ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Desarrollo de un simulador docente interactivo para el estudio de implementaciones TCP

AUTOR

Carlos María Heredia Conesa

DIRECTORA

Pilar Manzanares López

Septiembre **/** 2007

Autor	Carlos María Heredia Conesa
E-mail del Autor	letepetete@hotmail.com
Director(es)	Pilar Manzanares López
E-mail del Director	pilar.manzanares@upct.es
Codirector(es)	
Título del PFC	Desarrollo de un simulador docente interactivo para el estudio de implementaciones TCP
Descriptores	

Resumen

La incorporación de las tecnologías de la información y de la comunicación en la educación, y mayoritariamente en la educación universitaria, ha contribuido a crear nuevos marcos o modelos de enseñanza, que ofrecen una capacidad de aprendizaje permanente en tiempo y no presencial. Es por ello que surge la idea de desarrollar una aplicación docente interactiva para que el alumno adquiera, de forma autónoma y guida, los conocimientos propios del protocolo TCP y algunos de sus algoritmos más comunes de una forma sencilla, que además le permitirá ampliar el abanico de ejemplos de funcionamiento de las diferentes implementaciones del protocolo TCP estudiadas en clase.

Titulación	Ingeniería Superior de Telecomunicación
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Septiembre – 2007

Deseo realizar un agradecimiento especial a Francisco Jorge López Castilla por ayudarme desinteresadamente a la realización de este proyecto.

ÍNDICE

1. Introducción 1
1.1 Introducción 1
1.2 Objetivos 2
2. Teoría necesaria 3
2.1 TCP 3
2.1.1 Objetivo 3
2.1.2 Características 3
2.1.3 TCP proporcionando confiabilidad. 4
2.1.4 La ventana deslizante 5
2.1.5 Aperturas pasivas y activas 6
2.1.6 Segmentos, flujos y números de secuencia 7
2.1.7 Tamaño de la ventana y el control de flujo 8
2.1.8 Algoritmos de transmisión. 8
2.1.9 Conclusiones de los distintos algoritmos 15
2.1.10 Medida RTT (Algoritmo de Karn) 15
2.1.11 Máquina de estados TCP 15
2.2 Java 16
2.2.1 Java 2D 16
2.2.2 Renderizado en Java 2D 17
2.2.3 Contexto de Renderizado de *Graphics2D* 18
2.2.4 Métodos de Renderizado de *Graphics2D* 20
2.2.5 Sistema de Coordenadas 21
2.2.6 Formas 2D 21
2.2.7 Texto en Java 2D 22
2.2.8 Fuentes 22
2.2.9 Distribución de Texto 23
2.2.10 Dibujo y relleno de gráficos primitivos 23
2.2.11 Formas Geométricas 24
2.2.12 Dibujar Curvas 26
2.2.13 Estilos de Línea 26
2.2.14 Transformar Formas, Texto e Imágenes 28
2.2.15 Controlar la Calidad del Dibujo. Renderizado 29
2.2.16 Crear y Derivar Fuentes 29
2.2.17 Imágenes 30
3. Elementos utilizados en la realización del programa 33

3.1 Eclipse.. 33
3.2 CVS, instalación y configuración.. 34
3.3 Configuración y uso de Eclipse con CVS.. 41
4. Desarrollo de la aplicación 45
4.1 Introducción.. 45
4.2 Paquete *GUI* 47
4.2.1 *GUI.java* 47
4.2.2 *JCanvas.java*... 48
4.2.3 *JCanvasEstados.java*... 55
4.2.4 *JCanvasEstadosExtendido.java*... 56
4.2.5 *JCanvasVentana.java* 58
4.2.6 Graficos.java 59
4.2.7 *JPanelBotones.java*... 61
4.2.8 *miJTable.java* 63
4.2.9 *MiTabla.java*... 64
4.2.10 *CustomRenderer.java*.. 65
4.2.11 *listaObjetos.java* 66
4.2.12 *InternalFrameEstados.java*... 66
4.2.13 *InternalFrameLog.java*.. 67
4.2.14 *InternalFrameNuevo.java* 68
4.2.15 *InternalFrameOpciones.java* 69
4.2.16 *InternalFrameAyuda.java* 69
4.2.17 *JScrollPaneListener.java*.. 70
4.2.18 *Lineas.java* 71
4.2.19 *Punto.java*... 71
4.2.20 *Texto.java*... 72
4.2.21 *FiltroFicherosSIM.java*... 72
4.2.22 *FiltroFicherosPNG.java* 73
4.3 Paquete *ModeloTCP*.. 73
4.3.1 *ImplementacionTCP.java* 75
4.3.2 *SimpleTCP.java*.. 76
4.3.3 *Vanilla.java*... 77
4.3.4 *Tahoe.java*.. 78
4.3.5 *Reno.java* 78
4.3.6 *SACK.java*.. 78
4.3.7 *SujetoComunicacionTCP.java*.. 78
4.3.8 ModeloTCP.java... 79

4.3.9 *Estado.java* 80
4.3.10 *Evento.java* 81
4.3.11 *FEL.java* 82
4.3.12 *InformacionEmisor.java* 83
4.3.13 *MensajeAplicacion.java* 83
4.3.14 *SegmentoTCP.java* 84
4.4 Paquete *Observer* 84
4.4.1 *ObserverTCP.java* 84
4.4.2 *SujetoObservable.java* 85
4.5 Paquete *Configuracion* 85
4.5.1 *Configuracion.java* 85
4.6 *Principal.java* 86
5. Uso y configuración del simulador 89
5.1 Introducción 89
5.2 Partes y funciones del simulador 89
5.2.1 Ventana de intercambio de segmentos 90
5.2.2 Botones de acción 91
5.2.3 Log del sistema 92
5.2.4 Ventana deslizante 93
5.2.5 Máquina de estados 94
5.3 Menús de configuración 95
5.3.1 Archivo 95
5.3.2 Opciones 97
5.4 Menú de ayuda 97
5.5 Nota importante para la ejecución del simulador 98
6. Conclusiones 99
6.1 Conclusiones 99
6.2 Líneas de trabajo futuras 99
7. Bibliografía 101

ÍNDICE FIGURAS

Figura 2.1 Comunicación TCP sin pérdidas 4
Figura 2.2 Comunicación con pérdidas 4
Figura 2.3 Ventana de transmisión inicial 5
Figura 2.4 Ventana de transmisión inicial tras un reconocimiento 5
Figura 2.5 Ejemplo de comunicación con ventana deslizante 6
Figura 2.6 Ventana deslizante 6
Figura 2.7 Tipos de aperturas 7
Figura 2.8 Variación del tamaño de la ventana según reconocimientos con *Slow Start* 8
Figura 2.9 Variación del tamaño de la ventana según reconocimientos con *Congestion Avoidance* 10
Figura 2.10 Bloque SACK dentro del segmento TCP 12
Figura 2.11 Diagrama de estados TCP 16
Figura 2.12. Ejemplo de gráficos y charts mediante el uso del API 2D de Java 16
Figura 2.13. Filtrado de imágenes 17
Figura 2.14 Forma para obtener el objeto Java 2D 18
Figura 2.15 Atributos de renderizado de Graphics2D 19
Figura 2.17. Métodos generales de dibujado 20
Figura 2.16. Uso de gradiente mediante GradientPaint() 20
Figura 2.18. Formas geométricas 25
Figura 2.19. Curvas cúbicas y cuadráticas 26
Figura 2.20. Stroke: Estilos de unión 26
Figura 2.21. Stroke: Estilos de finales 27
Figura 2.22. Uso de GradientPaint() 27
Figura 2.23. Uso de TexturePaint() 27
Figura 2.25. Ejemplo. Escalado de imagen 28
Figura 2.26. Ejemplo. Obtener los nombres de las fuentes disponibles en el sistema 29
Figura 2.27. Ejemplo. Creación de un objeto Font 30
Figura 2.28. Ejemplo. Método para cambiar el estilo de fuente 30
Figura 2.29. Ejemplo. Aplicación del atributo Font al contexto Graphics2D 30
Figura 2.30. Carga fichero de imagen y dibujado en un objeto 31
Figura 3.1 Vista general del IDE Eclipse 34
Figura 3.2 Página Web del fabricante 35
Figura 3.3 Habilitar el uso compartido simple de archivos 36
Figura 3.4 Localización del panel de control del servidor de repositorio 36
Figura 3.5 Pestaña *About* del servidor CVSNT 37
Figura 3.6 Pestaña *Repository configuration* del servidor CVSNT 37
Figura 3.7 Ventana de opciones para añadir un repositorio 38
Figura 3.8 Aspecto final de la pestaña *Repository configuration* del servidor CVSNT una vez añadido un repositorio 38
Figura 3.9 Pestaña *Server Settings* del servidor CVSNT 39
Figura 3.10 Pestaña *Plugins* del servidor CVSNT 39
Figura 3.11 Ventana de configuración del protocolo *sserver* 40
Figura 3.12 Pestaña *Advanced* del servidor CVSNT 40
Figura 3.13 Ruta para añadir un proyecto al control de versiones 41
Figura 3.14 Selección de repositorio en *Eclipse* 42
Figura 3.15 Selección del módulo dentro del repositorio en *Eclipse* 42
Figura 3.16 Sincronización inicial de ficheros fuente del proyecto con el CVS 43
Figura 3.17 Visualización de estado de los ficheros en Eclipse 43
Figura 4.1 Diagrama de secuencia del patrón MVC 45
Figura 4.2 Diagrama de secuencia del patrón observador 46
Figura 4.3 Diagrama de dependencias de la clase *GUI.java* 47

Figura 4.4 Diagrama de clase de la clase *GUI.java* 48
Figura 4.5 Diagrama de dependencias de la clase *JCanvas.java*............................... 49
Figura 4.6 Objeto *BufferedPanel* 50
Figura 4.7 Área de dibujo sobre el objeto *JCanvas*.. 51
Figura 4.8 Áreas de *buffer* e impresión 52
Figura 4.9 Código para imprimir cada página.. 53
Figura 4.10 Método que recupera el texto del *tooltip* 53
Figura 4.11 Código para realizar la corrección de la posición del texto 54
Figura 4.12 Diagrama de clase de la clase *JCanvas.java*... 54
Figura 4.13 Visualización del *JCanvasEstados*.. 55
Figura 4.14 Diagrama de dependencias de la clase *JCanvasEstados.java*................ 56
Figura 4.15 Diagrama de clase de la clase *JCanvasEstados.java*.............................. 56
Figura 4.16 Diagrama de dependencias de la clase *JCanvasEstadosExtendido.java*. 57
Figura 4.17 Visualización del *JCanvasEstadosExtendido*... 57
Figura 4.18 Diagrama de clase de la clase *JCanvasEstadosExtendido.java*.............. 58
Figura 4.19 Visualización del *JCanvasVentana* 58
Figura 4.20 Diagrama de dependencias de la clase *JCanvasVentana.java* 59
Figura 4.21 Diagrama de clase de la clase *JCanvasVentana.java* 59
Figura 4.22 Resultado del empleo del método drawFlecha() 60
Figura 4.23 Resultado del empleo del método drawFlechaRellena()........................... 60
Figura 4.24 Resultado del empleo del método drawX() 60
Figura 4.25 Resultado del empleo del método drawInicio() 61
Figura 4.26 Diagrama de clase de la clase *Graficos.java* 61
Figura 4.27 Visualización del *JPanelBotones*.. 61
Figura 4.28 Diagrama de dependencias de la clase *JPanelBotones.java*.................... 62
Figura 4.29 Diagrama de clase de la clase *JPanelBotones.java*................................. 62
Figura 4.30 Visualización de la *miJTable.java*.. 63
Figura 4.31 Diagrama de dependencias de la clase *miJTable.java* 64
Figura 4.32 Diagrama de clase de la clase *miJTable.java* 64
Figura 4.33 Diagrama de dependencias de la clase *MiTabla.java* 65
Figura 4.34 Diagrama de clases de la clase *MiTabla.java* 65
Figura 4.35 Diagrama de clase de la clase *listaObjetos*... 66
Figura 4.36 Visualización del *InternalFrameEstados* 67
Figura 4.37 Diagrama de dependencias de la clase *InternalFrameEstados.java* 67
Figura 4.38 Visualización de *InternalFrameLog* 68
Figura 4.39 Visualización de *InternalFrameNuevo*... 68
Figura 4.40 Diagrama de dependencias de la clase *InternalFrameNuevo.java*........... 69
Figura 4.41 Visualización de *InternalFrameOpciones.java* 69
Figura 4.42 Visualización del *InternalFrameAyuda* 70
Figura 4.43 Diagrama de dependencias de la clase *JScrollPaneListener.java*............ 70
Figura 4.44 Diagrama de clase de la clase *Lineas.java* 71
Figura 4.45 Diagrama de clase de la clase *Punto.java* 71
Figura 4.46 Diagrama de clase de la clase *Texto.java*.. 72
Figura 4.47 Diagrama de dependencias de la clase *FiltroFicherosSIM.java*................ 72
Figura 4.48 Diagrama de dependencias del *ModeloTCP*.. 73
Figura 4.49 Diagrama de dependencias de los algoritmos TCP 74
Figura 4.50 Diagrama de dependencias de la *FEL* 74
Figura 4.51 Diagrama de clase de la interface *ImplementacionTCP.java*.................... 75
Figura 4.52 Diagrama de flujo del método recibirMsgTCP() 77
Figura 4.53 Diagrama de clase de la clase *SujetoComunicacionTCP.java*................. 79
Figura 4.54 Diagrama de clase de la clase *ModeloTCP* 80
Figura 4.55 Diagrama de clase de la clase *Estado.java*.. 81
Figura 4.56 Diagrama de clase de la clase *Evento.java*.. 82
Figura 4.57 Diagrama de clase de la clase *FEL.java* 82
Figura 4.58 Diagrama de clase de la clase *InformacionEmisor.java*........................... 83

Figura 4.59 Diagrama de clase de la clase *MensajeAplicacion.java* 83
Figura 4.60 Diagrama de clase de la clase SegmentoTCP.java 84
Figura 4.61 Diagrama de clase de la interface *ObserverTCP.java* 85
Figura 4.62 Diagrama de clase de la clase *SujetoObservable.java* 85
Figura 4.63 Diagrama de clase de la clase *Configuracion.java* 86
Figura 4.64 Diagrama de dependencias de la clase *Principal.java* 86
Figura 5.1 Ventana principal del sistema 90
Figura 5.2 Ventana de intercambio de segmentos 91
Figura 5.3 Ventana de acción 92
Figura 5.4 Registro del sistema 93
Figura 5.5 Registro del sistema 94
Figura 5.6 Zona de estados 94
Figura 5.7 Diagrama de la maquina de estados completa 95
Figura 5.8 Menú Archivo y Opciones 95
Figura 5.9 Parámetros de la nueva simulación 96
Figura 5.10 Opciones del simulador 97

ÍNDICE ECUACIONES

Ecuación 1 Tamaño de la ventana en *Slow Start* 8
Ecuación 2 Valor del umbral cuando salta *timeout* en *Congestion Avoidance* 9
Ecuación 3 Tamaño de la ventana en *Congestion Avoidance* 9
Ecuación 4 Valor del umbral cuando recibimos tres reconocimientos iguales 10
Ecuación 5 Valor de la ventana tras *Fast Retransmit* 11
Ecuación 6 Valor de la ventana con *Fast Recovery* 11
Ecuación 7 Valor de la ventana tras la recuperación del sistema 11
Ecuación 8 Valor del umbral cuando recibimos tres reconocimientos iguales 14
Ecuación 9 Valor de las variables internas para implementar el algoritmo SACK 14
Ecuación 10 Valor del tamaño de ventana una vez recuperado de las perdidas mediante SACK 14

ÍNDICE TABLAS

Tabla 1 Equivalencias entre algoritmos e implementaciones TCP ... 11
Tabla 2 Diagrama de segmentos del Ejemplo 1 utilizando SACK ... 13
Tabla 3 Diagrama de segmentos del Ejemplo 2 utilizando SACK ... 13
Tabla 4 Diagrama de segmentos del Ejemplo 2 utilizando SACK ... 14
Tabla 5 Clases del paquete java.awt.geom ... 21

1. Introducción

1.1 Introducción

La incorporación de las tecnologías de la información y de la comunicación (TIC) en la educación, y mayoritariamente en la educación universitaria, ha contribuido a crear nuevos marcos o modelos de enseñanza, que ofrecen una capacidad de aprendizaje permanente en tiempo y no presencial.

Entendemos por teledocencia a los marcos, modelos de enseñanza y aprendizaje a distancia, que dado el gran auge del desarrollo tecnológico y la mejora de las comunicaciones, han creado nuevos canales de aprendizaje y docencia consiguiendo una gran autonomía y flexibilidad en las partes.

Sin duda, la mayor ventaja que ofrece este nuevo fenómeno es la superación de los límites espaciales y temporales para realizar cualquier actividad asociada al proceso de aprendizaje, como consecuencia de las comunicaciones asíncronas y de la interconexión a través de las nuevas líneas de comunicación abiertas.

El teleformador ya no debe transmitir directamente los conocimientos, ahora su principal cometido en este nuevo medio es el de crear información y depositarla para que sea accesible al alumno, siendo guía de los mismos para poder generar un conocimiento a través de un proceso de auto-aprendizaje, teniendo además el rol de estimulador del alumno mediante la motivación y retroalimentando los conocimientos adquiridos por él.

La mayor conclusión del párrafo anterior atiende a la madurez del alumno para poder ser capaz de auto-formarse, teniendo simplemente los recursos para ello. Siendo el material adecuado y teniendo el apoyo de personal cualificado, el alumno debe ser capaz de aprender utilizando todo lo que está a su disposición. Es por esto por lo que este tipo de aprendizaje está especialmente orientado a alumnos universitarios y post-universitarios principalmente.

