Al finalizar el proceso de establecimiento de la conexión, se encola el identificador de la conexión en la cola de conexiones pendientes de accept, notificando (en caso de que haya) al primer thread que se haya quedado bloqueado por una llamada a accept. Al desbloquearse el thread se le copian todos los datos desde el socket temporal creado para la conexión.

4. Connect

   Cuando se realiza un connect, IP4JVMSocket verifica el estado del socket y los parámetros dados. Luego notifica de que se debe realizar una nueva conexión a la instancia de TCPSocketManager que se encuentra asociada a la red utilizada y además se bloquea esperando a que se establezca la conexión. La instancia de

TCPSocketManager verifica que pueda ser posible crear una conexión en las direcciones dadas y posteriormente crea una nueva instancia de TCB manteniéndola en su colección de TCBs. Esta instancia de TCB inicia el proceso de saludo de 3 vías para iniciar la conexión. Al finalizar el proceso de saludo de 3 vías, o al ocurrir un error al tratar de establecer la conexión, se notifica al socket asociado y este se desbloquea, permitiendo al thread invocante de la operación connect continuar con su procesamiento.

5. Send

Al realizar un send la instancia de IP4JVMInputStream pasa el pedido a la instancia de IP4JVMSocket, esta última verifica que se puedan mandar datos dado el estado del socket y pasa el pedido a la instancia de TCPSocketManager que redirigirá finalmente el pedido a la instancia correspondiente de la clase TCB. Esta última agregará el send como uno pendiente a realizar y con el paso del tiempo se irán enviando los datos cuando haya espacio suficiente en la cola de retransmisión.

6. Receive

La instancia correspondiente de IP4JVMOutputStream al recibir un pedido de este tipo pasa el mismo a la instancia de la clase IP4JVMSocket, esta verifica que se pueda realizar dado el estado del socket para posteriormente pasar el pedido a la instancia de TCPSocketManager. Luego de esto, la instancia de IP4JVMSocket se bloquea esperando la finalización de la recepción. Por otro lado la instancia de TCPSocketManager pasa el pedido a la instancia de TCB que corresponda y esta última la almacena dentro de una cola de pedidos de recepción pendientes. A medida que vayan llegando los datos para el receive esta instancia irá poniéndolos en el buffer dado para llenar. Cuando el mismo se llene u ocurra algún error se notifica a la instancia de IP4JVMSocket la cual se desbloqueará retornando así el control al thread que haya invocado el receive.

7. Close

La instancia de IP4JVMSocket al recibir un close, verifica que el mismo pueda ser realizado dado el estado del socket, pasa el pedido a la instancia de TCPSocketManager, bloqueando posteriormente al proceso que invocó el close. La instancia de TCPSocketManager entonces pasa el pedido a la instancia de TCB adecuada y esta última inicia el proceso de cerrado de la conexión. Al finalizar el proceso de cerrado de la conexión, la instancia de TCB notifica a la instancia de IP4JVMSocket y esta última desbloquea el thread invocador del close.

8. Recepción y envío de segmentos

En los puntos anteriores ya se explican algunos casos de recibo de segmentos. Aquí solo detallaremos los puntos que no se han mencionado.

La clase encargada de recibir los Jobs de TCP es la instancia del protocolo TCPProtocol, la cual con ayuda de la instancia de TCPProtocolHeader parsea los datos para luego pasar el Job nuevamente al stack agregando únicamente la instancia de TCPProtocol como protocolo que ha procesado el paquete. Luego la

instancia de TCPSocketManager recibe el Job y sacando la información parseada por el TCPProtocol realiza un procesamiento previo de validación del segmento, verifica que en el nodo donde actúa exista una conexión con el puerto y dirección IP que se indican en el segmento y finalmente pasa el control a la instancia TCB que corresponda. La cual procesa el segmento actualizando la información necesaria.

La búsqueda del TCB se encuentra optimizada para el caso en que se use varias veces contiguas el mismo TCB. Se mantiene la referencia a la última instancia de TCB usada, si matchea con los datos obtenidos del segmento simplemente se usa esta. Este proceso es bastante usual por lo que este cambio aporta una optimización no menor.

En el caso de que no se encuentre un TCB que corresponda con el segmento recibido la instancia de TCPSocketManager se encarga de enviar un segmento con la bandera RST inicializada a uno al origen del segmento original.

En el caso del envió, como ya se mencionó, la clase encargada de esto es TCB. Esta clase crea el Job con ayuda de la clase TCPProtocolHeader. El Job creado es posteriormente recibido por la instancia de TCPProtocol la cual no la modifica en ningún sentido y la retorna al stack tal cual viene del mismo.

9. Timers

Como ya se mencionó anteriormente existen en la instancia TCPSocketManager 2 timers implementados. Al vencer alguno, TCPSocketManager recorre todas las instancias de TCB existentes notificando del vencimiento del timer adecuado. La instancia de TCB actualiza los valores de los contadores de cada uno de los timers TCP asociados al timer dado, en el caso de que alguno llegue a cero significa que el mismo ha vencido por lo cuál se debe ejecutar el procesamiento necesario.

En el Apéndice 10.4 se explica mejor como se realiza la gestión de los timers.

Aspectos adicionales de la implementación

La implementación realizada contempla el uso de piggy backing lo cual permite hacer un mejor uso de los recursos de red al postergar por un período corto el envío de un ACK en el caso de que no haya datos para enviar. Esto se hace con la esperanza de que en ese período aparezca algún dato para enviar en el mismo segmento, si así no sucede se envía el segmento conteniendo únicamente el ACK.

En la implementación realizada se cubrió gran parte de los algoritmos utilizados por Net/3 para obtener un mejor aprovechamiento de los recursos de la red. Net/3 implementa Slow Start, Congestion Avoidance, Fast Retransmit y Fast Recovery. En todos los casos en los que se probó se obtuvo el comportamiento deseado con respecto a estos algoritmos.

En la siguiente figura se muestra cómo debería ser la evolución de la ventana de congestión

cuando estos algoritmos se encuentran implementados:

Ilustración 26: Evolución de Ventana de Congestión

Slow Start: Cuando la conexión se encuentra en Slow Start para cada ACK de segmento que llegue se incrementa la ventana de congestión en mss (maximum segment size de la conexión). Esto produce un crecimiento exponencial de la carga impuesta a la red mientras se corra este algoritmo. El criterio de parada de este algoritmo es básicamente la detección de la perdida de algún paquete en la red lo que puede llegar a ser un indicador de que la red se encuentra congestionada. Otro criterio es el pasaje de la marca de threshold que se usa para estimar cuanto es la capacidad de la red. Luego de pasar el threshold se realiza un crecimiento lineal de la ventana de congestión. Por el contrario si se detecta que se ha perdido algún paquete se setea la ventana de congestión igual a la mitad del valor actual (a este proceso se le denomina exponential backoff) y el threshold también es seteado a este valor.

Fast Retransmit: al llegar 3 ACK consecutivos repetidos TCP induce que se han perdido segmentos y es por esto que se están repitiendo los ACK por lo cual retransmite nuevamente todo desde el último segmento que se ha recibido ACK hasta el último segmento que se ha enviado.

Fast Recovery: luego de realizar Fast Retransmit se hace Congestion Avoidance pero no se hace Slow Start sino que se realiza un crecimiento lineal de la ventana de congestión. El tamaño de la ventana al reducirse lo hace a la mitad del tamaño actual más la cantidad de

bytes que el otro extremo de la conexión ha confirmado como recibidos mediante el envío de ACK.

A pesar que el stack no presenta el comportamiento esperado tomando en cuenta los tiempos, si los presenta tomando en cuenta otros factores determinantes de la evolución de la ventana, como lo es la evaluación de la ventana luego de la llegada de un ACK. En el Apéndice 10.4 se detalla y explica el comportamiento real del stack.

Se implemento el algoritmo de Nagle para evitar el síndrome de la ventana tonta. Este algoritmo utiliza un timer y una bandera. Cuando la ventana de recepción del otro extremo de la conexión es 0 envía de a un solo byte cada vez que vence el timer del algoritmo. El timer de retransmisión es deshabilitado durante este proceso. En la siguiente figura se muestra el problema del síndrome de la ventana tonta:

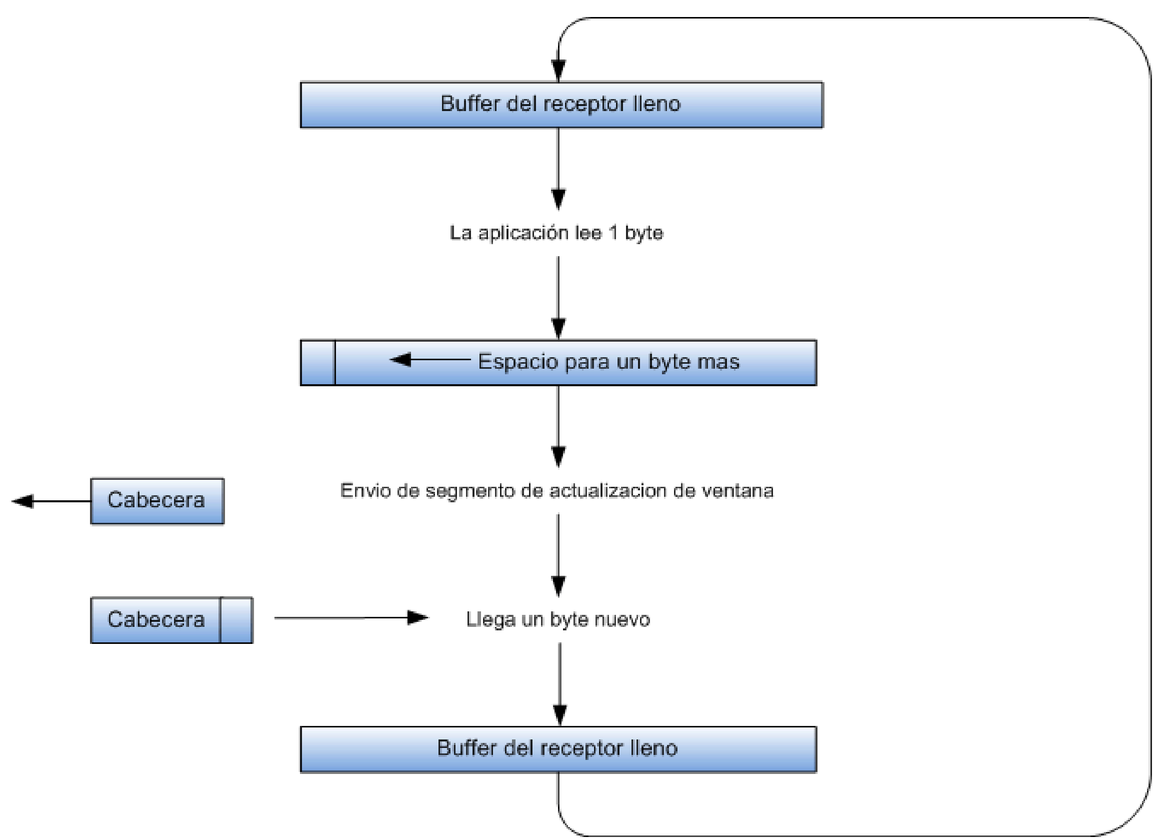


Ilustración 27: Síndrome de la ventana tonta

Para calcular el tiempo que debe ser utilizado en el timer de retransimision en la implementación se hace uso de la opción Timestamp. Para calcular el tiempo de retransmisión a partir de la última medida de tiempos de ida y vuelta de paquetes se aplican las siguientes formulas:

delta = nticks - srtt

srtt = srtt + g*delta
rttvar = rttvar + h*(|delta| - rttvar)
RTO = srtt + 4*rttvar

- delta
  Diferencia entre el tiempo de ida y vuelta medido, menos el actual valor del estimador suavizado de RTT (round trip time, tiempo de ida y vuelta).
- nticks
  Este valor es calculado a partir del tiempo de ida y vuelta obtenido gracias a medidas realizadas utilizando la opción de TCP Time Stamp mencionada en secciones anteriores.
- srtt
  Estimador suavizado de RTT
- g
  Ganancia aplicada al estimador RTT (se inicializa con el valor 1/8).
- h
  Ganancia aplicada al estimador de desviación media (inicializado en 1/4).
- rttvar
  Estimador de varianza promedio. Es una buena aproximación a la varianza estándar y además es más fácil de calcular ya que no requiere realizar operaciones de raíces cuadradas.

Cabe destacar que además al valor calculado se le aplica un exponential backoff que se encuentra detallado en el Apéndice 10.4.

Para una explicación detallada de esta fórmula se pueden consultar los documentos referenciados por [9] y [49].

Para la implementación del calculo de ISS (initial send sequence, secuencia inicial de envio) se utilizó un contador de timer que se inicia al iniciar el procesamiento de TCPSocketManager. Esta variable se incrementa en un valor de 64000 cada ½ segundo y además cuando se establece una conexión. Este comportamiento, junto con el tiempo de espera luego de la muerte de la conexión, combate el problema de que se confundan segmentos de conexiones ya inexistentes o cerradas con los de nuevas conexiones.

5.6 Ejecución de Tomcat en IP4JVM

Como se dijo con anterioridad, desde un inicio se planeo ejecutar sobre el stack implementado una aplicación que hiciera uso de la implementación TCP realizada, para probar exhaustivamente la implementación y así confirmar de forma empírica su correctitud. Para esto, primeramente se realizaron aplicaciones Java para envío y recepción de archivos utilizando conexiones TCP. Esto se realizó en paralelo con las etapas finales del desarrollo de TCP. Finalmente se ejecutó Apache Tomcat utilizando el stack, esta aplicación es muy popular por lo que nos pareció un buen ejemplo de aplicación que debería funcionar con nuestro stack para dar a este último el visto bueno. Para ésto fue necesario estudiar como configurar adecuadamente los parámetros ofrecidos por el servidor para que utilizara el stack adecuadamente, dentro de éstos podemos encontrar los puertos, direcciones IPv6, adaptadores utilizados, etc. Finalmente, compilando la máquina virtual y configurando el stack adecuadamente se consiguió ejecutar el servidor Apache Tomcat de modo tal que utilizase el stack implementado en IP4JVM. Los test realizados fueron hechos sobre las páginas de administración del servidor Tomcat.

Gracias a estas pruebas se encontraron algunas carencias del stack que debieron corregirse. Un ejemplo fue la función push, que en un inicio contenía errores en su implementación, pero que debido a que las pruebas realizadas hasta el momento con conexiones TCP entre dos nodos para envío y recepción de archivos no hacían uso de la misma, no habíamos dado nota del error. Posteriormente, al ejecutar Tomcat, siendo que éste hace uso extensivo de la misma, se debió implementar correctamente. Otro caso particular que surgió fue que al realizar el shutdown del servidor, Tomcat no cierra los sockets, por lo cuál cuando intentábamos bajar el servidor algunos procesos no se daban por finalizados. Debimos revisar el código del propio Tomcat para darnos cuenta que dicho aplicativo en vez de cerrar los sockets se quedaba esperando el vencimiento de tiempo estipulado para el *accept*, cosa que nosotros en nuestra primera implementación no habíamos tomado en cuenta.

Cabe recalcar que para ejecutar Tomcat utilizando el stack basta con configurar en el archivo de configuración del Tomcat los puertos y las direcciones IPs a ser utilizadas por los diferentes conectores y las mismas direcciones deben estar configuradas en IP4JVM. Ademas es necesario configurar algunas variables de ambiente para establecer IP4JVM como el stack por defecto a utilizar por aplicaciones Java. El proceso de configuración necesario para correr IP4JVM sobre Apache Tomcat es detallado en el Apéndice 10.5.

5.7 Plan de pruebas ejecutado

Las pruebas realizadas en las diferentes etapas del desarrollo proyecto se fueron creando y adecuando a los crecientes requerimientos de la herramienta a medida que esta crecía o se iban corrigiendo problemas existentes. Las pruebas realizadas para las diferentes etapas fueron:

- *Modificación para mejorar performance*

  Para esto se ejecutaron las pruebas existentes que permiten realizar ping entre la herramienta y una implementación Linux de IPv6 que se encuentre en la misma red LAN y configurando previamente a la ejecución la dirección IPv6 e interfaz utilizada por la herramienta. La otra aplicación (o prueba) utilizada fue la que permitia enviar un paquete UDP desde la herramienta a un stack IPv6 en las mismas condiciones descritas para el caso de ping. A la implementación IPv6 utilizada para probar la herramienta denominaremos nodo de referencia.

- *Agregado dinámico de interfaces y direcciones IP*

  En este caso se creo una aplicación Java que permitió integrar las aplicaciones que se contaba con anterioridad para probar la herramienta además de permitir agregar nuevas funcionalidades para probar nuevas prestaciones dadas por la herramienta. Con esta aplicación se probó creando interfaces dinamicamente y configurando direcciones IP a la misma. Esto también permitió probar el correcto funcionamiento de la herramienta utilizando más de una interfaz y más de una IP. En esta etapa se pudo probar también el correcto funcionamiento de la herramienta con un nodo de referencia corriendo Windows XP.

- *Stateless Address Autoconfiguration*

  Para probar el correcto funcionamiento de la auto configuración de direcciones IPv6 cuando un router IPv6 se encuentra en la LAN se modifico la aplicación utilizada para probar antes mencionada agregándole nuevas funcionalidades. El entorno utilizado se propago hasta el final del proyecto, estando conformado este por un router IPv6 con 2 interfaces de red en diferentes redes. La herramienta corre en una de estas redes mientras que la implementación IPv6 utilizada para probar se encuentra corriendo en la otra red. Mediante el uso de la dirección auto generada para pinguear desde el nodo de referencia (implementación utilizada para probar el la herramienta) se comprobó el correcto funcionamiento del protocolo de auto configuración de direcciones. Otras pruebas realizadas constaron de apagar el router para ver si las direcciones auto generadas caducaban. Los nodos de referencia utilizados en esta etapa fueron los mismos que los utilizados en la etapa anterior además de nodos con sistemas operativos Knoppix y OpenSuse. La aplicación de ruteo utilizada en el router es radvd.

- *Ruteo de paquetes*

  Se modificó la aplicación utilizada para realizar las pruebas para probar el funcionamiento de la herramienta configurando en esta diferentes rutas para las diferentes interfaces. Para esto se brindan en la aplicación comandos para agregar y quitar rutas de la misma manera que se realiza en sistemas Linux, luego de esto se pueden realizar pings y probar que los paquetes enviados se rutean adecuadamente por la interfaz configurada para cada caso.

- *Integración con máquinas virtuales Java*

  Como se mencionó anteriormente en este documento se probó la integración de la herramienta primero con SableVM en un sistema operativo Linux Fedora. Posteriormente se probó la integración del stack con la máquina virtual OpenJDK en el mismo sistema operativo. Como último paso se realizó la integración de la herramienta con OpenJDK en el sistema operativo OpenSuse, lo que permitió probar el correcto funcionamiento de la herramienta en dicho entorno.

- *Pruebas básicas de TCP*

  Al inicio de la implementación de TCP se creó una aplicación cliente que recibía mensajes de texto de un servidor (corriendo en el nodo de referencia) y los mostraba en pantalla. Luego de esto se probó el correcto funcionamiento de la aplicación servidor de mensajes en la herramienta corriendo el cliente en el nodo de referencia. El próximo paso dado fue la modificación de la aplicación que realizaba las pruebas de las otras etapas para incluir un cliente que requiriera archivos de un servidor de archivos situado en el nodo de referencia. A continuación, se incluyó también el servidor de archivos en la aplicación de pruebas utilizando el cliente de archivos desde el nodo de referencia.

- *Pruebas de Path MTU*

  Luego de haberse probado correctamente TCP en los casos anteriores se probo también el funcionamiento del protocolo de Path MTU, pues el mismo requería del envio de mensajes de un cierto tamaño para probar su correcto funcionamiento. Para probar Path MTU se utilizaba el servidor de archivos en el nodo de referencia y el cliente en la herramienta, luego de transferido un archivo se modificaba la propiedad de MTU en una de las interfaces utilizadas del router, a continuación se requeria nuevamente un archivo desde la herramienta para comprobar el correcto manejo del mensaje de ICMP Packet Too Big por parte de la herramienta.

- *Prueba con Tomcat*

  Como primer paso hacia la ejecución del servidor Tomcat conjuntamente con la herramienta se integró a la aplicación de prueba un servidor HTTP básico. Se realizaron consultas HTTP de archivos desde el nodo de referencia utilizando el entorno antes mencionado (con router) y el browser Konqueror incluido en los nodos de referencia. Con esta funcionalidad adicional se pudieron detectar errores varios en la herramienta que permitió que las pruebas posteriores con Tomcat fueran menores. Luego de esto se realizaron las pruebas utilizando directamente el

servidor Tomcat corriendo con la herramienta, realizando consultas desde un browser en el nodo de referencia.

- *Pruebas de Congestion Avoidance*

  Como último paso de las pruebas realizadas utilizando el web server y el file server integrados en la aplicación de pruebas se comprobó el correcto funcionamiento de los algoritmos de congestion avoidance de TCP. Para esto se agregó código a la herramienta para generar archivos de log para evaluar el comportamiento de la herramienta.

6. Resumen y Perspectivas

En la presente sección se presentará un resumen del proyecto en cada una de sus fases, junto a las conclusiones generales que se fueron obteniendo a lo largo del mismo. También en esta sección se encontrará una enumeración de trabajo futuro, que no fue abordado por el alcance de este proyecto pero que sin embargo creemos que podría ser bueno agregar para sumarle valor al producto obtenido en esta segunda iteración de IP4JVM.

6.1 Planificación Inicial

El desarrollo del proyecto en sí fue muy dispar en relación a lo estimado en la planificación inicial. Si bien los objetivos trazados fueron alcanzados, y hasta con un alto grado de satisfacción, los tiempos consumidos por las distintas actividades tuvieron una desviación importante en perspectiva con lo estipulado al comienzo del proyecto.

A continuación se muestra un diagrama de Gantt que procura ilustrar las tareas llevadas a cabo durante el proyecto junto a los tiempos que cada una de éstas duraron.

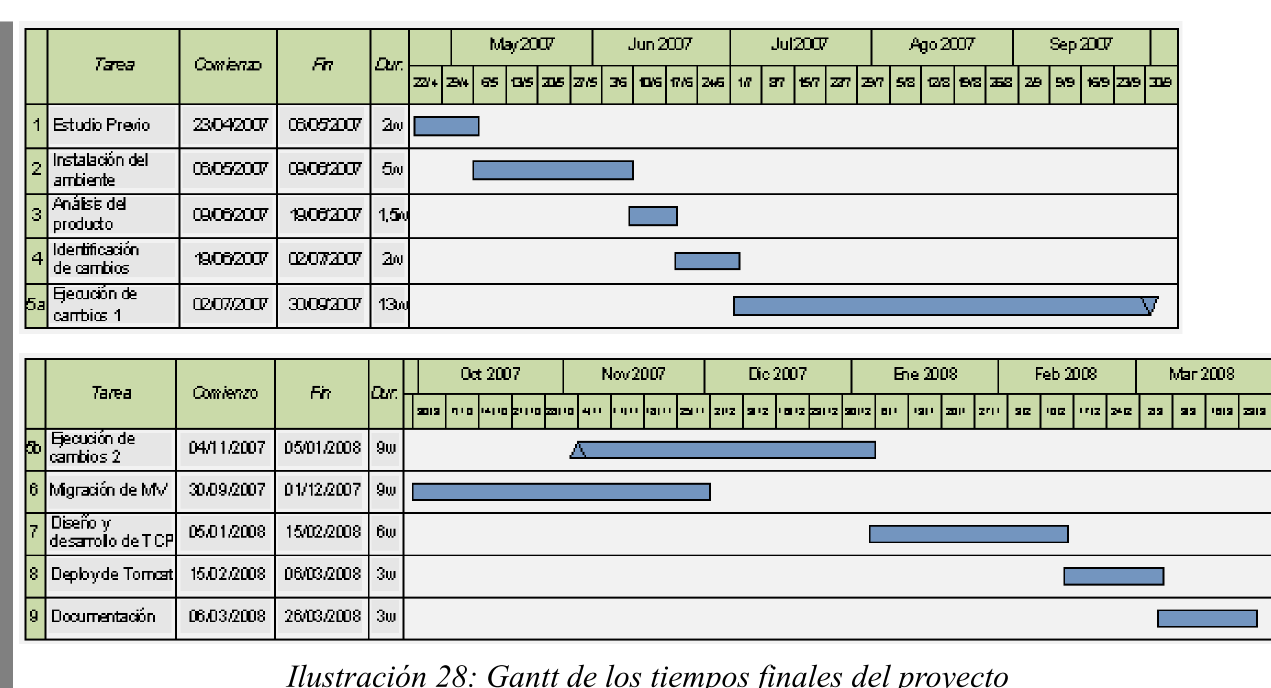


Ilustración 28: Gantt de los tiempos finales del proyecto

El primer aspecto que debe mencionarse es que la tarea de realización de cambios se dividió en dos etapas. La primera fue realizar todos los cambios de alto impacto, una vez obtenida una

versión bastante más performante del sistema se comenzó la migración hacia otras máquinas virtuales de la herramienta. Sin embargo, luego de cinco semanas de investigación de la compilación de las dos herramientas ya mencionadas (Harmony y OpenJDK) se vio que los avances sobre OpenJDK eran realmente alentadores, por ende, dado a que nuestro objetivo principal era la integración con esta máquina virtual (Harmony se investigó como modo de mitigar un posible fracaso en cuanto a la integración con OpenJDK), y dado que aún no había una versión liberada estable por parte de Harmony, ni mucha documentación o soporte que pudiesen ser de ayuda en cuanto a problemas de compilación surgidos, se decidió dividir los esfuerzos del grupo. Así pues, se decidió paralelizar el trabajo y las tareas de continuar la investigación de la integración con OpenJDK y la de finalizar los cambios que aún restaban efectuar se realizaron en simultaneo por los distintos integrantes del proyecto. Es por esto que vemos un solapamiento durante un período de cuatro semanas en cuanto a estas dos tareas.

A continuación se mostrará una tabla que indica, para cada etapa, los tiempos estimados y reales, para así poder analizar con mayor detenimiento las desviaciones mencionadas.

Etapa	Tiempo Estimado	Tiempo Insumido
Estudio Previo	2 semanas	2 semanas
Instalación de ambiente	2 semanas	5 semanas
Análisis del producto	1 semana	1,5 semanas
Identificación de Cambios	2 semanas	2 semanas
Realización de Cambios	14 semanas	20 semanas
Migración de MV	4 semanas	7 semanas
Diseño y Desarrollo de TCP	8 semanas	6 semanas
Ejec. Tomcat sobre IP4JVM	1 semanas	3 semanas
Documentación	2 semanas	3 semanas
Total	**36 semanas**	**49,5 semanas**

Tabla 1: Comparación de tiempos estimados y consumidos por etapa

Como vemos en la tabla anterior, la etapa inicial transcurrió según lo planeado. Sin embargo, la segunda etapa introduce tres semanas de desviación. Se llevaron cinco semanas para la instalación del ambiente frente a las dos planificadas. Este error de estimación se debió al profundo desconocimiento que teníamos en cuanto al esfuerzo necesario para compilar una máquina virtual. La falta de documentación sobre SableVM por ser éste un producto poco usado contribuyó a la dificultad presentada por la herramienta, ya que tuvimos varios problemas durante su compilación los cuáles no pudimos solucionar con tan sólo leer la documentación generada por el proyecto anterior. Por eso se precisó más tiempo y mucha ayuda por parte de nuestro tutor para superar el primer escollo encontrado en el proyecto.

El análisis del código supuso también un 50% más del tiempo estimado (1 semana y media contra la semana estimada), esto se debió también a que el funcionamiento del framework

codificado por Laura no resulta trivial a ojos de alguien que no está familiarizado con la herramienta.

La identificación de cambios se pudo realizar en las dos semanas previstas e incluso se generó un documento exhaustivo en donde se detallaban minuciosamente los cambios que nos parecían importantes realizar.

También se observa que la realización de cambios tomó seis semanas más de lo previsto. Hace falta aclarar que si bien en el diagrama de Gantt mostrado con anterioridad, la suma de las tareas referentes a los cambios totalizan 22 semanas, hay que tener en cuenta que 4 de esas semanas se realizaron por parte de un sólo integrante del grupo, y siendo que todos los tiempos expresados se basan en el trabajo de ambos integrantes, podemos decir que esas 4 semanas se traducen a 2 tomando en cuenta esto último.

Las 6 semanas de más que llevó la tarea de realización de cambios es entendible si se aprecia que al momento de la planificación inicial del proyecto no se había leído el código, no se conocía su estado más allá de lo comentado por nuestro tutor y, sobre todo, no se sabía la cantidad de cambios que deberían realizarse ni la complejidad e impacto de los mismos.

Otros de los errores de estimación importantes fue el de la etapa de migración de máquina virtual. Para esta etapa vale la misma apreciación que para la anterior, ya que durante 4 semanas contó con la dedicación de uno sólo de los integrantes del grupo, por lo cuál la dedicación no es de 7 semanas y no de 9 como parecería al ver el diagrama apresuradamente.

Una vez más debemos adjudicar este error a la nula experiencia que teníamos en cuanto a compilación de máquinas virtuales refiere. El estudio de las herramientas insumió demasiado tiempo, y si bien el soporte fue mayor que con SableVM, las respuestas a las preguntas efectuadas al mail list no tenían siempre una inmediata respuesta, pudiendo a veces demorarse unos días.

Además se puede ver que el tiempo que llevó el diseñar e implementar el protocolo TCP en nuestro stack fue menor al estimado. Esto se debió en parte a la motivación del grupo por ser la etapa final de desarrollo del proyecto y por tratarse de un diseño y desarrollo propio. Además otro factor que afectó positivamente, fue el hecho de que en esta altura del proyecto el funcionamiento de la herramienta ya era bastamente conocido, por lo cuál el integrar un protocolo más, fue casi natural.

En la etapa de ejecución de aplicaciones que usasen TCP (como ser Tomcat) sobre nuestro stack y la depuración del mismo también se requirió más tiempo, en este caso dos semanas más. Fundamentalmente por funcionalidades de TCP que eran esenciales para el aplicativo y que se encontraban mal implementadas en nuestro código, lo que requirió correcciones y cambios múltiples (como la función PUSH).

Por último la etapa de documentación también llevo mas tiempo de lo planeado inicialmente. Dentro de los aspectos que llevaron a la prolongación de esta tarea se encuentra la necesidad que tuvimos de generar documentación sobre la implementación de TCP una vez finalizado el

proyecto, ya que no fuimos documentando esta etapa a medida que la ejecutábamos y esto resultó ser un error estratégico. Otros documentos que si se habían ido realizando requerían mayor afinamiento y correcciones, además de que algunos habían resultado muy extensos y por ende difíciles de leer, por lo cuál procuramos resumirlos de la mejor manera para su mejor legibilidad y sin perder la información útil que ellos contenían.

6.2 Performance y Diseño

Si bien observamos una notoria mejoría en cuanto a performance y diseño con respecto a la versión inicial de la herramienta, aún hay cambios que podrían mejorar más dichos aspectos, como por ejemplo, se podría probar mantener un pool de threads que atendiesen múltiples jobs en paralelo lo cual permitiría que el stack siguiese atendiendo pedidos mientra procesa otros que hubiesen llegado con anterioridad.

El diseño no se modificó mucho debido a que no se quería reimplementar la herramienta de cero y los tiempos no eran los suficientes, por lo cuál estuvimos bastante atados a lo que era el diseño anterior, apenas haciendo aquellos cambios que juzgamos imprescindibles. Sin embargo tampoco parece necesario hacer un re diseño completo, ya que si bien es perfectible, el diseño actual no demuestra grandes carencias que limiten futuras evoluciones de la herramienta.

La performance de la herramienta es sustancialmente mejor a la anterior, y se ha conseguido un uso muy razonable del procesador, así como un estable uso de memoria RAM.

Cabe la posibilidad que en una segunda iteración se pudiesen hallar nuevos cambios, ya que tenemos un conocimiento más profundo de la herramienta y del objetivo de las funciones, estando así más capacitados para poder detectar un algoritmo perfectible. Así mismo, pruebas más exhaustivas y en redes más pobladas podrían llegar a mostrar cualidades no deseadas en el producto que llevarían a cambios en el mismo.

Es por esto que en un futuro trabajo deberían hacerse las pruebas mencionadas y a partir de ellas estudiar la viabilidad de cambios de diseño en función de el esfuerzo requerido y las ventajas obtenidas. En paralelo deberían analizarse nuevamente los algoritmos para detectar posibles mejoras en los mismos.

También deberá completarse los RFCs que aún están incompletos y aprovecharse del uso de OpenJDK para usar las bondades que Java incorporó a partir de su versión 1.5. Esto último no se ha hecho para no perder compatibilidad con SableVM, la cuál usa Java 1.4. Pero tal vez el precio a pagar por dicha compatibilidad sea alto al considerar las ventajas de tener la herramienta integrada con una máquina virtual poco usada como SableVM. Si se decidiese por usar las nuevas posibilidades de Java, podrían usarse colecciones tipadas, haciendo del código más fácilmente legible además de poder prescindir del uso de *casting*, el cuál si no se usa con responsabilidad puede introducir errores que no son fáciles de detectar.

6.3 Portabilidad

Como se detalló, la herramienta se encuentra integrada y probada tanto con SableVM como con OpenJDK compiladas tanto en Linux como en Windows.

Sería de interés continuar el trabajo de manera de poder integrar la herramienta con otras máquinas virtuales, como ser la propia Harmony. A su vez sería interesante poder probar la herramienta en otros sistemas operativos para poder comprobar su independencia del mismo.

Arquitecturas de 64bits sería otra prueba interesante, ya que todas las pruebas realizadas fueron sobre arquitecturas de 32bits.

6.4 Completitud / Correctitud

A pesar de actualmente contar con casi la totalidad de las implementaciones de los RFCs sobre ICMPv6, Stateless Address Autoconfiguration, Neighbor Discovery, IPv6, etc. hay puntos que aun faltan por implementar de los RFCs.

Uno de los principales aspectos que no se han implementado de los RFCs es el soporte que algunos de los mismos requieren para protocolos utilizados por routers. La implementación actual sólo soporta el uso del stack para nodos de tipo host, por lo que sería un paso a dar a futuro la implementación completa de estos RFCs para el caso de ejecución del stack para routers.

Otros puntos faltantes en las implementaciones de los RFCs son brindar la posibilidad al usuario de configurar determinadas variables de los protocolos. Este es un punto pendiente pero no muy complejo e importante, y como no se utilizaba en las pruebas necesarias sobre los protocolos para el alcance del proyecto no se realizaron para mantener la implementación mas simple.