Existen distintos marcos donde se puede entender la teledocencia. En nuestro caso, dicha teledocencia está orientada a un software específico en el que el alumno encontrará un conocimiento muy concreto. Sin embargo, el abanico es realmente amplio y donde realmente se puede comprender todo el marco de actuación de la teledocencia actual es fijándose en portales web, en los que la interacción entre alumnos-teleformador-material-interface es continua. En ese punto podemos observar las siguientes relaciones:

- **Estudiante-teleformador:** se trata de la interacción entre el estudiante y el experto que preparó el material.
- **Estudiante-estudiante:** se trata de la interacción formal o informal entre los estudiantes y que puede ser directa o indirectamente relacionada con el proceso de aprendizaje. Las funciones que cumple esta interacción van desde las sociales, hasta las de comparación de grupo.
- **Estudiante-contenido:** se trata de la interacción entre el que estudia y el contenido o tema de estudio.
- **Estudiante-interface:** Es realmente importante esta parte, ya que el alumno debe fácilmente ser capaz de acceder y entender el material, buscando simplicidad y eficiencia a la hora de usarlo.

1.2 Objetivos

El objetivo fundamental de este proyecto era crear una aplicación docente para su uso en las asignaturas que traten el tema desarrollado en el mismo. Es decir, es un proyecto cuya finalidad es puramente docente, donde el alumno encontrará una herramienta autónoma para adquirir los conocimientos del protocolo TCP (Transmision Control Protocol).

La idea surge ante el requerimiento de los alumnos de ampliar el abanico de ejemplos de funcionamiento de las diferentes implementaciones del protocolo TCP estudiadas en clase.

Por ello, se comienza el desarrollo de una aplicación donde el alumno puede, en una primera fase, adquirir conocimientos de forma sencilla y progresiva, para posteriormente probar los mismos aumentando la dificultad, así como plantearse nuevos casos donde no se tendría claro el funcionamiento del protocolo.

Analizando los conocimientos que se valoraban más interesantes, se definieron una serie de requisitos en ese proceso de aprendizaje, que son:

- Ventana deslizante.
- Diagrama de estados.
- Acciones a ejecutar.
- *Timeouts*.

Éstos son los conceptos y parámetros que el alumno necesitaba adquirir, pero además debía ser necesario aprenderlos en cuatro de los diferentes algoritmos de TCP como son:

- Vanilla.
- Reno.
- Tahoe.
- SACK (*Selective Acknowledgment*).

Por otro lado, el desarrollo de este proyecto final de carrera ha tenido como objetivo en el autor la adquisición de conocimiento en una parte muy específica del lenguaje de programación JAVA, el dibujo 2D y las interfaces gráfica GUI (*Graphical User Interface*).

2. Teoría necesaria

2.1 TCP

2.1.1 Objetivo

Es necesario un protocolo de propósito general que aísle a las aplicaciones de los detalles del trabajo con redes y que permita la definición de una interfaz uniforme para la transferencia de datos(Comer, 1996).

2.1.2 Características

Estamos ante un servicio de entrega confiable cuyas características podemos definir como:

- Orientación de flujo: El servicio de entrega de flujo en la máquina de destino pasa al receptor exactamente en la misma secuencia de octetos que le pasa el transmisor en la máquina origen.
- Necesidad de un circuito virtual: Antes de comenzar la transferencia, las aplicaciones negocian con el S.O. (Sistema Operativo) para evaluar si es factible la misma. Los módulos *software* del protocolo en cuestión se comunican intercambiando mensajes, informando a la aplicación en el caso de que ésta pueda establecerse. Es en este punto donde podemos asociar el término de "circuito virtual" a dicha conexión. Podríamos concebirlo como un canal dedicado, pero sólo visible a nivel de aplicación, entendiendo que esto no es así, de ahí el término "virtual", porque realmente depende únicamente del correcto funcionamiento del servicio de entrega de flujo de datos.
- Transferencia de memoria intermedia: Las aplicaciones envían un flujo de datos al módulo *software* del protocolo. Cada aplicación enviará datos del tamaño adecuado a la misma. Dicho *software* del protocolo puede dividir el flujo en segmentos. Para hacer eficiente la transferencia y minimizar el tráfico de red, las implementaciones deberán recolectar datos para hacer un segmento lo más largo posible. De igual manera, si la aplicación genera bloques de datos muy largos, el protocolo puede dividir en partes más pequeñas para su transmisión.
- Flujo no estructurado: El protocolo no entiende los datos que envía por la red. Debe ser cometido del programa de aplicación el entender el contenido del flujo de datos.
- Conexión *Full-Duplex*: Transferencia concurrente en ambas direcciones. Esto se puede explicar cómo dos flujos independientes que se mueven en direcciones opuestas, pudiendo llevar el control de flujo al origen reduciendo por tanto la carga en el canal.

2.1.3 TCP proporcionando confiabilidad.

Se basa en la técnica fundamental del acuse de recibo positivo con retransmisión. Esta técnica se fundamenta en que el receptor envía un mensaje de acuse de recibo, o lo que es lo mismo ACK (*Acknowledgement*), conforme recibe los datos. El transmisor guarda cada paquete hasta que el ACK confirme su recepción. Dicho ACK no sólo me confirmará los datos enviados, sino que además será necesario para continuar inyectando paquetes a la red. En el caso de que la recepción de dicho reconocimiento no se produjese en un determinado intervalo de tiempo, saltará un temporizador para proceder con el reenvío del paquete, ya que se entiende que se ha perdido el mismo.

Además de las pérdidas tenemos el problema de la duplicidad de paquetes, esto es debido a retransmisiones prematuras o pérdidas de reconocimientos, solucionándose mediante el uso de números de secuencia en los paquetes enviados.

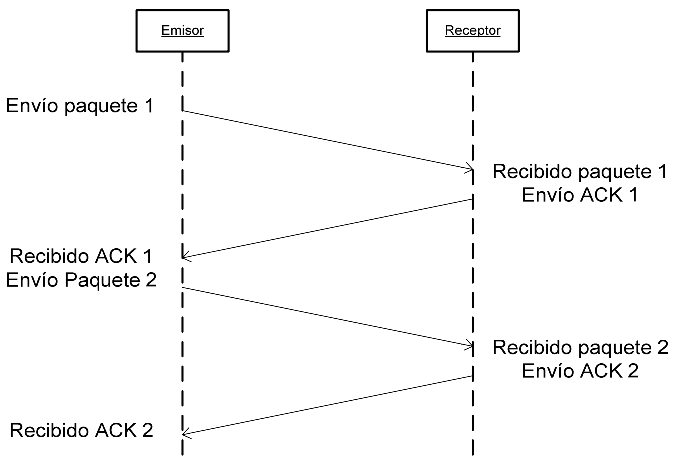


Figura 2.1 Comunicación TCP sin pérdidas

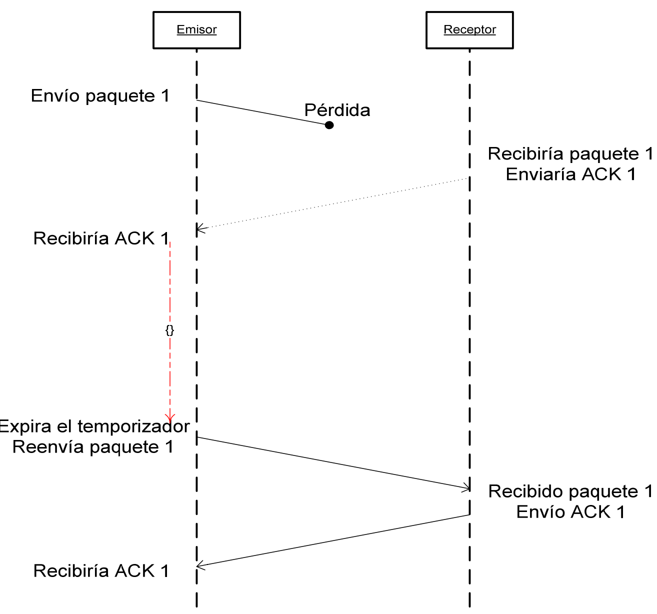


Figura 2.2 Comunicación con pérdidas

En la **Figura 2.1** mostramos un ejemplo de comunicación empleando la técnica de acuse de recibo positivo en donde no se producen problemas, es decir, no tenemos pérdidas de paquetes. Es en la **Figura 2.2** donde podemos observar que es lo que ocurre cuando se pierde un paquete. En este caso se pierde en el canal el paquete enviado dado que no se recibe el reconocimiento y expira el temporizador, procediendo a la retransmisión del mismo que posteriormente llega al receptor produciéndose su reconocimiento.

2.1.4 La ventana deslizante

Si utilizamos un protocolo simple de acuse de recibo positivo, gastamos un gran ancho de banda debido a que debemos retrasar el envío de un nuevo paquete hasta que se reciba el ACK del paquete anterior → Demasiado tiempo ocioso.

La idea fundamental es permitir al transmisor enviar varios paquetes sin esperar un acuse de recibo, punto donde se introduce el concepto de ventana de transmisión.

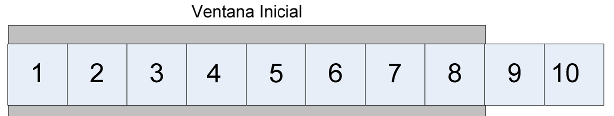


Figura 2.3 Ventana de transmisión inicial

Como podemos ver en la **Figura 2.3** el emisor parte de un tamaño de ventana inicial, que en este caso es de ocho paquetes. Una vez transmitida la ventana completa deberemos esperar el reconocimiento de alguno de ellos para poder enviar el noveno, ya que a la recepción de algún reconocimiento, la ventana se deslizará y nos permitirá enviar el mismo. En la **Figura 2.4** se recibe el reconocimiento del primer segmento, por lo que se permite lo anteriormente comentado.

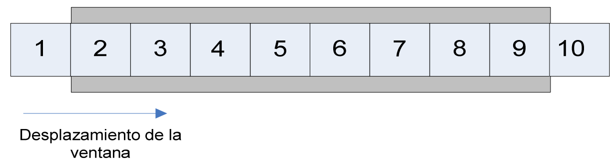


Figura 2.4 Ventana de transmisión inicial tras un reconocimiento

Utilizando los conceptos vistos hasta ahora, podemos hacer una primera aproximación de lo que sería una comunicación entre emisor y receptor utilizando la teoría de reconocimientos positivos y de ventana deslizante. Como nos muestra la **Figura 2.5** el emisor enviaría ocho paquetes, que son los permitidos por la ventana. Tras la recepción de cada uno de los paquetes por parte del receptor éste procede a enviar los reconocimientos; y por cada uno que recibe el transmisor la ventana se desliza, por lo que se permite el envío de un nuevo paquete, siendo en el primer caso el noveno.

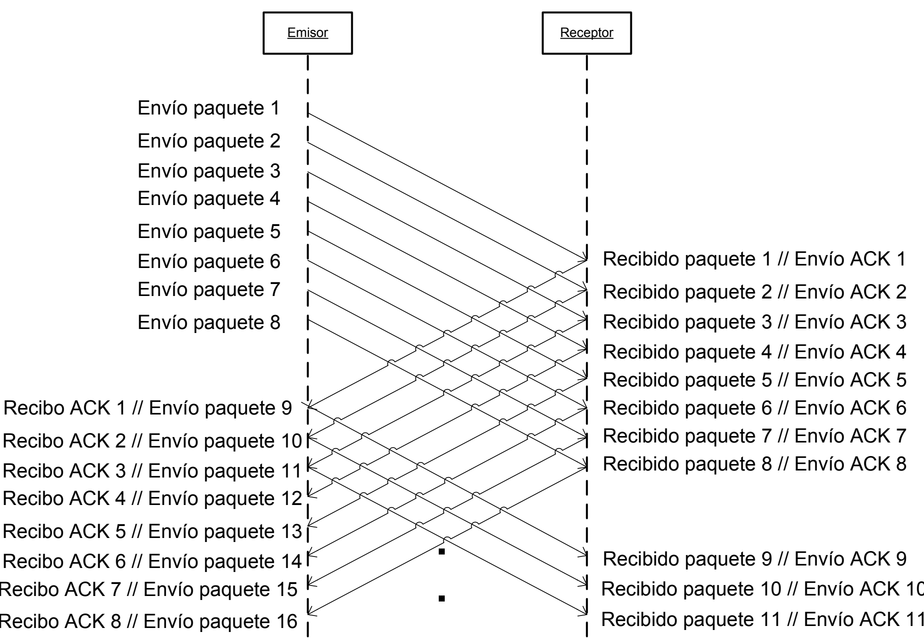


Figura 2.5 Ejemplo de comunicación con ventana deslizante

La **Figura 2.6** define por completo la ventana deslizante en transmisión. Esta ventana puede ser interpretada como dos punteros que delimitan tres zonas. La primera de ellas corresponde a los paquetes que fueron enviados y confirmado, la segunda contiene a los paquetes que se han enviado pero que están a la espera de reconocimiento, y la tercera inicia la zona de paquetes que todavía no pueden ser enviados.

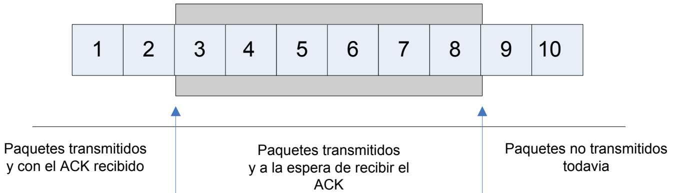


Figura 2.6 Ventana deslizante

2.1.5 Aperturas pasivas y activas

TCP es un protocolo orientado a conexión que requiere que ambos puntos extremos estén de acuerdo en participar. Para poder intercambiar información es necesario crear una conexión, que hemos definido anteriormente como circuito virtual.

Para crear dicha conexión se ha de establecer una negociación entre los extremos que van a participar en la misma. Definiéndose dicha negociación como apertura podemos entender dos tipos distintos, activa o pasiva.

Una apertura activa se produce cuando el programa de aplicación contacta con su S.O. pidiéndole expresamente la solicitud de un circuito para comunicarse con el otro extremo. En cambio, una apertura pasiva se producirá cuando se recibe una conexión entrante que notifica al S.O. la necesidad de un circuito de comunicación.

Como es lógico, lo natural será que un extremo realice una apertura activa mientras el otro sea forzado por este primero a realizar una apertura pasiva a través del intercambio de mensajes, esto es conocido como apertura simple. Pero es posible que ambos extremos decidan unilateralmente crear un circuito de comunicaciones entre ellos en el mismo instante de tiempo, por lo que ambos realizarán una apertura activa, estaríamos ante un caso de apertura simultánea. El intercambio de mensajes de lo anterior aparece reflejado en la **Figura 2.7**.

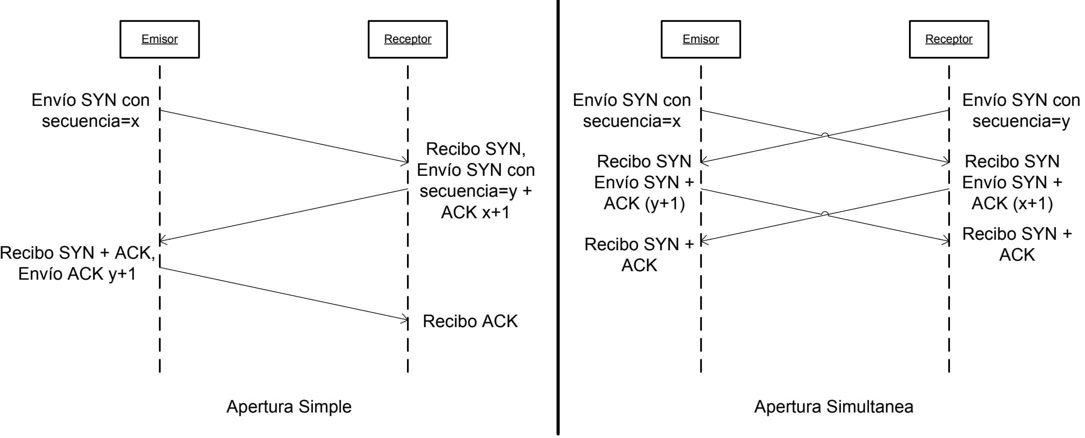


Figura 2.7 Tipos de aperturas

2.1.6 Segmentos, flujos y números de secuencia

Hasta ahora habíamos denominado a la unidad de transmisión entre emisor y receptor paquete, pero esta unidad debe ser correctamente denominada segmento. Es interesante explicar cómo el protocolo TCP genera dicho segmento, recogiendo el flujo de datos de la capa superior visualizados como una secuencia de octetos (bytes) y dividiendo el mismo en estas unidades para su transmisión.

En el caso que este flujo de bytes no tenga la cadencia necesaria para rellenar los segmentos, el protocolo puede esperar a que el nivel superior le mande suficientes datos o incluso instar a la aplicación que le envíe todo lo que tenga.

Como hemos comentado anteriormente una de la ventajas del uso de la ventana deslizante es el aumento de la eficiencia, pero además soluciona otro inconveniente, el control de flujo de extremo a extremo, ya que restringe la transmisión de mensajes hasta que la memoria intermedia del buffer permita la incorporación de más datos.

El mecanismo que gobierna dicha ventana deslizante opera a nivel de octeto (byte), no a nivel de segmento. Dichos octetos se numeran de manera secuencial, siendo conocidos como números de secuencia. Dentro de la ventana activa disponemos de tres punteros que apuntarán a distintos números de secuencia y que tienen el siguiente significado:

- La zona existente entre el primero de ellos y el segundo marca qué octetos se han enviado pero todavía no han sido reconocidos.
- El intervalo entre el segundo y el tercero muestra qué octetos dentro de la ventana estamos permitidos a enviar.
- Todo lo que quedaría antes del primer puntero son secuencias enviadas y reconocidas, mientras que las secuencias posteriores al tercero son aquellas secuencias que todavía no estamos autorizados a enviar.

2.1.7 Tamaño de la ventana y el control de flujo

Además del deslizamiento comentado anteriormente de la ventana de transmisión, el tamaño de la misma también varía en tamaño, creciendo o decreciendo dependiendo de la situación.

En cada acuse de recibo, que nos informa cuantos octetos se recibieron, lleva además un campo que advierte al emisor cuantos octetos adicionales está preparado para aceptar el receptor dado el tamaño de su buffer libre. Según este campo y mi tamaño actual de ventana decidiré cuantos octetos puedo inyectar todavía en la red para no saturar al receptor.

2.1.8 Algoritmos de transmisión.

En este punto comentaremos los cuatro algoritmos TCP implementados en el simulador desarrollado en este PFC. Comenzaremos con su versión más básica, conocida como *Slow Start*, desde la que iremos añadiendo mejoras que tendrán como resultado cada una de sus otras versiones.

2.1.8.1 *Slow Start*

El objetivo de este algoritmo es el de arrancar la conexión para averiguar a que tasa podemos transmitir. Se rige por la siguiente ecuación:

$$Ventana\ permitida = MIN\ (CNWD, ANWD)$$

Ecuación 1 Tamaño de la ventana en *Slow Start*

Siendo CNWD (*Congestion Window*) la ventana de congestión y ANWD (*Announced Window*) la ventana advertida por el receptor. En situaciones normales la ventana de transmisión es igual a CNWD.

Al iniciar la transmisión se fija el parámetro CNWD a uno, incrementándose en una unidad cada vez que se recibe un reconocimiento. Este sistema da un incremento exponencial como podemos observar en la **Figura 2.8**.

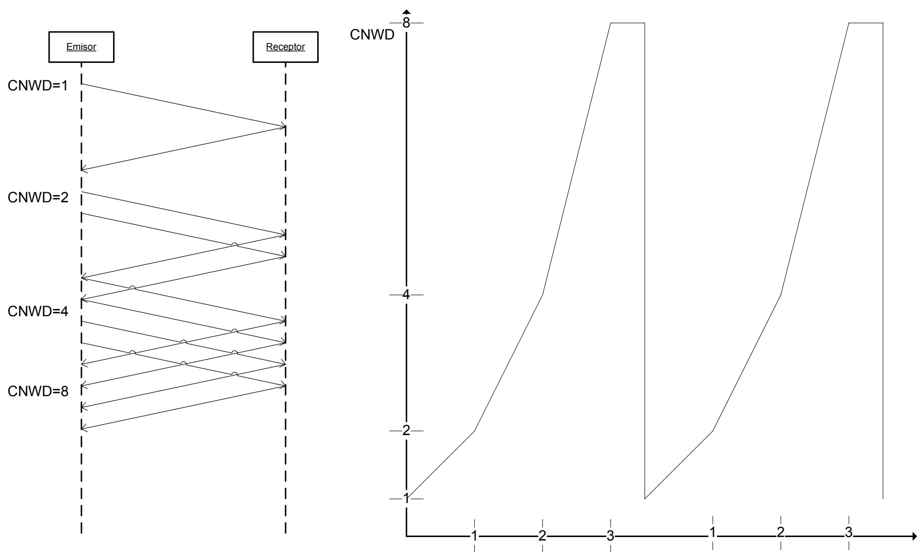


Figura 2.8 Variación del tamaño de la ventana según reconocimientos con *Slow Start*

Cuando se produce una pérdida se produce la expiración del *timeout*. Esto acarrea que el tamaño de la ventana se fije a uno y se proceda a continuar con el algoritmo, volviéndose a incrementar la ventana de manera exponencial hasta que se produzca una pérdida y rebajemos de nuevo el tamaño de la ventana al inicial. La forma producida por el tamaño de la ventana es conocida comúnmente como “diente de sierra”, ya que asemeja a la misma.

2.1.8.2 *Congestion Avoidance*

Este algoritmo introduce una variable adicional llamada *ssthresh*, que sirve para fijar el valor de CNWD cuando se pierden segmentos TCP.

Slow Start incrementa el valor de CNWD hasta que se alcance ANWD o se produzca una pérdida que derive en un *timeout*. Cuando esto se produce, el valor de *ssthresh* se fija:

$$ssthresh = MAX\left(2, \frac{MIN(CNWD, ANWD)}{2}\right)$$

Ecuación 2 Valor del umbral cuando salta *timeout* en *Congestion Avoidance*

Después de dicho *timeout* procedemos a iniciar de nuevo *Slow Start. Cuando el valor de* CWND supere *ssthresh* entraremos en la fase de *Congestion Avoidance*, donde por cada reconocimiento recibido se modificará el valor de la ventana de transmisión según la siguiente ecuación:

$$CNWD = CNWD + \frac{1}{CNWD}$$

Ecuación 3 Tamaño de la ventana en *Congestion Avoidance*

Tal y como vemos en la **Figura 2.9** el protocolo comienza en un estado de *Slow Start* y con un tamaño de ventana uno. Vamos enviando mensajes siguiendo las restricciones que nos marca la ventana y aumentando en uno el valor de la misma por cada reconocimiento positivo recibido. Cuando nuestra ventana tiene un tamaño de cuatro segmentos perdemos el primero, lo que producirá la expiración del *timer*. Es en ese momento cuando fijamos el valor de *ssthresh* a la mitad del tamaño de nuestra ventana, en este caso dos.

Continuando en *Slow Start* procedemos a la retransmisión del segmento perdido con un tamaño de ventana uno. Cuando recibimos aquel reconocimiento que hace que nuestra ventana tenga un tamaño de tres segmentos pasamos a *Congestion Avoidance*, donde procederemos a incrementar, como ya hemos comentado, el tamaño de la ventana según la **Ecuación 3**.

La diferencia de *Congestion Avoidance* con respecto a *Slow Start* es el incremento lineal de la ventana una vez que se pasa del valor especificado por *ssthresh* después de tener una expiración del *timer*. Con esto se tendrá un menor incremento en la zona de riesgo de pérdida de paquetes, incrementando así el tiempo con un tamaño de ventana mayor y por tanto el rendimiento del sistema.

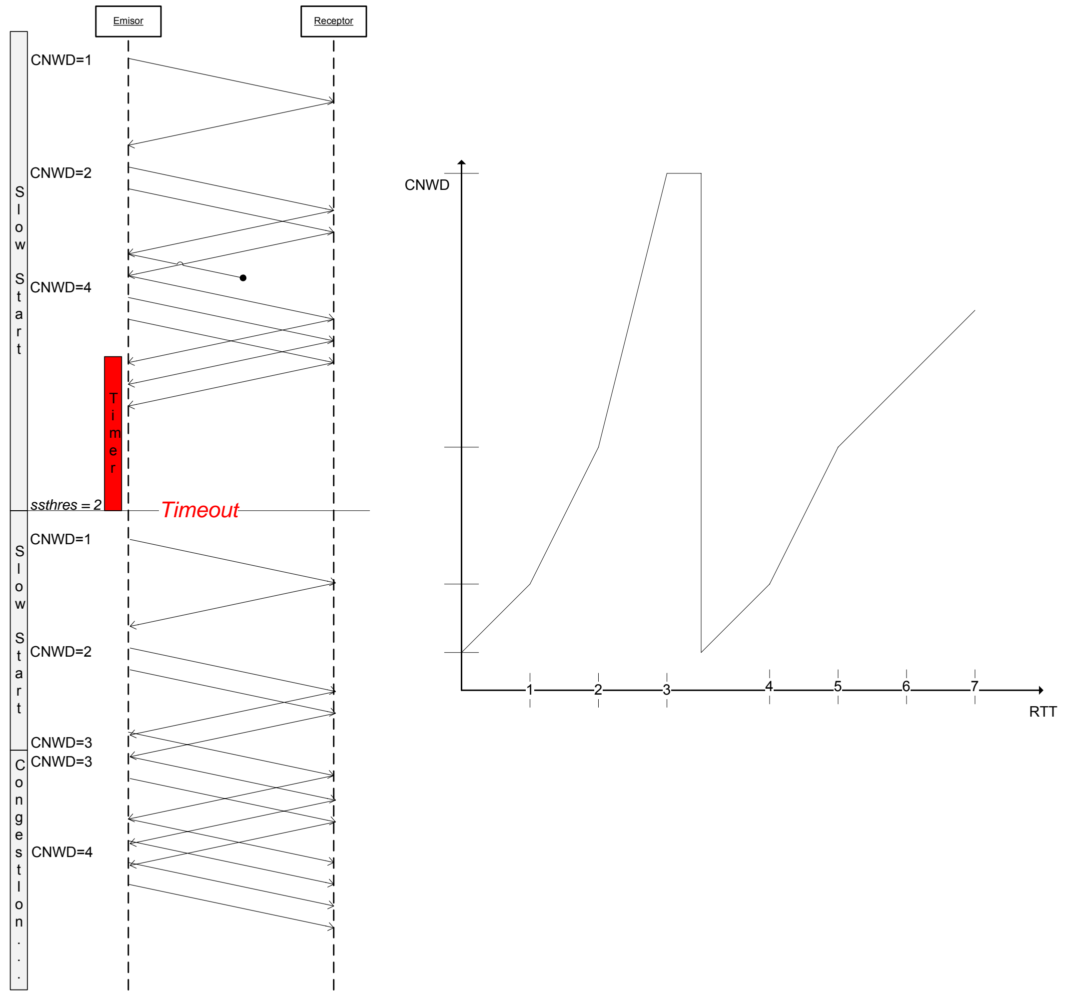


Figura 2.9 Variación del tamaño de la ventana según reconocimientos con *Congestion Avoidance*

2.1.8.3 *Fast Retransmit* y *Fast Recovery*

Dado lo visto hasta ahora es necesario implantar algoritmos que permitan hacer una rápida detección de errores y una rápida recuperación de los mismos. Para ello utilizamos estos dos algoritmos que pasamos a explicar.

El algoritmo *Fast Retransmit* se basa en que el sistema no esperará un *timeout* para decidir que se ha perdido un paquete, sino que al tercer reconocimiento repetido ya entiende que ese paquete se ha perdido y se procede a su retransmisión, ya que nos indica cuál es el siguiente que el receptor espera recibir. El funcionamiento es el siguiente:

1. A la recepción del tercer ACK igual fijamos:

$$ssthresh = MAX\left(2, \frac{MIN(CNWD, ANWD)}{2}\right)$$

Ecuación 4 Valor del umbral cuando recibimos tres reconocimientos iguales

2. Retransmisión del segmento perdido.

3. Fijamos:

$$CNWD = ssthresh\ + 3\ segmentos$$

Ecuación 5 Valor de la ventana tras *Fast Retransmit*

4. Por cada ACK repetido recibido:

$$CNWD = ssthresh\ + 1\ segmento$$

Ecuación 6 Valor de la ventana con *Fast Recovery*

5. Transmitimos nuevos paquetes si la ventana lo permite.

6. A la llegada del ACK del segmento perdido:

$$CNWD = ssthresh$$

Ecuación 7 Valor de la ventana tras la recuperación del sistema

7. Este ACK reconoce todos los segmentos intermedios desde el segmento perdido hasta la recepción del tercer ACK igual.

8. La transmisión sigue en *Congestión Avoidance*.

Visto el funcionamiento podemos resumir que el mecanismo de *Fast Retransmit* permite al protocolo adelantarse a la expiración del *timer*, entendiendo que al tercer reconocimiento repetido ese segmento se ha perdido, por lo que realiza su retransmisión. Mientras que *Fast Recovery* es una manera de seguir inyectando segmentos a la red mediante una aproximación del número de los mismos que se encuentran en el canal, lo que nos permite enviar nuevos segmentos cuando todavía no se ha recuperado del error.

2.1.8.4 Implementaciones

Hasta ahora hemos visto diferentes mecanismos de funcionamiento del protocolo estudiado TCP. Estos mecanismos se agrupan dando lugar a implementaciones del mismo, encontrándonos una gran cantidad de implementaciones de TCP.

En nuestro caso vamos a estudiar cuatro de ellas, que se muestran resumidas y con su equivalencia en cuanto a mecanismos implementados en la siguiente tabla.

Vanilla	*Slow-Start y Congestion Avoidance*
Tahoe	Añade *Fast Retransmit*
Reno	Añade *Fast Retransmit* y *Fast Recovery*
Sack	Añade *SACK option*

Tabla 1 Equivalencias entre algoritmos e implementaciones TCP

2.1.8.5 SACK

Estamos ante un tipo muy especial de implementación, donde se realizarán reconocimientos selectivos para informar al receptor de los paquetes que le han llegado correctamente, sólo retransmitiéndose así los paquetes perdidos.

Con esto evitamos realizar transmisiones de paquetes ya recibidos, por lo que ganamos en cuanto a rendimiento del protocolo, convirtiéndose además en un protocolo más robusto ante paquetes perdidos.

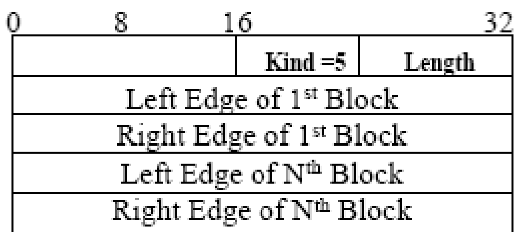


Figura 2.10 Bloque SACK dentro del segmento TCP

Cuando el receptor recibe algún paquete notando que le faltan secuencias procede de la siguiente manera:

1. El receptor informa de bloques de datos no contiguos recibidos, que en este caso denominaremos islas.

2. El receptor espera los datos que faltan para completar los huecos.

3. Entonces reconoce todos los datos recibidos, haciendo avanzar la ventana del emisor.

El receptor informa de unos cuantos bloques y cada bloque se define por dos campos de 32 bits:

- *Left* = primer *sequence number* del bloque.
- *Right* = siguiente *sequence number* al último del bloque.

Donde el significado de dichos bytes nos indica que *left*-1 y *right* no han sido recibidos.

Dada la especificación de TCP se define que en el campo *options* tiene cuarenta bytes, lo que nos arroja el siguiente cálculo. Si tenemos libres cuarenta bytes y cada bloque son ocho bytes, y además necesitamos dos bytes para notificar la utilización de SACK según el campo SACK-*permitted*, podremos tener como mucho cuatro bloques SACK.

Para comprender lo anteriormente descrito procederemos a realizar simples ejemplos ilustrativos. Para ello asumiremos inicialmente que el receptor de datos generará las opciones SACK. Que el borde izquierdo de la ventana es 5000 y que el transmisor de datos tiene una ventana de ocho segmentos, cada uno conteniendo 500 bytes de datos.

Ejemplo 1:

En este ejemplo los primeros cuatro segmentos se reciben y los siguientes cuatro se pierden, por lo que el receptor de datos enviará un segmento ACK normal notificando la recepción con el número de secuencia 7000. Aquí no se hace uso del SACK.

5000
5500
6000
6500
7000 (Perdido)
7500 (Perdido)
8000 (Perdido)
8500 (Perdido)

Tabla 2 Diagrama de segmentos del Ejemplo 1 utilizando SACK

Ejemplo 2:

En este caso el primer segmento se pierde pero los siete restantes se reciben. Después de recibir cada uno de ellos, el receptor de datos enviará un segmento ACK que notifica el número de secuencia 5000 y contiene la opción SACK especificando un bloque de datos que falta. La información aportada es redundante entre la que nos otorga SACK y el reconocimiento.

Segmento	ACK	*Left*	*Right*
5000 (Perdido)			
5500	5000	5500	6000
6000	5000	5500	6500
6500	5000	5500	7000
7000	5000	5500	7500
7500	5000	5500	8000
8000	5000	5500	8500
8500	5000	5500	9000

Tabla 3 Diagrama de segmentos del Ejemplo 2 utilizando SACK

Ejemplo 3:

Los segmentos segundo, cuarto, sexto y octavo, que en este caso es el último, se pierden. El receptor de datos envía un ACK normal por el primer paquete. El tercero, quinto y séptimo paquete lanzan opciones SACK de la forma indicada en la **Tabla 4** creando como podemos ver tres islas, lo que le permitirá al emisor reconocer los segmentos que le faltan al receptor para proceder a su reenvío.

		1er bloque		2do bloque		3er bloque	
Segmento	ACK	Left	Right	Left	Right	Left	Right
5000	5500						
5500 (Perdido)							
6000	5500	6000	6500				
6500 (Perdido)							
7000	5500	7000	7500	6000	6500		
7500 (Perdido)							
8000	5500	8000	8500	7000	7500	6000	6500
8500 (Perdido)							

Tabla 4 Diagrama de segmentos del Ejemplo 2 utilizando SACK

Un ejemplo de implementación podría ser como la que sigue:

1. A la recepción del tercer ACK igual fijamos:

$$ssthresh = MAX\left(2, \frac{MIN(CNWD, ANWD)}{2}\right)$$

Ecuación 8 Valor del umbral cuando recibimos tres reconocimientos iguales

2. Retransmisión del segmento perdido

3. Fijamos:

$$pipe = CNWD - ndup$$
$$CNWD = ssthresh$$

Ecuación 9 Valor de las variables internas para implementar el algoritmo SACK

Siendo ndup la estimación del número de paquetes en la red.

4. Si pipe < CNWD enviamos nuevo segmento o retransmitimos los indicados por SACK.

5. Por cada paquete enviado: pipe = pipe + 1.

6. Por cada nuevo ACK duplicado: pipe = pipe – 1.

7. A la llegada del ACK del segmento retransmitido: pipe = pipe - 2.

8. Este ACK reconoce todos los segmentos intermedios desde el segmento perdido hasta la recepción del tercer ACK igual.

9. La transmisión sigue en *Congestión Avoidance* con:

$$CNWD = ssthresh$$

Ecuación 10 Valor del tamaño de ventana una vez recuperado de las perdidas mediante SACK

2.1.9 Conclusiones de los distintos algoritmos

Después de ver los cuatro algoritmos que vamos a implementar en nuestro programa, podemos concluir que:

- Tahoe: Comportamiento igual con una, dos, tres y cuatro perdidas.
- Reno: Con más de dos pérdidas por ventana entra en *Slow Start* debido a un *timeout*.
- SACK: Solo entra en *timeout* si se pierden más de la mitad de segmentos de una ventana o bien se pierde un segmento retransmitido.

2.1.10 Medida RTT (Algoritmo de Karn)

El protocolo TCP ajusta las temporizaciones o *timeouts* de acuerdo al tiempo estimado de ida y vuelta de los datos o RTT (*Round Trip Time*) entre los puntos origen y destino, y según el camino actual que haya que recorrer para enlazarlos. La estimación del RTT es a menudo también llamada SRTT (*Smoothed Round Trip Time, Tiempo de Ida y Vuelta Promediado*) y se obtiene promediando el RTT medido en paquetes anteriores.

En la recomendación original RFC (Request For Comments) 793(IETF) del TCP, la temporización para la retransmisión o RTO (*Retransmit TimeOut*) se ajustaba a un valor el doble del SRTT. Pero aparece un problema por la forma en que se toman las muestras para el cálculo del promedio anterior: el TCP es incapaz de distinguir dos reconocimientos para el mismo número de secuencia de paquete, y que llegan separados. Si se produce un *timeout* antes de la recepción de un determinado ACK, el paquete correspondiente será reenviado.