También debería estudiarse la manera de obtener el *link mtu* mediante el uso de la librería Pcap para tener una implementación más portable de las funcionalidades que utilizan este atributo. Actualmente el mismo debe ser especificado por el usuario al momento de crear una interfaz. En caso de que no se especifique se toma por defecto el valor por defecto para redes Ethernets (1500).

Las partes faltantes del protocolo TCP tienen que ver con cosas que Java no permite, como es el tratamiento de las estadísticas que permitan brindar mejores funcionalidades de de la red (por ejemplo, la clase Socket no ofrece la funcionalidad de ver paquetes enviados, perdidos, etc.) o el correcto tratamiento de datos urgentes, por lo cuál no se implementó nada al respecto.

Otro aspecto que debe analizarse como parte de un plan a futuro, es la exhaustiva prueba de

UDP, ya que este protocolo ya estaba implementado en la herramienta y en nuestro proyecto no lo probamos ni usamos.

Como punto no tan importante pero que es bueno indicar, seria deseable estandarizar el formato de los archivos de configuración del stack, utilizando algún estándar como ser XML.

Otro punto a destacar es que los paquetes que son dirigidos a direcciones IP locales que no pertenecen a interfaces loopback no son tratados como paquetes que deben hacer loopback. Por lo tanto si se envía mediante nuestro stack un paquete a una dirección que pertenece a una interfaz loopback el stack lo trata como tal y lo procesa correctamente, sin embargo si la dirección destino del paquete es una dirección local pero que no pertenece a una interfaz loopback (o sea envío el paquete a una de mis propias direcciones locales asignada a una interfaz de tipo Eth) el stack de IP4JVM lo tratará como un paquete de salida y no le dará el procesamiento requerido para u paquete de entrada. Esto no es un requerimiento especificado en ningún RFC, por lo cual no se puede considerar como error, pero vale la pena indicarlo pues es una funcionalidad bastante utilizada en las implementaciones de IPv6 que si tratan los paquetes enviados a direcciones locales como paquetes loopback. Para implementar esto se debería agregar la lógica necesaria para identificar que la IP de destino es local, en cuyo caso se debería tratar al paquete como de loopback. Esta funcionalidad no se implementó pues agregaría un esfuerzo adicional y no es requerida por los RFCs.

Otro punto importante es que la implementación no se ha probado la herramienta con casos de test estándar que permitan aseverar con mayor certeza y propiedad que el stack se encuentra razonablemente completo, sino que los tests llevados a cabo fueron creados por nosotros mismos a partir de los RFCs involucrados. Se podría probar para esto ejecutar los tests especificados por IPv6 Ready Logo [65] el cuál contiene una especificación para probar implementaciones de los protocolos base de IPv6 [66].

7. Conclusiones del Trabajo

Se cumplieron con los principales objetivos del proyecto, dentro de los cuales se distinguían principalmente poder ejecutar Tomcat utilizando IP4JVM, para lo cual era necesario implementar TCP y cualquier otro requerimiento que esto conllevara, mejorar la performance de la herramienta, y adicionalmente realizar la integración de la herramienta con la máquina virtual OpenJDK. Como se ha explicado a lo largo de este informe se ha cumplido de buena manera con estos requerimientos pero insumiendo mucho más tiempo que el esperado al inicio del proyecto. Además de cumplir con los requerimientos, y en la mayor parte de los casos para cumplir con estos, se realizaron desarrollos adicionales como mejorar el diseño de la herramienta, completar implementaciones existentes (Stateless Address Autoconfiguration, IPv6 Protocol, Neighbor Discovery, etc) e implementar algunos que se creían en un inicio implementados (al menos parcialmente) pero no se encontraban así (Path MTU, utilización de varias interfaces, ruteo de paquetes, etc).

Se puede ver que la mayor parte del tiempo insumido se debió a los cambios iniciales, los cuales nos ayudaron a interiorizarnos con la herramienta pero que nos llevaron al mayor defasaje de la planificación. Todas aquellas tareas planificadas que requerían conocimiento de la herramienta anterior tuvieron retrazos en su desarrollo frente a lo planificado, mientras que aquellas tareas que resultaron tener mayor independencia con el código previo, como ser por ejemplo la implementación completa de TCP, se ajustaron perfectamente a los tiempos inicialmente estipulados. Esto demuestra que el Framework IP4JVM tiene mayor complejidad en su código que lo pensado al principio, y que esto no debe ser despreciado en futuros trabajos sobre la herramienta por parte de programadores que aún no han tenido contacto con la misma, como apunte para próximos desarrollos en la herramienta, entonces, podemos decir que se requiere mucho tiempo de investigación para conocer la herramienta para posteriormente realizar modificaciones en la misma. Adicionalmente se recomienda para futuros desarrollos evitar modificar el framework a no ser que el cambio sea requerido, y en caso de que esto sea necesario, evaluar el impacto del mismo. En el presente proyecto se realizaron varios cambios en el framework pues se consideraba que los mismos eran realmente necesarios para agregar claridad, extensibilidad y/o aplicabilidad, pero agregar complejidad al framework resultará en un mayor tiempo requerido en investigación para nuevos desarrollos, lo cual es uno de los factores menos deseados para una herramienta que pretende ser open source y que requiere por consiguiente que si alguien quiere modificar algo en el código no se necesite demasiado tiempo de introducción.

Otro punto a destacar para futuros desarrollos es que en algunos casos las implementaciones cuentan con carencias, pues a los efectos de los alcances de los proyectos llevados a cabo (incluido este) dichas funcionalidades no eran requeridas e implementarlas llevarían a mayor inversión de tiempo. Un ejemplo de este tipo de carencias es la falta de las capacidades necesarias para correr routers o implementar funcionalidades de los mismos en la herramienta.

Como punto final se destaca que el desarrollo realizado, y en consecuencia el progreso obtenido en la herramienta en el presente proyecto ha sido grande, a modo de ejemplo, en la versión previa sólo se podía realizar ping entre IP4JVM y otra implementación que se

encontrara en la misma LAN, mientras que en la versión actual se puede realizar transferencias de archivos y correr servidores web en la herramienta pudiendo la misma interactuar con implementaciones en otras redes. A pesar de esto queda mucho desarrollo por realizar para completar el stack (implementación de DNS, DHCP, IPsec, MIPv6, etc.). El desarrollo futuro de la herramienta nos parece de gran utilidad ya que tener una herramienta open source de un stack IPv6 brindará a la comunidad de un entorno para, dentro de otras cosas, la prueba de nuevas implementaciones de algoritmos, variantes de los existentes, un ejemplo de implementación del stack para aquellos desarrolladores que tengan curiosidad o requieran de los conocimientos de cómo se pueden implementar diferentes aspectos del stack. Es por esto que pensamos que realmente vale la pena invertir en el desarrollo del mismo. Además podemos agregar que en las dos iteraciones que ha sufrido la herramienta, esta ha adquirido un diseño que permite su crecimiento y que no la limita en absoluto en este sentido. En la presente iteración se mejoraron muchas de sus cualidades que permitieron obtener, no sólo una mejor performance, sino también aumentar la legibilidad del código, la portabilidad y escalabilidad del diseño, así como actualizar la versión Java que soporta mediante la integración con una nueva máquina virtual (OpenJDK). Todo este trabajo realizado durante nuestro proyecto, permite expandir el horizonte de la herramienta y asegurarnos la posibilidad de futuro crecimiento de IP4JVM como proyecto open source.

8. Glosario

ACK (ackowledgement): confirmación, acuse de recibo.

Array o arreglo: estructura utilizada generalmente en programación que permite hacer uso de un conjunto ordenado de datos.

Background: trasfondo, soporte, contenido que haga referencia a un punto específico.

Buffer: espacio finito de memoria donde se almacenan datos.

Busy Waiting: problema de programación en donde en un loop se realizan iteraciones innecesarias a causa del no uso de funcionalidades de sincronización como pueden ser semáforos o monitores.

C++: lenguaje de programación que extiende el popular lenguaje C agregando nuevas funcionalidades como permitir programar utilizando el paradigma de orientación a objetos.

Cabezal: información adjuntada al inicio de un conjunto de datos con el cometido de agregar información de control utilizada por el protocolo correspondiente.

Capa: unidad de organización jerárquica de un stack de protocolos. Es un conjunto de protocolos que brindan determinado conjunto de funcionalidades a las capas (o conjuntos de protocolos) superiores. Ejemplos de esto son la capa de red, capa transporte, etc.

Checksum: suma de verificación utilizada para verificar la integridad de datos.

Cliente: proceso o computador que pide servicios a un servidor.

Conector: es un alias de la palabra Socket, refiriéndose explicitamente al punto de conexión entre dos nodos TCP.

CPU (central process unit): unidad central de proceso y cálculo de un computador.

DHCP (dynamic host configuration protocol): protocolo que permite configurar de forma dinámica un nodo host de una red estableciendo por ejemplo la IP del mismo mediante la negociación con otros nodos de la red haciendo uso de este protocolo.

DNS (domain name service): protocolo de resolución de nombres que permite a partir de un nombre (una URL) establecer cuál es la dirección IP asociada a dicho nombre.

Ethernet II: protocolo de capa 2 (enlace de datos) utilizado para transmitir los datos de capa 3 (red) en el medio físico. Por lo general se denomina red ethernet a una red que hace uso de este protocolo.

Exponential Backoff: proceso por el cual se decrementa cada cierto tiempo o evento de forma exponencial (dividiendo entre 2 cada vez) el valor de un determinado atributo .

Framework: estructura de soporte que abstrae de complejidades subyacentes y se utiliza para implementar funcionalidades de una manera estructurada y más simple.

Herramienta, Implementación: cuando en el documento se hace referencia a "la herramienta" o a "la implementación" se hace referencia implícita a la implementación de IP4JVM.

Host: nodo de red con nombre identificador. O máquina huésped de determinado software.

HTTP(Hyper-Text Transfer Protocol): protocolo utilizado por Internet para la transferencia de paginas web.

ICMP (Internet Control Message Protocol): sub protocolo de IP utilizado para el envío y gestión de mensajes de control.

Input: datos que se reciben de entrada sobre las cuáles se ejecuta un proceso.

Internet: red mundial de computadores que hace uso de múltiples tecnologías y dispositivos para lograr el objetivo de interconectar los diferentes nodos de la misma.

Iteración: en el documento puede referir tanto a la estructura de programación que permite iterar sobre un conjunto de sentencias o también a uno de los proyectos realizados sobre el desarrollo de IP4JVM.

IP (Internet Protocol): protocolo de capa 3 utilizado para enviar datos desde un nodo de la red a otro.

Java: lenguaje de programación orientado a objetos utilizado para el desarrollo del stack de protocolos.

JNI (Java Native Interface): interfaz brindada por Java para la integración de librerías no implementadas en Java como ser archivos .dll o .so.

LAN: red pequeña de computadores de alcance reducido (empresas, establecimientos) que generalmente hace uso del protocolo Ethernet II.

Loop: sentencia o conjunto de sentencias ejecutadas repetidas veces durante la ejecución de un programa.

Mail List: dirección de correo electrónico a la cuál se subscribe para participar de un foro de discusión sobre un determinado tema entre los participantes del mismo.

Makefile: archivo con conjunto de comandos necesarios para compilar un determinado paquete.

Máquina de estados: diagrama que representa los diferentes estados y los eventos que hacen a un sistema pasar de unos estados a otros.

Multicast (multi-difusión): envío de información en una red a múltiples destinos simultáneamente.

Nodo de red: cualquier dispositivo conectado a una red donde se encuentre corriendo un stack de protocolos (el mismo no tiene por que estar completamente implementado). Se puede tratar de un computador personal, un router, un servidor, un dispositivo como un switcher, etc.

Offset: posición de un byte dentro de un conjunto ordenado de los mismos obtenida a partir de la cantidad de bytes que preceden al mismo.

Open Source: se denomina de esta manera a un software del cual se brinda de manera gratuita su código fuente y sus binarios. Estos software por lo general se encuentran sujetos a la licencia GPL.

Parsear: proceso mediante el cual a partir de una tira de bytes se obtienen los datos que esta tira representa.

PC (personal computer): refiere a un computador de uso personal, en el documento se refiere a cualquier computador para abreviar.

Performance: medida que contempla el tiempo de respuesta, uso de recursos, etc., de una determinada aplicación.

Piggy Backing: proceso mediante el cual se posterga el envío de un ACK hasta que haya datos para enviar conjuntamente con el ACK o hasta que venza un timer dado.

Ping: funcionalidad utilizada para verificar la conectividad con un nodo de la red mediante el envío de un mensaje ICMP del tipo ECHO REQUEST y la posterior recepción de un mensaje ICMP de tipo ECHO REPLAY a partir del nodo solicitado. En el documento también se utiliza para hacer referencia al uso de dicha funcionalidad.

Proceso: refiere al conjunto de pasos o algoritmia que permiten cumplir con cierto objetivo.

Prototipo, Versión: se refiere a las implementaciones obtenidas en las diferentes iteraciones de IP4JVM.

RFC(Request for Comments): documentos que detallan los requerimientos necesarios para ciertos protocolos de Internet.

Router: dispositivo, computador o aplicativo que anuncia los prefijos a ser utilizados en cierta red y que además realiza el ruteo de los paquetes de la red.

Ruteo: proceso mediante el cual se logra enviar en una red un paquete desde su origen hasta su destino pasando por diferentes nodos intermedios.

Script: archivo que contiene una secuencia de comandos a ser ejecutada en una consola de sistema operativo.

Servicios de networking: conjunto de servicios que hacen posible la utilización de los recursos de red.

Servidor: proceso o computador encargado de brindar servicios a otros procesos.

Servidor HTTP: aplicación servidor que atiende los HTTP Requests enviados por diferentes clientes HTTP (por lo general navegadores web).

Sistema Operativo: software que permite a aplicaciones de usuario hacer uso de los recursos de hardware disponibles en un computador.

Socket: se usa en dos posibles sentidos, tanto como punto de conexión entre dos nodos TCP así como haciendo referencia a la clase Socket brindada por Java.

Stack: se usa en varios sentidos.

- referenciando a la clase Stack implementada encargada de administrar los jobs
- referenciando a todo el framework implementado y su funcionamiento
- referenciando a un conjunto de protocolos organizado jerarquicamente en capas que brinda funcionalidades para el uso de redes.

Stream: cadena de datos.

SVN: sistema de control de versiones utilizado para mantener versiones actuales e históricas de archivos como ser código fuente, documentos, paginas web, etc.

TCB (transfer control block): estructura de datos utilizada por protocolos de capa de transporte para mantener el control sobre las transferencias que se están llevando a cabo por el mismo.

TCP (Transmission Control Protocol): protocolo de capa 4 utilizado para establecer conexiones de transmisión de datos confiable entre nodos de una red.

Telnet: protocolo utilizado para obtener acceso remoto a una computadora de la red y hacer uso de la misma y de sus recursos de forma remota.

Testing: proceso mediante el cual se comprueba el correcto funcionamiento de un software o dispositivo.

Thread: utilizado en dos sentidos, uno para referenciar a la clase Thread brindada por Java y otro para referenciar a la unidad de procesamiento paralelo denominada thread o hilo de ejecución.

Threshold: umbral o límite utilizada para acotar un determinado valor.

Timeout: vencimiento de un timer.

Tiempo de ida y vuelta, o RTT (round trip time): tiempo que demora determinado dato en ir de un extremo al otro y volver al origen.

Timer: temporizador, en algunos casos se refiere a la clase Timer brindada por Java.

Timestamp: marca de tiempo, puede tratarse de un entero contando la cantidad de ticks desde el inicio de algún proceso o también un indicador de la fecha y hora actual.

UDP (User Datagram Protocol): protocolo de capa 4 utilizado para enviar de datagramas entre dos nodos de una red.

VM (Virtual Machine): máquina virtual refiere a la utilización de un software que simula la existencia de una máquina real en un host.

9. Referencias

[1] Documentación del proyecto IP4JVM realizado por Laura Rodríguez -
https://gforge.inria.fr/plugins/scmsvn/viewcvs.php/doc/PrimeraIteracion/?root=ip4jvm
(último acceso 10/11/08)

[2] "Acerca de la tecnología Java", Pagina principal de Java -
http://www.java.com/es/about/
(último acceso 10/11/08)

[3] RFC 2460 – Internet Protocol, Version 6 (IPv6) Specification -
S. Deering y R. Hinden – 1998 -
http://www.faqs.org/rfcs/rfc2460.html
(último acceso 10/11/08)

[4] RFC 791 – Internet Protocol -
Information Sciences Institute University of Southern California – 1981 -
http://www.faqs.org/rfcs/rfc791.html
(último acceso 10/11/08)

[5] Instituto de Computación de la facultad de Ingeniería -
http://www.fing.edu.uy/inco
(último acceso 10/11/08)

[6] IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires) -
http://www.irisa.fr/home_html-en
(último acceso 10/11/08)

[7] INRIA (l'Institut National de Recherche en Informatique et en Automatique) -
http://www.inria.fr/index.en.html
(último acceso 10/11/08)

[8] Java Virtual Machine Technology -
http://java.sun.com/javase/6/docs/technotes/guides/vm/index.html
(último acceso 10/11/08)

[9] RFC 793 – Transmission Control Protocol -
Information Sciences Institute University of Southern California – 1981 -
http://www.faqs.org/rfcs/rfc793.html
(último acceso 10/11/08)

[10] RFC 768 - User Datagram Protocol -
J. Postel – 1980 -
http://www.faqs.org/rfcs/rfc768.html
(último acceso 10/11/08)

[11] OSI ISO/IEC 10731:1994, Conventions for the definition of OSI services –
ISO/IEC – 1994 -

ISBN: 0580240789

[12] MODELO TCP/IP, Rubén Borja Rosales - http://www.raap.org.pe/docs/RAAP2_Modelo_tcpip.pdf (último acceso 10/11/08)

[13] ISO (International Organization for Standardization) - http://www.iso.org (último acceso 10/11/08)

[14] Página principal del Departamento de Defensa de los Estados Unidos - http://www.defenselink.mil/ (último acceso 10/11/08)

[15] ARPANET Project - http://stinet.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA025293 (último acceso 10/11/08)

[16] A Pragmatic Report on IPv4 Address Space Consumption, Tony Hain - http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_8-3/ipv4.html (último acceso 10/11/08)

[17] IPv4 Address Consumption, Iljitsch van Beijnum - http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_10-3/103_addr-cons.html (último acceso 10/11/08)

[18] The Internet Protocol Journal, Cisco Systems - http://www.cisco.com/web/about/ac123/ac147/about_cisco_the_internet_protocol_journal.html (último acceso 10/11/08)

[19] Cisco Systems, página principal - https://www.cisco.com (último acceso 10/11/08)

[20] IEFT (Internet Engineering Task Force) - http://www.ietf.org/ (último acceso 10/11/08)

[21] RFC 1726 - Technical Criteria for Choosing IP The Next Generation (IPng) - C. Partridge y F. Kastenholz – 1994 - http://www.faqs.org/rfcs/rfc1726.html (último acceso 10/11/08)

[22] RFC 1752 – The Recommendation for the IP Next Generation Protocolo - S. Brander y A. Mankin – 1995 - http://www.faqs.org/rfcs/rfc1752.html (último acceso 10/11/08)

[23] ALE (Address Lifetime Expectation) Working Group - http://www.ietf.org/html.charters/OLD/ale-charter.html (último acceso 10/11/08)

[24] The libpcap project at SourceForge -

http://sourceforge.net/projects/libpcap/
(último acceso 10/11/08)

[25] Java Native Interface -
http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html
(último acceso 10/11/08)

[26] Ethernet II (DIX) -
http://www.yale.edu/pclt/COMM/ETHER.HTM
(último acceso 10/11/08)

[27] RFC 4443 – Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification -
A. Conta, S. Deering y M. Gupta – 2006 -
http://www.faqs.org/rfcs/rfc4443.html
(último acceso 10/11/08)

[28] The SableVM Project -
http://www.sablevm.org/
(último acceso 10/11/08)

[29] Página Principal de Windows -
http://www.microsoft.com/spain/windows/default.mspx
(último acceso 10/11/08)

[30] Apache Harmony -
http://harmony.apache.org/
(último acceso 10/11/08)

[31] The Apache Software Foundation -
http://apache.org
(último acceso 10/11/08)

[32] Apache Tomcat -
http://tomcat.apache.org/
(último acceso 10/11/08)

[33] The Open-Source JDK Community -
http://openjdk.java.net/
(último acceso 10/11/08)

[34] Sun Microsystems Homepage -
http://www.sun.com/
(último acceso 10/11/08)

[35] SVN del proyecto IP4JVM -
https://scm.gforge.inria.fr/svn/ip4jvm
(último acceso 10/11/08)

[36] Linux Fedora Core 6, Fedora Project -
http://fedoraproject.org/es/index
(último acceso 10/11/08)

[37] Eclipse - an open development platform -
http://www.eclipse.org/
(último acceso 10/11/08)
[38] WireShark Hompegae -
http://www.wireshark.org/
(último acceso 10/11/08)
[39] OpenSUSE.org -
http://es.opensuse.org/Bienvenidos_a_openSUSE.org
(último acceso 10/11/08)
[40] Briefing sobre el procesador AMD Sempron -
http://www.amd.com/es-es/Processors/ProductInformation/0,,30_118_11599_11603,00.html
(último acceso 10/11/08)
[41] Tecnología móvil AMD Turion 64 X2 – Resumen de producto -
http://www.amd.com/la-es/Processors/ProductInformation/0,,30_118_13909_13910,00.html
(último acceso 10/11/08)
[42] Java IcedTea (OpenJDK) - Fedora Unity -
http://fedorasolved.org/Members/jpmahowald/java-icedtea-openjdk
(último acceso 10/11/08)
IcedTea Wiki -
http://icedtea.classpath.org/wiki/Main_Page
(último acceso 10/11/08)
[43] Linux RedHat Homepage -
http://www.redhat.es/
(último acceso 10/11/08)
[44] Mail lists de OpenJDK -
http://mail.openjdk.java.net/mailman/listinfo/build-dev
(último acceso 10/11/08)
http://mail.openjdk.java.net/mailman/listinfo/net-dev
(último acceso 10/11/08)
[45] Reno TCP en Wikipedia -
http://es.wikipedia.org/wiki/Reno_TCP
(último acceso 10/11/08)
[46] RFC 3782 The NewReno Modification to TCP's Fast Recovery Algorithm -
S. Floyd, T. Henderson y A. Gurtov – 2004 -
http://www.faqs.org/rfcs/rfc3782.html
(último acceso 10/11/08)
[47] Vegas TCP Homepage -
http://www.cs.arizona.edu/projects/protocols/

(último acceso 10/11/08)
[48] 4.4 BSD Lite-based Operating Systems, Christopher Browne
http://cbbrowne.com/info/bsd.html
(último acceso 10/11/08)
[49] TCP/IP Illustrated, Volume 2: The Implementation -
Gary R. Wright y W. Richard Stevens – Addison Wesley – 1995 -
ISBN: 020163354X
[50] Computer Networks – Third Edition -
Andrew S. Tanenbaum – Pearson Education – 1996 -
ISBN: 0133942481
[51] RFC 1981 – Path MTU Discovery for IP version 6 -
J. McCann, S. Deering y J. Mogul – 1996 -
http://www.faqs.org/rfcs/rfc1981.html
(último acceso 10/11/08)
[52] RFC 2461 – Neighbor Discovery for IP Version 6 (IPv6) -
T. Narten, E. Nordmark y W. Simpson – 1998 -
http://www.faqs.org/rfcs/rfc2461.html
(último acceso 10/11/08)
[53] RFC 2462 – IPv6 Stateless Address Autoconfiguration -
S. Thompson y T. Narten – 1998 -
http://www.faqs.org/rfcs/rfc2462.html
(último acceso 10/11/08)
[54] RFC 3484 – Default Address Selection for Internet Protocol version 6 (IPv6) -
R. Draves – 2003 -
http://www.faqs.org/rfcs/rfc3484.html
(último acceso 10/11/08)
[55] RFC 4291 – IP Version 6 Addressing Architecture -
R. Hinden y S. Deering – 2006 -
http://www.faqs.org/rfcs/rfc4291.html
(último acceso 10/11/08)
[56] RFC 4311 – IPv6 Host-to-Router Load Sharing -
R. Hinder y D. Thaler – 2005 -
http://www.faqs.org/rfcs/rfc4311.html
(último acceso 10/11/08)
[57] IPv6 Essentials - Second Edition -
Silvia Hagen – O'Reilly – 2006 -
ISBN: 0596100582
[58] Página del proyecto IP4JVM en el gforge del INRIA -
https://gforge.inria.fr/projects/ip4jvm/
(último acceso 10/11/08)

[59] The history of the Internet
http://www.historyoftheinternet.com
(último acceso 10/11/08)

[60] RFC 3775 – Mobility support in IPv6 -
D. Johnson, C. Perkins y J. Arkko – 2004 -
http://www.faqs.org/rfcs/rfc3775.html
(último acceso 10/11/08)

[61] Linux IPv6 Router Advertisement Daemon (radvd) -
http://www.litech.org/radvd/
(último acceso10/11/08)

[62] Knoppix MIPL -
http://www.ipt.etsi.org/knoppix.htm
(último acceso 10/11/08)

[63] DNS Resources Directory -
http://www.dns.net/dnsrd/
(último acceso 10/11/08)

[64] RFC 854 - Telnet Protocol Specification -
J. Postel y J. Reynolds – 1983 -
http://www.faqs.org/rfcs/rfc854.html
(último acceso 10/11/08)

[65] IPv6 Ready Logo – home page –
http://www.ipv6ready.org/
(último acceso 10/11/08)

[66] IPv6 Ready Phase 1 / 2 Test Interoperability Specification Core Protocols –
http://www.ipv6ready.org/pdf/IPv6Ready_Base_Interop_version_4_0_0.pdf
(último acceso 10/11/08)

[67] UML (Unified Model Language) –
http://www.uml.org/
(último acceso 10/11/08)

[68] Linux Home Page –
http://www.linux.org/
(último acceso 10/11/08)

10. Apéndices

A continuación se adjuntan los distintos apéndices, los cuáles presentan con mayor detalle los principales aspectos sobre cada uno de las temáticas generales tratadas anteriormente en el documento.

10.1 Preparación de entorno (svn, eclipse, etc)

10.1.1 Introducción

El presente apéndice presenta aspectos necesarios para instalar el entorno de trabajo utilizado en Fedora Core 6 y OpenSuse 10.3. Los pasos aquí detallados se pueden trasladar a Windows y otros sistemas operativos considerando que la mayor diferencia existirá en la compilación de la máquina virtual. Para instalar algunas de las herramientas utilizadas en el entorno puede llegar a ser necesario tener permisos de super usuario, por lo que se recomienda realizar la instalación con un usuario que los tenga. Se recomienda además, en la mayoría de los casos excepto aquellos mencionados, utilizar las últimas versiones de los productos dados.

Como dato informativo, a continuación se presenta una tabla que resume los productos necesarios y las versiones utilizadas durante el desarrollo del proyecto.

Producto	Versión
Wireshark	1.0.4
Eclipse	3.2
Plugin Subclipse	1.2.x
Máquina Virtual Java JDK	1.5
SableVM	1.13
OpenJDK	1.7

10.1.2 Instalación

Wireshark

Esta herramienta es utilizada para ver el tráfico de la red. El mismo puede ser instalado haciendo uso de los administradores de paquetes que tienen los sistemas operativos mencionados. La versión utilizada de esta herramienta fue la 1.0.4. En la siguiente URL se puede obtener más información sobre esta herramienta:

http://www.wireshark.org/
(último acceso 10/11/08)

Máquina Virtual Java JDK

Es necesario tener instalada una máquina virtual Java JDK, se recomienda la versión 1.5 o superiores. En caso de que el sistema operativo utilizado no brinde una se puede descargar la versión 1.6 de la siguiente página:

https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewProductDetail-Start?ProductRef=jdk-6u4-b-oth-JPR@CDS-CDS_Developer
(último acceso 10/11/08)

La instalación de la máquina virtual consiste en ejecutar el archivo descargado e instalar la máquina virtual en la ruta que se desee.

Puede ser necesario incluir en la variable PATH la ruta a la máquina virtual. En Linux esto se logra ejecutando el siguiente comando: export PATH=<ruta a la máquina virtual>:$PATH. Este comando puede ser incluido en el archivo .bashrc que se encuentra en el directorio home del usuario. En Windows este paso podría ser llevado a cabo modificando las variables de entorno ingresando a "Mi PC – Propiedades – Avanzado – Variables de entorno" y aquí agregar a la variable PATH la ruta a la máquina virtual.

Para verificar que la instalación ha sido correcta se puede iniciar una consola y ejecutar el comando: java --version (en Windows java -version). Verificando que la información desplegada coincide con la versión descargada.

Eclipse

Para realizar cambios en el código del proyecto y descargarlo se utilizó el IDE Eclipse en su versión 3.2. Para esto es necesario haber instalado una máquina virtual Java acorde a la

versión. El mismo puede ser descargado de la siguiente página:

http://www.eclipse.org/downloads/
(último acceso 10/11/08)

La herramienta no requiere instalación, basta con descomprimir el archivo a una carpeta y ejecutar el comando eclipse en Linux situados en la carpeta descomprimida (puede llegar a ser necesario configurar permisos de ejecución sobre el mismo). En windows basta con dar doble click sobre el ejecutable eclipse.exe dentro de la carpeta descomprimida. Cuando la aplicación pregunta sobre el workspace a utilizar, usar el que se da por defecto.

A continuación para poder bajar el proyecto IP4JVM se debe instalar el plugin subclipse versión 1.2.x el cuál permite gestionar un repositorio SVN desde Eclipse. En la siguiente página se detalla cómo descargar e instalar dicho plugin:

http://subclipse.tigris.org/install.html
(último acceso 10/11/08)

Proyecto IP4JVM

Para bajar el proyecto se debe primero que nada tener un usuario con los permisos adecuados para gestionar el repositorio del proyecto IP4JVM ya que el mismo es un proyecto privado por el momento. El proyecto se encuentra en gforge cuya página es la siguiente:

https://gforge.inria.fr/projects/ip4jvm/
(último acceso 10/11/08)

Pasos a seguir luego:

1. A continuación en Eclipse ir al menú Window -> Open Perspective -> Others .... Seleccionar "SVN Repository Exploring".