En el caso de que en un breve instante después llegue el ACK para el primer paquete, el transmisor medirá un valor erróneo de RTT. Como consecuencia, el TCP del nodo transmisor tiene el riesgo de cometer el error de pensar que el ACK que acaba de llegar corresponde al paquete que acaba de reenviar, cuando en realidad pertenece al paquete que envió al principio.

Un entusiasta propuso la siguiente modificación al TCP:

- No tener en cuenta el RTT medido en los paquetes que hayan tenido que ser retransmitidos.
- En las sucesivas retransmisiones, establecer un *timeout* igual al doble del anterior.

La razón del primer punto es hacer frente al problema expuesto anteriormente, acerca de la incertidumbre que se tiene en asociar los paquetes retransmitidos con sus reconocimientos. Pero al desestimar las medidas de RTT de aquellos paquetes que fueron retransmitidos, se necesita un mecanismo adicional que siga manteniendo una estimación de *timeout* cuando esto ocurra. Este problema se resuelve usando una función binaria exponencial para calcular los sucesivos *timeouts*: si, por ejemplo, el valor actual de *timeout* es uno, el próximo será dos, el siguiente cuatro, le seguirá ocho, etc.

2.1.11 Máquina de estados TCP

TCP se comporta siguiendo el diagrama de estados mostrado en la **Figura 2.11**. Este diagrama es básico para comprender el intercambio de información así como la

apertura y cierre de conexiones, diferenciadas si te comportas como servidor o como cliente (activas o pasivas).

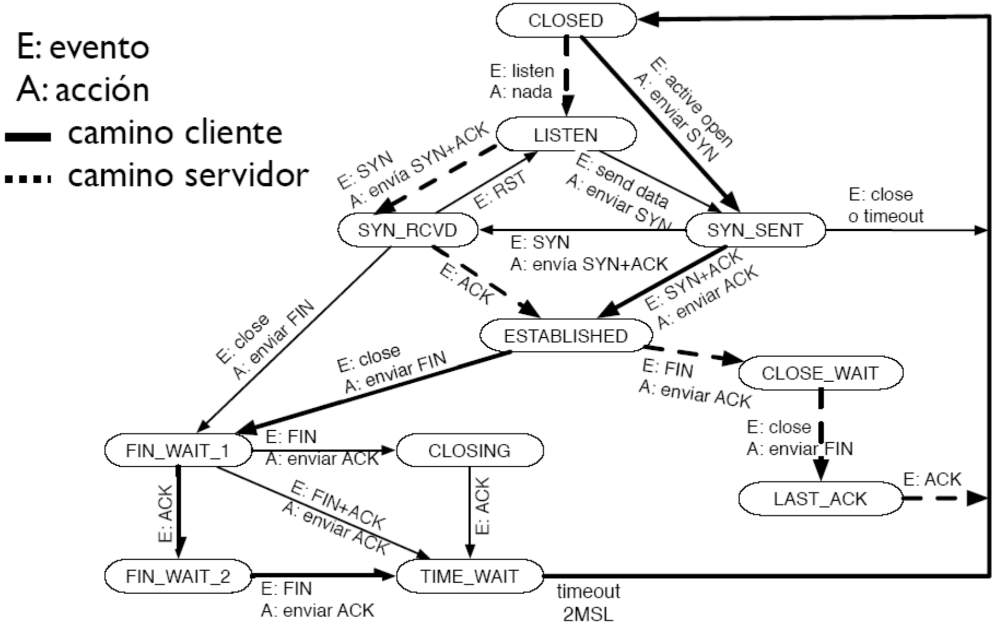


Figura 2.11 Diagrama de estados TCP

2.2 Java

No es objetivo de este apartado explicar que es Java o cómo funciona, sino buscamos hacer una aproximación a la metodología que sigue el lenguaje para dibujar y tratar los elementos gráficos que utilizaremos en la realización del programa.

2.2.1 Java 2D

El API 2D (Sun Microsystems, Inc, 2003-2007) de Java y nos enseña cómo mostrar e imprimir gráficos en dos dimensiones en nuestros programas Java. El API 2D de Java nos permite fácilmente:

- Dibujar líneas de cualquier anchura
- Rellenar formas con gradientes y texturas
- Mover, rotar, escalar y recortar texto y gráficos.
- Componer texto y gráficos solapados.

Por ejemplo, podríamos usar el API 2D de Java para mostrar gráficos y *charts* complejos que usan varios estilos de línea y de relleno para distinguir conjuntos de datos, como se muestra en la siguiente figura:

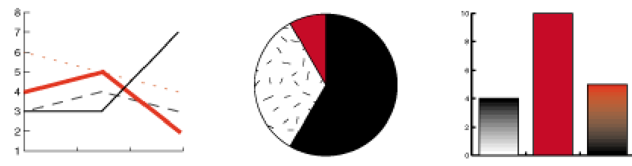


Figura 2.12. Ejemplo de gráficos y charts mediante el uso del API 2D de Java

El API 2D de Java también nos permite almacenar datos de imágenes (por ejemplo, podemos realizar fácilmente filtros de imágenes, como *blur* o recortado), como se muestra en la siguiente figura:

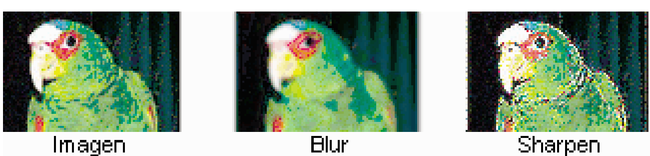


Figura 2.13. Filtrado de imágenes

Se describirán los usos más frecuentes del API 2D de Java y también algunas de las características más avanzadas.

El API 2D de Java que se introdujo en el JDK 1.2 proporciona gráficos avanzados en dos dimensiones, texto, y capacidades de manejo de imágenes para los programas Java a través de la extensión del AWT. Este paquete de *rendering, que no es más que un* proceso de cálculo complejo desarrollado por un ordenador destinado a generar una imagen 2D a partir de una escena 3D, soporta líneas artísticas, texto e imágenes en un marco de trabajo flexible y lleno de potencia para desarrollar interfaces de usuario, programas de dibujo sofisticado y editores de imágenes.

El API 2D de Java proporciona:

- Un amplio conjunto de gráficos primitivos geométricos, como curvas, rectángulos, y elipses y un mecanismo para renderizar virtualmente cualquier forma geométrica.
- Mecanismos para detectar esquinas de formas, texto e imágenes.
- Un modelo de composición que proporciona control sobre cómo se renderizan los objetos solapados.
- Soporte de color mejorado que facilita su manejo.
- Soporte para imprimir documentos complejos.

2.2.2 Renderizado en Java 2D

La filosofía principal para pintar en Swing es la llamada al método heredado de *JComponent* **`paint()`**, este método delega su trabajo en otros tres métodos:

- **`paintComponent()`**: encargado de redibujar el componente.
- **`paintBorder()`**: para redibujar el borde.
- **`paintChildren()`**: encargada de repintar los componentes contenidos en el actual.

Pero se sigue utilizando el método **`paint()`** para indicar como se debe dibujar el área de trabajo. Nunca se ha de llamar a **`paint()`** o **`paintComponent()`** explícitamente, sino que se llama automáticamente por parte del panel siempre que haga falta o bien se utiliza la función **`repaint()`** en el caso de que se necesite su llamada explícita (no se debe usar **`update()`** como ocurría en AWT).

Normas para dibujar en Swing son:

1. Para los componentes Swing, **`paint()`** se invoca siempre como resultado tanto de las órdenes del sistema como de la aplicación; el método **`update()`** no se invoca jamás sobre componentes Swing.

2. Jamás se debe utilizar una llamada a **`paint()`**, este método es llamado automáticamente. Los programas pueden provocar una llamada futura a **`paint()`** invocando a **`repaint()`**.

3. En componentes con salida gráfica compleja, **`repaint()`** debería ser invocado con argumentos para definir el rectángulo que necesita actualización, en lugar de invocar la versión sin argumentos, que hace que sea repintado el componente completo.

4. La implementación que hace Swing de **`paint()`**, reparte esa llamada en tres llamadas separadas: **`paintComponent()`**, **`paintBorder()`** y **`paintChildren()`**. Los componentes internos Swing que deseen implementar su propio código de repintado, deberían colocar ese código dentro del ámbito del método **`paintComponent()`**; no dentro de **`paint()`**. **`paint()`** se utiliza sólo en componentes grandes (como *JFrame*).

5. Es aconsejable que los componentes (botones, etiquetas, etc.) estén en contenedores distintos de las áreas de pintado de gráficos para evitar problemas. Es decir se suelen fabricar paneles distintos para los gráficos y para los componentes.

Para usar las características del API 2D de Java tenemos que forzar el objeto *Graphics,* pasado al método de dibujo de un componente, a un objeto *Graphics2D*.

```
public void paint (Graphics g) {
 Graphics2D g2 = (Graphics2D) g;
              ...
               }
```

Figura 2.14 Forma para obtener el objeto Java 2D

2.2.3 Contexto de Renderizado de *Graphics2D*

Al conjunto de atributos de estado asociados con un objeto *Graphics2D* se le conoce como “Contexto de Renderizado de *Graphics2D*”. Para mostrar texto, formas o imágenes, podemos configurar este contexto y luego llamar a uno de los métodos de renderizado de la clase *Graphics2D*, como **`draw()`** o **`fill()`**. El contexto de renderizado de *Graphics2D* contiene varios atributos:

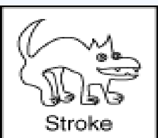
	El estilo de lápiz que se aplica al exterior de una forma. El atributo ***stroke*** nos permite dibujar líneas con cualquier tamaño de punto y patrón de sombreado y aplicar finalizadores y decoraciones a la línea.
	El estilo de relleno que se aplica al interior de la forma. Este atributo ***paint*** nos permite rellenar formas con colores sólidos, gradientes o patrones.
	El estilo de ***composición*** se utiliza cuando los objetos dibujados se solapan con objetos existentes.
	La ***transformación*** que se aplica durante el dibujado para convertir el objeto dibujado desde el espacio de usuario a las coordenadas de espacio del dispositivo. También se pueden aplicar otras transformaciones opcionales como la traducción, rotación escalado, recortado, a través de este atributo.
	El Clip que restringe el dibujado al área dentro de los bordes de la ***Shape*** se utiliza para definir el área de recorte. Se puede usar cualquier ***Shape*** para definir un *clip*.
	La ***fuente*** se usa para convertir cadenas de texto.
	Punto de renderizado que especifican las preferencias en cuanto a velocidad y calidad. Por ejemplo, podemos especificar si se debería usar ***antialiasing***, si está disponible.

Figura 2.15 Atributos de renderizado de Graphics2D

Para configurar un atributo en el contexto de renderizado de *Graphics2D*, se usan los métodos **`setAttribute()`**:

- **`setStroke()`**
- **`setPaint()`**
- **`setComposite()`**
- **`setTransform()`**
- **`setFont()`**
- **`setRenderingHints()`**

Cuando configuramos un atributo, se le pasa al objeto el atributo apropiado. Por ejemplo, para cambiar el atributo *paint* a un relleno de gradiente azul-gris, deberíamos construir el objeto *GradientPaint* y luego llamar a **`setPaint()`**:

```
gp = new GradientPaint(0f,0f,blue,0f,30f,green);
g2.setPaint(gp);
```

Figura 2.16. Uso de gradiente mediante GradientPaint()

Graphics2D contiene referencias a sus objetos atributos. Si modificamos un objeto atributo que forma parte del contexto *Graphics2D*, necesitamos llamar al método **`set()`** para notificarlo al contexto. La modificación de un atributo de un objeto durante el renderizado puede causar comportamientos impredecibles.

2.2.4 Métodos de Renderizado de *Graphics2D*

Graphics2D proporciona los siguientes métodos generales de dibujado que pueden usarse para dibujar cualquier primitivo geométrico, texto o imagen.

`draw()`	Dibuja el exterior de una forma geométrica primitiva usando los atributos *stroke* y *paint*.
`fill()`	Dibuja cualquier forma geométrica primitiva rellenado su interior con el color o patrón especificado por el atributo *paint*.
`drawString()`	Dibuja cualquier cadena de texto. El atributo *font* se usa para convertir la fuente a *glyphs* que luego se rellenan con el color o patrón especificados por el atributo *paint*.
`drawImage()`	Dibuja la imagen especificada.

Figura 2.17. Métodos generales de dibujado

Además, *Graphics2D* soporta los métodos de renderizado de *Graphics* para formas particulares, como **`drawOval()`** y **`fillRect()`**.

2.2.5 Sistema de Coordenadas

El sistema 2D de Java mantiene dos espacios de coordenadas:

- El **espacio de usuario** es el espacio en que se especifican los gráficos primitivos.
- El **espacio de** dispositivo es el sistema de coordenadas para un dispositivo de salida, como una pantalla, una ventana o una impresora.

El **espacio de usuario** es un sistema de coordenadas lógicas independiente del dispositivo: el espacio de coordenadas que usan nuestros programas. Todos los geométricos pasados a las rutinas Java 2D de renderizado se especifican en coordenadas de espacio de usuario.

Cuando se utiliza la transformación por defecto desde el espacio de usuario al espacio de dispositivo, el origen del espacio de usuario es la esquina superior izquierda del área de dibujo del componente. La coordenada ***x*** se incrementa hacia la derecha, y la coordenada ***y*** hacia abajo.

El **espacio de dispositivo** es un sistema de coordenadas dependiente del dispositivo que varía de acuerdo a la fuente del dispositivo. Aunque el sistema de coordenadas para una ventana o una pantalla podrían ser muy distintos al de una impresora, estas diferencias son invisibles para los programas Java. Las conversiones necesarias entre el espacio de usuario y el espacio de dispositivo se realizan automáticamente durante el dibujado.

2.2.6 Formas 2D

Las clases del paquete *java.awt.geom* definen gráficos primitivos comunes, como puntos, líneas, curvas, arcos, rectángulos y elipses:

Arc2D	Ellipse2D	QuadCurve2D
Area	*GeneralPath*	*Rectangle2D*
CubicCurve2D	*Line2D*	*RectangularShape*
Dimension2D	*Point2D*	*RoundRectangle2D*

Tabla 5 Clases del paquete java.awt.geom

Excepto para *Point2D* y *Dimension2D*, cada una de las otras clases geométricas implementa la *interface Shape*, que proporciona un conjunto de métodos comunes para describir e inspeccionar objetos geométricos bidimensionales.

Con estas clases podemos crear de forma virtual cualquier forma geométrica y dibujarla a través de *Graphics2D* llamando al método `draw()` o al método `fill()`.

- **Formas Rectangulares**

Los primitivos *Rectangle2D*, *RoundRectangle2D*, *Arc2D*, y *Ellipse2D* descienden del *RectangularShape*, que define métodos para objetos *Shape* que pueden ser descritos por una caja rectángular. La geometría de un *RectangularShape* puede ser extrapolada desde un rectángulo que encierra completamente el exterior de *Shape*.

- ***QuadCurve2D* y *CubicCurve2D***

La clase *QuadCurve2D* nos permite crear segmentos de curvas cuadráticos. Una curva cuadrática está definida por dos puntos finales y un punto de control.

La clase *CubicCurve2D* nos permite crear segmentos de curvas cúbicos. Una curva cúbica está definida por dos puntos finales y dos puntos de control. Las siguientes figuras muestran ejemplos de curvas cuadráticas y cúbicas.

- ***GeneralPath***

La clase *GeneralPath* nos permite crear una curva arbitraria especificando una serie de posiciones a lo largo de los límites de la forma. Estas posiciones pueden ser conectadas por segmentos de línea, curvas cuadráticas o curvas cúbicas. La siguiente figura puede ser creada con 3 segmentos de línea y una curva cúbica.

- **Áreas**

Con la clase *Area* podemos realizar operaciones booleanas, como uniones, intersecciones y substracciones, sobre dos objetos *Shape* cualesquiera. Esta técnica, nos permite crear rápidamente objetos *Shape* complejos sin tener que describir cada línea de segmento o cada curva.

2.2.7 Texto en Java 2D

Cuando necesitemos mostrar texto, podemos usar uno de los componentes orientados a texto, como los componentes *JLabels* o *JTextComponent* de *Swing*. Cuando se utiliza un componente de texto, mucho del trabajo lo hacen por nosotros--por ejemplo, los objetos *JTextComponent* proporcionan soporte interno para chequeo de pulsaciones y para mostrar texto internacional.

Si queremos mostrar una cadena de texto estática, podemos dibujarla directamente a través de *Graphics2D* usando el método `drawString()`. Para especificar la fuente, podemos usar el método `setFont()` de *Graphics2D*.

Si queremos implementar nuestras propias rutinas de edición de texto o necesitamos más control sobre la distribución del texto que la que proporcionan los componentes de texto, podemos usar las clases del paquete *java.awt.font*.

2.2.8 Fuentes

Las formas que una fuente usa para representar los caracteres de una cadena son llamadas *glyphs*. Un carácter particular o una combinación de caracteres podría ser representada como uno o más *glyphs*. Por ejemplo, "á" podría ser representado por dos *glyphs*, mientras que la ligadura "fi" podría ser representada por un sólo *glyph*.

Se puede pensar en una fuente como en una colección de *glyphs*. Una sola fuente podría tener muchas caras, como pesada, media, oblicua, gótica y regular. Todas las

caras de una fuente tienen características tipográficas similares y pueden ser reconocidas como miembros de la misma familia. En otras palabras, una colección de *glyphs* con un estilo particular forma una cara de fuente; y una colección de caras de fuentes forman una familia de fuentes; y el conjunto de familias de fuentes forman el juego de fuentes disponibles en el sistema.

Cuando se utiliza el API 2D de Java, se especifican las fuentes usando un ejemplar de *Font*. Podemos determinar las fuentes disponibles en el sistema llamando al método estático **`GraphicsEnvironment.getLocalGraphicsEnvironment()`** y preguntando al *GraphicsEnvironment* devuelto. El método **`getAllFonts()`** devuelve un *array* que contiene ejemplares *Font* para todas las fuentes disponibles en el sistema; **`getAvailableFontFamilyNames()`** devuelve una lista de las familias de fuentes disponibles.

GraphicsEnvironment también describe la colección de dispositivos de dibujo de la plataforma, como pantallas e impresoras, que un programa Java puede utilizar. Esta información es usada cuando el sistema realiza la conversión del espacio de usuario al espacio de dispositivo durante el dibujo.

2.2.9 Distribución de Texto

Antes de poder mostrar el texto, debe ser distribuido para que los caracteres sean representados por los *glyphs* apropiados en las posiciones apropiadas. Si estamos usando *Swing*, podemos dejar que *JLabel* o *JTextComponent* manejen la distribución de texto por nosotros. *JTextComponent* soporta texto bidireccional y está diseñada para manejar las necesidades de la mayoría de las aplicaciones internacionales.

Si no estamos usando un componente de texto *Swing* para mostrar el texto automáticamente, podemos usar uno de los mecanismos de Java 2D para manejar la distribución de texto.

- Si queremos implementar nuestras propias rutinas de edición de texto, podemos usar la clase *TextLayout* para manejar la distribución de texto, iluminación y detección de pulsación. Las facilidades proporcionadas por *TextLayout* manejan muchos casos comunes, incluyendo cadenas con fuentes mezcladas, lenguajes mezclados y texto bidireccional.
- Si queremos un control total sobre la forma y posición de nuestro texto, podemos construir nuestro propio objeto *GlyphVector* usando *Font* y renderizando cada *GlyphVector* a través de *Graphics2D*.

2.2.10 Dibujo y relleno de gráficos primitivos

Cambiando el punteado y los atributos de dibujo en el contexto de *Graphics2D*, antes del dibujo, podemos fácilmente aplicar diversos estilos de líneas y patrones de relleno para gráficos primitivos. Por ejemplo, podemos dibujar una línea punteada creando el objeto *Stroke* apropiado y llamando a **`setStroke()`** para añadirlo al contexto *Graphics2D* antes de dibujar la línea. De forma similar, podemos aplicar un relleno de gradiente a un *Shape* creando un objeto *GradientPaint* y añadiendo al contexto *Graphics2D* llamando a **`setPaint()`** antes de dibujar la *Shape*.

2.2.11 Formas Geométricas

Para dibujar formas geométricas usaremos los métodos *Graphics2D* **`draw()`** y **`fill()`**.

A continuación se muestran a modo esquemático diversas formas geométricas con el código fuente correspondiente. Las variables **`rectHeight`** y **`rectWidth`** definen las dimensiones del espacio en que se dibuja cada forma, en píxeles. La variables **`x`** e **`y`** cambian para cada forma.

```
g2.draw(new Line2D.Double(x, y+rectHeight-1, x +
rectWidth, y));
```

```
g2.setStroke(stroke);
g2.draw(new Rectangle2D.Double(x, y, rectWidth,
rectHeight));
```

```
g2.setStroke(dashed);
g2.draw(new RoundRectangle2D.Double(x, y, rectWidth,
rectHeight, 10, 10));
```

```
g2.setStroke(wideStroke);
g2.draw(new Arc2D.Double(x, y, rectWidth, rectHeight,
90, 135, Arc2D.OPEN));
```

```
g2.setStroke(stroke);
g2.draw(new Ellipse2D.Double(x, y,rectWidth,
rectHeight));
```

```
int x1Points[] = {x, x+rectWidth, x, x+rectWidth};
int y1Points[] = {y, y+rectHeight, y+rectHeight, y};
GeneralPath polygon = new
       GeneralPath(GeneralPath.WIND_EVEN_ODD,
        x1Points.length);
polygon.moveTo(x1Points[0], y1Points[0]);

for (int index = 1;
     index < x1Points.length; index++) {
        polygon.lineTo(x1Points index],
                       y1Points index]);
};

polygon.closePath();
g2.draw(polygon);
```

```
int x2Points[] = {x, x+rectWidth, x, x+rectWidth};
int y2Points[] = {y, y+rectHeight, y+rectHeight, y};
GeneralPath polyline = new
         GeneralPath(GeneralPath.WIND_EVEN_ODD,
                     x2Points.length);

polyline.moveTo (x2Points[0], y2Points[0]);

for (int index = 1; index < x2Points.length; index++)
{
       polyline.lineTo(x2Points[index],
                       y2Points index]);
};g2.draw(polyline);
```

```
g2.setPaint(red);
g2.fill(new Rectangle2D.Double(x, y,
                              rectWidth, rectHeight));
```

```
g2.setPaint(redtowhite);
g2.fill(new RoundRectangle2D.Double(x, y,
                                   rectWidth,
                                   rectHeight,
                                   10, 10));
```

```
g2.setPaint(red);
g2.fill(new Arc2D.Double(x, y, rectWidth, rectHeight,
90, 135, Arc2D.OPEN));
```

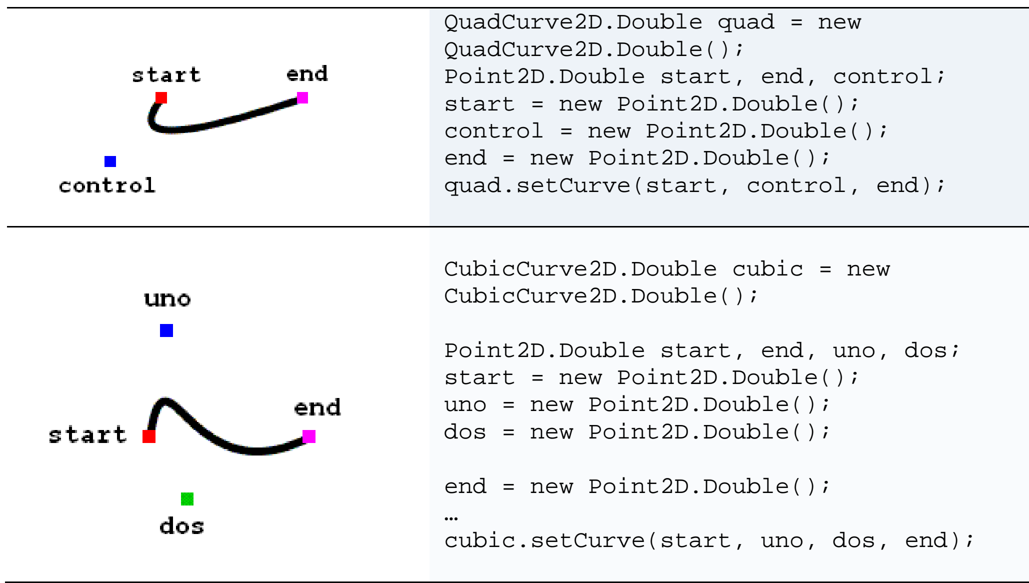
```
g2.setPaint(redtowhite);
g2.fill (new Ellipse2D.Double(x, y,
                              rectWidth,
                              rectHeight));
```

```
int x3Points[] = {x, x+rectWidth, x, x+rectWidth};
int y3Points[] = {y, y+rectHeight, y+rectHeight, y};

GeneralPath filledPolygon = new
         GeneralPath(GeneralPath.WIND_EVEN_ODD,
                     x3Points.length);
filledPolygon.moveTo(x3Points[0 , y3Points[0]);

for (int index = 1;
     index < x3Points.length; index++) {
      filledPolygon.lineTo(x3Points[index],
                           y3Points[index );
};
filledPolygon.closePath();
g2.setPaint(red);
g2.fill(filledPolygon);
```

Figura 2.18. Formas geométricas

2.2.12 Dibujar Curvas

Es posible dibujar curvas cúbicas y cuadráticas. Para ello se emplean puntos finales y puntos de control. Para dibujar una curva cuadrática se necesitarán tres puntos, mientras que para una curva cúbica, se necesitarán cuatro.




```
QuadCurve2D.Double quad = new
QuadCurve2D.Double();
Point2D.Double start, end, control;
start = new Point2D.Double();
control = new Point2D.Double();
end = new Point2D.Double();
quad.setCurve(start, control, end);
```

```
CubicCurve2D.Double cubic = new
CubicCurve2D.Double();

Point2D.Double start, end, uno, dos;
start = new Point2D.Double();
uno = new Point2D.Double();
dos = new Point2D.Double();

end = new Point2D.Double();
…
cubic.setCurve(start, uno, dos, end);
```

Figura 2.19. Curvas cúbicas y cuadráticas

2.2.13 Estilos de Línea

Los estilos de línea están definidos por el atributo *stroke* en el contexto *Graphics2D*. Para seleccionar el atributo stroke podemos crear un objeto *BasicStroke* y pasarlo dentro del método *Graphics2D* **`setStroke`**().

Un objeto *BasicStroke* contiene información sobre la anchura de la línea, estilo de uniones, estilos finales y estilo de punteado. Esta información se usa cuando se dibuja un *Shape* con el método **`draw()`**.

La **anchura de línea** es la longitud de la línea medida perpendicularmente a su trayectoria. La anchura de la línea se especificada como un valor *float* en las unidades de coordenadas de usuario, que es equivalente a 1/72 pulgadas cuando se utiliza la transformación por defecto.

El **estilo de unión** es la decoración que se aplica cuando se encuentran dos segmentos de línea. *BasicStroke* soporta tres estilos de unión:

Figura 2.20. Stroke: Estilos de unión

El **estilo de finales** es la decoración que se aplica cuando un segmento de línea termina. *BasicStroke* soporta tres estilos de finalización:

CAP_BUTT ***CAP_ROUND*** ***CAP_SQUARE***

Figura 2.21. Stroke: Estilos de finales

El **estilo de punteado** define el patrón de las secciones opacas y transparentes aplicadas a lo largo de la longitud de la línea. Este estilo está definido por un *array* de punteado y una fase de punteado. El *array* de punteado define el patrón de punteado. Los elementos alternativos en el *array* representan la longitud del punteado y el espacio entre punteados en unidades de coordenadas de usuario. El elemento 0 representa el primer punteado, el elemento 1 el primer espacio, etc. La fase de punteado es un desplazamiento en el patrón de punteado, también especificado en unidades de coordenadas de usuario. La fase de punteado indica que parte del patrón de punteado se aplica al principio de la línea.

2.2.13.1 Patrón de Relleno

Los patrones de relleno están definidos por el atributo *paint* en el contexto *Graphics2D*. Para seleccionar el atributo *paint*, se crea un ejemplar de un objeto que implemente el interface *Paint* y se pasa dentro del método *Graphics2D* `setPaint()`.

Tres clases implementan el interface *Paint*: *Color*, *GradientPaint*, y *TexturePaint*.

Para crear un *GradientPaint*, se especifica una posición inicial y un color y una posición final y otro color. El gradiente cambia proporcionalmente desde un color al otro a lo largo de la línea que conecta las dos posiciones.

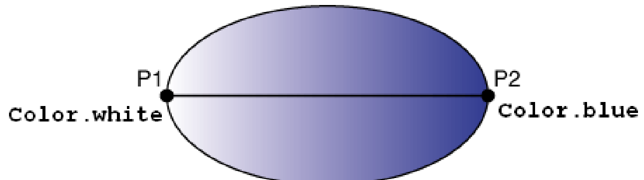


Figura 2.22. Uso de GradientPaint()

El patrón para una *TexturePaint* está definido por un *BufferedImage*. Para crear un *TexturePaint*, se especifica una imagen que contiene el patrón y un rectángulo que se usa para replicar y anclar el patrón.

```
Rectangle r = new Rectangle(0,0,10,50);
TexturePaint tp = new TexturePaint(Imagen_Patron, r);
```

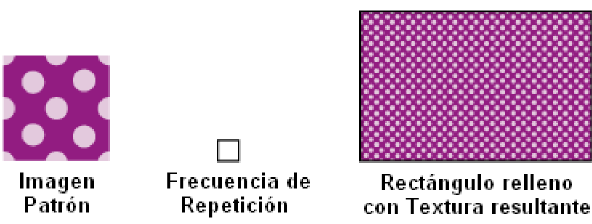


Figura 2.23. Uso de TexturePaint()

2.2.14 Transformar Formas, Texto e Imágenes

Podemos modificar el atributo *transform* en el contexto *Graphics2D* para mover, rotar, escalar y modificar gráficos primitivos mientras se están dibujando. El atributo *transform* está definido por un ejemplar de *AffineTransform*.

Graphics2D proporciona varios métodos para cambiar el atributo *transform*. Podemos construir un nuevo *AffineTransform* y cambiar el atributo *transform* de *Graphics2D* llamando al método **`setTransform()`**.

AffineTransform define los siguientes métodos para hacer más sencilla la construcción de nuevas transformaciones:

- **`getRotateInstance()`**
- **`getScaleInstance()`**
- **`getShearInstance()`**
- **`getTranslateInstance()`**

De forma alternativa podemos usar uno de los métodos de transformación de *Graphics2D* para modificar la transformación actual. Cuando se llama a uno de esos métodos de conveniencia, la transformación resultante se concatena con la transformación actual y se aplica durante el dibujado.

- **`rotate()`** especifica un ángulo de rotación en radianes.
- **`scale()`** especifica un factor de escala en direcciones *x* e *y*.
- **`shear()`** especifica un factor de compartición en direcciones *x* e *y*.
- **`translate()`**especifica un desplazamiento de movimiento en direcciones *x* e *y*.

También podemos construir directamente un *AffineTransform* y concatenarlo con la transformación actual llamando al método **`transform()`**.

El método **`drawImage()`** también está sobrecargado para permitirnos especificar un *AffineTransform* que se aplica a la imagen a dibujar. Especificar un *transform* cuando se llama a **`drawImage()`** no afecta al atributo *transform* de *Graphics2D*.

La **Figura 2.25** muestra un código de ejemplo donde escalamos una imagen:

```
AffineTransform tx = new AffineTransform();
double scale_x=0.8;
double scale_y=0.8;
tx.scale(scale_x, scale_y);
g.setTransform(tx);
g.drawImage(imagen,0,0,null);
```

Figura 2.24. Ejemplo. Escalado de imagen

2.2.15 Controlar la Calidad del Dibujo. Renderizado

Podemos usar el atributo *rendering hint* de *Graphics2D* para especificar si queremos que los objetos sean dibujados tan rápido como sea posible o si preferimos que se dibujen con la mayor calidad posible.
Para seleccionar o configurar el atributo *rendering hint* en el contexto, *Graphics2D* podemos construir un objeto *RenderingHints* y pasarlo dentro de **`Graphics2D.setRenderingHints()`.** Si sólo queremos seleccionar un *hint*, podemos llamar a **`Graphics2D.setRenderingHint()`** y especificar la pareja clave-valor para el *hint* que queremos seleccionar. Estas parejas están definidas en la clase *RenderingHints*.

RenderingHints soporta los siguientes tipos de *hints*:

- ***Alpha interpolation*** Por defecto, calidad, o velocidad.
- ***Antialiasing*** Por defecto, *on*, u *off*
- ***Color rendering*** Por defecto, calidad, o velocidad
- ***Dithering*** Por defecto, activado, o desactivado
- ***Fractional metrics*** Por defecto, *on,* u *off*
- ***Interpolation*** Vecino más cercano o bilinear
- ***Rendering*** Por defecto, calidad, o velocidad
- ***Text antialiasing*** Por defecto, *on,* u *off*.

2.2.16 Crear y Derivar Fuentes

Podemos mostrar una cadena de texto con cualquier fuente disponible en nuestro sistema, en cualquier estilo y tamaño que elijamos. Para determinar las fuentes disponibles en nuestro sistema, podemos llamar al método **`GraphicsEnvironment.getAvailableFontFamilyNames()`**. Este método devuelve un *array* de *strings* que contiene los nombres de familia de las fuentes disponibles. Se puede usar cualquiera de las cadenas, junto con un argumento tamaño y otro de estilo, para crear un nuevo objeto *Font*. Después de crear un objeto *Font*, podemos cambiar su nombre de familia, su tamaño o su estilo para crear una fuente personalizada.

El método **`getAvailableFontFamilyNames()`** de *GraphicsEnvironment* devuelve los nombres de familia de todas las fuentes disponibles en nuestro sistema.

```
GraphicsEnvironment gEnv =
GraphicsEnvironment.getLocalGraphicsEnvironment();

String envfonts[] = gEnv.getAvailableFontFamilyNames();
Vector vector = new Vector();

for ( int i = 1; i < envfonts.length; i++ ) {
      vector.addElement(envfonts[i]);
}
```

Figura 2.25. Ejemplo. Obtener los nombres de las fuentes disponibles en el sistema

El objeto *Font* inicial se crea con estilo *Font.PLAIN* y tamaño 10. Los otros estilos disponibles son *ITALIC*, *BOLD* y *BOLD+ITALIC*.

```
Font thisFont;
...
thisFont = new Font("Arial", Font.PLAIN, 10);
```

Figura 2.26. **Ejemplo. Creación de un objeto Font**

Un nuevo *Font* se crea a partir de un nombre de fuente, un estilo y un tamaño.

```
public void changeFont(String f, int st, String si){
      Integer newSize = new Integer(si);
      int size = newSize.intValue();
      thisFont = new Font(f, st, size);
      repaint();
}
```

Figura 2.27. Ejemplo. Método para cambiar el estilo de fuente

Para usar la misma familia de fuentes, pero cambiando uno o los dos atributos de estilo y tamaño, podemos llamar a uno de los métodos **`deriveFont()`**.

Para controlar la fuente utilizada para renderizar texto, podemos seleccionar el atributo *font* en el contexto *Graphics2D* antes de dibujarlo. Este atributo se selecciona pasando un objeto *Font* al método **`setFont()`**. En el ejemplo mostrado en la **Figura 2.29**, el atributo *font* se configura para usar un objeto *Font* recientemente construido y luego se dibuja la cadena de texto en el centro del componente usando la fuente especificada.

En el método **`paint()`**, el atributo *Font* del contexto *Graphics2D* se configura como el nuevo *Font*. La cadena se dibuja en el centro del componente con la nueva fuente.

```
g2.setFont(thisFont);
String cambio = "Fuente de prueba";
FontMetrics metrics = g2.getFontMetrics();
int width = metrics.stringWidth(cambio);
int height = metrics.getHeight();
g2.drawString(cambio, w/2-width/2, h/2-height/2);
```

Figura 2.28. Ejemplo. Aplicación del atributo Font al contexto Graphics2D

2.2.17 Imágenes

El API 2D de Java implementa un nuevo modelo de imagen que soporta la manipulación de imágenes de resolución fija almacenadas en memoria. Trabajaremos con una nueva clase *Image* que se puede extraer del paquete *java.awt.image*. Además podemos usar un objeto de tipo *BufferedImage*, para manipular datos de una imagen recuperados desde un fichero o una URL (Uniform Resource Locator).