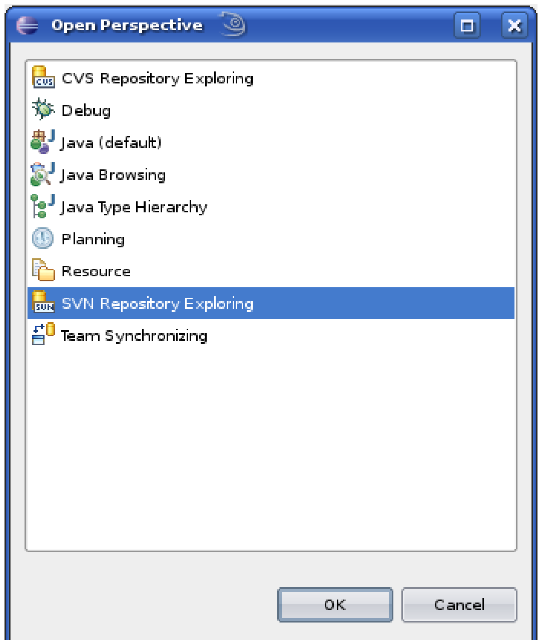


Ilustración 29: Menú Open Perspective

2. Luego de esto, dar click en el botón que se indica en la siguiente imagen:

Ilustración 30: Botón de creación de repositorio

3. Ingresar la ruta "https://scm.gforge.inria.fr/svn/ip4jvm" en el campo URL y hacer click en Finish en la siguiente pantalla:

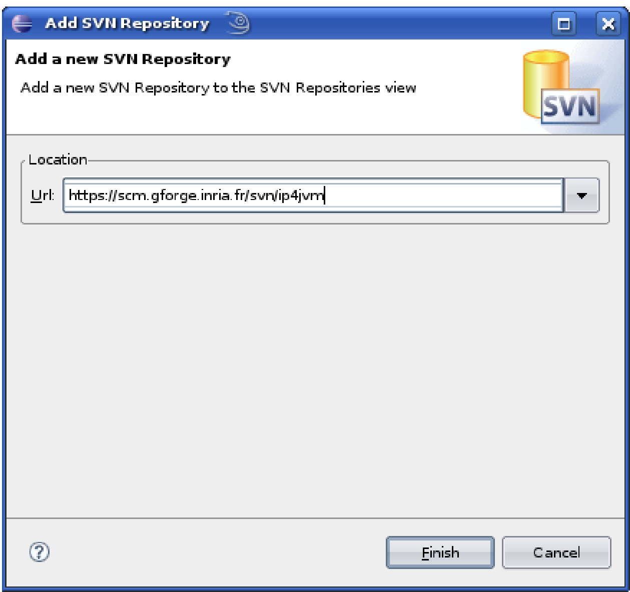


Ilustración 31: Pantalla de Add Repository

4. Abrir el árbol, dar click derecho en la carpeta doc y seleccionar Checkout

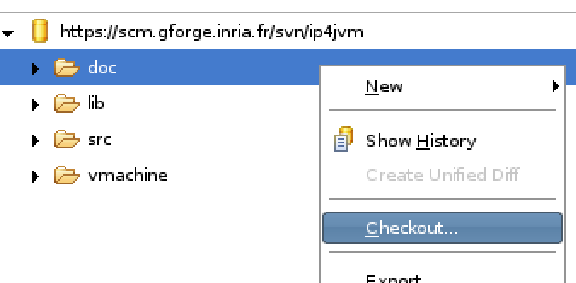


Ilustración 32: Checkout de doc

5. En la siguiente ventana hacer click en Finish:

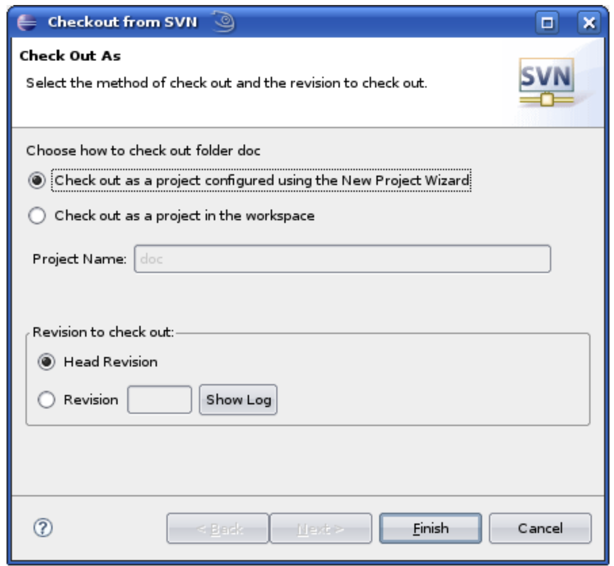


Ilustración 33: Ventana de checkout

6. A continuación seleccionar Project y dar en Next.

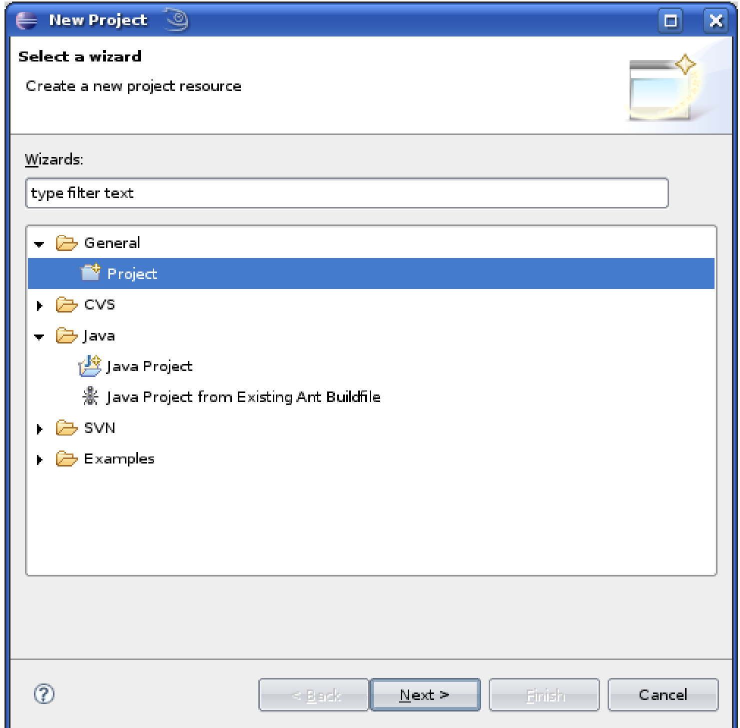
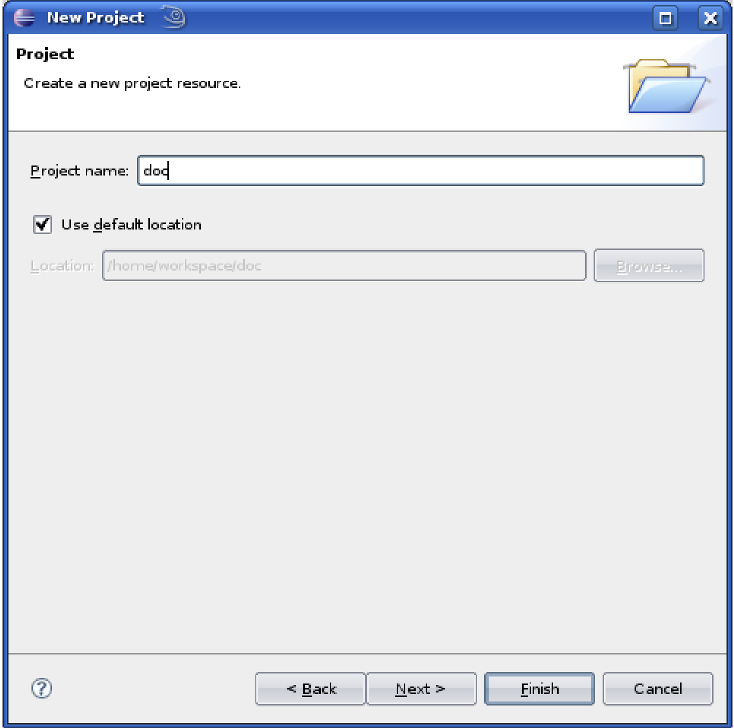


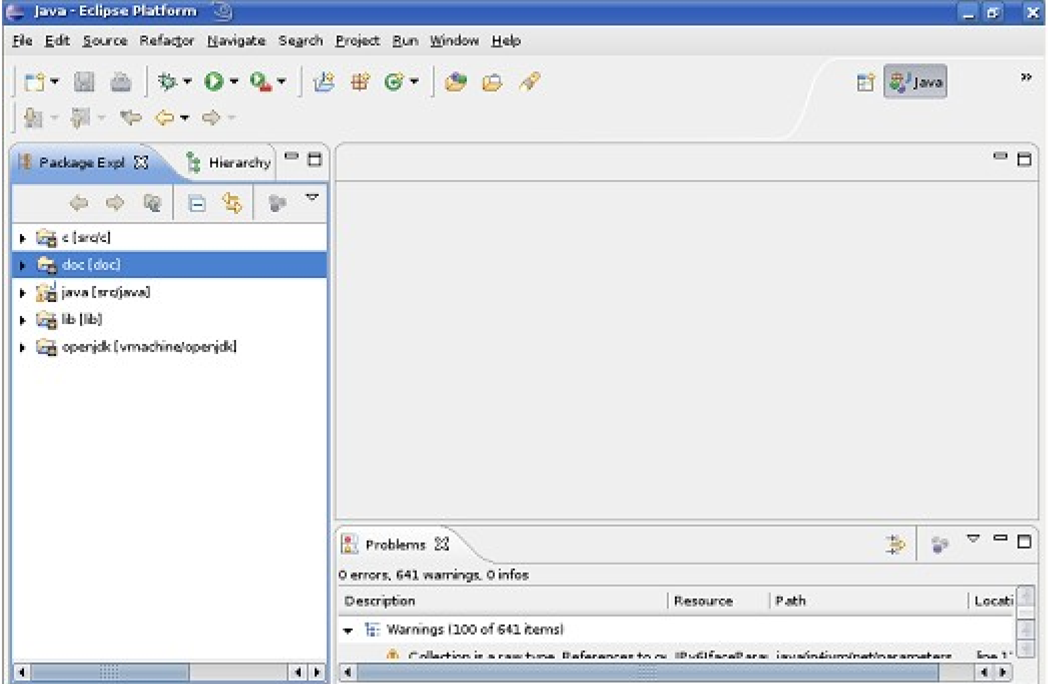
Ilustración 34: Creación de proyecto

7. En la siguiente ventana ingresar el nombre del proyecto (por ejemplo doc) y dar en Finish.



8. Repetir los pasos a partir del 4 pero esta vez para lib, src/c y vmachine.

9. Finalmente repetir los pasos desde el 4 al 7 para src/java pero seleccionando tipo de proyecto Java Project. Una vez finalizado el procedimiento, al ingresar a la vista de exploración de paquetes del eclipse, deberían verse todos los proyectos de la siguiente manera:



Compilación máquina virtual con modificaciones

En el siguiente apéndice (10.2) se detallan los pasos a seguir para compilar satisfactoriamente la máquina virtual con las modificaciones necesarias para incluir la herramienta IP4JVM.

10.2 Instalación de stack en SableVM y OpenJDK

10.2.1 Introducción

En el presente apéndice se presentan los principales lineamientos para la compilación de las máquinas virtuales Sable-VM y OpenJDK y las modificaciones necesarias en las mismas para integrar adecuadamente la herramienta IP4JVM.

10.2.2 Objetivos

Brindar un documento de apoyo para la instalación de la herramienta en las máquinas virtuales Sable-VM y OpenJDK.

10.2.3 Instalación en SableVM

La compilación de la máquina virtual SableVM fue realizada únicamente en el sistema operativo Fedora Core 6 por lo que los lineamientos aquí expuestos se refieren a dicho sistema operativo con la suite estándar de paquetes instalada. Se seleccionó este sistema operativo ya que a la hora de la realización de la compilación de la máquina virtual era el sistema operativo que se encontraba disponible en la facultad, por lo que pareció razonable alinearse a esta práctica. Adicionalmente esta instalación fue requerida ya que la herramienta IP4JVM era compilada originalmente en esta máquina virtual. La compilación en Windows fue dejada de lado debido a que teníamos como objetivo tener la herramienta funcionando sin importar, en un principio, el sistema operativo subyacente.

A continuación se brindaran la serie de pasos necesarios para compilar la máquina virtual. Para ejecutar algunos de los pasos es necesario tener permisos de super usuario por lo que se recomienda que se realicen con un usuario que tenga dichos permisos.

1. Descargar una versión SDK de la máquina virtual SableVM 1.13 o superior, o utilizar la brindada en el proyecto bajo la carpeta vmachine/sablevm.

   http://sourceforge.net/project/downloading.php?group_id=5523&use_mirror=superb-west&filename=sablevm-sdk-1.13.tar.gz&4224698
   (último acceso 10/11/08)

2. Descomprimir el archivo .tar.gz bajado de la máquina virtual en un directorio a elección.

3. Instalar los paquetes necesarios para la JVM. Para esto no es necesario bajarlos e instalarlos manualmente, esto se puede realizar de manera automática gracias a la herramienta "Administrador de Paquetes" incluida en Fedora Core 6. Los paquetes necesarios son:

- libcap-devel (utilizado por IP4JVM)
- libtool
- libtool-ltdl3-devel

No se brindan referencias para bajar dichos paquetes pues como ya se comentó se pueden instalar utilizando la utilidad "Administrador de Paquetes" que brinda Fedora Core 6.

En el archivo INSTALL contenido dentro de la carpeta sablevm (descomprimida a partir del .tar.gz en el paso anterior) se referencia las dependencias necesarias para compilar la máquina virtual. Esto puede ser de utilidad en el caso de que se quiera intentar compilar dicha máquina virtual en otro sistema operativo.

4. Con el código de IP4JVM disponible, copiar la carpeta vmachine/sablevm/sablevm-1.13/src a la carpeta sablevm/src (siendo la carpeta sablevm la descomprimida a partir del .tar.gz) sobre escribiendo todo archivo existente. Copiar además los archivos contenidos en la carpeta vmachine/sablevm/sablevm-classpath-1.13/lib a la carpeta sablevm-classpath/lib (esta carpeta se obtiene al descomprimir el archivo .tar.gz).

5. Compilar e instalar la máquina virtual ejecutando en una consola, posicionados en la carpeta descomprimida a partir del .tar.gz, los siguientes comandos:

- /.configure
- make
- make install

6. Se puede hacer uso del comando "make check" para verificar que la compilación haya sido exitosa. Además, luego de compilada la máquina virtual, deberían haberse creado las carpetas "classpath" y "sablevm" bajo la ruta /usr/local/bin.

7. Si se desea que Sable-VM sea la máquina virtual Java por defecto del sistema operativo, se debe modificar la variable de entorno PATH ejecutando el siguiente comando:

- export PATH=/usr/local/lib/sablevm/bin:$PATH.

En caso de que se desee automatizar el proceso de manera tal que por defecto al iniciar el sistema operativo se realice el seteo antes mencionado se debe copiar el comando antes dado al final del archivo .bashrc bajo la carpeta home del usuario.

8. Se debe setear la variable CLASSPATH utilizando el siguiente comando:
    - export CLASSPATH=<ruta a la carpeta descomprimida de Sable-VM>/sablevm-classpath/lib/:$CLASSPATH.

   Se puede cambiar también en este caso el archivo .bashrc para que el comando sea ejecutado cada vez que se inicia la sesión del usuario.

9. Luego se deben compilar las librerías netmanageread y netmanagerwrite utilizadas por IP4JVM para enviar y recibir segmentos hacia y desde la red usando pcap. Para esto se deben ejecutar los siguientes comandos en una consola de comandos que se encuentre posicionada en la ruta src/c del proyecto IP4JVM:
    - make -f makefile.read install
    - make -f makefile.write install

   La ejecución de estos comandos creará 2 archivos llamados “libnetmanagerread.so” y “libnetmanagerwrite.so” bajo la carpeta lib del proyecto IP4JVM.

10. Como paso final se debe setear la variable de entorno LD_LIBRARY_PATH haciendo que la misma apunte a la ruta de las librerías antes creadas (esto se realiza de manera análoga al seteo de las variables PATH y CLASSPATH).

En caso de que se desee verificar el correcto funcionamiento de la aplicación se puede ejecutar cualquiera de los programas Java de los cuales se encuentra el código en la carpeta test del proyecto IP4JVM. Si los mismos no dan errores de carga de clases o librerías significa que la herramienta se encuentra correctamente integrada a la máquina virtual.

Un ejemplo de comando de ejecución de uno de los test sería:

- java test/TestNetwork ping6 eth0 fe80::211:11ff:fe20:b763 5

Recordar que para ejecutar cualquiera de los test se debe crear un archivo llamado ip4jvm.config el cual contenga las direcciones MAC a ser utilizadas.

10.2.4 Instalación en OpenJDK

A continuación se detallaran los pasos a seguir para compilar OpenJDK en los sistemas operativos Fedora Core 6 y OpenSuse 10.3 e integrar la herramienta IP4JVM a dicha máquina virtual. Se intentó su compilación en Windows, requiriendo para ello instalar varios aplicativos, como ser Visual Studio 6, de los cuales no teníamos, en un principio, los instaladores. Luego de conseguirlos no se pudo instalar adecuadamente estos aplicativos y priorizando las tareas, se decidió seguir con otras de mayor relevancia y dejar esta de lado. Se debe contar con permisos de super usuario en el usuario utilizado para realizar los pasos que se indican.

1. Bajar los fuentes (src) y enchufes binarios (binary plugs) de OpenJDK. En este documento se utilizó la versión 1.7 b22 de OpenJDK, los fuentes y binary plugs para arquitectura de procesadores x86 y para sistemas operativos Windows y Linux se encuentran en la carpeta vmachine/openjdk del proyecto IP4JVM.

   Ruta para descargar OpenJDK 1.7 b22:
   http://download.java.et/openjdk/jdk7/promoted/b22
   (último acceso 10/11/08)

2. Bajar e instalar ant. Luego de bajar el archivo .tar.gz, descomprimirlo en una carpeta a elección (Por ejemplo USER_HOME/Desktop, donde USER_HOME es la ruta al directorio home del usuario) y luego inicializar la variable de entorno ANT_HOME (si se descomprimió en Desktop: export ANT_HOME=USERT_HOME/Desktop). Si se quiere automatizar el proceso de inicialización de la variable se debe ingresar el comando utilizado al final del archivo .bashrc (que se encuentra debajo de la carpeta USER_HOME).

   Ruta para descargar ant:
   http://ant.apache.org/bindownload.cgi
   (último acceso 10/11/08)

3. Si no se tiene instalado JDK 1.6 o superior, instalarlo. Se aconseja instalarlo bajo la carpeta /opt ya que los scripts de compilación de OpenJDK han sido modificados de manera tal de tomar esta carpeta como carpeta de instalación. Además en este documento se utilizó la versión jdk1.6.0_03, si se utiliza otra versión o se instala en otra ruta será necesario modificar el script jdk_generic_profile.sh, dado, de manera que refleje los cambios.

   Ruta para descarga de JDK 1.6:
   https://cds.sun.com/is-bin/INTERSHOP.enfinity/WFS/CDS-CDS_Developer-Site/en_US/-/USD/ViewFilteredProducts-SingleVariationTypeFilter;pgid=QLNAWvO8KBJSR0EUreYksEPG0000Yjy0GFFP;sid=Ne7RWvZwPW7RWLG6kQOZXxnZbzSVkTs1YAwem-tubzSVkQ==
   (último acceso 10/11/08)

4. Se requiere tener instalados todos los paquetes de los cuales depende OpenJDK. Esto se puede realizar con la utilidad de gestión de paquetes que brindan los sistemas operativos antes mencionados, Fedora Core 6 y OpenSuse 10.3. Los paquetes que son necesarios para los sistemas operativos nombrados con la versión b22 de OpenJDK son:
   - cups, cups-devel
   - gnu make
   - gcc, gcc-c++

- glibc, glibc-devel
- alsa, alsa-devel
- freetype2, freetype2-devel
- zip
- openmotif, openmotif-devel (y sus dependencias)

Puede que no sea necesario instalar todos estos paquetes dependiendo de los que se tengan ya instalados.

No se brindan referencias para descargar los paquetes pues como se mencionó anteriormente los mismos pueden ser descargados con las herramientas de gestión de paquetes que brindan los sistemas operativos nombrados con anterioridad.

5. Descomprimir el archivo .zip de los fuentes de OpenJDK donde se desee (en el presente documento se refiere a dicha ruta como OPENJDK_HOME y se considerara como /root/Desktop/openjdk).

6. Ejecutar el archivo .jar de los binary plugs correspondiente (en este caso el de Linux) y seleccionar la ruta de instalación que se prefiera. El archivo jdk_generic_profile.sh contenido en la carpeta vmachine/openjdk/make está modificado de tal manera que los binary plugs deben estar instalados en /opt, en caso de que se instalen en otra ruta se debe asignar el valor de la ruta a la variable ALT_BINARY_PLUG usada en el script dado. En Suse puede llegar a ser necesario correr el siguiente comando para poder ejecutar java -jar: export LIBXCB_ALLOW_SLOPPY_LOCK=1. Este comando puede ser agregado al archivo .bashrc antes descrito si se desea que el comando sea ejecutado automáticamente al iniciar la sesión del usuario.

7. Copiar el directorio src/java/ip4jvm contenido dentro del proyecto IP4JVM al directorio OPENJDK_HOME/j2se/src/share/classes (esta ruta debería existir). Luego copiar los directorios j2se y control contenidos en la carpata vmachine/openjdk/openjdk a la ruta OPENJDK_HOME. Si en alguno de los dos casos el sistema pregunta si se deben sustituir los datos existentes, confirmar (tal vez sea propicio respaldar los archivos originales antes de sustituirlos).

8. En OpenSuse 10.3 es necesario aplicar el patch const_strings.patch que se encuentra bajo la carpeta /vmachine/openjdk/suse del proyecto IP4JVM.

9. Ejecutar en una consola posicionada en la ruta OPENJDK_HOME/control/make el comando: ./jdk_generic_profile.sh. Si el archivo no es ejecutable se debe asignar los permisos de ejecución al mismo.

Si todo va bien, al rededor de una hora (dependiendo del computador utilizado) se tendrá todo satisfactoriamente compilado. Se pueden verificar los mensajes que la compilación ha enviado a la salida estándar abriendo el archivo salida2.txt, y los errores

y advertencias se pueden ver en el archivo errores2.txt (estos dos archivos se generan bajo la ruta OPENJDK_HOME/control/make).

Luego de compilada la máquina virtual tal vez se desee poner la nueva máquina virtual Java en la variable PATH. La carpeta de máquina virtual JDK Java compilada se encuentra en la ruta OPENJDK_HOME/control/build/linux-i586/j2sdk-image/bin.

10. Ir a la carpeta src/c del proyecto IP4JVM y en una consola (posicionada en esta ruta) correr los comandos "make -f makefile.read install" y "make -f makefile.write install". Luego de esto deberían crearse 2 archivos llamados libnetmanagerread.so y libnetmanagerwrite.so bajo la carpeta lib del proyecto IP4JVM.

En el caso de que se haya instalado la máquina virtual base (JDK 1.6) en otro directorio diferente de /opt se deberá modificar el valor asignado a la variable JAVA_HOME en estos archivos antes de compilarlos.

11. Como último paso copiar los archivos libnetmanagerread.so y libnetmanagerwrite.so a la carpeta OPENJDK_HOME/control/build/linux-i586/j2sdk-image/jre/lib/i386.

Luego de compilada la máquina virtual y de inicializada la variable de entorno PATH de manera tal que apunte a la nueva máquina virtual compilada, se puede correr el programa TestUDPNet y testPing6v2 modificando el código fuente del primero y el archivo de configuración de manera que contengan las direcciones MAC deseadas.

Para correr testPing6v2 se debe ejecutar en la linea de comandos el siguiente comando cambiando la dirección de destino del ping: "java testPing6v2 ping6 eth0 fe80::20c:29ff:fe30:5433". Este programa solo verificará que las clases del stack IP4JVM se encuentran en el class path por defecto.

Para correr TestUDPNet no es necesario pasar parámetros al programa (pero si modificar el código). Este programa verifica la correcta integración entre la máquina virtual y el stack utilizando por defecto el stack dado por IP4JVM como el stack por defecto para Internet. Esto usa la clase por defecto DatagramSocket modificada para usar el stack de IP4JVM. El programa únicamente envía dos mensajes sin datos.

Una manera más fácil que permite ejecutar todos los test que se desen es usar la aplicación Tester que se encuentra bajo la carpeta test/multiTester. El mismo es muy facil de correr y es auto explicativo.

10.2.5 Conclusiones

Se logró compilar satisfactoriamente las máquinas virtuales Sable-VM y OpenJDK en los sistemas operativos Fedora Core 6 y OpenSuse 10.3 (este último solo para la máquina virtual OpenJDK). Además se pudo integrar satisfactoriamente el stack implementado en el proyecto

IP4JVM a ambas máquinas virtuales.

10.2.6 Trabajo Futuro

Aspectos que quedan para la investigación y experimentación son:

- Probar con otros sistemas operativos.
- Probar con otras arquitecturas de procesadores.
- Integrar el stack con otras máquinas virtuales Java.

10.3 Mejoras realizadas sobre código existente y planificación de ejecución.

El contenido de este apéndice no es más que el documento generado como salida en la tarea de identificación de cambios a realizar en la herramienta. Dicha tarea se llevó a cabo en Junio de 2007 y fue la primera aproximación profunda al código subyacente a la herramienta; por ende los tiempos verbales de la redacción de este apéndice se ajustan al momento en que fue generado el documento, o sea al finalizar la identificación de cambios mediante la lectura detallada del código y previo al inicio de la realización de los mismos.

10.3.1 Introducción:

En el presente documento se listarán los aspectos del código que se hallaban implementados para el proyecto IP4JVM que, según nuestro juicio, ameritaban un análisis detallado que podrían derivar en una posible modificación del mismo.

La estructura del documento está conformada por tres secciones que listan los distintos tipos de cambios que a nuestro entender debían llevarse a cabo con el fin de mejorar la calidad del software, ya sea desde el punto de vista de la performance u otros aspectos importantes como la escalabilidad o mayor claridad del código en sí, más una sección de conclusiones generales donde se clasifican cambios según ciertos criterios y se esboza un plan de ejecución de los mismos. Recordemos que este documento sirvió como punto de arranque de la fase de ejecución de cambios y de ahí surgió la importancia de tener las modificaciones plenamente identificadas, detalladas, priorizadas y ordenadas.

La primera de las secciones abarca los *Criterios Generales*, la misma refiere a aquellos aspectos donde vimos que sería útil tomar un criterio consistente y usarlo coherentemente a lo largo del desarrollo, para, de esta manera, ser consistentes en la solución de problemas parecidos y no dejarlo a la libre voluntad de cada desarrollador.

La segunda sección, *Diseño*, contiene aquellas mejoras que a nuestro entender se le podían hacer al diseño del producto en pos de obtener un software con mejores cualidades. El principal objetivo en este sentido era tener un diseño que aumentara la cohesión, ya que identificamos clases que realizaban tareas que no parecían inherentes a las mismas. Crear clases más específicas en el tratamiento de los datos de entrada y salida y otras que se especializaran a tratar los vencimientos de tiempos, por ejemplo, nos ayudaría a tener un diseño más cohesivo y más fácilmente entendible y legible, haciéndolo a su vez más fácil de mantener y extender en futuras iteraciones.

En la sección *Clases*, se detallan minuciosamente por cada clase, aquellos cambios que sugerimos; cambios estos que a su vez se dividen en 3 tipos: Definición (por ejemplo un cambio de nombre para obtener mayor nemotecnicidad); Atributos (cambio de tipos, o agregar o sacar atributos que aportaban o dejaban de aportar); y Algoritmia (cambios en los algoritmos de las funciones que permitieran optimizar el rendimiento).

Para cada uno de los cambios listados, también se adjuntará una estimación primaria del esfuerzo necesario para realizarlo y el impacto del mismo en el software en su conjunto. Una tercera clasificación de los cambios, se da a partir del grado de dependencia con el resto del código, que intentaba dar una idea general de la cantidad de puntos donde este cambio impactaría. Estas tres propiedades (Esfuerzo, Impacto y Dependencias) se clasificarán en una de las tres categorías Alto, Medio y Bajo y deben ser tomadas como una estimación primaria, que se estimó al momento de identificar los cambios y que puede distar bastante del real esfuerzo necesario y del verdadero impacto de llevar a cabo el cambio en la realidad.

Por último en la sección Conclusión se hará un resumen de los cambios propuestos para ilustrar de mejor manera y de forma resumida cuál sería el plan óptimo de trabajo sobre los mismos.

10.3.2 Criterios generales:

A continuación se listarán aquellas convenciones que suponemos de utilidad para los programadores, con el fin de mejorar el código en cuanto a su legibilidad y consistencia, así como también en la performance del producto. Estos criterios, pensamos, deberían respetarse a lo largo de todo el código, y no refieren a un método o fragmento de código en particular, sino a todo el proyecto en sí.

El esfuerzo para llevar a cabo estas modificaciones ha de ser Alto, ya que refieren a cambios sobre conceptos repetidos constantemente a lo largo de toda la implementación.

10.3.2.1 Colecciones tipadas.

a) **Descripción:**
Pensamos que el uso de la tipificación de colecciones, aprovechando esta reciente característica de Java, da una mayor claridad al código, así como también permite una mejor posibilidad de detección de faltas, en tiempo de compilación, debidas al uso erróneo de *casting*. Hace falta remarcar que el uso de colecciones tipadas fue agregado en la versión 1.5 de Java, la cual no se hallaba disponible aún al momento de realización del proyecto anterior.

b) **Esfuerzo:** Alto.
c) **Impacto:** Bajo.
d) **Dependencias:** Alto.

10.3.2.2 Uso consistente de colecciones de objetos.

a) **Descripción:**
Vimos que a lo largo de todo el código, al iterarse con listas se utiliza un índice que va desde el valor 1 hasta el tamaño de la lista, y los objetos de la misma se obtienen mediante el método `get(indx)`. Esto, en el uso de `java.util.LinkedLists`, es poco eficiente, ya que el método `get(indx)` justamente recorre la lista nuevamente hasta encontrar el i-ésimo elemento indicado por el parámetro. A nuestro criterio esto no debería ser implementado de esta manera y nos motivó a plantear la siguiente convención:

- Si el uso del método `get(indx)` es imperante, se debe usar un `java.util.Vector` y

no una `java.util.LinkedList`.

- Para iterar sobre listas deberá hacerse uso de iteradores.

- Cuando la iteración sobre listas no tiene condición de parada, o sea que necesariamente debe recorrerse toda la lista, se deberá usar otra de las nuevas características incluidas a partir de Java 1.5 como lo es el `foreach()`.

b) **Esfuerzo:** Alto.
c) **Impacto:** Alto.
d) **Dependencias:** Alto.

10.3.2.3 Extensión de colecciones.

a) **Descripción:**
No nos parece del todo prolijo que varias clases extiendan a la clase `java.util.LinkedList`, sobre todo porque estas clases no son listas en sí, sino que poseen una lista de elementos pero también poseen atributos simples. En lugar de esto, deberían contener un atributo que contenga la colección extendida y realizar la especificación de los métodos y comportamientos definiendo nuevos métodos usando ese atributo.

b) **Esfuerzo:** Alto.
c) **Impacto:** Bajo.
d) **Dependencias:** Alto.

10.3.2.4 Iteraciones.

a) **Descripción:**
No son pocas las ocasiones en el código en donde se hay un ciclo recorriendo alguna colección y se recorre toda, aún cuando existe algún condicionante que ameritaría la parada de la recorrida. Esto junto a otros detalles que a nuestro entender son posibilidades de mejorar el código, se listan a continuación:

- Agregar las condiciones faltantes en los ciclos que así lo requieran
- Las sentencias `for()`, que puedan ser optimizadas con un `while()` que contenga una condición de parada, deberán ser cambiadas.

b) **Esfuerzo:** Alto.
c) **Impacto**: Alto.
d) **Dependencias:** Alto.

10.3.2.5 Constructores.

a) **Descripción:**
Pensamos que deben implementarse constructores de los objetos que reciban por parámetro todos los atributos del mismo, para no tener que acceder a los mismos luego de construido el objeto. A continuación se adjunta una fracción del código que deja en evidencia nuestras bases para creer necesario tal cambio:

```
NetParameterNetwork nip = new NetParameterNetwork();
nip.ipAddress = ipAddress;
nip.tentative = tentative;
```

b) **Esfuerzo:** Bajo.
c) **Impacto:** Bajo.
d) **Dependencias:** Alto.

10.3.2.6 Conocimiento del Nivel.

a) **Descripción:**
No nos parece adecuada ni natural la decisión de que una aplicación sepa en que nivel está. Intuitivamente sería más una responsabilidad del stack u otra entidad.

Network sabe cuales son las aplicaciones y layers y en que nivel van por lo que las aplicaciones y layers no tendrían por que saber a que nivel pertenecen.

Deberíamos definir constantes para aquellos valores incambiados que se usan para retornar el nivel en el que va una aplicación o layer

b) **Esfuerzo:** Medio.
c) **Impacto:** Bajo.
d) **Dependencias:** Alto.

10.3.3 Diseño.

A continuación detallamos aquellos cambios que sugerimos como necesarios en el diseño del software para aumentar la legibilidad del código y mejorar sus cualidades para futuros usos del mismo.

10.3.3.1 ip4jvm.javafwrk.NetLevel.ItemNetLevelLayer / ip4jvm.javafwrk.NetLevel.ItemNetLevelApplication.

a) **Descripción:** A nuestro entender, debería contarse con una interfaz a la cuál ambas clases realicen, ya que siempre están diferenciadas en el código, siendo que la única diferencia que tienen en realidad es el método `receive()`. Con esta mejora podríamos incluir métodos en la interfaz que permitan a cada una de las clases implementarlos , según corresponda, evitar cosas como la que sigue (y que se repiten mucho en el código):

```
if (getFirst() instanceof ItemNetLevelLayer){
      ItemNetLevelLayer l =(ItemNetLevelLayer)getFirst();
      return l.getLayer().getLevel();
}else if (getFirst() instanceof ItemNetLevelApplication){
      ItemNetLevelApplication l =(ItemNetLevelApplication)getFirst();
      return l.getApplication().getLevel();
}
```

pudiendo, por ejemplo, cambiar todas estas líneas de código por una sola linea, que sería la siguiente:

```
return getFirst().getLevel();
```

El esfuerzo estimado para este cambio es de Medio, ya que deberían hacerse algunas modificaciones en todos aquellos algoritmos que tienen este tipo de diferenciación, además de crear la interfaz y adaptar las clases existentes para que encajen en esta.

b) **Esfuerzo:** Medio.

c) **Impacto:** Bajo.

d) **Dependencias:** Alto.

10.3.3.2 Manejo de Jobs.

e) **Descripción:** Nos parece pertinente que no sea la clase `ip4jvm.javafwrk.NetStack` quien procese los jobs de principio a fin, ya que por ejemplo, podría tener un pool de threads y que sean éstos los responsables del procesamiento. La única responsabilidad de la clase `ip4jvm.javafwrk.NetStack` entonces sería administrar el pool de threads.

   De esta manera se está desacoplando y delegando responsabilidades, además de aprovechar el paralelismo para una mejor performance general. La idea sería que el `ip4jvm.javafwrk.NetStack` permaneciera dormido y despierte cuando llegue algún `ip4jvm.javafwrk.Job`, y que asignara el mismo a el thread correspondiente (habría un thread por cada `ip4jvm.javafwrk.NetLevel`).

f) **Esfuerzo:** Alto.
   Esta claro que el esfuerzo necesario para realizar este cambio es de Alto, ya que incluye el manejo de threads y todo los dolores de cabezas del paralelismo que ello conlleva.

g) **Impacto:** Alto.

h) **Dependencias:** Alto.

10.3.3.3 Métodos de la Clase ip4jvm.javafwrk.NetLevel.

a) **Descripción:** Los métodos `getClassificator(indx)`, `getLayer(indx)` y `getApplication(indx)` reciben un índice como parámetro, a partir de este índice iteran i veces sobre la lista correspondiente y devuelven el i-ésimo objeto de la misma. Estos métodos son usados únicamente en el método `ProcessIncoming()` donde se itera con un índice y se van llamando estos métodos pasando este mismo índice como parámetro. Esto es notoriamente ineficiente, y la solución que proponemos es la de eliminar dichos métodos y utilizar iteradores en el `ProcessIncoming()` para recorrer cada una de las listas, lo cuál mejoraría el orden de ejecución del algoritmo.

b) **Esfuerzo:** Bajo.

c) **Impacto:** Medio.

d) **Dependencias:** Bajo.

10.3.3.4 ip4jvm.javafwrk.TimerServer.

a) **Descripción:** Hoy por hoy todo lo que tiene que ver con los timeouts de vencimientos,

ya sean de fragmentos, caché de tablas, aplicaciones, MTU, neighbor discovery, etc, se tratan mediante la constante verificación, a la hora de procesarlos, de si ha vencido su tiempo o no, por lo cuál se entra en una bifurcación donde se tratan ambas posibilidades (que haya vencido o que no).

A nuestro parecer esto es poco eficiente y a su vez un poco confuso, nuestra propuesta es aplicar el patrón de diseño *Singleton* y crear una clase llamada `ip4jvm.javafwrk.TimerServer`, donde cada vez que se desee controlar algo que venza, se agende en este objeto. Por ejemplo, supongamos que debemos tratar un fragmento que debe vencer en cierto tiempo, entonces agregamos un temporizador en nuestro `ip4jvm.javafwrk.TimerServer` que apunte al fragmento, que contenga el nombre del objeto y del método a ejecutar y eventualmente una lista de parámetros.

Luego de hacer esto podremos olvidarnos y trabajar normalmente con el fragmento. Una vez que el tiempo preestablecido transcurra, el temporizador se activará y el `ip4jvm.javafwrk.TimerServer` sabrá que método ejecutar, lo cuál podría, por ejemplo, resolver mediante reflection.

Debemos aclarar que será necesario evitar problemas de concurrencia que podrían introducirse a partir de este nuevo enfoque, ya que si un temporizador se activa y quiere, por ejemplo, eliminar un fragmento con el cuál justo se está trabajando, debería postergarse esta acción para no obtener resultados inesperados.

b) **Esfuerzo:** Alto.
c) **Impacto:** Alto.
d) **Dependencias:** Alto.

10.3.4 Clases.

En esta sección listaremos las clases que nos parecen que deben sufrir ciertas modificaciones. Para cada clase, como ya explicamos, listamos tres categorías de cambios (Definición, Atributos, Algoritmia).

10.3.4.1 ip4jvm.javafwrk.NetStackState.

a) Definición:

- **Descripción:** Esta clase representa a los jobs que aún están pendientes de procesar o que cuyo procesamiento aún no ha sido completado. Por esto pensamos que un nombre más mnemotécnico para esta clase sería, por ejemplo, `ip4jvm.javafwrk.Job`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

10.3.4.2 ip4jvm.javafwrk.Application.

b) Definición:

- **Descripción:** Si bien el nombre realmente hace referencia a aplicaciones que tiene o se brindan, nos parece que el término *Applications* en inglés puede llevar a confusión ya que puede confundirse con código que corre en capa de aplicaciones, cuando esto no es así. Es por esto que proponemos cambiarlo a algún otro como puede serlo `ip4jvm.javafwrk.Service`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

10.3.4.3 ip4jvm.javafwrk.NetManagerRead.

a) Algoritmia:

- **Descripción:** Pensamos que no es apropiado el uso del método `Java_ip4jvm_javafwrk_NetManagerRead_connect()` pues realiza un busy wait. Por otra parte, hay un método llamado `loop()` implementado en la librería `libpcap` que no realiza busy waits.
- **Esfuerzo:** Medio.
- **Impacto:** Alto.
- **Dependencias:** Bajo.

10.3.4.4 ip4jvm.javafwrk.NetParameters.

a) Definición:

- **Descripción:** Eliminar la extensión a `java.util.LinkedList`. Proponemos que los parámetros se mantengan en atributos del tipo `java.util.HashMap` referenciados por IP, por MAC por igualdad, por MAC match y por Iface. Esto se debe a que son constantemente accedidos y mantener tal estructura beneficiaría mucho a los algoritmos que implementa la clase.
- **Esfuerzo:** Alto.
- **Impacto:** Alto.
- **Dependencias:** Alto.

b) Atributos:

- **Descripción:** Poner un atributo que apunte al `ip4jvm.javafwrk.NetParameter` por defecto. Con esto nos aseguramos que haya únicamente uno y sacamos el atributo `isDefault` que ahorraría memoria pues se repite innecesariamente en todos los `ip4jvm.javafwrk.NetParameter`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

c) Algoritmia:

- **Descripción:** En los métodos `hasThisScope()` y `getIface()` se itera con índices, deberían eliminarse y usarse iteradores en donde hoy se las llama.
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.

- **Dependencias:** Medio.

10.3.4.5 ip4jvm.javafwrk.NetStack.

a) Atributos:

- **Descripción:** el atributo `toSend` se usaba posiblemente antes, pero por lo visto ya no es necesario en la actual implementación, por lo cuál proponemos eliminarlo de la clase .
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

b) Algoritmia:

- **Descripción:** Como ya explicamos, deberían tratarse los distintos `ip4jvm.javafwrk.Jobs` mediante la asignación de los mismos a hilos y no como se realiza ahora. Por lo cuál se contaría con una cantidad constante de hilos a los cuáles se les van asignando `ip4jvm.javafwrk.Jobs`, y al finalizar avisan al padre de la finalización y se duermen esperando un nuevo trabajo.
- **Esfuerzo:** Alto.
- **Impacto:** Alto.
- **Dependencias:** Alto.

- **Descripción:** En los métodos `getNextToSend()` y `run()` se usa código como el siguiente:

```
toSend.getFirst();
toSend.remove(0);
```

  el cuál podríamos cambiar por `toSend.removeFirst()` y reducir la cantidad de líneas innecesarias en el código.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

- **Descripción:** En el método `run()` hay varias iteraciones de este tipo:

```
while ((iter.hasNext())&&(ns.nextProtocol>0)){
```

```
        Integer level = (Integer)iter.next();
        if ((level.intValue()) == ns.level){
           NetLevel netLevel = (NetLevel)levels.get(level);
           ns = netLevel.processIncoming(ns);
           ns.level ++;
        }
    }
```

donde se itera hasta encontrar el nivel. Se podrían buscar derecho el nivel para que lo procese ya que se tienen en un hash, por lo cuál el código resultante (notablemente más eficiente) sería:

```
    while ((levels.containsKey(ns.level)) && (ns.nextProtocol>0)){
        NetLevel netLevel = (NetLevel)levels.get(ns.level);
        ns = netLevel.processIncoming(ns);
        ns.level ++;
    }
```

- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Medio.

10.3.4.6 ip4jvm.javafwrk.NetLevel.

a) Definición:

- **Descripción:** Debería eliminarse la extensión a `java.util.LinkedList` por parte de esta clase. Por lo explicado en la sección Criterios Generales.
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

b) Atributos:

- **Descripción:** Deberían agregarse dos nuevos atributos que sean del tipo lista, uno para las `ip4jvm.javafwrk.NetLevel.ItemNetLevelLayer` y otra para las `ip4jvm.javafwrk.NetLevel.ItemNetLevelApplications`. Hoy por hoy se tratan ambas clases en una sola lista, pero sería muy útil tenerlos en listas separadas. Esto supone un esfuerzo Alto, ya que cambiarían muchos algoritmos y suponemos que la dificultad no radica en la complejidad del cambio en sí, la cuál es baja, sino a la cantidad de ocurrencias del mismo.
- **Esfuerzo:** Alto.
- **Impacto:** Medio.
- **Dependencias:** Medio.

- **Descripción:** Sería de utilidad agregar en la clase un puntero al `ip4jvm.javafwrk.NetLevel.ItemNetLevelLayer` por defecto en vez de tener el atributo booleano `isDefault` repetido en todos los elementos.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

- **Descripción:** Agregar un atributo que sea el nivel del `ip4jvm.javafwrk.NetLevel`, lo cuál nos permitiría mejorar el método `getLevel()`, que a nuestro entender, hoy por hoy tiene un funcionamiento erróneo ya que delega la responsabilidad de saber el nivel a clases que no deberían saberlo.
- **Esfuerzo:** Bajo.
- **Impacto:** Medio.
- **Dependencias:** Alto.

10.3.4.7 ip4jvm.javafwrk.NetConfigs.

a) Definición:

- **Descripción:** Eliminar la extensión a `java.util.LinkedList` (ver sección Criterios Generales).
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

b) Atributos:

- **Descripción:** Los atributos están declarados como de visibilidad de paquete, a nuestro entender deberían ser privados, lo cuál supondría agregar métodos `get()` y `set()` para todos los atributos, además de cambiar las partes de código donde se acceda directamente a los mismos.
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

c) Algoritmia:

- **Descripción:** Los atributos `ipAddresses` y `macAddresses` son listas, sin embargo se

implementan los métodos `get(indx)` para las dos donde se pasa un índice y se itera hasta obtener el i-ésimo elemento. Lo mismo con los métodos `set(indx, val)` donde además del valor que deberá tomar elemento, se pasa el lugar de la lista donde se encuentra el elemento al cuál se le desea cambiar su valor. Estos métodos deberían dejar de existir ya que son invocadas siempre dentro de ciclos que iteran con un índice.
En cada punto donde se invoquen, debería sustituirse la iteración sobre el índice por un iterador que recorra y obtenga o asigne un valor al elemento, reduciendo así el orden notoriamente.

- **Esfuerzo:** Alto.
- **Impacto:** Alto.
- **Dependencias:** Alto.

10.3.4.8 ip4jvm.net.applications.SocketManager.

a) Atributos:

- **Descripción:** El atributo `sockets` debería ser un `java.util.HashMap` indexado por identificador de envío en vez de una `java.util.LinkedList`, para mejorar los accesos, ya que se usan con mucha frecuencia. Esto cambiaría alguna lógica externa donde se debía conocer el lugar de la lista donde estaba el socket abierto para poder acceder al mismo y conllevaría un esfuerzo Medio.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Bajo.

b) Algoritmia:

- **Descripción:** En el método `add()` existe el siguiente código:

```
ItemSocket i = new ItemSocket();
if (port <= 0) port = getPortAvailable();
i.port = port;
i.type = type;
i.socket = socket;
i.state = ItemSocket.STATE_CREATED;
i.netconfig = new NetConfigs();
```

el cuál debería ser sustituido por un constructor que reciba por parámetro el valor de los atributos.

- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.9 ip4jvm.net.DatagramSocket.

a) Atributos:

- **Descripción:** Pensamos que sería una buena opción mantener en un atributo la instancia del `ip4jvm.net.applications.SocketManager` ya que se utiliza mucho y está constantemente pidiéndose al `ip4jvm.net.NetworkFactory`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

b) Algoritmia:

- **Descripción:**

```
(SocketManager)NetworkFactory.getNetwork().getApplication("socketmanager")
```

En el código anterior debería usarse una constante, posiblemente definida en la clase `ip4jvm.net.Network` que tenga el valor "socketmanager".

- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.10 ip4jvm.net.Network.

a) Definición:

- **Descripción:** Deberían ponerse los nombres de las aplicaciones, de los protocolos y la ruta del archivo de configuración como constantes, ya que se usan repetidas veces y en cada uso se vuelven a reescribir dichos valores.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

b) Algoritmia:

- **Descripción:**

```
if (first){
   stack.addConfiguration(name, mac, myip, true, true);
```

```
      first = false;
   }else{
      stack.addConfiguration(name, mac, myip, true, false);
   };
```

En el código anterior, si la variable “first” tiene el valor “true” se invoca al método `stack.addConfiguration()` con el último de los parámetros en “true” y luego se asigna el valor “false” a la variable “first”, y si “first” vale “false” se hace lo mismo pero con el último parámetro en “false”. Claramente este código puede sustituirse por las siguientes sentencias con el fin de tener un código más legible y con menos líneas innecesarias:

```
   stack.addConfiguration(name, mac, myip, true, first);
   first = false;
```

- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.11 ip4jvm.net.applications.Ping6.

c) <u>Algoritmia:</u>

- **Descripción:** En el método `rcv()` encontramos el siguiente código:

```
   for(int i=0; i<netstate.netconfig.size(); i++){
      icmpRey.setCheckSum(checksum.getCheckSum(
                (Inet6Address)netstate.netconfig.getIP(i),
                (Inet6Address)ipv6.getSourceAddress(),
                icmpRey.getLenght(),ICMPv6Type.ICMPv6,
                icmpRey.toBytessinCheckSum()));
      NetConfigs net = new NetConfigs();
      net.add(netstate.netconfig.getMAC(i),
      netstate.netconfig.getIP(i));
      NetStackState ns = new NetStackState(ICMPv6Type.ICMPv6, 3,
                          new NetMessage(icmpRey.toBytes()),
                          NetStackState.TYPE_OUTMSG, net,

            ipv6.getSourceAddress());
      synchronized (jobs){
         jobs.add(ns);
      }
   }
```

el cuál debería se sustituido por un `foreach()` para obtener un código más prolijo (ver sección Criterios Generales).

- **Esfuerzo:** Bajo.

- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.12 ip4jvm.net.applications.NeighborDiscovery.

a) <u>Algoritmia:</u>

- **Descripción:** Tanto en el método `rcv()` como en `enableInterfaces()` se usan reiteradamente índices sobre listas dentro de un ciclo, lo cuál ya marcamos varias veces a lo largo de este documento que redunda en una ineficiencia innecesaria. Se deberían cambiar por iteradores.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Media.

- **Descripción:** El siguiente código:

```
LinkedList prefixes = icmpRA.options.getPrefixInfo();
for (int i=0; i<prefixes.size(); i++){
      PrefixInfo prefixInf = (PrefixInfo)prefixes.get(i);
      if (prefixInf.getOnLinkFlag()){
         long lifetime = prefixInf.getValidLifeTime();
            if (lifetime>0){
                  prefixes.add(prefixInf);
            }else{
                        prefixes.remove(prefixInf);
             }
         }
      }
```

  contiene, a nuestro entender, una falta lógica ya que si se elimina un elemento se saltea al siguiente en la próxima iteración, debido a que cambiará su posición relativa en la lista y el índice igual es aumentado.
- **Esfuerzo:** Bajo.
- **Impacto:** Alto.
- **Dependencias:** Bajo.

10.3.4.13 ip4jvm.net.protocols.headers.EthernetIIHeader.

a) **Definición:**

- **Descripción:** Deberíamos colocar valores constantes utilizados para los tamaños de los campos en definiciones de constantes a nivel de la clase.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.14 ip4jvm.net.protocols.ICMPv6.

a) **Definición:**

- **Descripción:** El método `toBytessinCheckSum()` contiene la palabra "sin" que pertenece al idioma español, ya que su significado en inglés no tiene que ver con lo que se desea denotar. El nombre de dicho método debería cambiarse.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.15 ip4jvm.net.protocols.headers. UDPProtocolHeader.

a) **Definición:**

- **Descripción:** Definir constantes de tamaño de puerto, checksum, largo.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

- **Descripción:** También aquí se encuentra el método `toBytessinCheckSum()`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

b) Atributos:

- **Descripción:** Existe la definición de un atributo entero llamado `lenght`, que debería ser cambiado por `length`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

10.3.4.16 ip4jvm.net.protocols.extras.FragmentIPv6.

a) Algoritmia:

- **Descripción:** Existen en esta clase 6 instancias donde se itera sobre listas usando índices. Deberían usarse iteradores.
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Alto.

10.3.4.17 ip4jvm.net.protocols.extras.FragmentsIPv6.

a) Definición:

- **Descripción:** Si se cambia por una estructura (por ejemplo un `java.util.HashMap`) a la cuál se pueda acceder por destino, por origen y por identificador se bajarían los órdenes promedio drásticamente, y dado a la frecuencia del uso de estos algoritmos, la performance del software en general sería notoriamente beneficiada.
- **Esfuerzo:** Alto.
- **Impacto:** Medio.
- **Dependencias:** Bajo.

b) Algoritmia:

- **Descripción:** Existen en esta clase 5 pares de métodos que hacen exactamente lo mismo uno a uno. Deberíamos eliminar 5 de ellas y en donde estas se llamen cambiarla por su equivalente.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Medio.

- **Descripción:** En vez de usar constantemente el método `isFragmentTimeOut()`, deberíamos colocar los fragmentos en el `ip4jvm.javafwrk.TimerServer` (ver sección 10.3.3.4) así cuando el fragmento venza el `ip4jvm.javafwrk.TimerServer` sabrá que rutina ejecutar y podremos así despreocuparnos de preguntar continuamente si se venció el tiempo para el mismo.
- **Esfuerzo:** Alto.
- **Impacto:** Alto.
- **Dependencias:** Bajo.

- **Descripción:** Existen a lo largo de la clase iteraciones por índices, condiciones de paradas omitidas y algunos ciclos que sería conveniente modificarlos por sentencias `foreach()`.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Alto.

10.3.4.18 ip4jvm.net.applications.neighDisco. NeighborCache.

a) <u>Algoritmia:</u>

- **Descripción:** También en este caso debería usarse el `ip4jvm.javafwrk.TimerServer`, esbozado en el cambio de diseño referido en la sección 10.3.3.4 del presente documento, y así tratar de mejor manera los vencimientos de los `ip4jvm.net.applications.neighDisco.processNeighbors`.
- **Esfuerzo:** Alto.
- **Impacto:** Alto.
- **Dependencias:** Bajo.

- **Descripción:** El método `printf()` debería cambiarse por uno llamado `toString()` y que el invocante imprima, si desea, el `java.lang.String` retornado y a su vez debería usarse un `foreach()` en el algoritmo de este método y no un `for()` común como se hace hoy en día.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.19 ip4jvm.net.protocols.headers.icmps.ICMPv6RouterSolicitation.

a) Algoritmia:

- **Descripción:** En el método `toBytesPayload()` encontramos el siguiente código:

```
for (int i=0; i<options.size(); i++){
   len = options.getLenght(i);
   System.arraycopy(options.toBytes(i), 0, b, offset, len);
   offset = offset + len;
};
```

  donde nuevamente se usa índices en vez de iteradores.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.20 ip4jvm.net.protocols.headers.icmps.ICMPv6NDOptions.

a) Definición:

- **Descripción:** Remover la extensión a la clase `java.util.LinkedList` y agregarle un atributo `java.util.HashMap` indexado por tipo de opción, lo cuál mejora los órdenes promedio de ejecución de aquellos algoritmos donde se accede a una opción determinada.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Alto.

- **Descripción:** Corregir el nombre del método `getLenght()` a `getLength()`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

b) Atributos:

- **Descripción:** Agregar un atributo del tipo `java.util.LinkedList` para poder tener

los `ip4jvm.net.applications.neighDisco.PrefixInfo` en una lista aparte.

- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

c) **Algoritmia:**

- **Descripción:** Todos los métodos de esta clase obtienen un elemento y habiéndolo ya obtenido no paran la iteración hasta que termina el índice alcance el tamaño total de la lista. Debería parar antes, para lo cuál deberían agregarse dichas condiciones de parada.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Medio.

10.3.4.21 ip4jvm.net.protocols.ipv6.routingHeaders.RoutingHeader.

a) **Definición:**

- **Descripción:** Poner las constantes como tales.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

b) **Atributos:**

- **Descripción:** Sería más adecuado usar un `java.util.Vector` para guardar los datos en vez de una lista para mejorar los accesos por posición y aprovechando que tiene una longitud máxima conocida.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Medio.

10.3.4.22 ip4jvm.net.applications.neighDisco.RouterList.

a) **Definición:**

- **Descripción:** Remover la extensión a la clase `java.util.LinkedList` y hacer que sea un `java.util.LinkedHashMap` indexado por `java.net.Inet6Address` para

mejorar el orden promedio de los accesos.

- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.23 ip4jvm.net.applications.neighDisco.PrefixInfoList.

a) Definición:

- **Descripción:** Remover la extensión a la clase `java.util.LinkedList` y hacer que sea un `java.util.HashMap` indexado por `java.net.Inet6Address` ya que lo único que se hace con esta lista es agregar elementos, quitarlos y ver si alguno cumple una propiedad. Claramente el cambio propuesto mejora el orden promedio de los algoritmos implementados por la clase.
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.24 ip4jvm.net.applications.neighDisco.DestinationCache.

a) Definición:

- **Descripción:** Remover la extensión a la clase `java.util.LinkedList` y hacer que sea un `java.util.LinkedHashMap` indexado por `java.net.Inet6Address` para mejorar el orden promedio de los accesos, ya que el caché, como concepto, tiene como principal objetivo optimizar los accesos.
- **Esfuerzo:** Medio.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.25 ip4jvm.net.applications.neighDisco.RoutingTableIPv6.

a) Algoritmia:

- **Descripción:** En el método `getIface()` vemos el siguiente código:

```
for (int i=0; i<table.size(); i++){
   RoutingEntryIPv6 entry = (RoutingEntryIPv6)table.get(i);
   if (entry.destination.equals(destination)){
      return entry.iface;
   }else if (scopeD == Ipv6Utils.getScope(entry.destination)){
      possible.add(entry);
   }
}
```

Este es un nuevo caso donde deberían usarse iteradores.

- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

- **Descripción:** En el mismo método vemos que se recorren todas las entradas de la tabla agregando todas las posibles a una lista, sin embargo se devuelve la primera de la lista, y no la lista completa. Por esto pensamos que debería obviarse la confección de esta lista y gracias a esto poder parar la iteración ni bien encuentro la primera posible.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

10.3.4.26 ip4jvm.net.protocols.Ipv6Protocol.

a) Definición:

- **Descripción:** Corregir el nombre del método `getPayloadLenght()` a `getPayloadLength()`.
- **Esfuerzo:** Bajo.
- **Impacto:** Bajo.
- **Dependencias:** Bajo.

b) Algoritmia:

- **Descripción:** Se utilizan muchos ciclos que iteran sobre el índice de listas, deberían cambiarse todas estas ocurrencias.
- **Esfuerzo:** Medio.
- **Impacto:** Medio.
- **Dependencias:** Medio.

- **Descripción:** Vemos que el método `run()` hace *busy waiting* lo cuál debe ser solucionado mediante el uso de *threads*.
- **Esfuerzo:** Alto.
- **Impacto:** Alto.
- **Dependencias:** Bajo.

- **Descripción:** Debería usarse el servidor de *timers* anteriormente propuesto (`ip4jvm.javafwrk.TimeServer`) para el procesamiento de las actualizaciones al `ip4jvm.net.applications.neighDisco.NeighborCache`.
- **Esfuerzo:** Alto.
- **Impacto:** Medio.
- **Dependencias:** Medio.

10.3.5 Conclusión:

En esta sección se hace un resumen más gráfico de los cambios realizados, para que el lector tenga una visión más global y genérica de los mismos. Se comienza por detallar el uso de threads propuesto para mejorar aspectos de performance lo cuál ayudará a entender un poco más la solución propuesta.

Luego se muestra una tabla con datos de cada uno de los cambios que ayuden a entender el orden lógico de los mismos, así como facilitar su priorización.

Por último se adjuntará un orden de ejecución de cambios que se deriva a partir de estos datos.

10.3.5.1 Uso de Threads.

Sabemos que el costo, hablando en términos de tiempos, de crear un thread es alto debido al cambio de contexto, por lo cuál, si bien hay problemas de enfoque que deben ser resueltos mediante el uso de los mismos, no deberá abusarse en su utilización si queremos mantener un software performante.

En nuestra solución propuesta existirán los siguientes threads, los cuáles nos parecen los básicos imprescindibles para mejorar el producto existente y obtener una mejora significativa en la performance:

a) NetStack: Deberá ser un único thread que pueda dormir y despertar únicamente cuando reciba un Job para poder evitar el busy waiting en el cuál se incurre actualmente. A su vez, como explicamos en la sección 10.3.3.2 de este apéndice, contará con un pool de threads (uno por cada NetLevel) entre los cuáles distribuirá los jobs según corresponda antes de volverse a dormir a la espera del próximo Job. La cantidad de threads con la que contará el pool, al corresponderse con la cantidad de niveles, será constante y nunca se crearán nuevos, ni se eliminarán, ahorrando de esta manera el tiempo necesario para su creación.

b) NetLevel: Serían los threads que describimos arriba y que procesarían los Jobs que el NetStack les asigne.

c) ManagerRead: Consideramos que debe ser un thread, ya que está continuamente leyendo de la red, y en realidad debería despertarse únicamente cuando llega algo a la misma. A su vez, el hecho de tener un thread a parte, nos permite poder leer paquetes grandes de la red sin que el NetStack frene su ejecución hasta que se le devuelva el control permitiendo esto un mejor provecho del paralelismo.

d) ManagerWrite: De la misma manera, el manejo de un thread para el ManagerWrite nos permite que su ejecución sea asincrónica con respecto al NetStack y por lo tanto no dejar a este último en una espera innecesaria. También nos evitaría el uso de un busy waiting para su algoritmia.

e) TimerServer: En este ya hemos ahondado en las anteriores secciones e inclusive cuenta con una sección propia en este apéndice (10.3.3.4).

10.3.5.2 Tabla de Cambios.

En la presente sección se ilustran los cambios a realizados, su esfuerzo e impacto, así como la dependencia que tiene el resto del código con los mismos.

La tabla que se coloca a continuación tiene una fila por cada uno de los cambios que hemos sugerido en las secciones anteriores. Las columnas se corresponden con el esfuerzo necesario para realizarlo, el impacto que éste tiene sobre el software en general, la interdependencia que tiene con respecto a otras secciones de código y una bandera que indica si el cambio debe necesariamente realizarse al inicio o si esto no es necesario.

Las tres primeras columnas pueden tomar valores entre (1, 2, 3) que equivalen a (Bajo, Medio y Alto). Mientras que la columna “Inicio” tendrá valores entre Si y No.

Es válido destacar que para medir de alguna forma la dependencias de un cambio, nos hemos ayudado de la búsqueda de referencias que provee la herramienta Eclipse. El criterio utilizado es que si el método/atributo/clase en cuestión es referenciado menos de cinco veces tiene baja dependencia, si las referencias están entre las cinco y las nueve será media y si excediesen las diez será alta.

		Esfuerzo / complejidad	Impacto	Dependencias	Inicio
Diseño	1	2	1	3	No
	2	3	3	3	Si
	3	1	2	1	No
	4	3	3	3	Si
Classes	1	1	1	3	No
	2	1	1	3	No
	3	2	3	1	Si
	4.a	3	3	3	Si
	4.b	1	1	1	No
	4.c	2	1	2	No
	5.a	1	1	1	No
	5.b	3	3	3	Si
	5.c	1	1	1	No
	5.d	2	2	2	No
	6.a	2	1	3	Si
	6.b	3	2	2	No
	6.c	1	1	2	No
	6.d	1	2	3	Si
	7.a	2	1	3	Si
	7.b	2	1	3	No
	7.c	3	3	3	No
	8.a	2	2	1	No
	8.b	1	1	1	No
	9.a	1	1	2	No
	9.b	1	1	1	No
	10.a	1	1	3	No
	10.b	1	1	1	No
	11	1	1	1	No
	12.a	2	2	2	No
	12.b	1	3	1	No
	13	1	1	1	Si
	14	1	1	1	No
	15.a	1	1	2	Si
	15.b	1	1	2	No
	15.c	1	1	2	No
	16	2	1	3	No
	17.a	3	2	1	No
	17.b	2	2	2	Si
	17.c	3	3	1	No
	17.d	2	2	3	No
	18.a	3	3	1	No
	18.b	1	1	1	No
	19	1	1	1	No
	20.a	2	2	3	Si
	20.b	1	1	2	No
	20.c	2	1	1	No
	20.d	2	2	2	No
	21.a	1	1	1	Si
	21.b	1	1	2	No
	22	2	1	1	Si
	23	2	1	1	Si
	24	2	1	1	Si
	25.a	1	1	1	No
	25.b	1	1	1	No
	26.a	1	1	1	No
	26.b	2	2	2	No
	26.c	3	3	1	No
	26.d	3	2	2	No

Ilustración 35: Tabla de clasificación de cambios

Para llenar la columna "Inicio" consideramos aquellos cambios que afectarán no solo al método en cuestión, sino que cambiarán a todos aquellos puntos del código desde donde se invoquen. Esto se debe a que de no hacer estos cambios al comienzo, podrá incurrirse en la desventura de implementar nuevas cosas que usen estos métodos y luego, al efectuarse los cambios deban cambiarse también las nuevas lineas de código, redundando en un re trabajo innecesario.

En la anterior tabla se ve que 35 de los 58 cambios identificados es de alto impacto en el aplicativo, lo cuál supone que la realización de los 58 cambios mejorarán el desempeño de la herramienta. Veamos a continuación, gráficamente, la porción de los cambios que corresponde a cada grado de impacto.

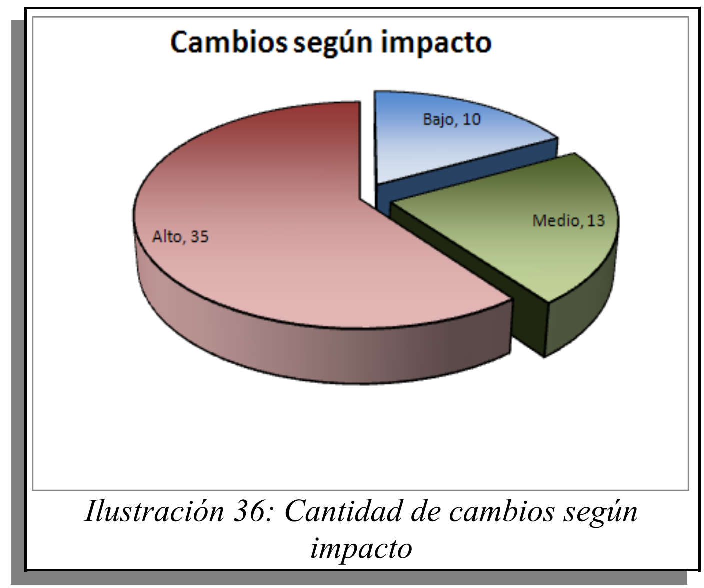


Ilustración 36: Cantidad de cambios según impacto

Como decíamos, 48 de los 58 cambios identificados son de alto o medio impacto; de la mano de este dato es que 47 de los 58 son de alta complejidad en su realización. Veamos como se discriminan los cambios en función de este segundo atributo (la complejidad):

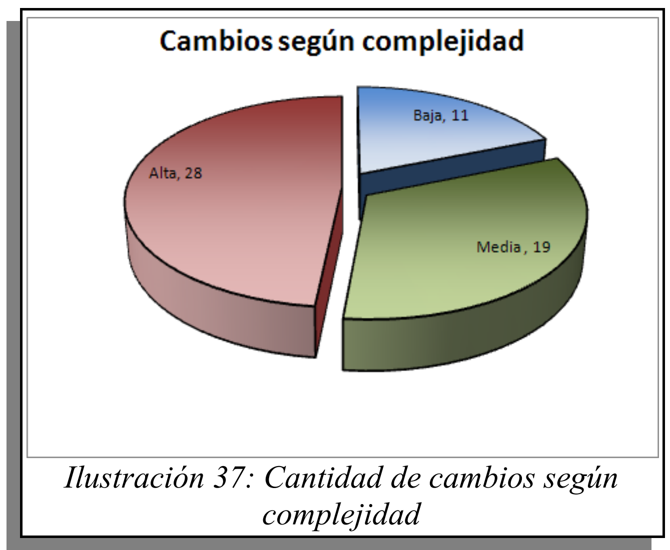


Ilustración 37: Cantidad de cambios según complejidad

Por último veamos como se reparten las cantidades de cambios en función del grado de dependencia:

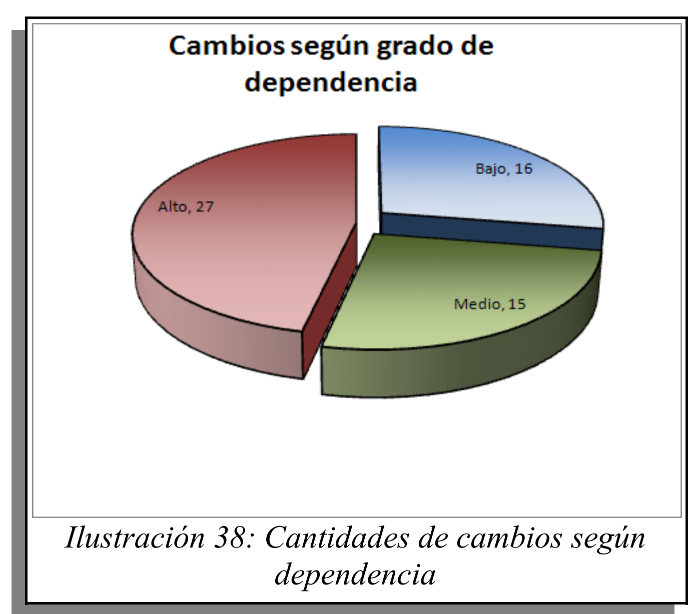


Ilustración 38: Cantidades de cambios según dependencia

Como conclusión general en base a estas tres gráficas, podemos decir que la mayoría de los 58 cambios identificados tiene una elevada complejidad y un alto grado de dependencias, lo cuál no es un dato menor si se quiere cumplir con los tiempos estimados en la planificación inicial. Pero en contrapartida, podemos observar que estos cambios tendrán un impacto alto, lo cuál hace que el esfuerzo invertido vea sus réditos en un mejor producto obtenido.

10.3.5.3 Ejecución de Cambios.

Los cambios detectados son 58, a efectos de tener una idea más precisa de la evolución temporal de los mismos, resulta necesario prever que durante la ejecución de los mismos, probablemente surjan otros que hoy no hemos percibido y que alteren los planes que podamos esbozar al día de la fecha.

De los cambios identificados, hemos resaltado 16 como iniciales, lo que implica que deben incluirse en las primeras fases de cambios. Ni bien se finalicen estos cambios, pensamos que lo correcto es seguir adelante con aquellas modificaciones que tengan un alto impacto, priorizando entre éstos los que tengan esfuerzo más bajo. Una vez ejecutados estos cambios se seguirán con las de impacto medio y por último con las de bajo impacto, siempre manteniendo el criterio de ejecutar las de bajo esfuerzo en primera medida.

Hechas estas aclaraciones de criterio procedemos a detallar las fases de ejecución de cambios. En cada fase se listan los cambios a realizar en orden de ejecución según conveniencias de esfuerzo, impacto y dependencias:

a) Fase I (Cambios Iniciales de Diseño)
- 2.2
- 2.4

b) Fase II (Cambios Iniciales en clases con impacto alto)
- 3.3
- 3.4a
- 3.5b

c) Fase III (Cambios Iniciales en clases con impacto medio o bajo)
- 3.6d
- 3.17b
- 3.20a
- 3.13
- 3.15a
- 3.21a
- 3.6a
- 3.7a
- 3.22
- 3.23
- 3.24

d) Fase IV (Cambios Generales de alto impacto)
- 3.12b

- 3.7c
- 3.17c
- 3.18a
- 3.26c

e) Fase V (Cambios Generales de impacto medio)

- 2.3
- 3.5d
- 3.8a
- 3.12a
- 3.17d
- 3.20d
- 3.26b
- 3.17a
- 3.6b
- 3.26d

f) Fase VI (Cambios de complejidad baja, dependencias bajas e impacto bajo)

- 3.4b
- 3.5a
- 3.5c
- 3.8b
- 3.9b
- 3.10b
- 3.11
- 3.14
- 3.18b
- 3.19
- 3.25a
- 3.25b
- 3.26a

g) Fase VII (Cambios con impacto bajo y no triviales)

- 2.1
- 3.1
- 3.2
- 3.4c
- 3.6c
- 3.7b

- 3.9a
- 3.10a
- 3.15b
- 3.15c
- 3.16
- 3.20b
- 3.20c
- 3.21b

Como primer apunte importante de esta división de fases, debemos resaltar que todos aquellos cambios identificados como iniciales se encuentran en las tres primeras fases, por lo cuál al finalizar estas etapas ya se habrá cumplido con estas modificaciones tal y como se había estipulado de antemano.

Otro aspecto a notar es que las primeras fases intentan dar cumplimiento a las modificaciones de alto impacto, mientras que a medida que se avanza en ellas la cantidad de modificaciones medias y bajas va aumentando. En la siguiente gráfica vemos la evolución de los porcentajes de cambios de impacto alto, medio y bajo según la fase:

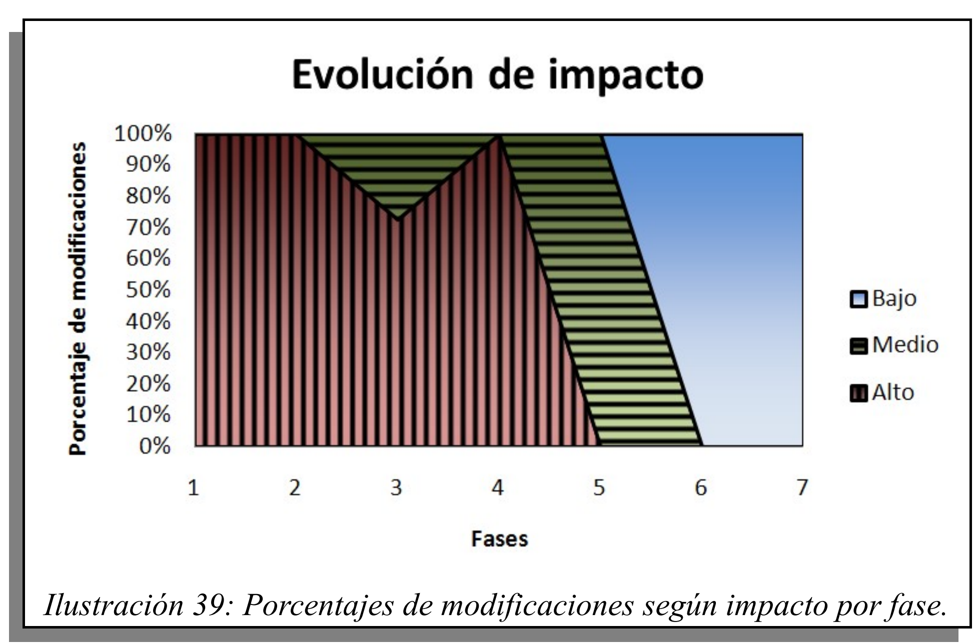


Ilustración 39: Porcentajes de modificaciones según impacto por fase.