Por ejemplo, se puede usar un *BufferedImage* para implementar *doble buffer*, los elementos gráficos se dibujan fuera de la pantalla en el *BufferedImage* y después se copian a ella a través de llamadas al método **`drawImage()`** de *Graphics2D*.

Las clases *BufferedImage* y *BufferedImageOp* también permiten realizar una gran variedad de operaciones de filtrado de imágenes como *blur*. El modelo de imagen productor/consumidor proporcionado en las versiones anteriores del JDK sigue siendo soportado por razones de compatibilidad.

Para crear un *BufferedImage* desde un fichero de imagen, se debe cargar el fichero en un objeto *Image* y posteriormente dibujar el *Image* en el objeto *BufferedImage*:

```
Image img = Toolkit.getDefaultToolkit().getImage("picture.gif");

int w = img.getWidth(this);
int h = img.getHeight(this);

BufferedImage b = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);

Graphics2D g2 = b.createGraphics();
g2.drawImage(img, 0, 0, null);
```

Figura 2.29. Carga fichero de imagen y dibujado en un objeto

3. Elementos utilizados en la realización del programa

3.1 Eclipse

Estamos ante un *framework*, que no es más que un conjunto de herramientas integradas para el rápido desarrollo de aplicaciones o entornos de trabajo, de código abierto e independiente de la plataforma donde se ejecute.

Eclipse se lanzó originalmente en Noviembre de 2001, producto de una inversión de cuarenta millones de dólares para su desarrollo por parte de *IBM,* antes de ofrecerlo como un producto de código abierto al consorcio *Eclipse.org*, integrado por algunas de las empresas más importantes del sector como *Rational*, *HP* o *Borland.*

La característica clave de Eclipse es la extensibilidad, ya que es una gran estructura formada por un núcleo y muchos *plugins* que van conformando la funcionalidad. La forma en que los *plugins* interactúan con el núcleo, es mediante interfaces o puntos de extensión totalmente abiertos a los desarrolladores, de manera que las nuevas aportaciones se integran sin dificultad ni grandes conflictos.

Otro de los puntos fuertes de Eclipse es la independencia del lenguaje de programación, que nos libera de las restricciones del contenido y permite programar en lenguajes tan distantes como HTML, Java, C, JSP, EJB, XML, etc.

Para obtener el IDE (*Integrated Development Environment*) Eclipse podemos descargarlo directamente del sitio web oficial del Proyecto Eclipse (Eclipse Foundation) o desde cualquier *mirror* autorizado, sin coste alguno y con total funcionalidad. El único requisito es la existencia de un JRE (Java Runtime Environment) (Java Microsystems) instalado previamente en el sistema, ya que está escrito casi en su totalidad en Java.

La instalación de Eclipse es tan sencilla como descomprimir el archivo descargado en el directorio que se estime conveniente, y aunque la descarga básica del entorno Eclipse incluye los *plugins* básicos, puede que sea deseable obtener alguna funcionalidad extra. Para ello, es necesario instalar nuevos *plugins* accediendo al apartado *Community* del sitio web oficial de Eclipse, donde se pueden encontrar enlaces a cientos de *ellos*. Para añadir estos elementos basta con descomprimir el archivo descargado en el subdirectorio *Plugins* de la carpeta donde está instalado Eclipse. La próxima vez que se ejecute Eclipse, automáticamente se reconocerán y añadirán dicha funcionalidad.

Las versiones que se pueden descargar del sitio web de Eclipse vienen con un ejecutable, el cual permite lanzar directamente el IDE Eclipse. Antes de ejecutarlo es importante verificar que se tienen permisos de escritura en el directorio, ya que la primera vez que se ejecuta tiene que crear las carpetas en las que guardará información sobre *workspaces*, *logs*, etc.

En la siguiente figura podemos ver la pantalla principal del IDE Eclipse, el cual no esconde ningún secreto para aquellos que estén acostumbrados a programar. No parece necesario explicar cada una de las zonas del IDE, ya que podemos encontrar los elementos típicos, un área de edición de código, una zona destinada a alojar los recursos disponibles en cada proyecto, barra de herramientas, zona de registro, salida de información, etc.

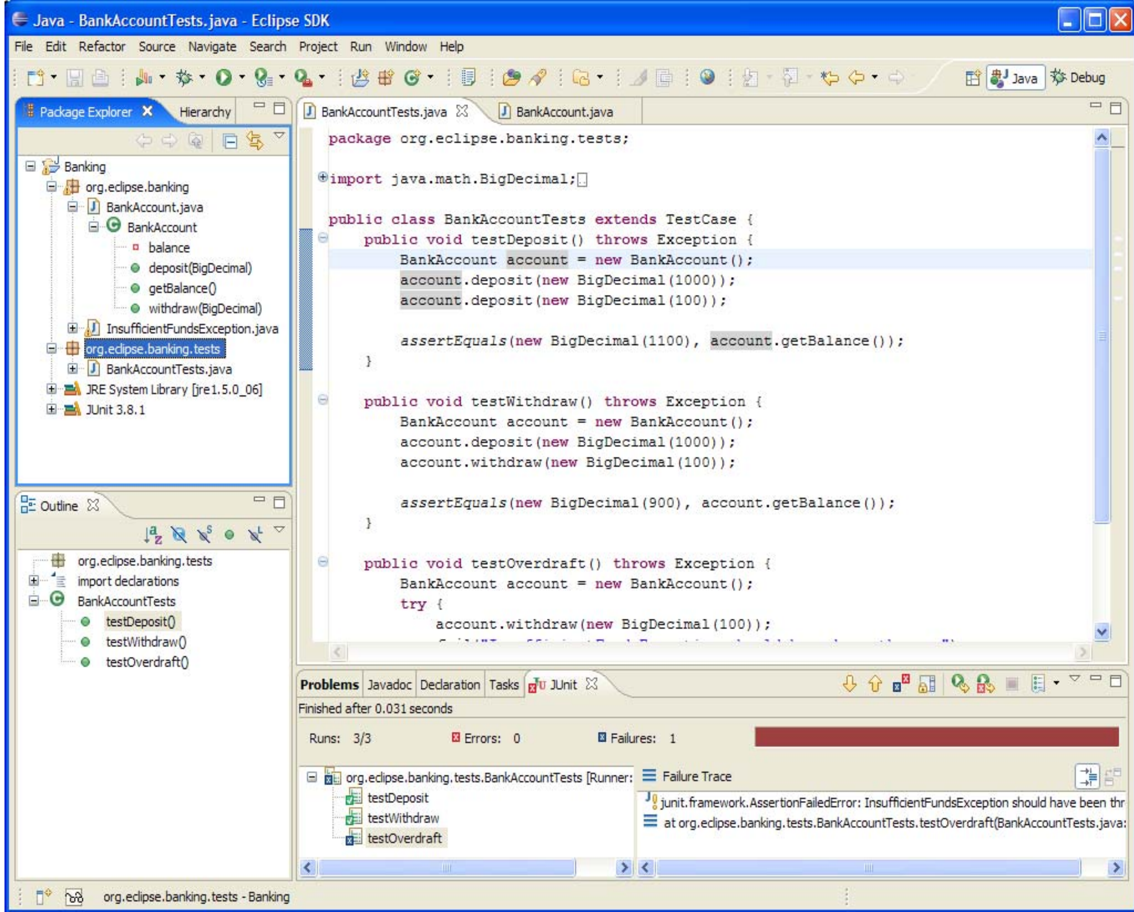


Figura 3.1 Vista general del IDE Eclipse

3.2 CVS, instalación y configuración.

CVS (*Concurrent Versions System*) es un sistema de mantenimiento de código fuente, que mantiene registrados los cambios producidos en dicho código, pudiendo asociar hitos o puntos de importancia, como versiones, lo que convierte a este tipo de herramientas en extraordinariamente útil para grupos de desarrolladores que trabajan en un mismo proyecto.

Se basa en una arquitectura cliente-servidor, en donde el servidor se encarga de guardar cada una de las versiones y los clientes se conectan a él para acceder a copias de esas versiones. De igual forma, una vez modificada dicha copia en local, se inserta en el servidor.

Además de poder acceder a las copias de la última versión, nos permite recuperar versiones anteriores de los ficheros, lo que puede ser realmente interesante a la hora de realizar posibles modificaciones al código sin necesidad de preocuparse por realizar copias de seguridad.

No es un requisito que sean ficheros fuentes de código, ya que simplemente trabaja con ficheros ASCII, por lo que cualquier tipo los mismos puede ser utilizado en este tipo de programas.

En cuanto a la topología de este tipo de sistemas puede ser diversa, pudiendo tener dos tipos básicos, trabajar de forma local (repositorio y copias de trabajo en el mismo sistema) o remota (el repositorio está en un sistema servidor y la copia local en otro que es cliente del primero).

Los dos conceptos claves a la hora de trabajar con CVS son:

- Repositorio: Jerarquía de directorios alojada en el servidor CVS que contiene diferentes módulos a disposición de los usuarios.
- Módulo: Árbol de directorios que forma parte del repositorio. Cuenta con un nombre identificador gracias al cual podremos trabajar con él de forma selectiva.

Una vez hecha una breve introducción de lo que es un CVS pasamos a ver el software utilizando como servidor, que en nuestro caso al trabajar en Windows ha sido CVSNT (March Hare Pty Ltd & CVSNT Project), que es el software más completo y avanzado *open-source* que hay en el mercado.

Para descargar este programa es necesario ir a la página del fabricante e indicar simplemente el tipo de S.O. del que disponemos, como podemos ver en la **Figura 3.2**.

Figura 3.2 Página Web del fabricante

Una vez descargado procederemos a la instalación del mismo, para ello deberemos ejecutar el asistente y seguir los pasos para realizar con éxito esta etapa. No cabe destacar nada sobre este punto ya que es la típica instalación de *Windows* en la que deberemos preocuparnos simplemente de la ruta de instalación, que debe ser sobre un sistema de archivos NTFS (*New Technology File System*).

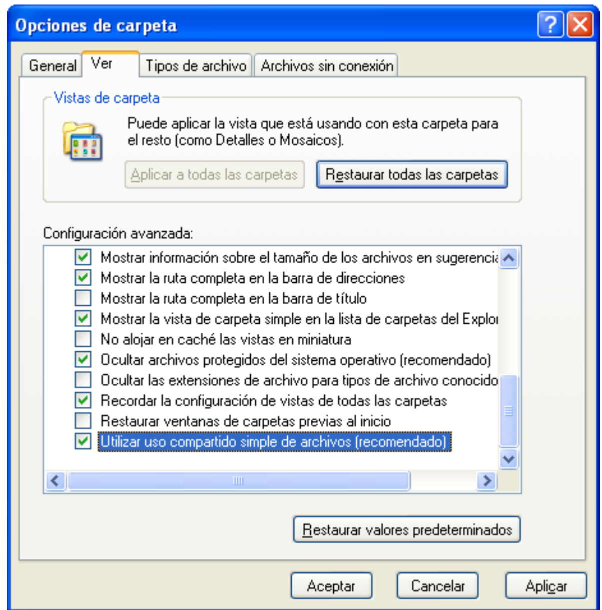


Figura 3.3 Habilitar el uso compartido simple de archivos

Antes de proceder a la configuración del servidor es necesario comprobar que tenemos seleccionada la casilla *Utilizo uso compartido simple de archivos*, que podemos encontrar en la ruta *Panel de control/Opciones de carpeta/Ver* tal y como podemos ver en la **Figura 3.3**.

Una vez habilitada dicha opción comenzamos con la configuración del servidor, para ello accedemos a su panel de control mediante el menú *Inicio* de *Windows*.

Figura 3.4 Localización del panel de control del servidor de repositorio

Arrancada la interfaz gráfica del servidor nos encontramos con la pestaña *About*, en donde lo más significativo es la posibilidad de poder arrancar y parar el propio servidor. Como vamos a proceder a su configuración es necesario tener el *CVSNT Service* parado, por lo que deberemos pulsar el botón de *Stop* en dicho servicio, dando igual el estado de *CVSNT Lock Service*.

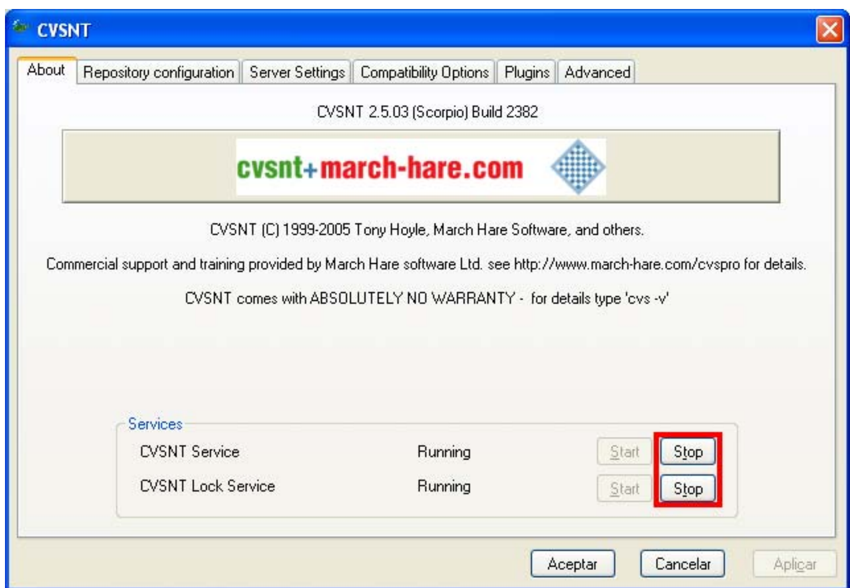


Figura 3.5 Pestaña *About* del servidor CVSNT

El siguiente paso será añadir un repositorio, que como hemos comentado anteriormente es la jerarquía de directorios alojada en el servidor CVS que contiene diferentes módulos a disposición de los usuarios, para ello nos situaremos en la pestaña *Repository configuration* y pulsaremos sobre el botón marcado y denominado *Add*, tal y como aparece en la **Figura 3.6**. Como observación señalar que podemos ver el nombre del servidor en el apartado *Server Name*.

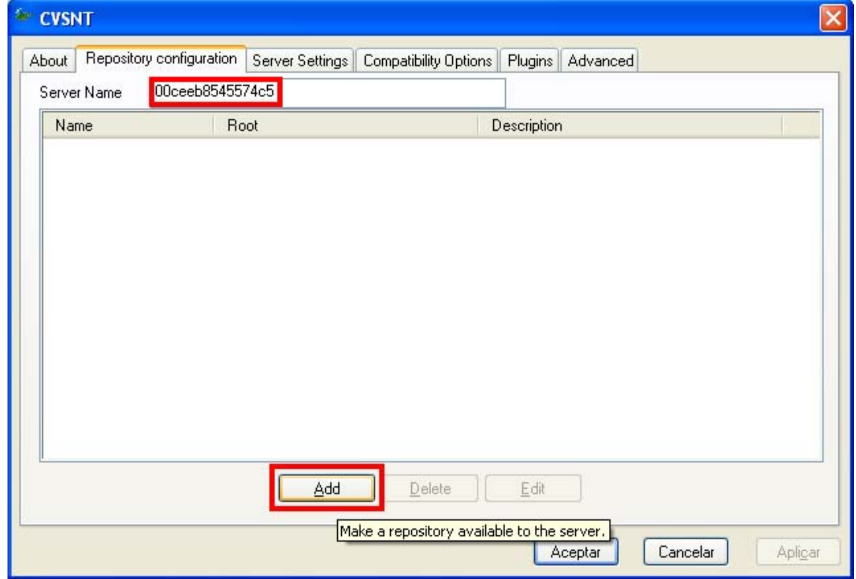


Figura 3.6 Pestaña *Repository configuration* del servidor CVSNT

Una vez pulsado nos aparece una ventana que mostramos en la siguiente **Figura 3.7**, y en donde deberemos rellenar los distintos campos. El primero de ellos es *Location*, donde podremos seleccionar la ruta donde va a estar localizado nuestro repositorio, no pudiendo contener espacios en la misma. El siguiente parámetro, *Name*, es el nombre que deseamos darle a este repositorio en concreto. En cuanto a *Description* se trata

de un parámetro sin importancia, ya que simplemente dará una descripción del repositorio pero sin influencia alguna en su instalación o ejecución.

En cuanto a las tres casillas que se encuentran en la parte inferior deberemos marcarlas, pues queremos que esté visible, activo y que sea el repositorio por defecto.

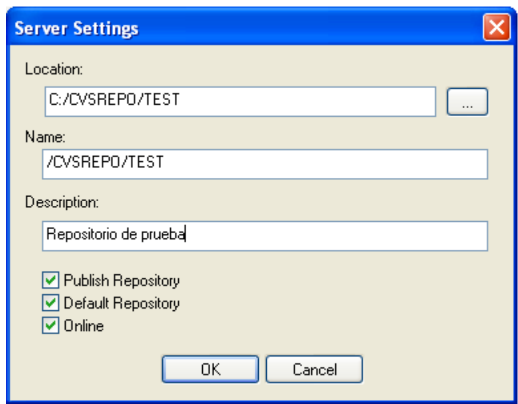


Figura 3.7 Ventana de opciones para añadir un repositorio

Cuando tengamos todos los campos rellenos y las pestañas activas podemos aceptar la configuración, tras ello aparecerá la pestaña anterior debidamente rellenada, tal y como podemos ver a continuación.