Así mismo, esto llevó a que los primeros cambios tengan esfuerzo alto, ya que la mayoría de los cambios de alto impacto así lo demandaban. Sin embargo, una vez que los cambios de alto impacto comenzaban a finalizarse, dando lugar a la ejecución de los cambios de medio impacto, se priorizaron entre estos últimos los de menor complejidad. Vemos esto reflejado en la siguiente gráfica, que análogamente a la anterior, muestra la evolución, durante el transcurso de las fases, de la complejidad de los cambios:

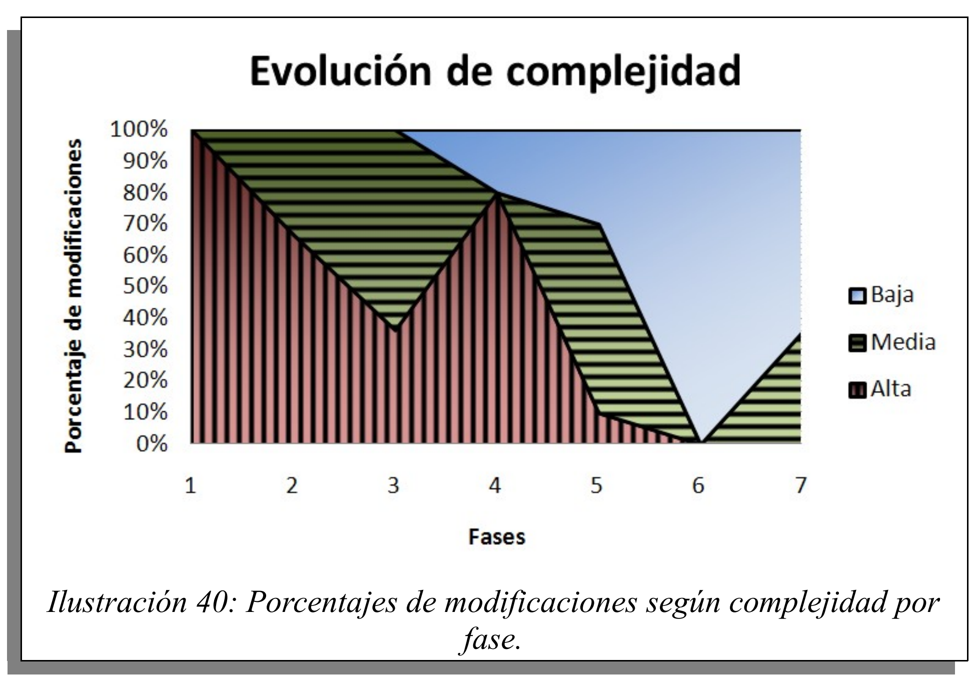


Ilustración 40: Porcentajes de modificaciones según complejidad por fase.

Aquí se aprecia lo antedicho, la primera fase tiene un 100% de cambios de alta complejidad, por tratarse de aquellos de máximo impacto. Luego ese número comienza a bajar a medida que comienzan a incluirse cambios de impacto medio en las fases, para luego volver a repuntar en la fase 4, donde los cambios de medio impacto que aún restan efectuar son únicamente los de alta complejidad.

Por último presentamos una gráfica similar a las dos anteriores, pero mostrando la evolución de los porcentajes de cambios de cada fase según dependencias con el resto del código. Este dato fue el tercero en orden de prioridad a la hora de decidir el orden de los cambios, por lo cuál se vio muy sujeto a el ordenamiento de los dos campos anteriores (impacto y complejidad).

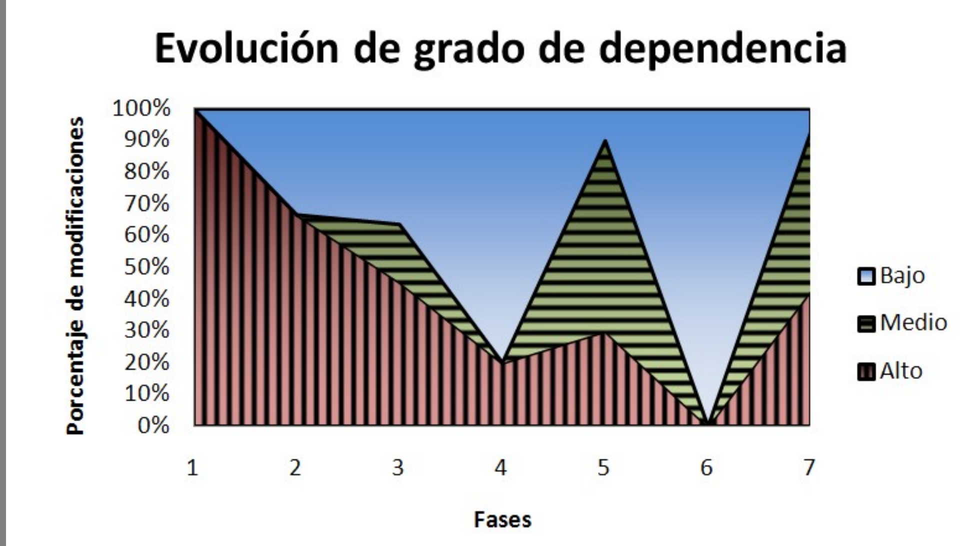


Ilustración 41: Porcentaje de modificaciones según dependencias por fase

Cabe destacar como punto final que no todos los cambios fueron realizados, ya que algunos al transcurrir la fase de modificación de código fueron detectados como irrelevantes para los objetivos estipulados y el tiempo con el que se contaba.

10.4 Implementación TCP

10.4.1 Introducción

En el presente apéndice se detallaran los aspectos referentes a la implementación del protocolo de transporte TCP. Al inicio se explicará la importancia de la implementación del mismo, luego se verán los objetivos deseados de lograr, a continuación se brindarán una serie de conceptos básicos para el buen entendimiento del apéndice, luego de lo cual se verán los principales aspectos del RFC que describe las características requeridas, para luego si explicar los principales detalles del diseño e implementación realizadas. Al final del apéndice se plantearan líneas a seguir para un trabajo futuro.

10.4.2 Motivación

TCP (Transmission Control Protocol) es uno de los protocolos fundamentales de Internet. TCP provee transferencia confiable y ordenada de streams de datos sin perdida ni duplicación de los mismos. Además de esto se encarga de obtener el máximo rendimiento posible de los recursos de red dados los datos a enviar y/o recibir.

TCP brinda la posibilidad de establecer conexiones entre aplicaciones y es altamente utilizado por aplicaciones web, como lo son aquellas que trabajan con HTTP, transferencia de archivos y e-mails. Esto hace de TCP un protocolo extremadamente importante para el funcionamiento de la Internet que conocemos.

10.4.3 Objetivos

El objetivo fundamental es realizar una implementación TCP utilizando el framework IP4JVM y los protocolos ya implementados. Luego de esto integrar la implementación a la máquina virtual Java OpenJDK de manera tal de permitir, mediante el uso de parámetros pasados a la máquina virtual, utilizar la implementación realizada.

10.4.4 Conceptos Básicos

TCP brinda la posibilidad de realizar una conexión a las aplicaciones de la capa aplicación mediante el uso de sockets. Los sockets son la interfaz que brinda TCP (y también UDP) a la capa aplicación para exponer sus servicios, cada socket (o punto de conexión con la capa 5) se identifica por un entero denominado puerto y una dirección IP local. Una conexión queda determinada por un par de sockets (el local y el remoto).

Los servicios brindados por TCP a través de la interfaz básicamente son: establecer una conexión, escuchar en determinado puerto para establecer conexiones, envío de datos, recepción de datos y cierre de conexión. La forma habitual de uso de la conexiones TCP se conoce con el nombre de Cliente – Servidor donde una aplicación requiere servicios de otra que es la proveedora de servicios. Un cliente se puede conectar a un servidor mediante el uso de TCP, pero para esto deberá conocer el puerto y la dirección IP del servidor.

Existen tres tipos de puertos: los denominados bien conocidos (1-1024), los registrados (1025-4999) y los dinámicos/privados (5000 en adelante). Los bien conocidos son definidos por la IANA (Internet Assigned Number Authority) en los cuales escuchan pedidos de conexiones aplicaciones Servidores que brindan servicios estándar como lo son : FTP, HTTP, Telnet, ssh, etc. Los puertos registrados son usualmente puertos efímeros (auto generados por TCP) de los cuales hacen uso las aplicaciones clientes cuando éstas no especifican un puerto local, además son usados por servicios de terceros que registran los puertos. Finalmente los puertos dinámicos/privados pueden ser usados por aplicaciones de usuario.

10.4.5 RFC 793

El RFC por excelencia que describe las funcionalidades y estructuras requeridas por una implementación TCP es el RFC 793. El mismo describe tanto requerimientos de estructuras de datos, interfaces a brindar, así como también definición de algunos de los timers que se pueden utilizar, una máquina de estados que describe las transiciones entre los estados de una conexión y también una descripción detallada de qué se debe hacer frente a los diferentes eventos. Presenta además algunos de los algoritmos utilizados para el correcto funcionamiento del protocolo.

Aunque el RFC 793 cubre muchos detalles, quedan muchos por resolver (debido principalmente a la complejidad de TCP) esto se ve reflejado en los RFCs complementarios posteriormente publicados. Dentro de estos RFCs se encuentra el 1122, el que corrige algunos errores del RFC 793, 1323, el cual presenta algunas extensiones y 2581, que presenta lineamientos para el control de congestión de la red. Dentro de las principales carencias del RFC 793 se encuentra el no brindar ningún tipo de lineamiento a seguir para obtener un adecuado uso de los recursos de red y evitar congestión de la misma.

Existen en la actualidad muchas implementaciones TCP, la mayor parte de la diferencia entre estas implementaciones radica en el algoritmo utilizado para evitar el congestión de la red. Entre los algoritmos más usados para evitar congestión de datos se encuentran: Tahoe, Reno, New Reno, Vegas, Hybla y FAST TCP.

A continuación en esta sección se detallaran los principales aspectos del RFC 793.

10.4.5.1 Formato de Datos

TCP como cualquier otro protocolo principal de una capa de un stack de protocolos, agrega a

los datos que pasan por el mismo, información de control (la cual se denomina cabecera), para administración y comunicación entre los nodos TCP participantes de una conexión. TCP al recibir datos a ser enviados, los secciona en segmentos, los que son utilizados como la unidad de trabajo del protocolo. Cada segmento generado contiene la cabecera TCP que lo identifica y los datos a enviar (en caso que se requiera). Como es de suponer un nodo TCP también deberá poder, a través de una serie de segmentos, re-ensamblar los datos originalmente enviados y brindar dichos mensajes a la aplicación de capa aplicación que corresponda.

A continuación se pasa a detallar el formato del cabezal de TCP:

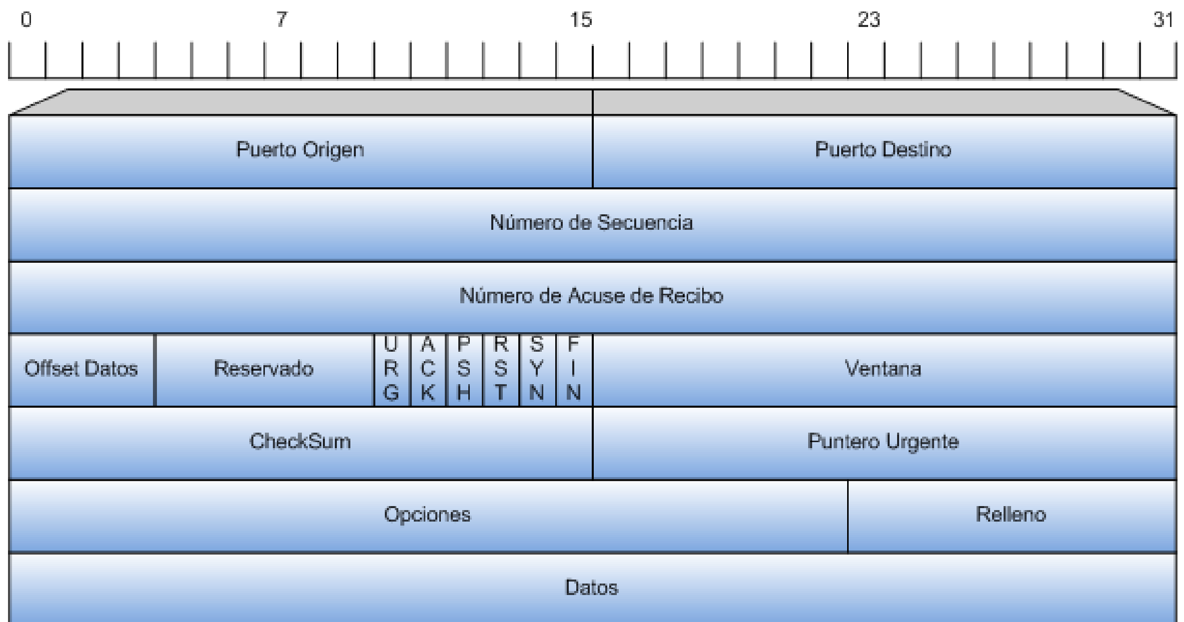


Ilustración 42: Cabezal TCP

- Número de Secuencia

  Este número representa el número de secuencia del primer byte de datos contenido en este segmento. Comúnmente hace referencia al mismo como SN debido a las siglas de su nombre en inglés “sequence number”. En el caso de que la bandera denominada SYN (que será explicada posteriormente) se encuentre seteada a uno, el número representa el número de secuencia inicial de la conexión (ISN: 'initial sequence number' también llamado ISS), en este caso el primer octeto de datos es ISN+1. El ISN se establece al iniciar la conexión, y se obtiene en función del timestamp actual del sistema.

- Número de Acuse de Recibo

  Si el bit de control ACK (“acknowledgement”, explicado más adelante) está inicializado en uno, este campo (número de acuse de recibo, o también llamado simplemente ACK) contiene el valor del siguiente número de secuencia que el emisor del segmento espera recibir. Una vez que una conexión queda establecida,

este número es enviado correctamente inicializado en cada segmento enviado.

- Offset Datos

  Este número representa la cantidad de palabras de 32 bits que ocupa la cabecera de TCP, lo cuál indica dónde comienzan los datos transferidos por el segmento. La cabecera de TCP (incluso una que lleve opciones) esta compuesta siempre un número entero de palabras de 32 bits.

- URG (bandera de datos urgentes)

  Bandera que indica si los datos que contiene el segmento son urgentes. Estos datos son tratados de manera diferente que los normales y al recibir datos con esta bandera inicializada a uno se debería notificar a la aplicación receptora que los datos son urgentes. Esta funcionalidad no es soportada en Java ya que la interfaz de sockets brindada no dispone de indicador alguno sobre este tipo de datos. Para evitar la perdida de estos datos la interfaz brindada por Java permite especificar la opción OOBInline que trata a los datos urgentes como datos normales, en el caso de que esta opción no sea especificada los datos urgentes serán descartados sin ninguna acción adicional. El RFC 793 tiene algunos aspectos no especificados con respecto a como se tienen que administrar estos datos.

- ACK (bandera de acuse de recibo)

  Bandera que indica si el ACK (número de acuse de recibo) dado en el cabezal se debe procesar.

- PSH (bandera de función push)

  Bandera que indica la utilización de la función push. Cuando llega un segmento con esta bandera inicializada en uno indica que los datos de este segmento y los que hayan llegado anteriormente para la conexión y que aun no se hayan entregado al usuario deben ser entregados inmediatamente a este último. Esta bandera modifica el comportamiento estándar de la recepción de segmentos que define que los segmentos serán entregados a la aplicación de usuario destinataria al llenar el buffer de recepción dado, al momento de solicitar la recepción de datos, por la aplicación receptora, o cuando ocurra algún error en la comunicación.

- RST (bandera de función reset)

  Bandera utilizada para reiniciar la conexión. Esta bandera es utilizada principalmente cuando la conexión se encuentra en un estado indebido al recibir un determinado segmento o cuando ocurre algún error en la conexión que amerite abortar la misma. Al recibir esta bandera inicializada en uno la conexión correspondiente debe ser abortada, descartando todos los datos sobre la misma, por parte del receptor del segmento.

- SYN (bandera de inicio de conexión)

  Bandera utilizada para establecer una conexión. La utilización de esta bandera será

explicada en la explicación de la función Open detallada en la sección de interfaces estándares de TCP

- FIN (bandera de fin de conexión)

  Bandera utilizada para finalizar una conexión. El uso de esta bandera se puede apreciar posteriormente en la explicación de la función Close en la sección de interfaces estándares de TCP.

- Ventana

  Cantidad de bytes a partir del actual número de ACK que el emisor del segmento está dispuesto a aceptar. Esto indica la cantidad de espacio en el buffer del receptor que se encuentra actualmente libre. Un nodo de TCP debería controlar la cantidad de información que envía al receptor, dependiendo del valor de ventana que el receptor indique. Este es uno de los métodos para poder evitar realizar un mal uso de los recursos de red existentes.

  A continuación se muestra en la imagen cómo sería un uso simple de la ventana:

Transmisor
Receptor
La aplicación hace una escritura de 2K
Buffer del receptor
0
4K
Vacio
2K
SEQ = 0
2K
ACK = 2048 WIN = 2048
La aplicación hace una escritura de 3K
2K
SEQ = 2048
Lleno
Transmisor bloqueado
ACK = 4096 WIN = 0
La aplicación lee 2K
ACK = 4096 WIN = 2048
2K
El transmisor puede enviar hasta 2K
1K
SEQ =4096
1K
2K

Ilustración 43: Administración de ventana

- CheckSum

  Suma de comprobación de integridad de los datos del segmento, o sea, permite establecer al recibir un segmento que el mismo no ha sido modificado intencional o accidentalmente. Se calcula teniendo en base a un pseudo-cabezal el cual se muestra en la siguiente figura:

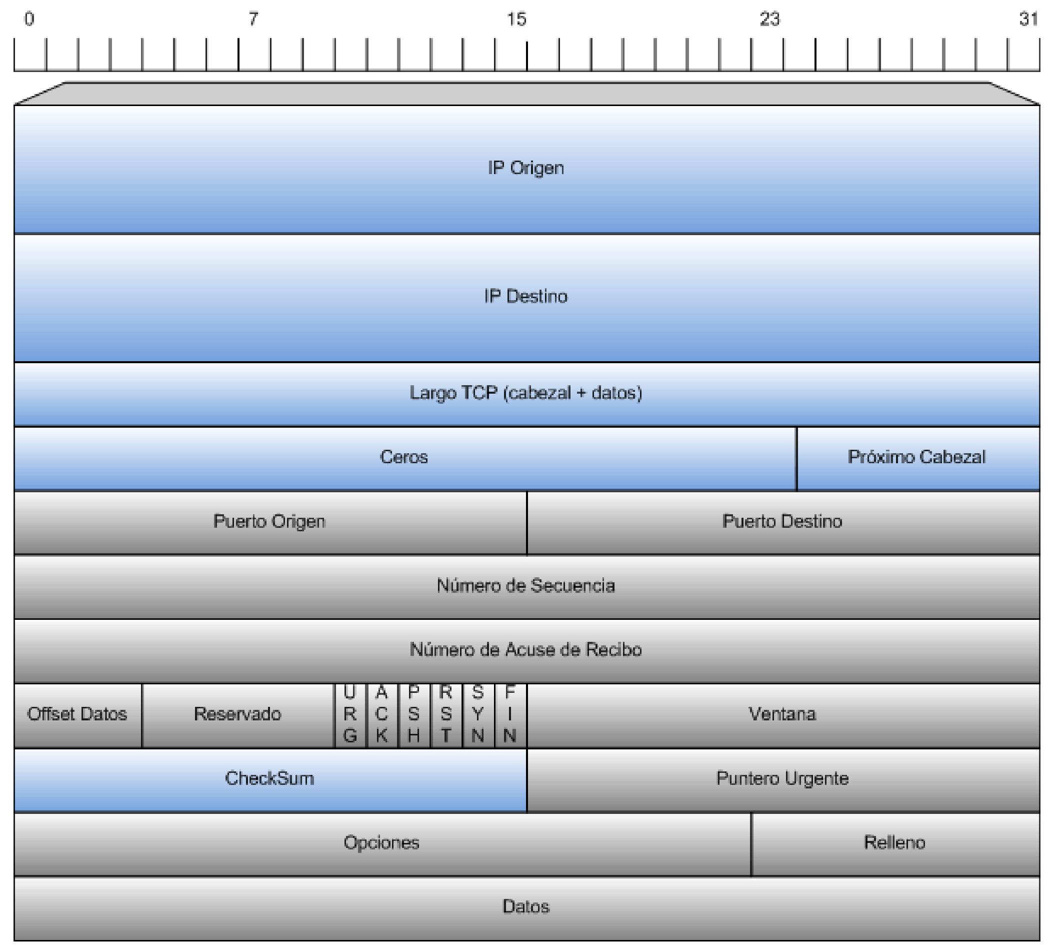


Ilustración 44: Pseudo Cabezal TCP

Los campos marcados en gris son aquellos que se setean con los valores que se obtienen del cabezal TCP. El checksum en este pseudo cabezal es seteado en 0. El campo "Próximo Cabezal" se setea con el valor de TCP (el valor para TCP de este cabezal es 6). Este checksum es calculado de acuerdo al RFC 1141.

- Puntero Urgente

  Offset con respecto del número de secuencia dado en el cabezal a partir del cual empiezan los datos urgentes del segmento.

- Opciones

  Campo de tamaño variable que contiene parámetros adicionales a ser utilizados por los nodos TCP. Si la lista de opciones ocupa un espacio no múltiplo de 4 bytes se rellena con ceros (esto es el campo relleno).

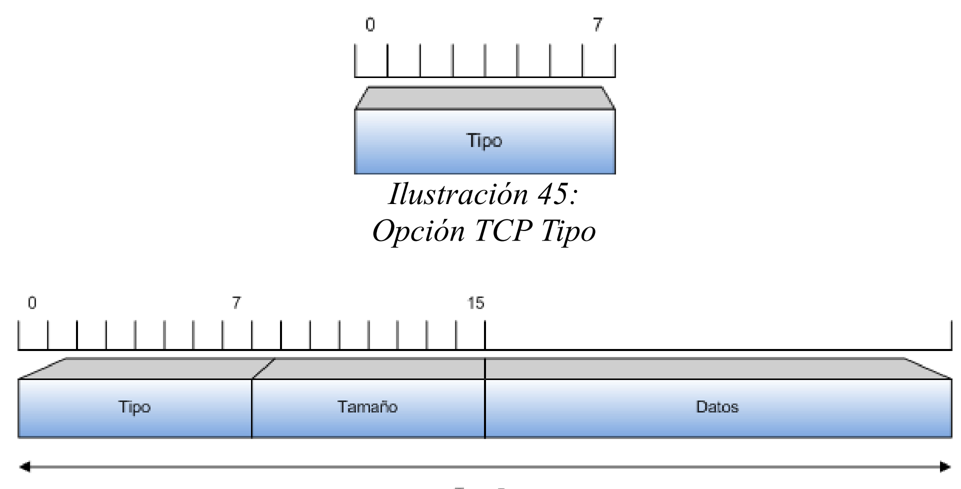


Ilustración 45: Opción TCP Tipo

Ilustración 46: Opción TCP Tipo y Datos

Se establecen para las opciones dos posibles formatos representados por las siguientes figuras:

Tipo: identifica a la opción.

Tamaño: tamaño en bytes de toda la opción (incluyendo los dos campos mencionados y los datos contenidos).

En el RFC 793 se especifican tres opciones básicas: No Operation, List End y Maximum Segment Size. Además de estas tres, como se verá más adelante, se implementaron tres más, dos de las cuales están especificadas en el RFC 1323.

10.4.5.2 Estados

Una conexión TCP pasa por varios estados desde el momento en que se abre la conexión hasta el momento en que la misma se cierra para finalizar su uso. Dependiendo de estos estados es cómo responde TCP a los diferentes eventos que pueden llegar a ocurrir. A continuación se presenta una máquina de estados (similar a la presentada en el RFC 793), donde se detallan los estados por los cuales puede pasar una conexión y algunas de las transiciones posibles que se pueden dar frente a algunos de los posibles eventos:

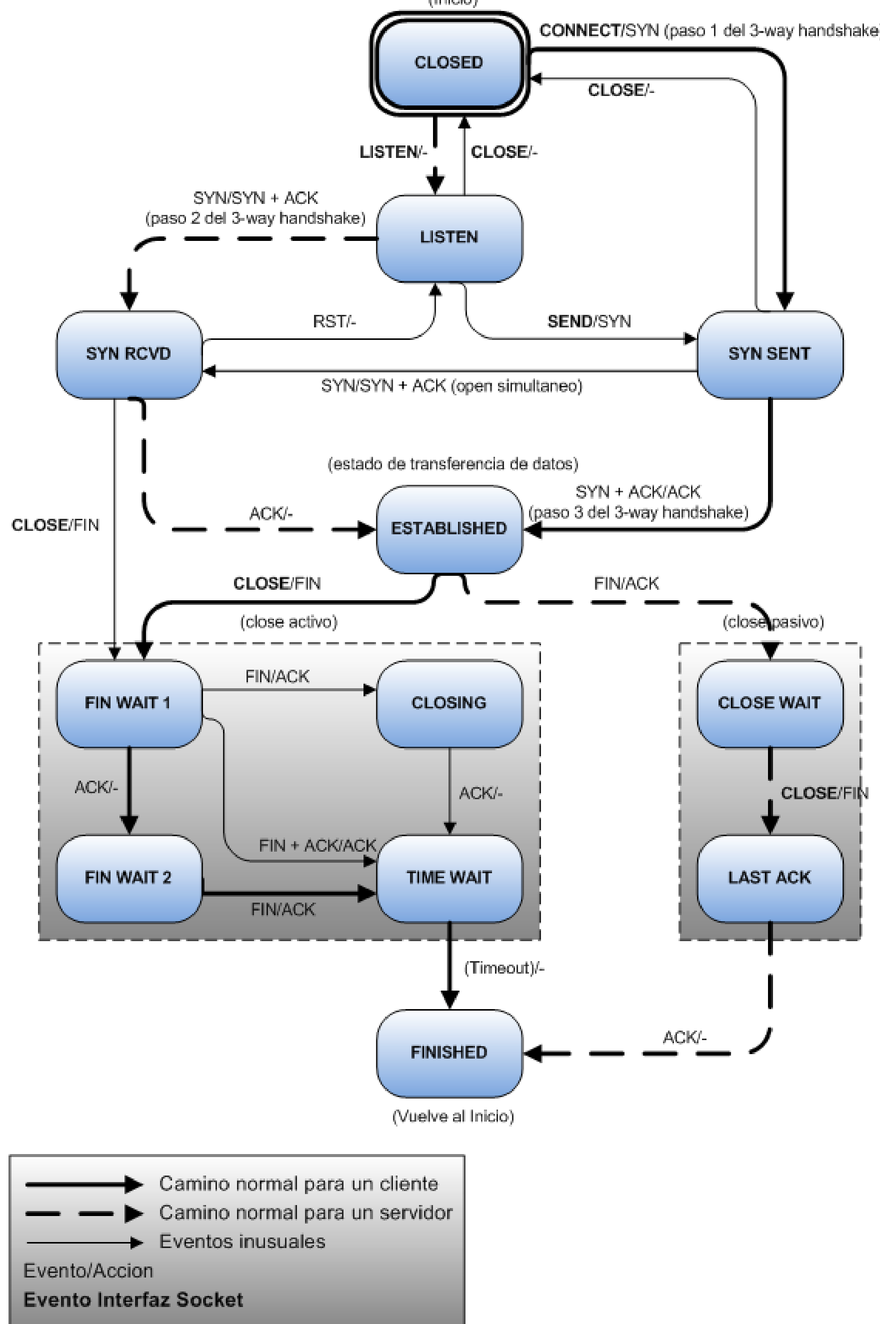


Ilustración 47: Máquina de estados de una conexión TCP

En el diagrama aquí presentado existen algunas diferencias con el presentado en el RFC 793:

- Existen algunas transiciones adicionales
- Se marcan los caminos típicos recorridos por aplicaciones Clientes y aplicaciones Servidores.

- Se cambian los nombres de open pasivo y open activo por Listen y Connect respectivamente lo cuál es más aproximado a la implementación final que se realizará.
- Otra diferencia es que se especifica el estado Finished, que en realidad es el estado Closed pero se decidió utilizar este nombre (Finished) para diferenciar claramente entre el inicio de una conexión y el fin de la misma, pero se debe considerar que los dos estados (Finished y Closed) en realidad se tratan de uno solo.

Las transiciones entre los estados y la explicación de la máquina de estados se vera implícita en la explicación de las operaciones definidas en la interface con la capa de aplicación, descrita en la siguiente sección.

10.4.5.3 Interfaces estándares de TCP

TCP como cualquier otra capa de un stack de protocolos debe brindar una interfaz a ser utilizada por la capa superior (capa aplicación). Esta interfaz, como ya se dijo, se brinda a través de las operaciones definidas en los sockets. A continuación se pasan a detallar las operaciones requeridas por el RFC 793 a ser implementadas en la interfaz de TCP y en algunos casos los principales algoritmos implicados:

1. *OPEN (puerto local, conector remoto, activo/pasivo [, tiempo de espera] [, prioridad] [,seguridad/compartimentación] [, opciones]) -> id de la conexión local*

   Llamada utilizada para abrir una conexión TCP. Esta llamada usualmente, como se mencionó antes, se divide en dos llamadas diferentes (dependiendo del parámetro que indica si es una llamada pasiva o activa) a la hora de implementar los sockets. Una es el Connect y la otra el Listen. A continuación se pasará a detallar cada uno de estos casos por separado, para finalmente indicar cuales son los restantes parámetros pasados a la llamada Open que son compartidos entre la llamada Connect y la llamada Listen.

   - **Listen**

     Al realizar un Listen (u open pasivo) el proceso quedara esperando a que otro nodo quiera establecer una conexión con este proceso (este estado es el representado por el estado LISTEN en la máquina de estados de la ilustración 47). Esta llamada es usada habitualmente por los procesos servidores que esperan a que clientes se conecten a ellos realizando pedidos para que los primeros los atiendan.

     Cuando se realiza un Listen se debe especificar el puerto local en el cuál se desea escuchar. En el caso donde no se especifique el conector remoto, se aceptaran peticiones de conexiones de cualquier nodo. Al establecerse una conexión, este conector remoto quedará determinado por la IP y el puerto del otro participante de la conexión.

En la máquina de estados se puede ver un escenario típico del funcionamiento de este método siguiendo el camino marcado desde el estado CLOSED al estado ESTABLISHED por las flechas punteadas. En la imagen ee muestran los estados por los que pasa y los segmentos que se envían.

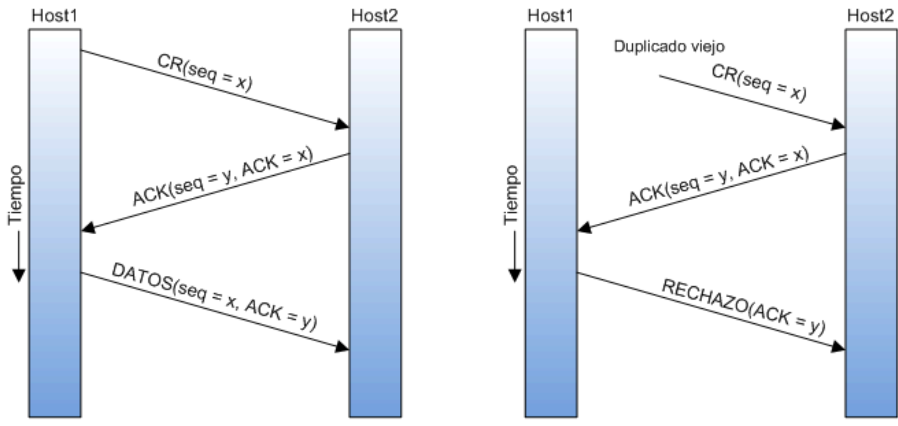
- **Connect**

Al realizar un Connect se inicia el proceso de inicio de conexión que consta en un saludo de tres vías. Los nodos, al establecer una conexión deben ponerse de acuerdo en los números de secuencias iniciales que van a usar, para así poder administrar adecuadamente las ventanas de recepción y envío y no confundir los datos con datos de conexiones ya cerradas. Para esto, este protocolo establece como procedimiento normal:

> Al iniciar la conexión el Host1, éste escoge un número de secuencia, “x”, y envía una connection request (en el caso de TCP marcando la bandera SYN del cabezal) que lo contiene. El Host2 responde con un connection accepted (en el caso de TCP marcando las banderas SYN y ACK), reconociendo “x” y enviando su propio número de secuencia inicial, “y”. Por último, el Host1 reconoce “y” en el primer envío de datos del Host2.

Este proceso resulta muy importante en TCP, ya que gracias al mismo se puede establecer la conexión para posteriormente hacer uso de las restantes funcionalidades.

En la siguiente imagen se muestra como reacciona el acuerdo de tres vías frente a un caso normal, un caso donde aparece una solicitud de conexión duplicada vieja y en el caso donde aparece una solicitud de conexión duplicada y ACK duplicado.



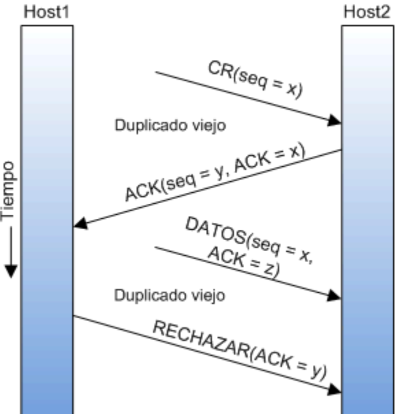


Ilustración 48: Saludos de 3 vías (común, CR Duplicado, CR y ACK Duplicado)

A continuación se muestra el establecimiento de conexión de tres vías tal cuál se realiza en TCP y un ejemplo de colisión de llamadas (lo cual está contemplado en el proceso):

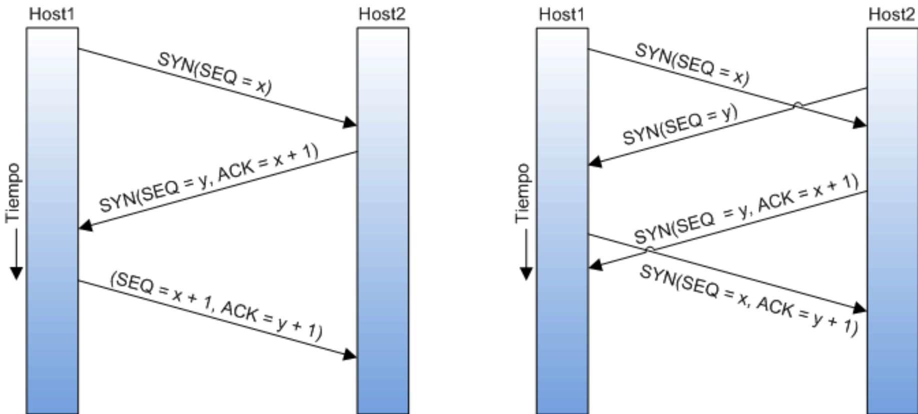


Ilustración 49: Saludo de 3 vías en TCP

Esta llamada es típicamente la utilizada por los procesos clientes que desean establecer una conexión con un servidor. El puerto local en este caso no es requerido. En el caso de que no se especifique TCP asignará uno efímero (auto generado) para su uso en la conexión. Por otro lado, el conector remoto es necesario para saber a que servidor se desea conectarse.

Parámetros de Open:

- Puerto local: puerto que usará el proceso que ejecutó la llamada open en el nodo en el que se encuentra.
- Conector remoto: puerto y dirección IP de la máquina destino a la cuál se quiere establecer la conexión.
- Activo/Pasivo: indica si se quiere realizar un open pasivo o activo.
- Tiempo de espera: tiempo luego de cada envío de datos dentro del cuál los datos tienen que haberse enviado por completo. De lo contrario la conexión se cierra. El valor por defecto para este parámetro es de 5 minutos.
- Prioridad y Seguridad/Compartimentación: Parámetros utilizados en la conexión para especificar prioridad de los paquetes y la seguridad, que permite identificar que la información es sensible y debe ser recibida únicamente por usuarios que hayan establecido una conexión con seguridad apropiada. En el caso de la implementación realizada, estos parámetros no se encuentran disponibles.
- Opciones: opciones a especificar en la conexión. Algunos ejemplos son: especificar el uso de algún algoritmo específico por parte de la implementación, uso de timer de cierre y especificación del tiempo del mismo, tamaño de los segmentos, etc.

La llamada retorna un identificador de la conexión establecida en caso de que se haya establecido correctamente, en caso contrario, deberá informar al proceso invocador del error que causó que no se pueda establecer la conexión.

2. *SEND(id de la conexión local, dirección del buffer, contador de bytes, indicador PUSH, indicador URGENT [, tiempo de espera])*

   Llamada utilizada para enviar datos al otro extremo de la conexión luego de que una conexión se encuentra establecida. Los datos, como ya se indicó anteriormente, se segmentan y se envían adjuntando el cabezal de TCP. En el cabezal se setean los puertos de origen y destino, el SN con el número de secuencia que identifica el primer byte del segmento enviado, la cantidad de bytes libres en la ventana de recepción, el ACK con el SN del último paquete recibido más 1 y los restantes valores y opciones como corresponda. Al llegar los datos al receptor éste los almacena en su buffer de recepción para re-ensamblar los datos. Si el emisor no recibe el ACK de los datos enviados luego de un cierto periodo (llamado tiempo de retransmisión) este los reenvía. Este tiempo de retransmisión es calculado dinámicamente por el protocolo midiendo los tiempos de ida y vuelta de los paquetes. El cálculo del tiempo de retransmisión será explicado con mayor detalle en la sección Timers.

   Parámetros:

   - Id de la conexión local: conexión por la cuál se quieren enviar los datos.
   - Dirección del buffer: puntero a los datos que se quieren enviar
   - Contador de bytes: cantidad de bytes de los datos que se quieren enviar
   - Indicador Push: bandera utilizada para indicar que el receptor debe subir los datos que posea de forma inmediata a la capa de aplicación.
   - Indicador Urgent: parámetro que indica que los datos que se están enviando son urgentes. Estos datos deberían ser notificados como urgentes del lado del receptor a la capa aplicación al ser entregados.
   - Tiempo de espera: parámetro que permite modificar el tiempo de espera establecido en la llamada open.

3. *RECEIVE(id de la conexión local, dirección del buffer, número de bytes) -> número de bytes, indicador 'urgent', indicador 'push'*

   Llamada utilizada para recibir en un buffer todos los datos enviados por el otro nodo de red participante en la conexión. Al recibir un segmento, este se almacena en un buffer de reensamblado a partir del cual se reconstruyen los mensajes a ser entregados al buffer de recepción. Cuando se llena el buffer de recepción o cuando se recibe un mensaje con el indicador de push los datos son devueltos al llamador del receive indicando la cantidad de bytes que se recibieron.

   Parámetros:

   - Id de la conexión local: identificador de la conexión desde la cual se quieren recibir datos.

- Dirección del buffer: puntero al buffer donde se deben poner los datos que lleguen.
- Número de bytes: capacidad del buffer de recepción.
- Número de bytes retornado: cantidad de bytes que se recibieron.
- Indicador 'urgent': indica si los datos que han llegado son urgentes.
- Indicador push: indica si el nodo emisor ha realizado un push.

4. *CLOSE(id de la conexión local)*

Función utilizada para iniciar el proceso de cierre de una conexión. En TCP el cierre de conexiones afronta el problema de los dos ejércitos, por lo cuál se utiliza un timer, que luego de vencer cierra la conexión forzosamente. El problema de los dos ejércitos se encuentra detallado en el libro Redes de Computadoras, escrito por Andrew S. Tanenbaum, en su página 499 de la 3ª edición.

El comportamiento luego de la invocación de esta función se puede visualizar en la máquina de estados presentada. Básicamente, al cerrar la conexión se continua mandando los datos de llamadas a sends y receives pendientes, se permite realizar nuevas llamadas a receives, pero no sends. Luego que el otro nodo también cierre la conexión es cuando entra en juego el timer de cierre de la conexión, y efectivamente se cierra. El protocolo de desconexión utilizado es un acuerdo de tres vías (como el saludo de tres vías), el cual se detalla a fondo en la referencia al libro Tanenbaum antes mencionada.

En la siguiente figura se detallan cuatro situaciones y el funcionamiento del protocolo de liberación de conexión frente a las mismas:

Ilustración 50: Liberación de conexión (común, perdida del último ACK, respuesta perdida, respuesta perdida y subsecuentes DR)

En el diagrama se representa con DR al pedido de desconexión (Disconnection request). Esta llamada se traduce a paquetes con la bandera FIN activada en TCP.

Luego de cerrarse una conexión conviene esperar un cierto tiempo antes de establecer una conexión con los mismos sockets, para evitar que segmentos ya enviados sean confundidos con segmentos de la nueva conexión.

5. *STATUS(id de la conexión local) -> datos de estado*

Retorna información sobre el estado de la conexión.

Parámetros:

- Id de la conexión local: identificador de la conexión de la cuál se quieren obtener los datos de estado.

Los "datos de estado" son el conjunto de datos que se retorna luego del llamado de la función, estos pueden variar según la implementación. En el RFC 793 se enumeran los siguientes como posibles datos a mostrar:

conector local,
conector remoto,
nombre de la conexión local,
ventana de recepción,
ventana de envío,
estado de la conexión,
número de buffers en espera del acuse de recibo,
número de buffers pendientes de recepción,
estado urgente,
prioridad,
seguridad/compartimentación,
y tiempo de espera de transmisión.

6. *ABORT(id de la conexión local)*

Llamada utilizada para abortar definitivamente una conexión y liberar todos los recursos que ésta tenga. Todos los sends pendientes y/o receives serán cancelados. Esta función no notifica de ningún modo al otro nodo acerca del llamado. Luego de llamada esta función, el stack responde de manera estándar a cualquier segmento que este destinado a una conexión cerrada o inexistente: envía un paquete de reset. Según la implementación, puede ser que las aplicaciones que hayan realizado un send y/o receive en el socket reciban un mensaje indicando el aborto de la conexión. Se debe tener especial cuidado con el uso de esta llamada pues produce pérdida de datos, lo cual no sucede con la llamada close.

10.4.5.4 Timers

En el RFC 793 se especifican los siguientes timers:

1. *Tiempo de espera de usuario*

Temporizador inicializado cada vez que se envía algún dato por parte del usuario o cuando se establece la conexión. En caso de que llegara a espirar se debe abortar la conexión informando adecuadamente al thread de usuario. El tiempo de espera es dado por el usuario a la hora de realizar el Connect o un Send. En caso de que no sea especificado se tomará uno por defecto de 5 minutos.

2. *Tiempo de espera de retransmisión*

Temporizador inicializado cada vez que se transmite algún segmento. Al vencer este temporizador se reenvía el segmento que se encuentre al principio de la cola de retransmisión moviéndolo al final de la misma. En el RFC 793 no se especifica explicitamente como debe ser implementado este temporizador ni como se mediría el tiempo de ida y vuelta que debería utilizarse para calcularlo. Por lo que queda abierta la implementación del mismo.

3. *Tiempo de espera en estado TIME-WAIT*

Temporizador inicializado al entrar en el estado TIME-WAIT. Al vencer este timer se aborta la conexión. Este temporizador se inicializa con el valor 2 MSL, siendo MSL (Maximum Segment Lifetime) el máximo tiempo de vida esperado de los segmentos en la red (este último valor es estimado en 60 segundos).

10.4.6 Diseño e Implementación

En esta sección se detallarán los aspectos concernientes al diseño y la implementación realizadas de TCP. La misma se basa en la implementación Net/3 explicada detalladamente en el libro tomado como referencia TCP/IP Illustrated Volume 2 de 1995 de Gary R. Wright y W. Richard Stevens.

10.4.6.1 Características Generales

Como es sabido la implementación realizada se debe integrar al modelo ya propuesto por el framework de IP4JVM y específicamente debe ser realizada en Java por la misma restricción.

La imposición de realizar la implementación con este lenguaje nos llevó a respetar la interfaz propuesta por Java para el manejo de sockets.

En el siguiente diagrama se muestran las principales clases brindadas por Java para hacer uso de los sockets del sistema:

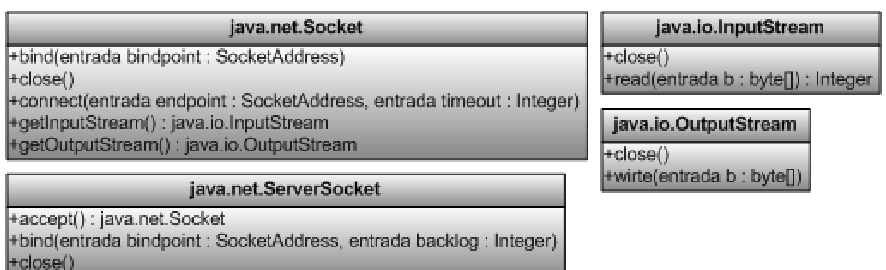


Ilustración 51: Clases Java para manejo de sockets

En este diagrama se omiten todas las operaciones utilizadas explicitamente para setear u obtener atributos de la conexión y además aquellas operaciones realizables a partir de las aquí mostradas.

La clase InputStream se utiliza para recibir datos desde el socket del cual se haya obtenido la instancia. Además permite cerrar la conexión mediante el uso de la operación close.

La clase OutputStream es usada para enviar datos a través del socket del cual se obtuvo la instancia. Al igual que la clase InputStream permite cerrar la conexión.

La clase Socket es la utilizada por aplicaciones clientes para establecer conexiones, obtener una instancia de InputStream para recibir datos, o una instancia de OutputStream para enviar datos. Además brinda la operación bind que permite especificar la dirección IP y el puerto local a ser utilizados en la conexión. En el caso de que no se realice un bind antes de establecer la

conexión, TCP asigna un puerto efímero y la IP de una de las interfaces locales al socket automáticamente. La operación connect permite iniciar el proceso de establecimiento de conexión y recibe como parámetros la especificación del conector remoto (dirección IP y puerto) y además el valor con el cual inicializar el temporizador de envío de datos de usuario. La operación close permite cerrar la conexión asociada al socket. Otro uso de la clase Socket, como se vera a continuación, es en el establecimiento de conexiones pasivas.

La clase ServerSocket permite, a aplicaciones servidores, esperar en un open pasivo el establecimiento de conexiones por el pedido de alguna aplicación en otro nodo. Brinda, al igual que la clase Socket, la operación bind que permite establecer el puerto y dirección IP locales. La operación bind difiere de la de la clase Socket en que debe ser invocada necesariamente para especificar un puerto, además recibe un nuevo parámetro denominado backlog que permite especificar cuantas conexiones simultaneas puede estar atendiendo el mismo socket. Adicionalmente brinda la operación accept la cual permite bloquear el thread invocador hasta que se establezca la conexión, luego de establecida la operación retorna una instancia de Socket para manejar el resto de la vida de la conexión. La operación close dada permite cerrar todas las conexiones asociadas al ServerSocket.

A continuación se muestra un ejemplo de dos aplicaciones (una cliente y otra servidor) implementadas en Java que hacen uso de las clases y operaciones mostradas:

```
public class Servidor{

public static void main(String[] args){

    ServerSocket server = new
ServerSocket(2004, 10);

    do{
        Socket conn = server.accept();

        OutputStream os=
conn.getOutputStream();

        ObjectOutputStream out = new
ObjectOutputStream();

        try{
            out.flush();
            out.write("HOLA");
            out.flush();
        } catch(IOException e){
            e.printStackTrace();
        } finally{
            try{
                out.close();
                connection.close();
            } catch(IOException e){
                e.printStackTrace();
            }
        }
    }while(true);
}

}
```

```
public class Cliente{

public static void main(String[] args){

    InetAddress add=
InetAddress.getByName("2001::1");

    Socket cli = new Socket(add,2004);
    InputStream is= cli.getInputStream();

    ObjectInputStream in = new
ObjectInputStream(is);

    try{
        String msg= in.readObject();
        System.out.println(msg);
    } catch(IOException e){
        e.printStackTrace();
    } finally{
        try{
            in.close();
            cli.close();
        } catch(IOException e){
            e.printStackTrace();
        }
    }
}

}
```

El constructor mostrado de la clase ServerSocket, implícitamente realiza el bind con los parámetros pasados al constructor (puerto 2004 y backlog 10). La operación flush mostrada fuerza el envío inmediato de todos los datos que se encuentren en buffers.

El servidor básicamente espera en el puerto 2004 hasta 10 conexiones en paralelo. Luego de haberse establecido una conexión con una aplicación cliente el servidor manda el texto "HOLA" y cierra la conexión esperando indefinidamente por nuevas conexiones.

El constructor de la clase Socket mostrado hace implícitamente un connect al conector dado con los parámetros pasados (dirección IP 2001::1 y puerto 2004).

El cliente establece una conexión con el servidor que escucha en la IP 2001::1 y puerto 2004. Al establecerse la conexión el cliente recibe el mensaje que el servidor tenga para mandarle y lo muestra en la salida estándar. Finalmente cierra la conexión finalizando su ejecución.

Como base para realizar la implementación se siguieron los lineamientos y requerimientos impuestos por el RFC 793 y adicionalmente se tomó como base la implementación Net/3 de la familia de sistemas operativos Linux BSD-Lite 4.4. La implementación Net/3 se encuentra detalladamente explicada y documentada en el libro TCP Illustrated Vol 2 de Addison-Wesley.

10.4.6.2 Clases Implicadas

A continuación se muestra una imagen con las principales clases implicadas en el diseño e implementación de TCP y la relación entre las mismas. Luego se comentarán sus roles y los detalles de clases no mostradas en esta imagen, omitidas para evitar la complejidad que significaría mostrar todas las clases participantes en un único diagrama:

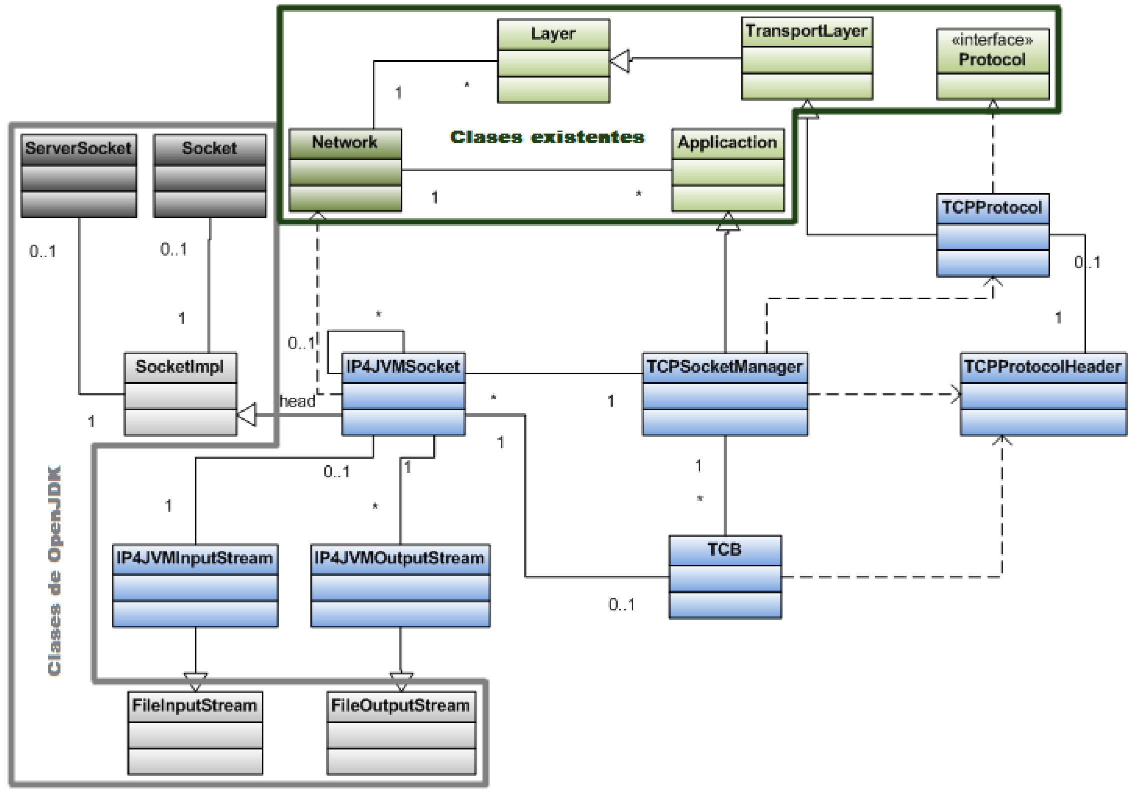


Ilustración 52: Principales clases

Leyendas de los colores para este diagrama y los subsiguientes que presentan clases:

- Las clases marcadas en gris son las ya provistas por la máquina virtual Java.
- Las que están en verde claro son clases que ya se habían implementado en etapas anteriores del desarrollo del stack.
- Las clases que están en gris oscuro son las clases que se tuvieron que modificar de la máquina virtual para poder hacer que fuera posible que nuestro stack fuera utilizado.
- Las clases de verde oscuro son las clases que se modificaron de las que ya se habían implementado en el stack.
- Las clases celestes son las nuevas, que se implementaron de cero.

Los principales grupos de clases se dividen en:

- Integración a la máquina virtual: donde se encuentran ServerSocket, Socket, SocketImpl, IP4JVMSocket, IP4JVMInputStream (con FileInputStream) e IP4JVMOutputStream (con FileOutputStream). El cometido principal de estas clases es la integración de lo implementado en el resto de las clases, con la máquina virtual Java.

- Integración con el framework de IP4JVM: de las cuales forman parte Network, TCPSocket Manager, TCPProtocol, Protocol, Application y Layer. Estas clases permiten

integrar la lógica de TCP a lo ya implementado en el stack de IP4JVM y utilizar las funcionalidades que brinda este último, todo esto mediante el uso del framework existente.

- Administración de conexiones: donde están TCPSocketManager y TCB (Transfer Control Block). Estas clases se encargan de casi la totalidad de la lógica necesaria para implementar TCP.

- Administración de formato de datos: en este grupo se encuentran las clases TCPProtocol y TCPProtocolHeader. Este grupo de clases es el encargado de parsear los datos de control del cabezal de TCP y las opciones a partir de una tira de bytes. Además de, a partir de la información de control y los datos que se deseen incluir en un segmento, crear la tira de bytes correspondiente al segmento.

Ahora se pasará a detallar cada uno de estos grupos.

Integración a la máquina virtual

Para integrar lo implementado al Socket, se debieron definir las clases y cambios mostrados en la siguiente imagen:

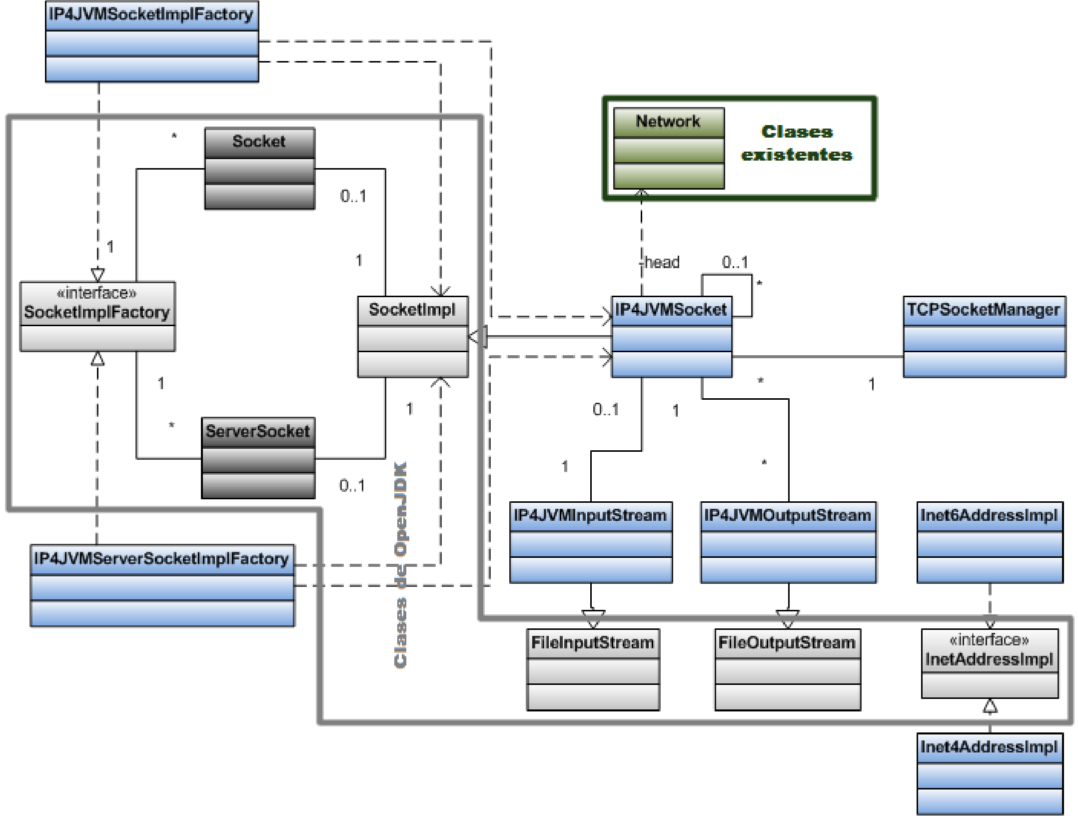


Ilustración 53: Clases Implicadas en la integración con la máquina virtual

Descripción de las clases presentadas:

- InetAddressImpl

  Interfaz que especifica las operaciones a ser implementadas por cualquier InetAddress.

- Inet6AddressImpl e Inet4AddressImpl

  Clases implementadas para realizar las operaciones necesarias para InetAddress, utilizando la información configurada en el stack IP4JVM. Se debe setear previamente a la máquina virtual el parámetro “impl.prefix” de manera que sea igual a “ip4jvm”.

- SocketImpl

  Clase abstracta utilizada por las clases ServerSocket y Socket para realizar la lógica correspondiente a las operaciones de los mismos. Esta clase implementa todas las operaciones antes vistas para las clases Socket y ServerSocket además de dos operaciones adicionales, setOption y getOption, que permiten establecer atributos de la conexión, además de obtener datos sobre los mismos.

- Socket y ServerSocket

  Se cambiaron ambas clases de manera tal de hacer que pudiera ser posible, mediante el pasaje del parámetro "impl.prefix" a la máquina virtual, especificar la SocketImplFactory a ser utilizada. Estas clases son brindadas por la máquina virtual Java y proveen las operaciones descritas más arriba, cuando se explicaron las clases dadas por Java para el manejo de Sockets.

  Estas clases, en sus constructores crean una instancia de SocketImpl, la cual es utilizada posteriormente en cada una de las operaciones que las clases brindan. En su estado original la máquina virtual creaba para todos los constructores una instancia de la clase SocksSocketImpl, ahora bien, esto se modificó para que si se especifica el parámetro anteriormente mencionado a la hora de correr el programa, se utilice la implementación dada por la factory (instancia de SocketImplFactory) referenciada por el parámetro y la clase invocante (si es que existe una clase factory que efectivamente sea referenciada en estas condiciones). En caso de que el parámetro no sea especificado se seguirá utilizando la implementación estándar (SocksSocketImpl).

  Un ejemplo de uso del parámetro: Si se especifica "impl.prefix" como igual a "IP4JVM" entonces la clase Socket utilizará como factory de SocketImpl la clase "java.net.IP4JVMSocketImplFactory" en caso de que esta exista. En el caso de ServerSocket la clase utilizará (en el en caso de que exista) la clase "java.net.IP4JVMServerSocketImplFactory" como factory de las instancias SocketImpl.

- SocketImplFactory

  Interfaz que deben cumplir las clases encargadas de brindar instancias de SocketImpl.

- IP4JVMScoketImplFactory y IP4JVMServerSocketImplFactory

  Clases que implementan la interfaz y que crean las instancias de IP4JVMSocket con los parámetros adecuados. La clase IP4JVMScoketImplFactory pasa como parámetro un boolean en falso al constructor de SocketImplFactory, indicando que no se trata de un servidor, por el contrario la clase IP4JVMServerSocketImplFactory pasa dicho parámetro en verdadero indicando de que se trata la implementación de un servidor.

- FileInputStream y FileOutputStream

  Clases abstractas brindadas por Java para el manejo de archivos y sockets haciendo uso de estos últimos como si fueran archivos.

- IP4JVMInputStream

  Clase que extiende la clase FileInputStream, y mediante la cuál se pueden realizar recepción de datos desde el socket. Esta clase mantiene una referencia a la instancia de IP4JVMSocket que la creó y redirige los pedidos de recepción a esta

última.

- IP4JVMOutputStream

  Clase que extiende la clase FileOutputStream y permite realizar envíos a través del socket. Así como IP4JVMInputStream, mantiene una referencia a la instancia de IP4JVMSocket a la cuál redirige los pedidos de envío.

- IP4JVMSocket

  Implementación de SocketImpl que define todas las operaciones de los dos tipos de sockets que brinda la máquina virtual. Esta clase recibe como parámetro en su constructor un indicador que indica si la implementación está asociada a una instancia de ServerSocket o de Socket. Esta clase se encarga de controlar el estado del socket, realizar verificaciones sobre el estado del socket cada vez que se invoque una operación, y redirigir adecuadamente indicando la conexión que se esta utilizando (en el caso de que se esté usando una) a la clase TCPSocketManager que será la encargada de gestionar dicho pedido adecuadamente o redirigirlo a la clase TCB. La instancia de TCPSocketManager es obtenida a partir de la instancia de Network que mantiene todas las aplicaciones y protocolos utilizados en el stack.

  Las instancias de IP4JVMSocket que correspondan con una implementación de Socket mantienen el identificador de la conexión que están utilizando (en el caso de que se haya establecido una), las opciones seteadas al socket y una instancia de IP4JVMInputStream, la cual será retornada cuando se requiera la misma. Cada vez que se pide una instancia de OutputStream se crea una nueva instancia de IP4JVMOutputStream.

  Las instancias que correspondan con implementaciones de ServerSocket mantienen los mismos datos que el caso de Socket agregando además una lista de Sockets de la cuál se explicará su utilización en la sección de atención de eventos.

Integración con el framework de IP4JVM

La integración con el framework de IP4JVM fue realizada modificando e implementado las clases que se muestran en la siguiente figura:

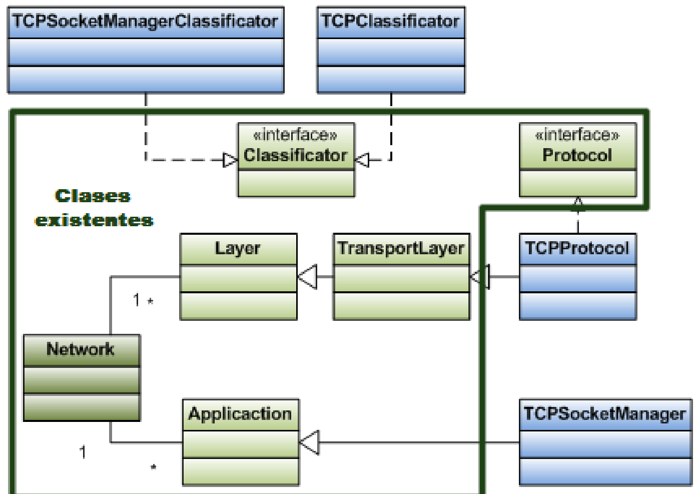


Ilustración 54: Clases implicadas en integración con el framework

Descripción de las clases:

- Network

  Esta clase fue modificada de manera tal de incluir la clase TCPSocketManager como Application y la clase TCPProtocol como Protocol y TransportLayer. También registra los Classificators que corresponden a cada una.

- TCPSocketManagerClassificator

  Clase encargada de verificar al llegar un Job, si este debe ser procesado por la Application TCPSocketManager.

- TCPProtocolClassificator

  Esta clase se encarga de verificar, a partir de un Job, si este debe o no ser procesado por el Layer TCPProtocol.

- TCPProtocol

  El propósito de esta clase es brindar un Protocolo de capa 4 para que los Jobs de TCP lleguen a la clase TCPSocketManager. TCPProtocol se encuentra casi desprovista de métodos ya que su principal objetivo es tomar los Jobs entrantes desde la red y pedirle a la instancia de la clase TCPProtocolHeader, que tiene asociada, que procese los datos entrantes, para luego retornar el Job modificándolo sólo de manera tal que incluya a la instancia de TCPProtocol como capa que procesó el Job.

  Como se verá más adelante la clase TCPSocketManager conjuntamente con la clase TCB implementan la mayoría de la lógica de TCP. Aunque esto parecería naturalmente ser una responsabilidad de la clase TCPProtocol, debido a las

restricciones impuestas por el modelo del framework, y para estar alineados a lo ya realizado para el protocolo UDP, se decidió diseñar y posteriormente implementar de esta manera. El modelo del framework demanda que un protocolo retorne un Job a ser procesado por una capa superior o una aplicación, y no existe forma para, de manera prolija, integrar los sockets como aplicaciones del stack.

Administración de conexiones

Las clases implicadas en la administración de conexiones se dividen entre, las que colaboran con la clase TCPSocketManager y las que lo hacen con la clase TCB. La imagen que se muestra a continuación presenta las clases que colaboran con la clase TCPSocketManager:

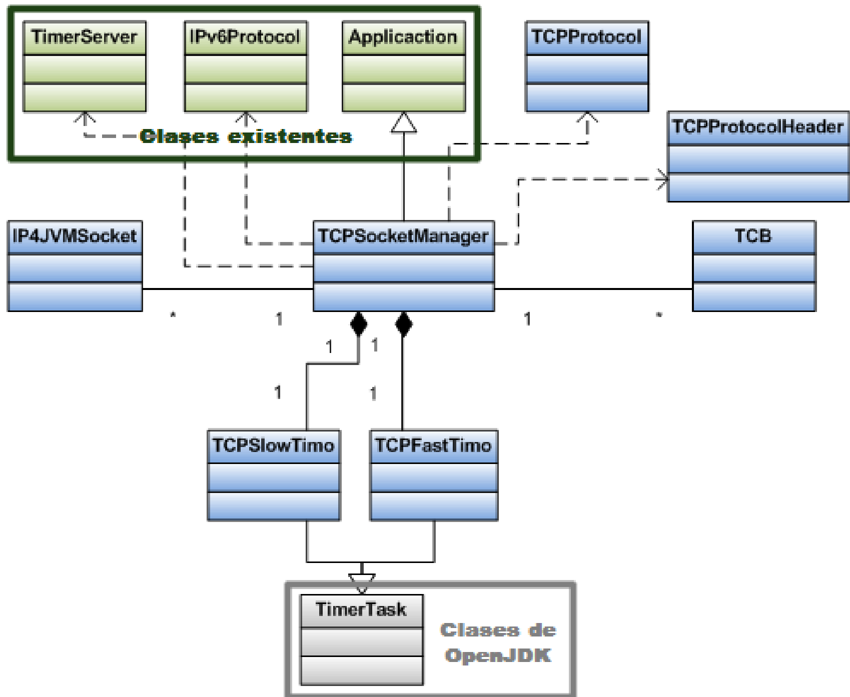


Ilustración 55: Clases que colaboran con TCPSocketManager

Descripción de las clases:

- TCPSocketManager

  Clase encargada de gestionar todas las conexiones que son administradas, como se verá más adelante, por la clase TCB. Los cometidos de esta clase son:

  - Recibir los paquetes que vienen de la red, procesarlos para luego decidir si pertenecen a alguna de las conexiones existentes, en cuyo caso se envían los datos a la conexión, o si se debe crear una conexión.
  - A partir de los pedidos de los socket, redirigir esos pedidos a las conexiones que correspondan.

  - Crear y eliminar conexiones.
  - Brindar las funciones a ser invocadas por los timers de TCP.

- TCPSlowTimo y TCPFastTimo

  En la implementación y diseños realizados se optó por la implementación de únicamente dos temporizadores, los cuales implementan todos los temporizadores necesarios para la implementación TCP. Esto fue realizado así pues, crear una instancia de la clase TimerTask por cada temporizador utilizado en cada conexión TCP insumiría demasiados recursos de procesamiento y memoria, lo cuál tal vez no sea tan notable en un proceso cliente que maneja una única conexión, pero en procesos servidores que manejan un conjunto potencialmente grande de conexiones seguramente esto afectaría de muy mala manera el desempeño de la implementación.