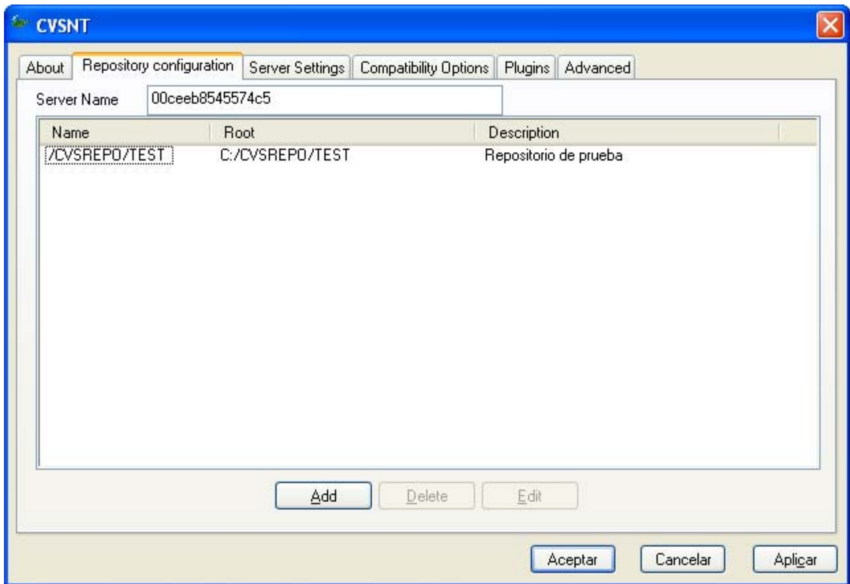


Figura 3.8 Aspecto final de la pestaña *Repository configuration* del servidor CVSNT una vez añadido un repositorio

Una vez añadido el repositorio pasamos a configurar la siguiente pestaña, denominada *Server Settings*, en donde dejaremos todo como está, excepto el directorio temporal que deberemos modificarlo a nuestro antojo. El resultado final de dicha configuración se puede observar en la **Figura 3.9**.

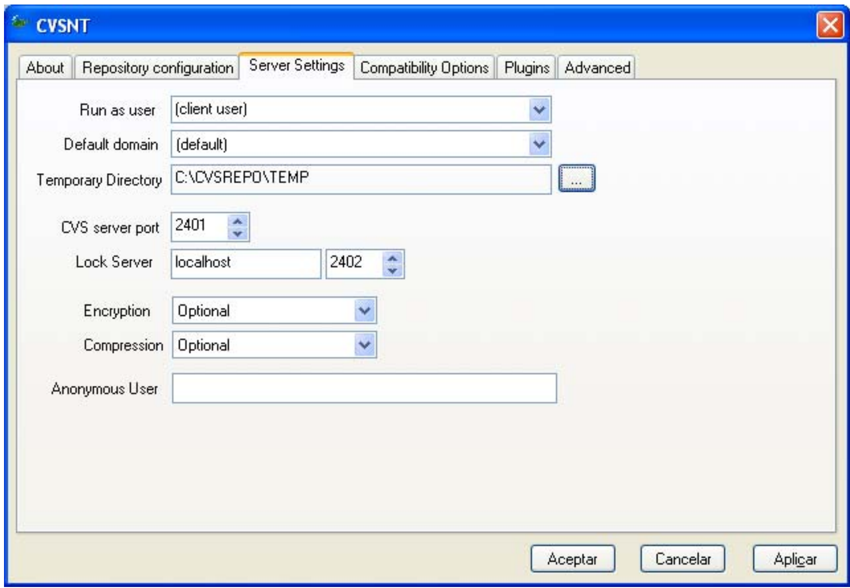


Figura 3.9 Pestaña *Server Settings* del servidor CVSNT

No es necesario tocar nada en la pestaña de *Compatibility Options*, por lo que la obviamos y pasamos directamente a ver que se debe realizar sobre la de *Plugins*, donde únicamente debemos comprobar que el protocolo *sserver* está activo. Para ello nos situamos sobre dicho protocolo y pulsamos el botón *Configure* como mostramos en la **Figura 3.10**.

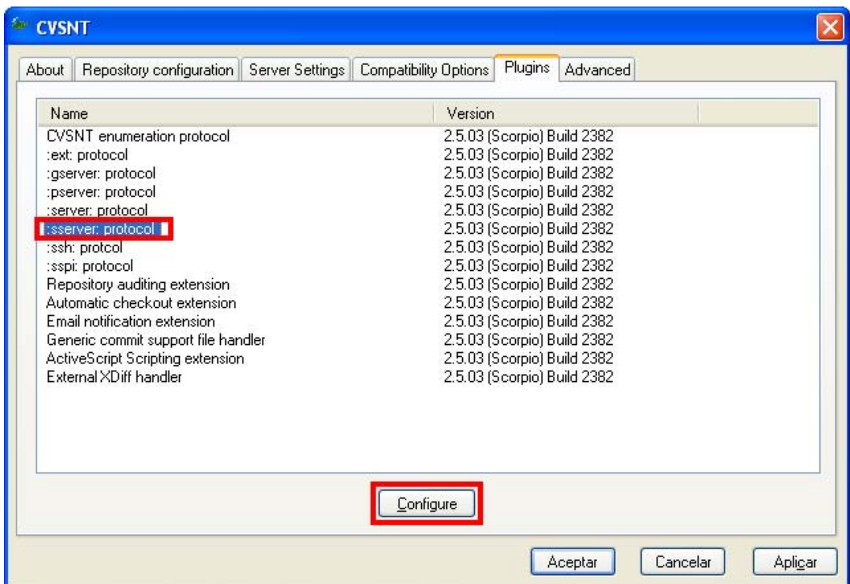


Figura 3.10 Pestaña *Plugins* del servidor CVSNT

Tal y como podemos ver en la siguiente figura podemos comprobar que el *plugin* está activo y que además sólo pedimos una clave de acceso asociado a un usuario que generaremos en un paso posterior, para poder acceder al repositorio a través de dicho protocolo.

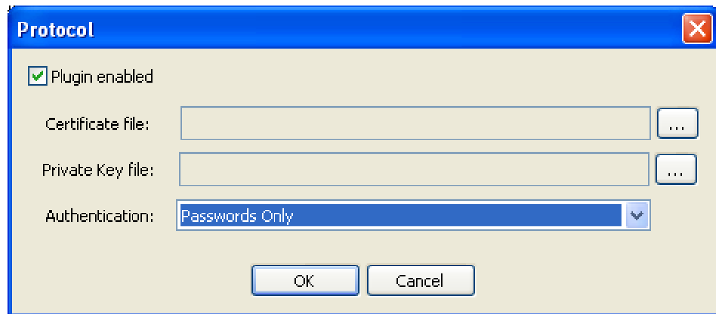


Figura 3.11 Ventana de configuración del protocolo *sserver*

Una vez comprobado que tenemos correctamente configurado el protocolo y que está activo para el acceso al repositorio, comprobamos que la configuración de la última pestaña es correcta, tal y como se muestra en la **Figura 3.12**. Normalmente esta es la configuración por defecto pero es necesario comprobarlo para no tener problemas en los siguientes pasos.

Por último es necesario aplicar los cambios para posteriormente arrancar el sistema, desde la pestaña *About* como al inicio de la configuración. Una vez realizada esta operación, podemos aceptar dicha configuración y el estado del sistema saliendo del panel de control del servidor.

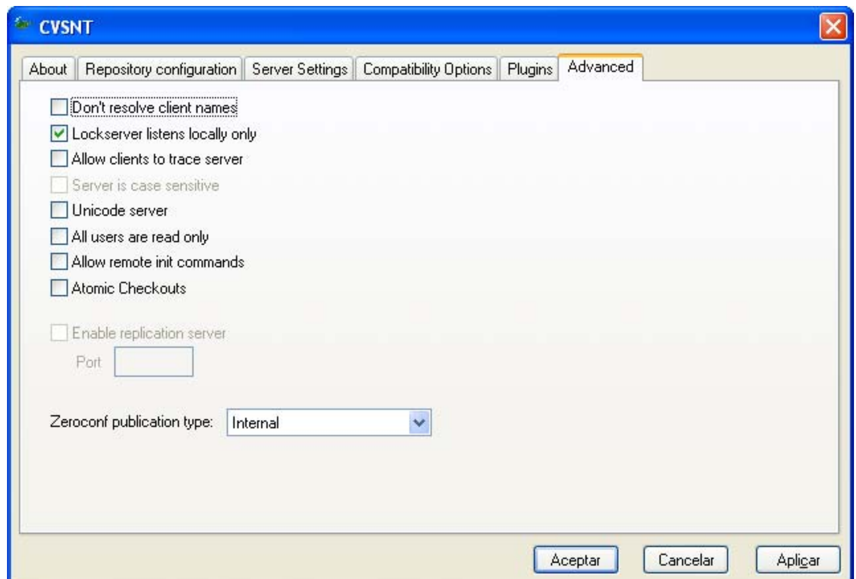


Figura 3.12 Pestaña *Advanced* del servidor CVSNT

Los siguientes pasos a realizar se deben realizar desde línea de comandos, por lo que accederemos a la misma e introduciremos los siguientes:

```
set cvsroot=:sspi: 00ceeb8545574c5:/CVSREPO/TEST
```

Este comando fija la variable de entorno *cvsroot*, que deberemos configurar para usar el acceso al servidor y ruta adecuados a través de la interfaz *sspi*. Una vez ejecutado

el comando procedemos a crear un usuario con un alias determinado para el acceso al repositorio, fijando además la contraseña para dicho acceso.

Para ello introducimos y ejecutamos:

```
cvs passwd -r cvsuser -a charlie
Adding user charlie@00ceeb8545574c5
New Password: *****
Verify Password: *****
```

Con esto ya podemos dar por concluida la configuración de nuestro repositorio, por lo que llega el momento de configurar nuestro entorno de desarrollo para el acceso al servidor.

3.3 Configuración y uso de Eclipse con CVS

Este manual supone la existencia de un proyecto básico de Eclipse el cual queremos añadir a nuestro control de versiones. Para ello debemos pulsar el botón derecho sobre el nombre del proyecto en la pestaña *Package Explorer*. Una vez realizado esto aparecerán las acciones que podemos realizar sobre dicho proyecto, interesándonos el grupo *Team* y la opción dentro del grupo de *Share Project*.

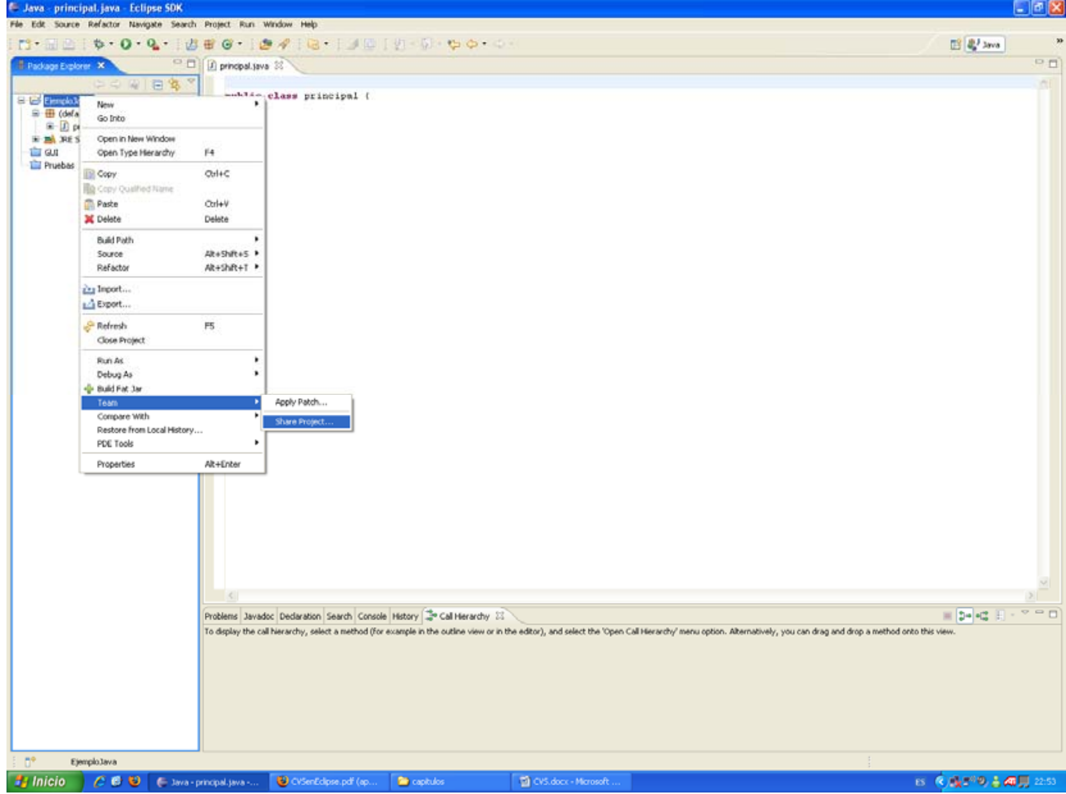


Figura 3.13 Ruta para añadir un proyecto al control de versiones

Seleccionada dicha acción nos aparecerá la lista de repositorios, **Figura 3.12**, que existen en el sistema. Nosotros en nuestro caso seleccionaremos aquel que hemos configurado anteriormente, tras lo que nos pedirá el *password* asociado al usuario del repositorio.

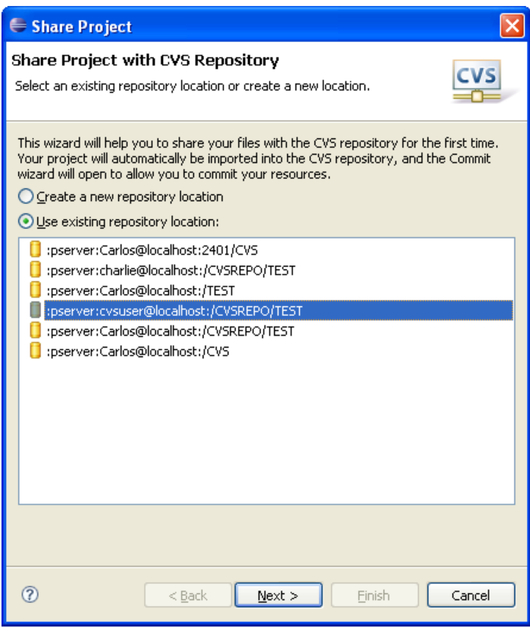


Figura 3.14 Selección de repositorio en *Eclipse*

Tras la selección del repositorio es necesario seleccionar el módulo donde insertaremos nuestro proyecto, en este caso generaremos un nuevo módulo de nombre como el proyecto.

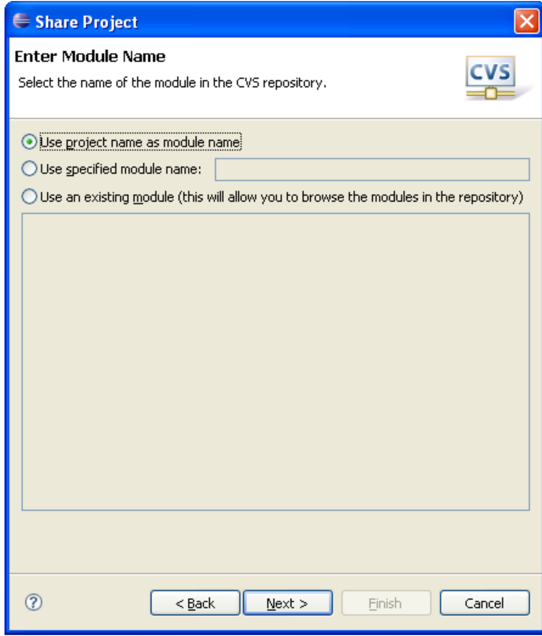


Figura 3.15 Selección del módulo dentro del repositorio en *Eclipse*

Sólo nos queda seleccionar qué ficheros queremos sincronizar con el control de versiones, pudiendo introducir una reseña inicial.

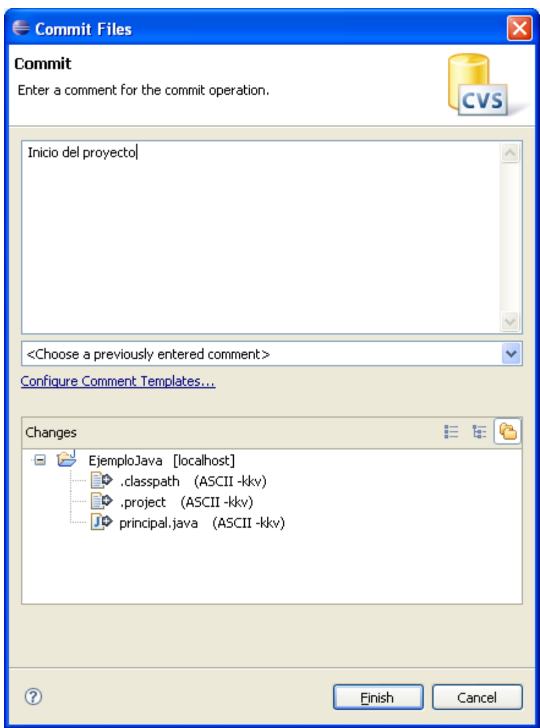


Figura 3.16 Sincronización inicial de ficheros fuente del proyecto con el CVS

Después de agregar el proyecto al control de versiones pasamos a explicar cómo utilizar el CVS para controlar los cambios en el proyecto. Una vez hecho algún cambio a los archivos, es necesario subirlos al repositorio. Eclipse marca con un “>” aquellos archivos que han cambiado desde el último *update* con el repositorio. Por ejemplo, podemos ver en la **Figura 3.17** que el fichero *principal.java* ha sido modificado respecto a la versión que existe en el repositorio.

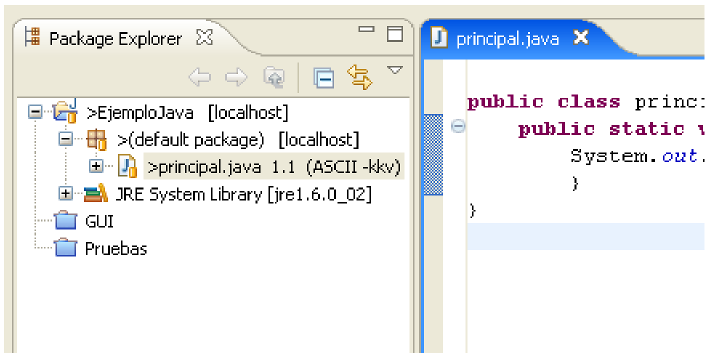


Figura 3.17 Visualización de estado de los ficheros en Eclipse

Para subir los cambios al CVS primero tenemos que hacer un *update*, para que Eclipse baje los nuevos archivos presentes en el repositorio y actualice nuestros archivos con los cambios existentes.