  Las clases TCPSlowTimo y TCPFastTimo extienden la clase TimerTask brindada por Java y se ingresa una instancia de cada una como tareas a ser ejecutadas periódicamente por la instancia de TimerServer. La instancia de TCPSlowTimo se programa para ser ejecutada cada medio segundo y al vencer invoca la función slowtimo de la instancia de TCPSocketManager a la cual está asociada. La instancia de TCPFastTimo se programa para ser ejecutada periódicamente cada 200 mili segundos y al pasar dicho tiempo la misma invoca a la función fasttimo de la instancia TPCSocketManager asociada.

  Al ser invocadas, ya sea la función slowtimo como fasttimo, la clase TCPSocketManager recorre todas las instancias de TCB que tiene asociadas, y para cada una invoca una función que notifica que ha vencido determinado temporizador.

Ahora se pasará a mostrar en la siguiente figura las clases que colaboran con la clase TCB:

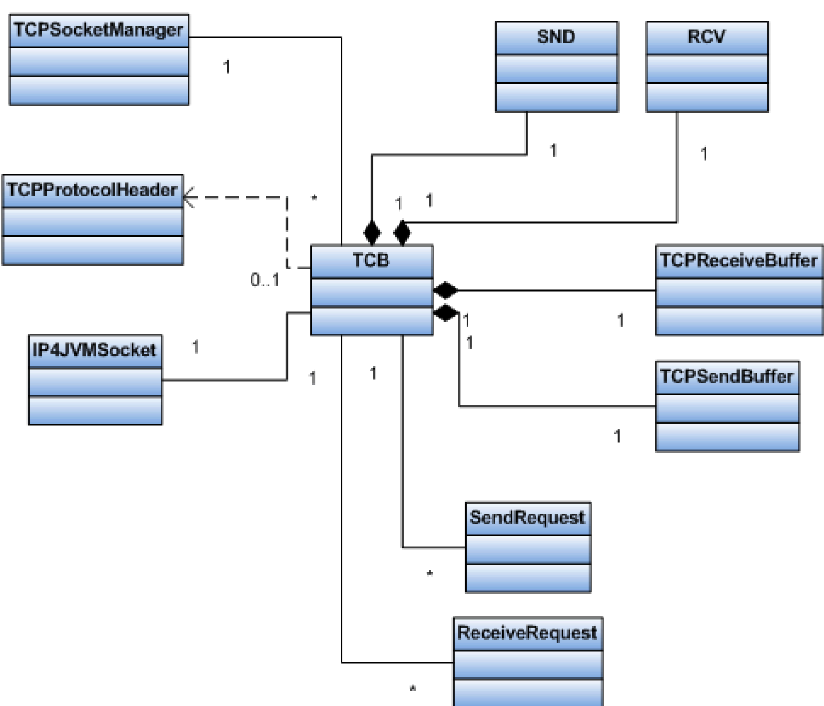


Ilustración 56: Clases que colaboran con TCB

Descripción de las clases mostradas:

- TCB

  Clase encargada de la mayor parte de los aspectos de una conexión, desde el establecimiento de la conexión pasando por el envío y recepción de datos, además del cierre de la conexión y algoritmos de control de congestión de la red. Mantiene una gran cantidad de variables para el control de congestión, ventanas, timers, buffers, etc. Aunque tiene algunas clases que colaboran con esta para disminuir la complejidad del código implementado en TCB, igualmente, dada la complejidad y gran cantidad de detalles que cubre TCP, el código de la clase TCB es extenso.

  Como se puede ver, los buffers de recepción y envío son administrados por la clase TCB, y no por la clase IP4JVMSocketManager como lo hace la implementación de Net/3. Esto se decidió realizar de esta manera para centralizar la mayor parte de los aspectos de la conexión en la clase TCB y dejar únicamente a la clase IP4JVMSocket la responsabilidad de comunicarse con las aplicaciones que invocan las funciones de sockets, saber como enviar las solicitudes a la conexión correcta y administrar los aspectos que ya se han visto al describir a esta clase.

  TCB mantiene una asociación con el socket al cual está asociado, para notificarle al mismo de eventos, como ser la finalización de un proceso de establecimiento de conexión, el llenado de un buffer de recepción, etc.

- SND

  En esta clase se mantienen gran parte de las variables usadas por TCP para administrar el envío de datos. Para el envío de datos se definen en el RFC 793 las siguientes variables que son implementadas en la clase SND:

  - SND.UNA

    Marca el primer SN enviado que aún no ha sido confirmado (no ha llegado el ACK de este SN) por el otro nodo.

  - SND.NXT

    Indica el SN del próximo segmento a ser enviado (el máximo SN enviado dentro de la ventana actual de envío más uno).

  - SND.WND

    Tamaño de la ventana de envío. Esta ventana es actualizada de acuerdo al valor de la ventana de recepción indicada por el otro extremo.

  - SND.UP

    Puntero al primer SN de datos urgentes del buffer de envío.

  - SND.WL1

    Mantiene el SN del segmento utilizado en la última actualización de la ventana de envío.

  - SND.WL2

    Mantiene el ACK del segmento utilizado en la última actualización de la ventana.

  - ISS

    SN de envío inicial ('initial send sequence'). Este número es calculado al inicio de la conexión y a partir de este SN se establecen los SN subsiguientes.

Estas variables dividen el espacio de los SN de envío en las áreas mostradas por la siguiente figura:

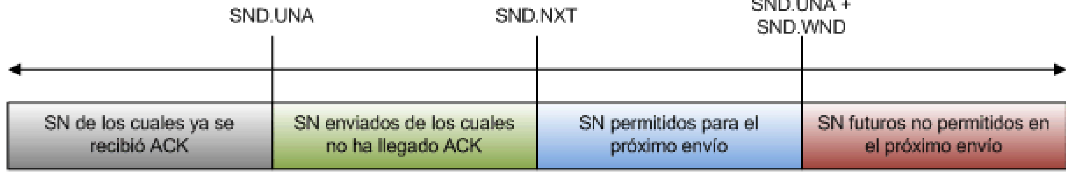


Ilustración 57: Áreas de SN de envío

En la implementación realizada, además de estas variables, se mantienen variables para control de la congestión (tamaño de la ventana de congestión y threshold), el cuál será explicado más adelante en la sección de Congestion Avoidance.

- RCV

  Esta clase mantiene variables utilizadas en la recepción de datos por la implementación TCP. Las variables definidas en el RFC 793 son las siguientes:

  - RCV.NXT

    SN esperado de inicio del segmento de la próxima recepción.

  - RCV.WND

    Tamaño de la ventana de recepción, esta es la cantidad de buffer disponible en TCPReceiveBuffer, el cual cómo se verá más adelante, es el buffer utilizado para el re ensamble de los datos. Esta ventana es la que se brinda a través del cabezal TCP al emisor para que este controle el tráfico adecuadamente.

  - RCV.UP

    Puntero al inicio de los datos urgentes en el buffer de recepción.

  - IRS

    SN de recepción inicial ('initial receive sequence') dado por el otro extremo de la conexión al inicio de la misma.

  Las variables definidas en el RFC 793 dividen el espacio de SN de los segmentos recibidos en las áreas detalladas en la siguiente figura:

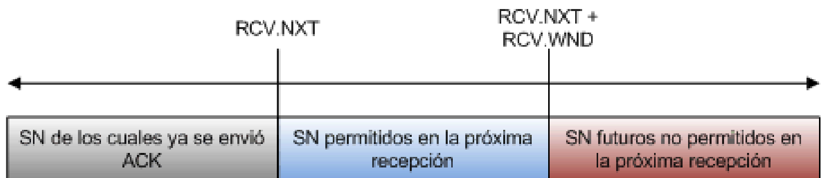


Ilustración 58: Áreas de SN de recepción

- SendRequest

  TCB mantiene una cola de instancias de SendRequest que representan todos los pedidos de envío realizados por el socket asociado. Estos send se irán pasando al TCPSendBuffer a medida que este último vaya obteniendo espacio al ir recibiendo los ACK correspondientes. Este proceso se detalla mejor en la explicación de la clase TCPSendBuffer.

- ReceiveRequest

  TCB además mantiene una cola de todos los pedidos de receive realizados por el socket, cada uno de estos receive es almacenado en una instancia de ReceiveRequest. El buffer del primer receive se irá llenando con los datos del TCPReceiveBuffer como se explica en los comentarios sobre esta última clase. Cuando el buffer del primer receive de la cola se llena se notifica a la instancia de

socket que se encargará de desbloquear al thread que hizo el receive.

- TCPReceiveBuffer

  En esta clase se van almacenando todos los datos que han llegado y que aún no se han subido al buffer de algún pedido de receive, además de la lista de pushes realizados por el nodo del otro extremo de la conexión. El uso que da TCB a esta clase es el re ensamblado de los datos recibidos. Al recibir un segmento TCB busca segmentos contiguos (ya sea anteriores y/o siguientes) al recibido en esta cola, si existen se fusionan en uno solo. La manera en que se fusionan se muestra en la figura que se presenta en la siguiente imagen.

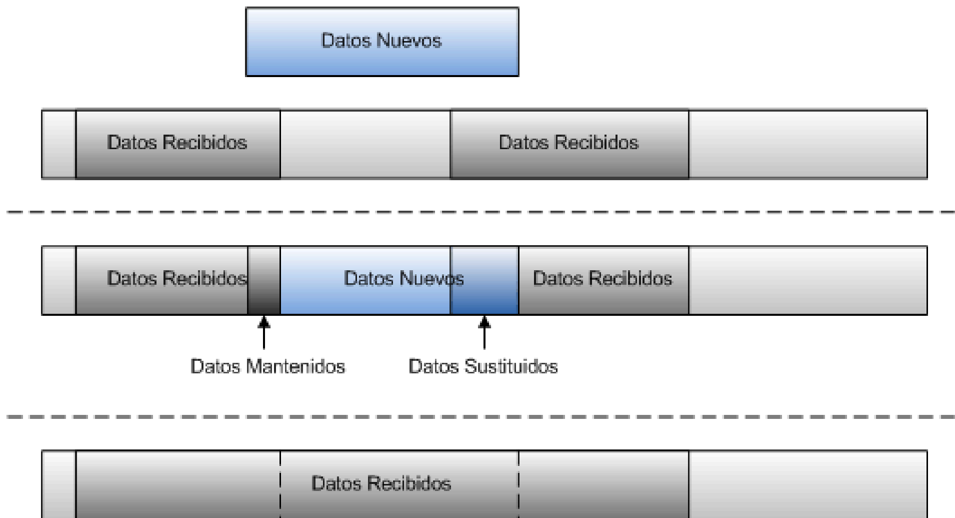


Ilustración 59: Fusión de datos en Receive Buffer

Si luego de este proceso, los datos inician donde comienza el receive buffer, los mismos son quitados del buffer de recepción, el inicio de buffer es corrido hasta el final de los datos quitados y luego los datos quitados son repartidos entre los SendRequest del TCB en el orden en que estos se crearon.

El proceso antes mencionado se muestra en la siguiente imagen:

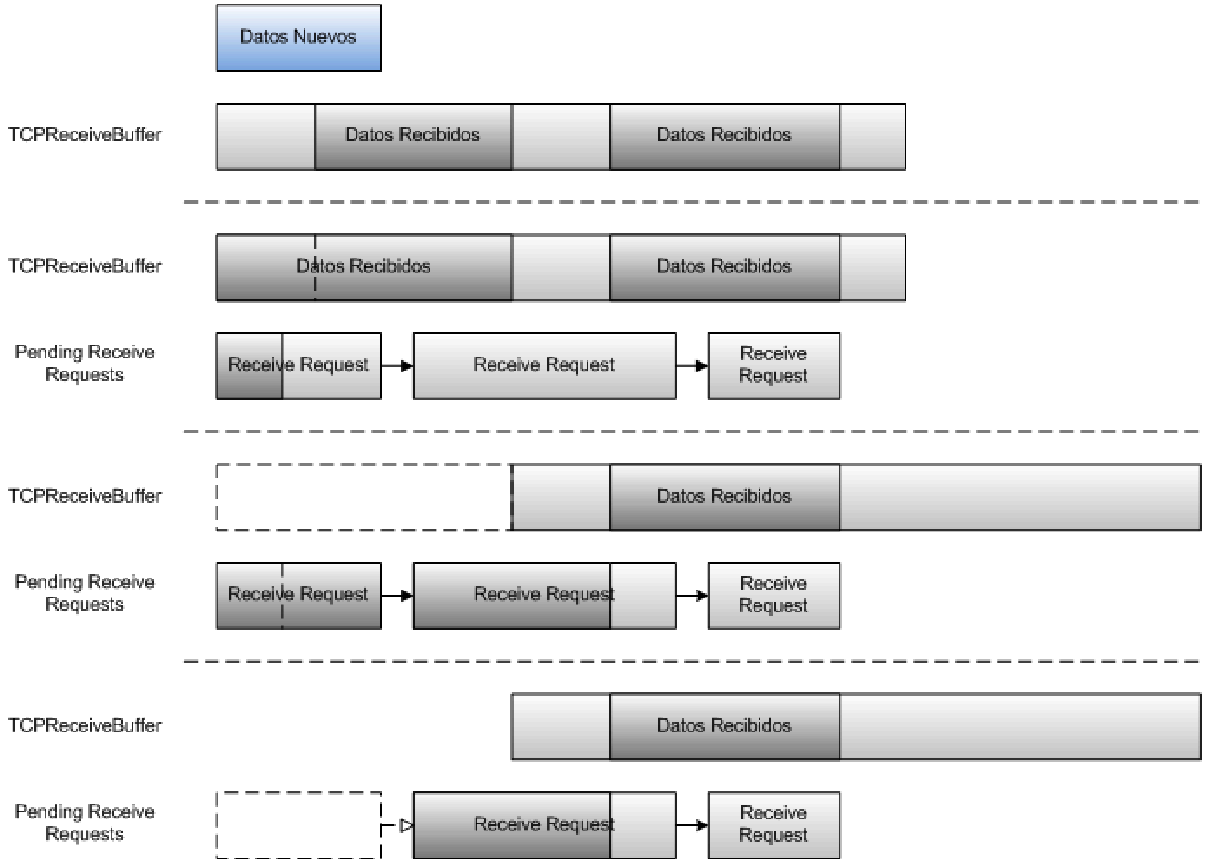


Ilustración 60: Transferencia entre TCPReceiveBuffer y ReceiveRequests

Un caso particular ocurre cuando llega un segmento con la bandera de push activada, indicando que todos los datos que no se han entregado al usuario deben ser entregados. Al llegar este tipo de segmentos, lo primero que se hace, es almacenar el SN del segmento en la lista de pushes realizados, esta lista se encuentra ordenada por SN. Este elemento es mantenido en la lista hasta que se hallan recibido todos los datos hasta el SN almacenado, en cuyo caso además de pasarse los datos al receive request correspondiente se le indica al mismo que se ha realizado un push y se despierta el socket asociado. Este procesamiento puede ser apreciado en la siguiente imagen donde se muestra un ejemplo de ejecución:

Datos Nuevos con Push
TCPReceiveBuffer
Datos Recibidos
Datos Recibidos
Pushes recibidos
Push 9
Push 21

Datos Nuevos
TCPReceiveBuffer
Datos Recibidos
Datos Recibidos
Pushes recibidos
Push 9
Push 12
Push 21

TCPReceiveBuffer
Datos Recibidos
Datos Recibidos
Pushes recibidos
Push 9
Push 12
Push 21
Pending Receive Requests
Receive Request
Receive Request
Receive Request

TCPReceiveBuffer
Datos Recibidos
Pushes recibidos
Push 21
Pending Receive Requests
Receive Request con Push
Receive Request con Push
Receive Request

TCPReceiveBuffer
Datos Recibidos
Pushes recibidos
Push 21
Pending Receive Requests
Receive Request

Ilustración 61: Administración de pushes

- TCPSendBuffer

  Esta clase es la encargada de mantener todos los datos que se han enviado (o se están por enviar) y de los cuales no ha llegado confirmación de recepción del otro nodo. Esta clase implementa el buffer de retransmisión y mantiene todos los datos que se encuentran dentro de la ventana de envío. En el caso donde se achica la ventana de envío, este buffer también mantiene aquellos datos que se encontraban en la ventana de envío anterior (esto se realiza mediante el uso de la variable max de envío que mantiene el máximo SN de datos enviado).

  Como se mencionó en la explicación de la clase SendRequest, al llegar el pedido de envío, el mismo es encolado junto con los otros SendRequest. En el caso de que no haya otros SendRequest y haya espacio en TCPSendBuffer, se pasan los datos desde el SendRequest al TCPSendBuffer. Esto también ocurre en el caso donde se acaba de liberar espacio en el TCPSendBuffer por la llegada de un ACK que se estaba esperando, en cuyo caso se corre el TCPSendBuffer hasta el ACK que ha llegado y queda espacio libre a ser asignado por datos del primer (o primeros) SendRequest de la cola.

  En la siguiente imagen se muestra este proceso:

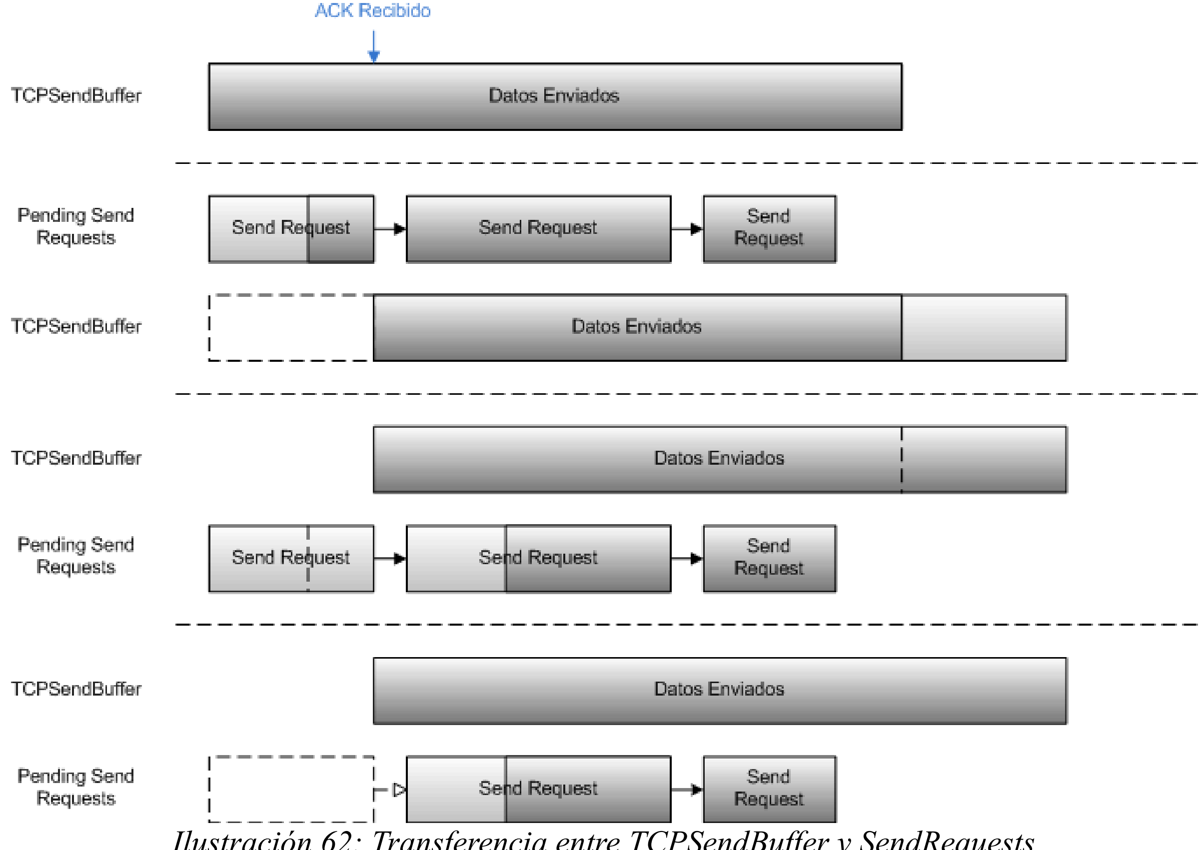


Ilustración 62: Transferencia entre TCPSendBuffer y SendRequests

Administración de formato de datos

A continuación se muestran las clases implicadas en la administración del formato de los datos:

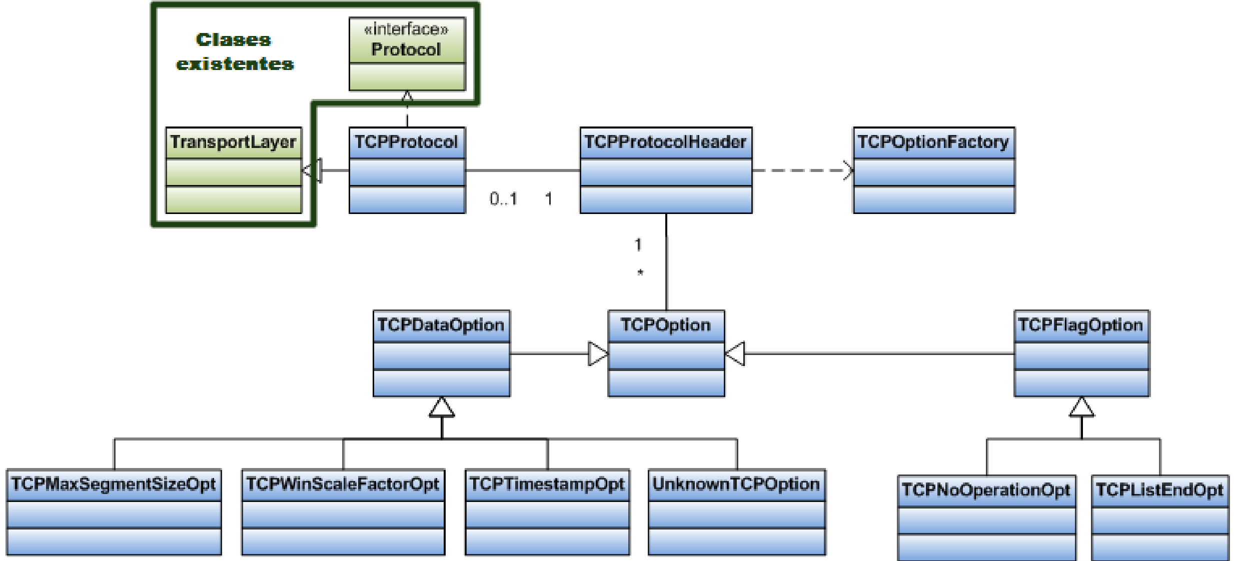


Ilustración 63: Clases de Administración de formato de datos

En este diagrama no se detallan todas las dependencias desde TCPOptionFactory a cada una de las clases de las opciones TCP.

Descripción de las clases:

- TCPOptionFactory

  Esta clase es utilizada para, a partir de una tira de bytes, parsear todas las opciones (y obtener la lista de instancias de clases correspondiente) contenidas en la misma.

- TCPProtocolHeader

  Esta clase es la encargada de, a partir de una tira de bytes, obtener la información de control del cabezal de TCP además de los datos incluidos. Adicionalmente permite a partir de la información deseada del cabezal y los datos a incluir, construir la tira de bytes del segmento correspondiente.

- TCPOption

  Clase abstracta utilizada para representar una opción TCP.

- TCPFlagOption

  Clase abstracta utilizada para representar aquellas opciones TCP que no contienen

datos. Estas opciones tienen tamaño uno.

- TCPNoOperationOpt

  Clase que representa la opción de TCP No Operation (que se define en el RFC 793), la cuál permite especificar que dicha opción debe ser salteada sin realizar procesamiento alguno. Es usualmente utilizada para rellenar y alinear opciones a palabras de 32 bits. Con esta clase se puede parsear a partir de una tira de bytes esta opción y además obtener la tira de bytes que la representa.

- TCPListEndOpt

  Con esta clase se representa la opción List End (la cual se define en el RFC 793), utilizada para marcar el fin de una lista de opciones. Al igual que la clase anteriormente descrita se utiliza para parsear esta opción a partir de una tira de bytes y obtener la tira de bytes que la representa.

- TCPDataOption

  Clase abstracta que representa todas las opciones TCP que contienen datos.

- TCPMaxSegmentSizeOpt

  Clase utilizada para representar la opción Maximum Segment Size (definida en el RFC 793) que permite especificar el máximo tamaño de segmento a ser utilizado en la conexión a la hora de ser establecida. Al igual que las otras clases que representan opciones, permite parsear a partir de una tira de bytes los datos de la opción y a demás a partir de los datos de la opción obtener la tira de bytes correspondiente.

  El tamaño máximo de segmento a ser negociado con el otro extremo debería es calculado a partir de la capacidad de la red gracias a la información dada por el protocolo Path MTU sobre los MTU de los diferentes caminos. Como inicialización la implementación toma un tamaño máximo de segmento establecido por defecto y luego de negociar la conexión, establece su tamaño máximo de segmento como el mínimo entre el valor por defecto y el nuevo valor ofrecido por el otro nodo. Luego, debido a cambios en la red, este tamaño puede ir cambiando debido a llegada de mensajes ICMPv6 PacketTooBig.

- TCPWinScaleFactorOpt

  Clase utilizada para representar y procesar la opción Win Scale Factor definida en el RFC 1323. Esta clase permite utilizar ventanas de tamaño no representable con 16 bits, lo cual se utiliza en redes con altos anchos de banda o con grandes delays.

- TCPTimeStampOpt

  Esta clase permite procesar la opción Time Stamp que se define en el RFC 1323 utilizada para calcular el tiempo de ida y vuelta para luego ser utilizado en el cálculo

de tiempo de retransmisión.

- UnknownTCPOption

  Esta opción ficticia permite omitir todas aquellas opciones no reconocidas por el stack.

10.4.6.3 Atención de Eventos

A continuación, en esta sección, se detallará como se realiza el flujo de control principal entre las clases implementadas, a la hora de suceder alguno de los posibles eventos.

Creación de socket

Cuando se crea una instancia de la clase Socket o ServerSocket, como ya se mencionó, esta crea una instancia de la implementación a utilizar en base al SocketImplFactory inicializado en la clase. En nuestro caso consideraremos que el SocketImplFactory será alguno de los implementados. A continuación la instancia de SocketImplFactory crea una instancia de IP4JVMSocket y esta última inicializa las estructuras y variables que se utilizaran posteriormente para cumplir con las funcionalidades requeridas según el tipo de socket creado. Entre los principales aspectos de inicialización se encuentran: la inicialización de valores de timers por defecto (conexión y envío de datos en 5 minutos, tiempo de cierre de conexión, colección de socket, etc). Además esta instancia pide a la instancia de TCPSocketManager que cree una instancia de TCB y le retorne el identificador de la misma.

La clase TCPSocketManager mantiene un entero que incrementa en uno cada vez que crea una instancia TCB el cuál es utilizado como identificador de la instancia.

Bind

Al invocarse esta función en la clase Socket, la misma verifica que se pueda realizar dado el estado del socket, principalmente verifica que no se haya realizado un bind anteriormente. Además hace verificaciones superficiales sobre la dirección pasada y luego invoca al método bind de la implementación, para finalmente cambiar su estado a uno que indica que ya se realizó un bind. Al ser invocado el método bind de la instancia de IP4JVMSocket, esta verifica las restricciones sobre los puertos y direcciones, en caso de que no se haya asignado un puerto local pide a la instancia de TCPSocketManager que le de uno libre. En el caso que la dirección dada ya se encuentre en uso o que sea una dirección invalida (ya que la IP dada no pertenece a alguna de las interfaces) se tira una excepción conteniendo un mensaje indicativo. Luego de este procesamiento de direcciones, se pasan los datos del bind a la instancia de TCPSocketManager que actualiza la información almacenada sobre la conexión correspondiente y le pasa la nueva dirección y puertos locales a la instancia de TCB para que esta actualice también su información.

Para la asignación de puertos efímeros, la instancia de TCPSocketManager mantiene un entero. Al pedir un nuevo puerto efímero se va incrementando recorriendo todos los puertos desde el último brindado hasta el 5000 (el máximo puerto utilizado para puertos efímeros), si no se encuentra ninguno libre se continua desde el 1025 (el primero no reservado) hasta recorrer todos o encontrar uno que no se esté utilizando.

En el caso de que el bind se realice a una instancia de la clase ServerSocket, además de lo dicho con anterioridad, la instancia de ServerSocket invoca al listen de la implementación correspondiente. Al recibir esta llamada, la instancia de la clase IP4JVMSocket setea el atributo de cantidad de conexiones en paralelo máxima y pasa el llamado a la instancia TCPSocketManager que la redirige a la instancia de TCB adecuada. Finalmente la instancia de TCB se pone en estado Listen (como es indicado en la máquina de estados que se mostró en la sección que explica el RFC 793).

Accept

Cuando se invoca un accept, la instancia de ServerSocket verifica que se pueda realizar dado el estado del socket. Luego crea una instancia de la implementación actualmente utilizada y pasa como parámetro esta instancia a la invocación de accept que de la implementación de socket asociada. Al recibir esta invocación la instancia de IP4JVMSocket, verifica que haya conexiones pendientes en espera de que se realice un accept de las mismas, en el caso de que no haya, se bloquea el procesamiento hasta que aparezca alguna. Luego de esto se invoca a una función de la instancia de TCPSocketManager con el indicador del TCB de la conexión que está esperando a ser aceptada, la cuál invoca a otra función de la instancia TCB que se encarga de copiar los datos al socket dado a partir de un socket temporal que se había asignado a la instancia de TCB.

Al llegar un pedido de conexión desde la red, la instancia de TCPSocketManager crea una instancia TCB y otra de IP4JVMSocket temporal para la conexión. Al finalizar el proceso de establecimiento de la conexión, si existen instancias de IP4JVMSocket pendientes de accept, se notifica a la instancia de IP4JVMSocket del socket padre de esta conexión (socket padre se llamará al socket que se encuentra en estado listen esperando conexiones) la cual ingresa el identificador del TCB a la cola de conexiones pendientes de ser aceptadas y notifica cualquier thread (solo notifica a uno) que esté esperando en la cola.

Connect

Cuando se realiza un connect en una instancia de la clase Socket, esta verifica superficialmente que la dirección dada sea correcta, que el parámetro de timeout también lo sea y además el estado del socket (no se haya cerrado y no esté conectado). Luego de esto la instancia redirige el pedido a la instancia de SocketImpl configurada y pone su estado como conectado. IP4JVMSocket verifica que se pueda realizar el connect dado el estado del socket (no se esté ya en proceso de conexión o conectado), si el puerto dado es igual a cero se invoca al bind para que asigne uno y una dirección IP. Finalmente se redirige el pedido a la instancia de TCPSocketManager que lo redirige a la instancia de TCB correspondiente indicando el ISS a utilizar (habiendo incrementado dicho valor en 64000), y la instancia de IP4JVMSocket bloquea el thread invocador hasta que se establezca la conexión o hasta que venza el tiempo de espera

establecido. La instancia de TCB inicializa las variables necesarias e inicia el proceso de saludo de tres vías. Luego de finalizado el proceso de saludo de tres vías la instancia de TCB notifica a la instancia de IP4JVMSocket que desbloquea el thread que invocó primeramente al connect.

Envío de datos

Como se mostró con anterioridad los sockets brindan una función que permite obtener una instancia de OutputStream a ser utilizada para enviar datos a través de los mismos. Para esto la clase Socket (no se brinda esta operación en los ServerSocket) verifica primero que el estado del socket sea adecuado (se haya conectado) para realizar cualquier envío. Luego de esto se redirige el pedido de instancia de OutputStream a la instancia de implementación configurada. En el caso de IP4JVMSocket se verifica que el socket no esté cerrado o cerrándose y en el caso de que no lo esté se retorna una instancia de IP4JVMOutputStream.

La instancia de IP4JVMOutputStream al recibir un pedido de envío de datos (un write) redirige el mismo a la instancia de IP4JVMSocket asociada, esta verifica que se puedan enviar datos dado el estado del socket, y en el caso de que así sea, redirige el pedido a la instancia de la clase TCPSocketManager la cuál lo redirige a la instancia de TCB correspondiente. La instancia de TCB, como ya se explicó en la sección donde se detallan las clases, crea una instancia de SendRequest con los datos del pedido realizado y pone dicha instancia al final de la cola de SendRequests que mantiene. Con el tiempo los datos de este SendRequest se irán pasando a la instancia de TCPSendBuffer y enviándose hasta que lleguen satisfactoriamente a su destino.

Recepción de datos

La recepción de datos es muy similar al envío. Para esto los sockets brindan una función para obtener el InputStream utilizado para recibir datos. La clase Socket (en este caso tampoco se brinda la operación para ServerSocket) verifica que sea posible recibir datos dado el estado del socket. A continuación se pide la instancia de InputStream a la implementación de socket configurada. IP4JVMSocket verifica que dado el estado del socket (no esté cerrado ni cerrándose) se puedan recibir datos y finalmente retorna una instancia de IP4JVMInputStream.

Cuando se invoca un read (para la recepción de datos) en la instancia de IP4JVMInputStream esta invoca al método receive de la instancia de IP4JVMSocket asociada la cuál redirige el pedido adecuadamente a la instancia TCB a través de la instancia de TCPSocketManager y bloquea el thread invocador del receive. Como también se explicó en la sección donde se detallaban las clases, la instancia de TCB crea una instancia de ReceiveRequest con los datos sobre el pedido de recepción y encola dicha instancia a la cola de los pedidos de recepción pendientes. Con el tiempo el buffer del ReceiveRequest se irá llenando, luego de lo cual será quitado de la cola y se notificará a la instancia de IP4JVMSocket sobre el fin de la recepción la cuál entonces desbloqueará el thread que invocó el receive.

Close

El close puede ser invocado, como se vio antes, desde cualquiera de las 4 clases brindadas

por Java para la utilización de sockets. Las clases al recibir este pedido verifican que no se esté cerrando ya o esté cerrado el socket. En caso de que no sea así, invocan de una forma u otra al método close de la instancia de IP4JVMSocket asociada. Esta, si se trata de un servidor, aborta todas las conexiones que tenga pendientes de aceptación o que se estén estableciendo por un connect realizado desde otro nodo. Luego se invoca al método disconnect de TCPSocketManager que redirige el pedido a la instancia de TCB que inicia el proceso de desconexión. Además la instancia de IP4JVMSocket bloquea el thread invocador hasta que se cierre efectivamente la conexión o hasta que venza el temporizador de linger (utilizado para esperar durante un cierto tiempo el cierre de la conexión), este temporizador se puede setear mediante el uso de la opción SO_LINGER del socket.

Recepción de segmentos desde la red

En primera instancia los segmentos que vienen desde la red (capa IP) son recibidos por la clase TCPProtocol que pasa los datos del Job correspondiente a la instancia de TCPProtocolHeader para que los parsee adecuadamente. Luego se agrega a si mismo a la lista de protocolos que han procesado el Job y retorna el Job al stack. Luego de esto el Job es recibido por la clase TCPSocketManager que verifica la integridad de los datos del segmento (mediante el uso de la suma de comprobación que contiene el cabezal). A continuación la instancia de TCPSocketManager compara las direcciones y puertos del cabezal con los del TCB que se uso por última vez, en caso de que coincidan se utiliza este TCB (esta es una optimización importante en el caso de que se cuente con muchas instancias de TCB pues, como se verá, la búsqueda del destino de los datos recorre todas las instancias). En caso de que no sea esta, se itera sobre todas las instancias de TCB que se tienen registradas. El algoritmo de búsqueda utilizado se muestra a continuación:

```
Function tcb_lookup(int dPort,IP dIP, int oPort, IP oIP) returns int;

int wildcard;
int matchwild=3;
int match= 0;
TCB currTcb;
//se obtienen los TCBs que están asociados al puerto destino dado
List l = obtener_TCBs(dPort);
while (l.hasNext() and matchwild!=0){
    currTcb= l.next();
    wildcard= 0;
    //si la IP local del TCB es cualquiera de las locales (o no esta especificada)
    if (not currTcb.getLocalIPAddress().isAnyLocalAddress()){
        //si la IP de destino es cualquiera de las locales (o no esta especificada)
        if (dIP.isAnyLocalAddress())
            wildcard++;
        //sino si la dirección local no es igual a la dirección destino del segmento
        else if (not currTcb.getLocalIPAddress() = dIP)
            //pasar a probar con otro TCB
            continue;
    } else {
        //si la IP de destino es cualquiera de las locales (o no esta especificada)
        if (not dIP.isAnyLocalAddress())
            wildcard++;
    }

    //si la IP remota del TCB es cualquiera de las locales (o no esta especificada)
    if (not currTcb.getRemoteIPAddress().isAnyLocalAddress()){
        //si la IP origen es cualquiera de las locales (o no esta especificada)
        if (oIP.isAnyLocalAddress())
```

```
            wildcard++;
        //sino si la dirección remota no es igual a la dirección origen del segmento
        // o si el puerto remoto no es igual al puerto origen
        else if (not currTcb.getRemoteIPAddress() = oIP) or
                not currTcb.getRemotePort = oPort)
            //pasar a probar con otro TCB
            continue;
    } else {
        //si la IP de origen es cualquiera de las locales (o no esta especificada)
        if (not oIP.isAnyLocalAddress())
            wildcard++;
    }

    //Si currTCB matchea mejor (utiliza menos wildcard para matchear)
    // que el último hallado entonces es el mejor hasta ahora hallado
    // y un candidato a ser el destinatario
    if (wildcard<matchwild){
        match= currTcb.getId();
        matchwild= wildcard;
    }
}

//Se retorna el destinatario del paquete, en caso de que no se haya encontrado ninguno
// se retornará cero
return match;

end-function;
```

En el caso de que no se encuentre ningún TCB que corresponda se responde, como lo indica el RFC 793, enviando un RST.

Si la conexión está cerrada se descarta el paquete.

Si el socket asociado es instancia de ServerSocket entonces se crea una nueva conexión temporal para procesar el segmento que ha llegado. Esta conexión perdurará en el caso de que el segmento recibido sea uno de pedido de establecimiento de conexión de lo contrario se abortará la conexión.

A continuación de todo este procesamiento, en cualquier caso que se deba continuar con el procesamiento del segmento, el mismo es pasado al TCB que corresponda. Antes de realizar un procesamiento paso a paso de los datos que contiene el segmento, se implementó un algoritmo de predicción de cabezales con el cuál se evita todo el procesamiento innecesario en el caso de tratarse de una comunicación unidireccional. Básicamente el algoritmo contempla dos casos:

- si el segmento que ha llegado es puramente datos, son los datos esperados, no hay nada en la cola de re ensamble (TCPReceiveBuffer) y hay espacio como para recibirlo, se pasa directamente al TCPReceiveBuffer sin mas procesamiento, el cual lo pasará como ya se mostró a los ReceiveRequest que estén pendientes.

- Si el segmento que ha llegado es puramente un ACK, entonces se corre la ventana de envío y se pasan datos al TCPSendBuffer como ya se vio a partir de los SendRequest pendientes y se envían los datos.

El algoritmo, aunque es corto, se prefiere no presentarlo en detalle, el mismo se encuentra detallado en el libro TCP Illustrated Vol 2 de Addison-Wesley.

Gestión de timers

Como se vio anteriormente, en el RFC 793 se especifican 3 timers, pero estos no son los únicos timers utilizados en esta implementación. Como ya se mencionó todos los timers utilizados en esta implementación se gestionaron con 2 instancias de TimerTask, una para el timer denominado fasttimo y otra para el slowtimo. Además de estas dos instancias de timers cada TCB mantiene una variable que indica si se desea realizar piggy backing (explicado más adelante) y un arreglo de enteros para el cuál cada posición representa la cantidad de ticks que restan para que el timer de esa posición venza. En la implementación realizada se contemplaron los siguientes temporizadores:

- Establecimiento de conexión: temporizador iniciado luego del envío del primer SYN para el establecimiento de la conexión, si no se recibe una respuesta en 75 segundos la conexión se aborta.

- Retransmisión: es seteado cuando TCP envía datos. El valor asignado es calculado de manera dinámica a partir de la siguiente formula:

  delta = nticks - srtt
  srtt = srtt + g*delta
  rttvar = rttvar + h*(|delta| - rttvar)
  RTO = srtt + 4*rttvar

  - delta
    Diferencia entre el tiempo de ida y vuelta medido, menos el actual valor del estimador suavizado de RTT (round trip timer, tiempo de ida y vuelta).
  - nticks
    Este valor es calculado a partir del tiempo de ida y vuelta obtenido gracias a medidas realizadas utilizando la opción de TCP Time Stamp mencionada en secciones anteriores.
  - srtt
    Estimador suavizado de RTT
  - g
    Ganancia aplicada al estimador RTT (este valor es inicializado en 1/8).
  - h
    Ganancia aplicada al estimador de desviación media (inicializado en 1/4).
  - rttvar

Estimador de varianza promedio. Es una buena aproximación a la varianza estándar y además es más fácil de calcular ya que no requiere realizar operaciones de raíces cuadradas.

A la formula antes planteada se aplica exponential backoff. Esto consiste en ir multiplicando por un valor que crece exponencialmente en la serie 1,2,4,8,.... Esto hace que cada vez que vence el temporizador, el mismo se vuelva a iniciar, pero en esta ocasión con aproximadamente el doble del valor utilizado la última vez. El valor multiplicado a la fórmula crece hasta llegar a 64, después de lo cual se mantiene constante.

Si el temporizador vence los datos son re transmitidos.

Para una explicación detallada sobre el uso de este timer y la fórmula que define el valor utilizado para su inicialización se puede consultar el RFC 793 y el libro TCP Illustrated Vol 2 de Addison-Wesley.

- Persist Timer: este timer es utilizado en el algoritmo que evita el problema de la ventana tonta descrito en la siguiente sección (Algoritmia adicional). En dicha sección se detallará su cometido. Por lo pronto cabe destacar que el valor con el cual se inicializa se calcula de la misma manera que el timer de retransmisión.

- Keepalive: este temporizador es utilizado para, que luego de pasadas 2 horas de inactividad, enviar un paquete especial para verificar que la conexión con el otro nodo se mantiene establecida.

- FIN_WAIT_2: se utiliza este temporizador para evitar que la conexión quede por siempre esperando la llegada de un segmento de FIN en el caso de que nunca llegue. Luego de entrar al estado FIN_WAIT_2 se inicializa este timer, si al correr de 10 minutos no sucede nada el timer vence y se vuelve a iniciar pero esta vez en 75 segundos, si llega a vencer luego de estos 75 segundos la conexión es cerrada definitivamente.

- TIME_WAIT: también llamado timer 2MSL debido a que se inicializa en este valor al entrar en el estado TIME_WAIT. Es utilizado para que la conexión no quede indefinidamente en este estado. Luego de vencido el timer la conexión se cierra definitivamente.

- ACK retardado: este timer permite la implementación de Piggy Backing. El proceso de piggy backing permite hacer un mejor uso de los recursos de red a la hora de enviar un ACK a un nodo. En el caso de que no se tengan datos para enviar a la hora de enviar un ACK, se espera un cierto tiempo hasta que o bien venza el timer de piggy backing (o ACK retardado) o lleguen datos a ser enviados con el ACK. Esto permite hacer un mejor uso de la posibilidad que brinda TCP de mandar datos conjuntamente con un ACK a la hora de mandar un segmento.

Los primeros 6 timers son implementados por el timer slowtimo y el último por fasttimo.

Al vencer slowtimo, como ya se comentó, este invoca a la función slowtimo de la instancia TCPSocketManager. Esta, además de incrementar en uno el timestamp del sistema (utilizado en las opciones TCP Time Stamp) e incrementar en 64000 el valor de la variable utilizada para inicializar los ISS, recorre todos los TCBs que tiene asociados invocando para cada uno la función slowtimo. Al invocarse esta última función en una instancia TCB esta decrementa en uno todas las posiciones del arreglo que no sean cero. Si una posición llega (luego del decremento) a cero, entonces significa que el temporizador ha vencido y se realiza el procesamiento necesario para manejar este evento.

Los 6 temporizadores son distribuidos en el arreglo en 4 posiciones. Esto se puede realizar de esta manera ya que algunos de los temporizadores son mutuamente excluyentes.

A continuación se brinda una tabla donde se muestra cómo se distribuyen los temporizadores en cada una de las posiciones y cómo se denomina a cada una de las posiciones:

	Estable. de conexión	Retransmi_ sión	Persist Timer	Keepalive	TIME_WAIT _2	TIME_WAIT
TCPT_RXMT		X				
TCPT_PERSIST			X			
TCPT_KEEP	X			X		
TCPT_2MSL					X	X

Como también ya se comentó al vencer fasttimo se invoca a la función fasttimo de la instancia de TCPSocketManager. A continuación esta instancia invoca para cada uno de los TCBs que tiene asociados la función fasttimo de las mismas. Estas últimas luego de invocada esta función verifican si se encuentra levantada la bandera indicadora de que se encuentra un ACK pendiente de envío (debido al uso de piggy backing) en cuyo caso se baja la bandera y se envía un segmento conteniendo el ACK y sin datos (pues no hay, porque si habría ya se hubiera mandado el ACK con esos datos).

10.4.6.4 Algoritmia adicional

En esta sección se detallan aquellos aspectos que se agregaron a la lógica descrita en el RFC 793 y que no se han comentado aún.

Congestion avoidance

En la implementación realizada se cubrió gran parte de los algoritmos utilizados por Net/3 para obtener un mejor aprovechamiento de los recursos de la red. Net/3 implementa Slow Start, Congestion Avoidance, Fast Retransmit y Fast Recovery. En todos los casos en los que se

probó se obtuvo el comportamiento deseado con respecto a estos algoritmos.

En la siguiente figura se muestra cómo debería ser la evolución de la ventana de congestión cuando estos algoritmos se encuentran implementados:

Evolución de Ventana de Congestión

Ilustración 64: Evolución de Ventana de Congestión

Slow Start: Cuando la conexión se encuentra en slow start para cada ACK de segmento que llegue se incrementa la ventana de congestión en mss (maximum segment size de la conexión). Esto produce un crecimiento exponencial de la carga impuesta a la red mientras se corra este algoritmo. El criterio de parada de este algoritmo es básicamente la detección de la perdida de algún paquete en la red, lo que puede llegar a ser un indicador de que la red se encuentra congestionada, o el pasaje de la marca de threshold que se usa para estimar cuanto es la capacidad de la red. Luego de pasar el threshold el algoritmo de congestion avoidance realiza un crecimiento lineal de la ventana de congestión. Por el contrario si se detecta que se ha perdido algún paquete se setea la ventana de congestión igual a la mitad actual de la que tiene y el threshold se setea a este valor.

Fast Retransmit: al llegar 3 ACK consecutivos repetidos TCP induce que se han perdido segmentos y es por esto que se están repitiendo los ACK, por lo cual retransmite nuevamente todo desde el último SN que ha sido confirmado como recibido (excluyendo este último) hasta el último que se ha enviado.

Fast Recovery: luego de realizar Fast Retransmit se hace Congestion Avoidance pero no se hace Slow Start sino que se realiza un crecimiento lineal de la ventana de congestión. El tamaño de la ventana al reducirse lo hace a la mitad del tamaño actual más la cantidad de bytes que el otro extremo de la conexión ha confirmado como recibidos mediante el envío de

ACK.

A pesar que el stack no presenta el comportamiento esperado tomando en cuenta los tiempos, si los presenta tomando en cuenta otros factores determinantes de la evolución de la ventana, como lo es la evaluación de la ventana luego de la llegada de un ACK. Esto se puede apreciar en las siguientes imágenes donde se muestra, primero la evolución del tamaño de la ventana de congestión por unidad de tiempo y por ACK recibido en un caso normal, y a continuación las mismas gráficas pero en el caso que suceden retransmisiones de paquetes.

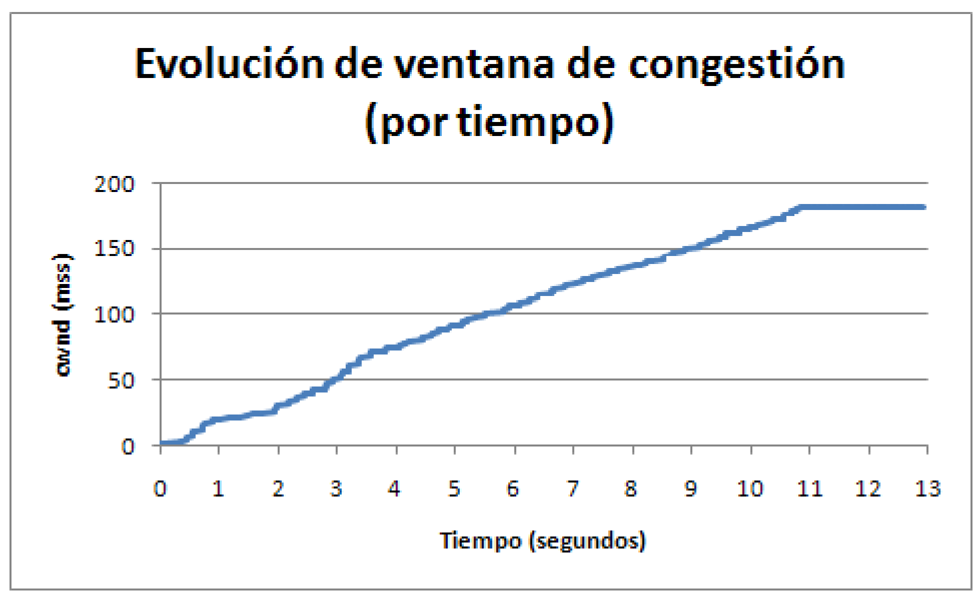


Ilustración 65: Evolución de ventana de congestión (por tiempo)

Leyendas:

- cwnd (congestion window): tamaño de la ventana de congestión.
- mss (maximum segment size): tamaño máximo de segmento.

En la imagen mostrada (de una de las transmisiones) se puede ver que no se presenta el comportamiento que se debería apreciar según el comportamiento "ideal" de slow start. Sin embargo en la imagen que se muestra a continuación se puede ver claramente que el crecimiento de la ventana de congestión presenta una relación lineal con respecto a la cantidad de ACK recibidos (se incrementa en 1 mss el cwnd por cada ACK recibido), lo cual es el comportamiento esperado del algoritmo (slow start). Esto nos lleva a concluir que la evolución de cwnd con respecto al tiempo se ve afectada por delays o llegada tardía de ACKs, lo que lleva a una evolución casi lineal de cwnd con respecto al tiempo (cuando debiera ser exponencial).

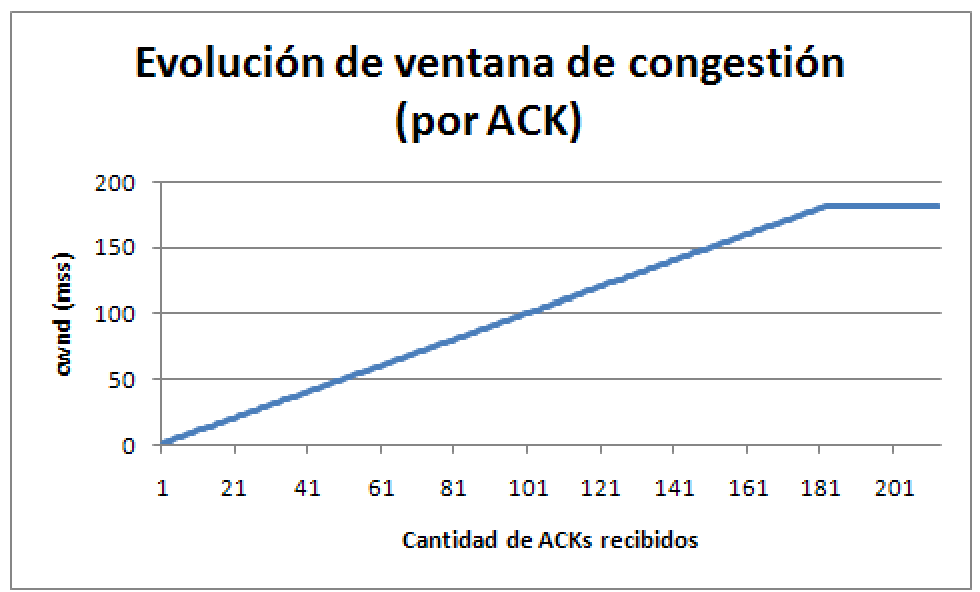


Ilustración 66: Evolución de ventana de congestión (por ACK)

A modo de anotación cabe destacar que el threshold usado en la implementación se inicializa con un valor máximo el cual no será nunca alcanzado. Luego de vencer un timer de retransmisión, o llegar 3 ACKs duplicados, esta se actualiza con el valor correspondiente, como ya se explicó.

Otro aspecto que se puede ver en ambas imágenes, y que no se había detallado anteriormente, es que existe un valor máximo para la ventana de congestión dado por la formula MAX_WND_SIZE (el máximo representable en 16 bits) multiplicado por un factor de escalado obtenido en la negociación de la conexión a través de la opción TCPWinScaleFactorOpt.

En las imágenes mostradas a continuación se ve el comportamiento obtenido en una de las conexiones cuando se suceden retransmisiones de paquetes:

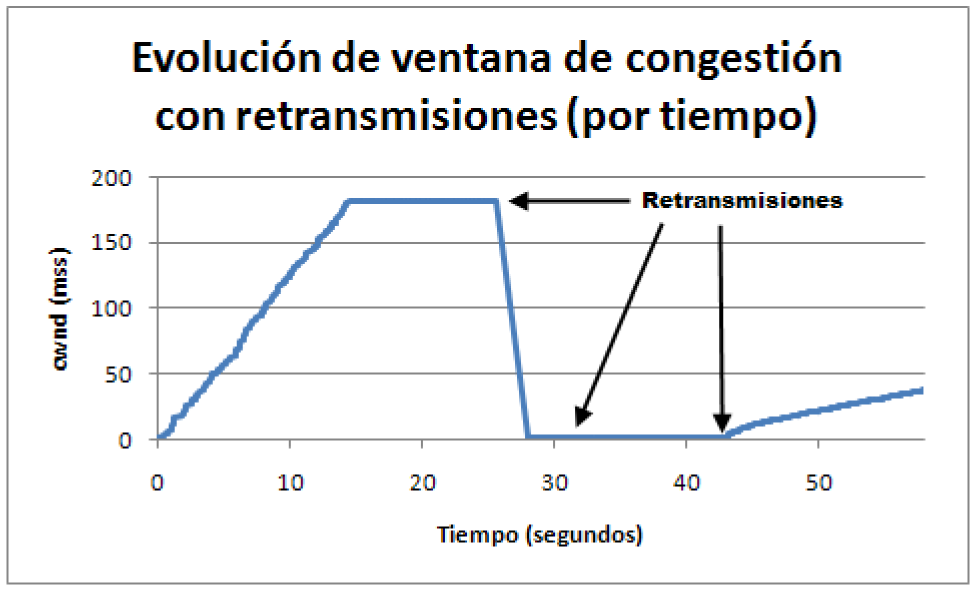
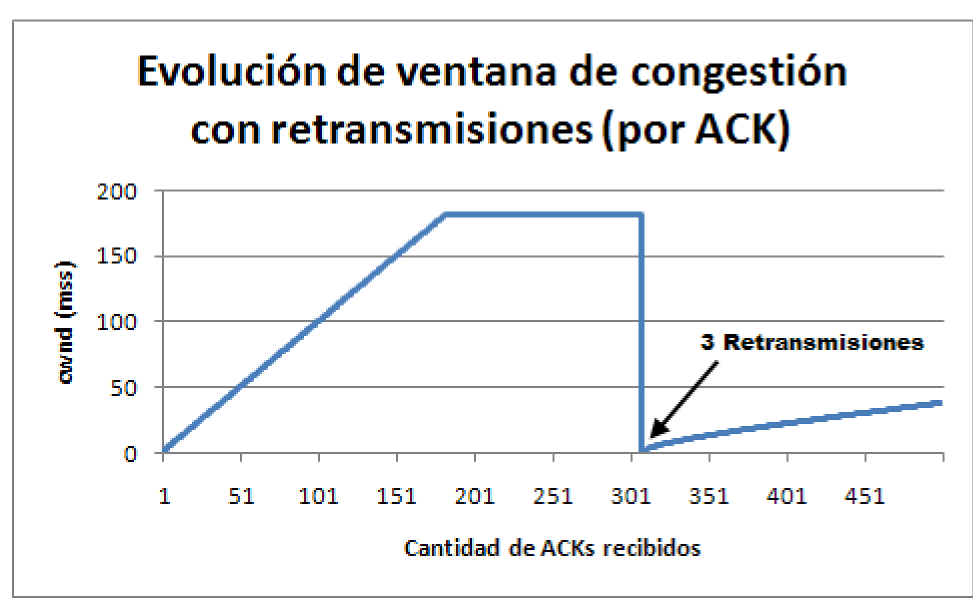


Ilustración 67: Evolución de congestión con retransmisiones (por tiempo)

En la imagen mostrada se puede ver la disminución de cwnd luego de la primer retransmisión realizada. Un aspecto a anotar, es que el threshold luego de la primer retransmisión se inicializa a ½ del tamaño de cwnd, luego de la segunda, el cwnd se encuentra inicializado a 1 mss, por lo que el threshold luego de esta retransmisión se sitúa a 2 mss (el valor mínimo). Es por esto que al reiniciar la transferencia, el cwnd crece de una forma lineal pero con menor inclinación que en un slow start (pues aquí se encuentra en acción el algoritmo de fast recovery). En la siguiente imagen se muestra la evolución en función de la cantidad de ACKs recibidos:



Algortimo de Nagle

Se implemento el algoritmo de Nagle para evitar el síndrome de la ventana tonta. Este algoritmo utiliza un timer y una bandera. El algoritmo de Nagle consiste en que cuando la ventana de recepción del otro extremo de la conexión es 0, se envía de a un solo byte cada vez que vence el timer del algoritmo. El timer de retransmisión es deshabilitado durante este proceso. En la siguiente figura se muestra el problema del síndrome de la ventana tonta:

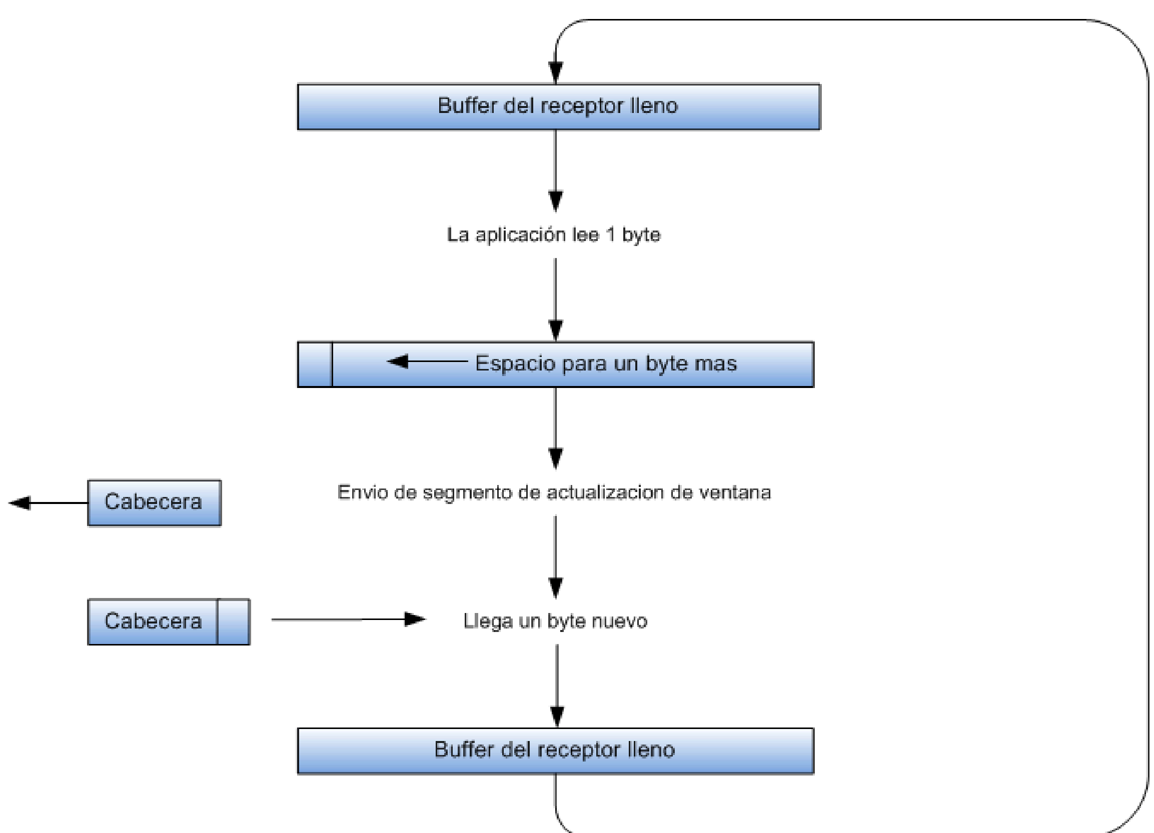


Ilustración 68: Síndrome de la ventana tonta

Algoritmo de PAWS (protection against wrapped sequence numbers)

Este algoritmo permite a través del uso de la opción TCP de Timestamp definir si un segmento es válido, se valida tanto el SN como el ACK. Esto es especialmente importante en redes de alta velocidad donde el número de secuencia puede llegar a repetirse luego de 17 segundos. El algoritmo plantea que al llegar un segmento que contenga respuesta de la opción Timestamp, el mismo será descartado si el campo val de la opción Time Stamp es menor que el valor de la variable ts_recent. Esta variable es inicializada al llegar un pedido válido de respuesta de la opción Time Stamp con el valor del campo val de dicha opción. Este algoritmo es descrito en el RFC 1323 y en el libro TCP Illustrated Vol2 de Addison-Wesley.

10.4.7 Conclusión del trabajo realizado

Se logró desarrollar una implementación casi completa de TCP, integrada al framework existente de IP4JVM. Además se pudo integrar dicha solución a la implementación de sockets estándar de la máquina virtual y hacer que fuera parametrizable, a la hora de la ejecución de la máquina virtual, la selección de la implementación de socket a utilizar.

Algunos aspectos quedaron sin implementar como el uso de algunos parámetros y funcionalidades que brinda la interfaz Java de sockets, como por ejemplo recibir paquetes con bandera urgent seteada en uno como si fueran normales o descartarlos en otro caso (en la actual implementación se tratan como datos normales). Otros aspectos quedaron sin testear adecuadamente, como lo fueron los algoritmos utilizados para congestion avoidance. Pero se pudieron realizar pruebas de transferencia de archivos y el posterior soporte del servidor Tomcat, lo cuál indica que la implementación realizada es buena y cumple con las expectativas.

10.4.8 Trabajo futuro

Quedan algunos aspectos por cubrir en caso de que se desarrolle un trabajo futuro. Dentro de estos aspectos se pueden identificar los siguientes:

- Testear más exhaustivamente la correctitud de los algoritmos de congestion avoidance viendo las cargas impuestas a la red.
- Testear la implementación con una configuración de la capa de red que contenga varias MACs e IPs. En las pruebas realizadas se utilizo una única MAC.
- Mejorar el manejo de errores, y opciones TCP lo cuál no fue cubierto en su totalidad.
- Probar la implementación con otras clases utilizadas para el manejo de sockets y la información a utilizar y además con otras aplicaciones diferentes de Tomcat.
- Realizar pruebas de performance de la implementación para tener un estimativo de cuan eficiente es.

10.5 Configuración y ejecución de Apache Tomcat con IP4JVM

10.5.1 Introducción

En este apéndice se presentan pasos a seguir para configurar, ejecutar y posteriormente testear la ejecución del servidor http Apache Tomcat haciendo uso del stack IP4JVM.

10.5.2 Pasos a Seguir

1. Se debe tener el entorno de desarrollo correctamente instalado y la máquina virtual compilada e instalada como se indica en los apéndices 10.1 y 10.2.

2. Descargar la última versión disponible de Apache Tomcat. Al momento de la realización del presente documento la versión utilizada fue la 6.0.18:

http://www.eng.lsu.edu/mirrors/apache/tomcat/tomcat-6/v6.0.18/bin/apache-tomcat-6.0.18.tar.gz
(último acceso 13/09/08)

3. Descomprimir el archivo en un directorio de preferencia. A la ruta del directorio creado al descomprimir el archivo denominaremos en el documento como TOMCAT_HOME.

4. Copiar dentro del directorio TOMCAT_HOME/bin los archivos de lenguaje necesarios. Se pueden copiar desde el directorio java/src del proyecto IP4JVM y son aquellos que tienen el formato “IP4JVMmessages<algo>.properties”, donde “<algo>” puede ser el string vacío o puede se un string que identifique el idioma.

5. Se deberá contar con un archivo de configuración de IP4JVM, el cual puede estar situado en cualquier ruta. A esta ruta, incluyendo el nombre del archivo, llamaremos IP4JVM_CONFIG. Para el presente documento se tomara como referencia un archivo de configuración de IP4JVM con la siguiente información:

```
#Configuration File
#The format of the file must be respected, one line starting with '-' is a division between kinds of options
# one starting with '#' is a line of comments
```

```
#General Settings (identifier, value)
#this option enables the use of MIPv6 support
#mipv6Enabled 1

-----------------------------
#Interfaces (identifier, type (1-eth, 2-lo, 3- tun), specific options)
#Specific Options:
#interfaces eth: physical interface identifier, MAC, mtu (optional, Default:1500)
#interfaces lo: (no options)
#interfaces tun: mapped interface, exit point, entry point, exit point prefix length

jth0 1 eth0 00:0C:29:E4:EA:88

-----------------------------
#IPS (identifier, ip, tentative (optional, Default:true))


jth0 2001:660:5503:276a::3
-----------------------------
```

6. A continuación se deberá configurar correctamente el servidor Tomcat para que utilice la dirección IP deseada (en el caso del archivo de configuración antes descrito esta será 2001:660:5503:276a::3). Para esto será necesario modificar el archivo server.xml contenido en la ruta TOMCAT_HOME/conf. A continuación se muestra en rojo los cambios que fueron necesarios realizar en el archivo de configuración teniendo en cuenta el archivo de configuración de IP4JVM:

```
<Connector address="2001:660:5503:276a::3" port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />
<Connector address="2001:660:5503:276a::3" port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

7. Ahora será necesario configurar adecuadamente las variables de entorno para utilizar adecuadamente IP4JVM en la máquina virtual OpenJDK compilada. Para esto será necesario configurar las siguientes variables de entorno como se muestra en la imagen:

```
export PATH= OPENJDK_COMPILED/bin:$PATH
export JRE_HOME= OPENJDK_COMPILED
export JAVA_HOME= OPENJDK_COMPILED
export JAVA_OPTS="-Djava.net.preferIPv6Addresses=true -Djava.net.preferIPv4Addresses=false -Djava.library.path=IP4JVM_LIB -Dip4jvm.config=IP4JVM_CONFIG -Dimpl.prefix=IP4JVM
```

En el anterior ejemplo se debería sustituir OPENJDK_COMPILED con la ruta de la máquina virtual compilada (como se especifica en el Apéndice 10.2). IP4JVM_LIB refiere a la ruta sistema operativo donde se encuentran las librerías compiladas

netmanagerread y netmanagerwrite (ver Apéndice 10.2).

8. Iniciar el servidor Tomcat, para esto es necesario ejecutar en una linea de comandos situada en la ruta TOMCAT_HOME/bin el script “startup.sh”.

9. Luego de esto sería necesario verificar que el servidor funciona correctamente. Como primer punto a se puede verificar el contenido del archivo “catalina.out” contenido en la ruta TOMCAT_HOME/logs. A continuación en el browser de otra máquina se puede realizar la consulta HTTP para verificar que la página de administración de tomcat funciona correctamente conjuntamente con todos los ejemplos contenidos en la misma. Para esto es necesario configurar adecuadamente la máquina para que pueda establecer la comunicación con el servidor tomcat, para esto es requerido que se asigne una dirección IP que pertenezca a la misma red que la configurada para el stack (prefijo de red 2001:660:5503:276a::/64) a alguna de sus interfaces de red, un ejemplo sería configurar la dirección 2001:660:5503:276a::4/64. Se debería verificar que el ping desde la nueva máquina hacia la dirección de IP4JVM, en nuestro caso 2001:660:5503:276a::3/64, responde correctamente. Luego de esto abriendo un navegador en la nueva máquina y escribiendo en la barra de direcciones la ruta hacia el servidor Tomcat, en nuestro ejemplo “http://[2001:660:5503:276a::3]:8080”, debería mostrar la siguiente página:

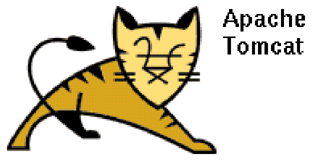


Ilustración 69: Página de bienvenida de Apache Tomcat

10. Probar con los diferentes ejemplos contenidos en los links marcados en la siguiente imagen:

Tomcat Online

Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC

Miscellaneous

Servlets Examples
JSP Examples
Sun's Java Server Pages Site
Sun's Servlet Site

the INSTALL file.

NOTE: For security reasons, using the administration webapp is restricted to users with role "admin". The manager webapp is restricted to users with role "manager". Users are defined in `$CATALINA_HOME/conf/tomcat-users.xml`.

Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.

Tomcat mailing lists are available at the Tomcat project web site:

- **users@tomcat.apache.org** for general questions related to configuring and using Tomcat
- **dev@tomcat.apache.org** for developers working on Tomcat

Thanks for using Tomcat!