Para hacer el *update*, marcamos nuestro proyecto y con el botón derecho seleccionamos *Team->Update*. Aparecerá una ventana preguntando con que rama del módulo queremos sincronizarlo. Una vez terminado el proceso, estamos listos para hacer el *commit*. Para ello seleccionamos el proyecto y hacemos *Team->Commit*. Eclipse nos preguntará si queremos agregar al repositorio los nuevos archivos y nos permitirá añadir una descripción para el mensaje del *commit*.

Estas son las acciones básicas para el correcto uso del control de versiones, pero la funcionalidad del mismo no se queda en ese punto, sino que es mucho más amplia permitiéndonos examinar las diferencias entre distintas versiones, pasar a una nueva rama, etc.

4. Desarrollo de la aplicación

4.1 Introducción

En este capítulo trataremos el desarrollo *software* de nuestro sistema. Introduciremos cada paquete y las clases que encontramos en los mismos, prestando especial atención en los puntos más importantes para la implementación del simulador.

Para la programación de esta aplicación se han seguido dos patrones básicos. El primero de ellos es el patrón MVC (Modelo-Vista-Controlador), que se basa en la separación de la interface de usuario (vista) de la funcionalidad (modelo).

- Modelo: Problema que tratamos de resolver. Este problema suele ser independiente de cómo queramos que nuestro programa recoja los resultados o cómo queremos que los presente.
- Vista: Presentación visual que queramos hacer del juego. Lo llamaremos interface gráfica por ser lo más común, pero podría ser de texto, de comunicaciones con otro programa externo, con la impresora, etc.
- Controlador: La tercera parte de código es aquel código que toma decisiones, algoritmos, etc. Es código que no tiene que ver con las ventanas visuales ni con las reglas de nuestro modelo, sólo debe saber cuáles son los puntos de entrada del modelo y cuando ejecutarlos.

Es interesante hacer notar que las flechas sólo van en sentido de vista a modelo y controlador, y del controlador al modelo. Esto puede ser contemplado en la **Figura 4.1**, en donde el usuario inicia el proceso mediante la interacción con la vista, produciendo un evento sobre el modelo, el cual interactuará con el controlador internamente para resolver el problema.

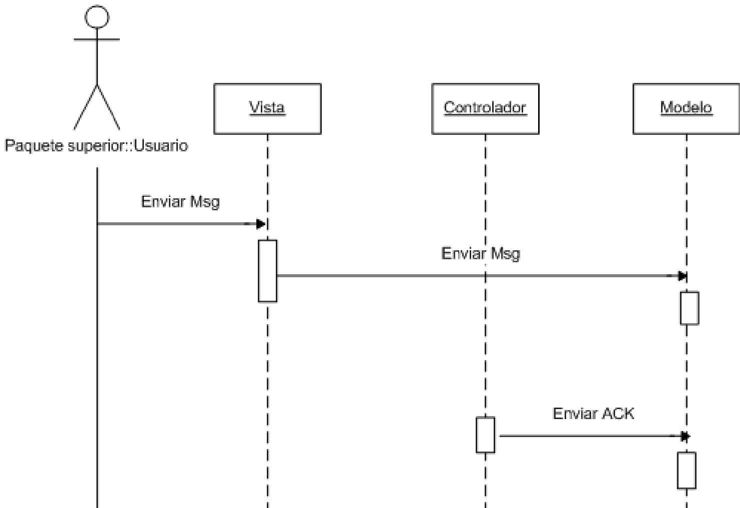


Figura 4.1 Diagrama de secuencia del patrón MVC

Ahora surge una pregunta. Si el controlador decide enviar un mensaje en el modelo TCP, ¿cómo se entera la interface gráfica para visualizar dicho mensaje en pantalla? Debemos tener en cuenta que ni el modelo ni el controlador ven a la vista, por lo que no pueden llamar a ninguna clase ni método de ella para que se actualice.

Para este tipo de problemas, tenemos otros patrones de diseño, por ejemplo, el patrón observador. Debemos hacer una interface que tenga un método de notificación de eventos, en este caso un método **update()**. Llamemos a esta interface *ObserverTCP*.

Esta interface formaría parte del modelo, de forma que las clases del modelo sí pueden verla. La clase del modelo debe tener una lista de objetos que implementen esta interface. La clase del modelo debe tener métodos **addObserver (*ObserverTCP*)** y **removeObserver (*ObserverTCP*) y notifyObserver()**. Estos métodos añadirían o borrarían el parámetro que se les pasa de la lista y que es un objeto que implementa la interface.

Tanto el controlador como la vista, deben implementar esta interface y deben llamar al método **addObserver ()** del modelo. A partir de este momento, cada vez que el se realice una acción debe llamar al método **update()** de todos los *ObserverTCP* que tenga en su lista.

En la **Figura 4.2** podemos ver lo anteriormente comentado. La clase *ModeloTCP* llama al método de la clase *SujetoComunicacionTCP* **enviarMsg()**, dicha clase realiza su proceso interno para la ejecución del mismo, llegando a un punto donde debe llamar a su método **notify()** que ejecutará un **update()** sobre los sujetos que han decidido observar el estado del sujeto en cuestión.

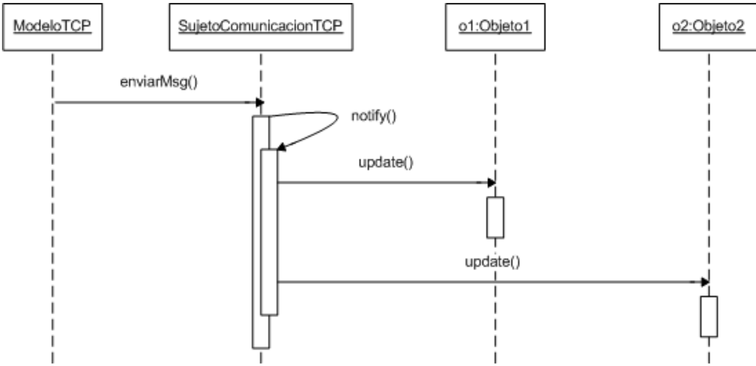


Figura 4.2 Diagrama de secuencia del patrón observador

El proyecto está dividido en paquetes, un total de cuatro, y un fichero que actúa como clase principal aglutinando al total de los objetos que se deben crear para la correcta generación del programa. Los paquetes son:

- Configuración.
- GUI.
- ModeloTCP.
- Observer.

4.2 Paquete *GUI*

El mayor trabajo ha sido realizar una interface de usuario (GUI), intuitiva y potente a la vez. Para ello se define una clase principal llamada GUI.java de donde se crean todos los elementos gráficos, y se añaden a la misma.

4.2.1 *GUI.java*

A grandes rasgos la *GUI* se compone de cinco elementos, cada uno independiente del resto, es decir, que no están comunicados entre sí, solamente son controlados a partir de la *GUI* en su creación, ya que el verdadero control de los mismos queda supeditado al método **update()** del patrón observador.

Tal y como se muestra en el siguiente diagrama de dependencias UML la *GUI* tiene, como elementos más importantes, los siguientes:

- *JCanvasEstados*
- *JPanelBotones*
- *JCanvasVentana*
- *JCanvas*
- *miJTable*

De todos ellos hablaremos extensamente e independientemente en puntos posteriores.

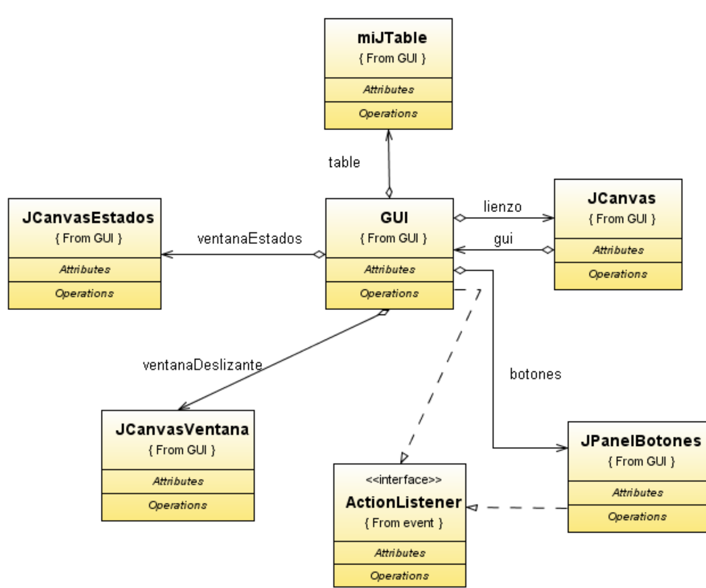


Figura 4.3 Diagrama de dependencias de la clase *GUI.java*

Esta clase extiende de *JFrame* e implementa un *ActionListener* para la barra de menú que se encuentra en la parte superior de la ventana, de donde podremos acceder a crear una simulación nueva, las opciones de visualización, imprimir, guardar, etc.

La clase *GUI.java* no tiene ningún tipo de control y acción, simplemente sirve de contenedor para los demás objetos que genera e incrusta en su área. Es interesante comentar que la GUI está definida a un tamaño fijo de 1280x1024 *pixeles*. Si la resolución de la tarjeta gráfica donde deseemos ejecutar el programa es inferior a dicha resolución, el sistema nos enviará un mensaje de error indicándonos que no es una resolución válida, saliendo inmediatamente de la aplicación.

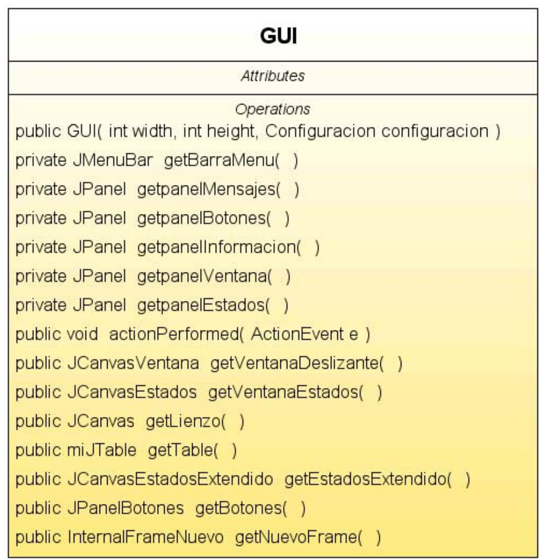


Figura 4.4 Diagrama de clase de la clase *GUI.java*

Esto puede parecer un requisito demasiado duro, pero es necesario comprender que para una correcta visualización de los contenidos es requisito fundamental una gran zona de visualización, debido a la gran cantidad de información que la aplicación debe mostrar al usuario.

En la **Figura 4.4** observamos el diagrama de clase de la clase *GUI.java*. Como podemos observar no existen métodos que, como hemos comentado anteriormente, otorguen funcionalidad alguna. Simplemente crea objetos que se incrustan en el propio *JFrame*.

4.2.2 *JCanvas.java*

Es en la clase *JCanvas.java* donde se han centrado los mayores esfuerzos y trabajos, ya que es en ella donde se presenta el diagrama temporal de intercambio de mensajes entre emisor y receptor.

Inicialmente podemos encontrar en ella el bloque emisor y el bloque receptor, cada uno de ellos con una línea temporal vertical dividida en slots temporales. Conforme vayamos simulando se dibujaran sobre el *JCanvas* los mensajes intercambiados y la información relevante de la misma.

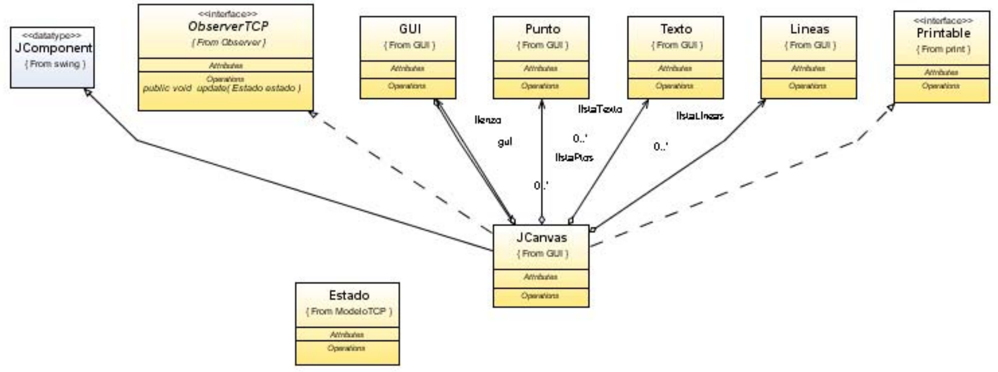


Figura 4.5 Diagrama de dependencias de la clase *JCanvas.java*

En la **Figura 4.5** podemos ver los objetos más importantes que tienen relación con la clase estudiada en este punto. Estos objetos se irán referenciando según avance el apartado, dejando claro las necesidades y dependencias con la misma.

El objeto JCanvas necesita saber cuál es su tamaño para realizar un dibujo generalista, es decir, en función de su tamaño. Por ello, cuando se genera a través de la clase *GUI.java* recoge su tamaño en su primer pintado, restringiéndose al asignado por la misma.

Hay que especificar muchas consideraciones hechas sobre este componente. La primera reside en que el componente de pintado en *AWT* era un *Canvas*, clase integrada como clase en las propias bibliotecas de dibujo. En *Swing* esta clase desaparece por diversos problemas y se deja al usuario la creación de cualquier tipo de componente gráfico avanzado a través de la clase *JComponent*, que hereda de la clase *Container*, la cual deberá ser extendida para adquirir las funcionalidades de cualquier objeto gráfico, debiendo implementar su propio código para obtener la funcionalidad deseada.

Los componentes *Swing* heredan las siguientes funcionalidades a partir de la clase *JComponent.java* :

- Bordes.
- Doble *buffer.*
- *ToolTips.*
- Propiedades.
- Aspecto.

De los puntos anteriores, la funcionalidad con más relevancia a la hora de tratar gráficos es la implementación por defecto del doble *buffer*, que se realiza de forma automática, que tiene como finalidad eliminar el incomodo parpadeo que se obtenía anteriormente en la clase *Canvas* de *AWT*.

Una de las funcionalidades que se le deseaban otorgar al *JCanvas* era la posibilidad de dibujar en diferentes capas, para poder mostrarlas según la elección del usuario, es decir, crear capas con el fondo, líneas, texto, etc.

Inicialmente se pensó que esto vendría incluido en alguna clase de Java para manejar este tipo de elementos. Tras un estudio exhaustivo se observó que Java no incluía esa funcionalidad, por lo que se decidió implementarla manualmente.

La idea era crear diversos objetos pertenecientes a la clase *BufferedImage* e incluirlos en una clase maestra que los controlase denominada *BufferedPanel* (ver **Figura 4.6**). Tras diversas pruebas se implementó con éxito, pero se obtuvo un gran problema a la hora de la ejecución del programa. Cada uno de los *BufferedImage* en su inicio tenía un peso de memoria equivalente a cuatro Mbytes, multiplicándose cada vez que el área visible crecía. Si teníamos cuatro *BufferedImage* y cada uno con una pequeña simulación pesaba alrededor de cincuenta Mbytes, colapsábamos la memoria de intercambio entre el sistema y la máquina virtual de Java, produciéndose una excepción.

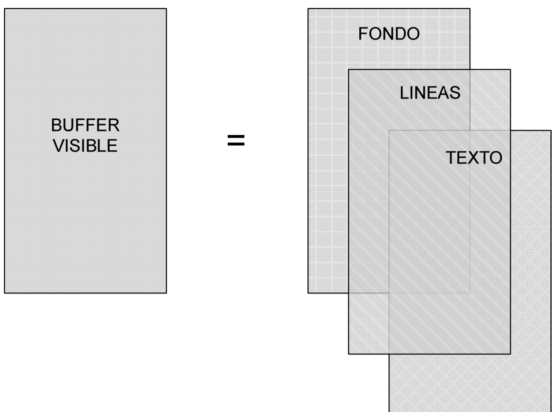


Figura 4.6 Objeto *BufferedPanel*

El problema anterior se podía solucionar aumentando la memoria de intercambio mediante un argumento a la hora de ejecución del simulador que afecta directamente sobre dicha memoria. Sin embargo, el inconveniente seguía siendo el alto peso que tenía dicho elemento, que repercutía directamente sobre el resultado final. Estábamos ante un programa lento y demasiado pesado que colapsaba cualquier CPU del mercado.

El principal objetivo de este proyecto era crear una aplicación útil y amigable para el usuario, y una aplicación tan pesada frustraba cualquier intento por conseguirlo. Había pues que cambiar la filosofía a la hora de pintar sobre dicho elemento, y por tanto, el elemento en sí.

La solución que a priori resultaba más eficaz era pintar directamente sobre el *JCanvas* mediante la obtención de su objeto gráfico, es decir, sin introducir ningún objeto de tipo *BufferedImage*. Para dotar al simulador de la opción de escoger diferentes elementos a pintar se requería algún tipo de sistema de control, por lo que se requería de unas variables que actuasen de activadores para escoger dichos elementos.

Además, es lógico pensar que es inútil pintar absolutamente todo. Solamente sería deseable pintar aquellas zonas visibles por el usuario en una determinada vista.

Con todo esto se consiguió un comportamiento realmente extraordinario para los recelos iníciales que planteaba trabajar con gráficos sobre Java, ya que es un lenguaje poco preparado para ello y que presenta grandes cargas computacionales si se desea realizar.

En conclusión, pintamos directamente sobre el *JComponent* mediante la obtención de su objeto gráfico, y solamente repintamos aquellas zonas que están visibles al usuario, entendiendo que es necesario pintar no sólo lo que se vea en su totalidad, sino que se ha de pintar totalmente aquello que se pueda ver parcialmente.

Como podemos ver en la **Figura 4.7** sólo imprimimos lo que se encuentra en el área visible (en la figura indicado pro el recuadro rojo). Es decir, nos evitamos hacer las llamadas para pintar todo aquello que no sea visible. Comentar que también se pintan completamente aquellos puntos que tienen influencia dentro del recuadro, aunque parcialmente no se vean. Es esto lo que le otorga una fluidez gráfica al programa, ya que sólo pintamos la zona visible y esto hace al sistema evitar pintar todas las líneas, pensando que pintar consume gran cantidad de recursos y tiempo.

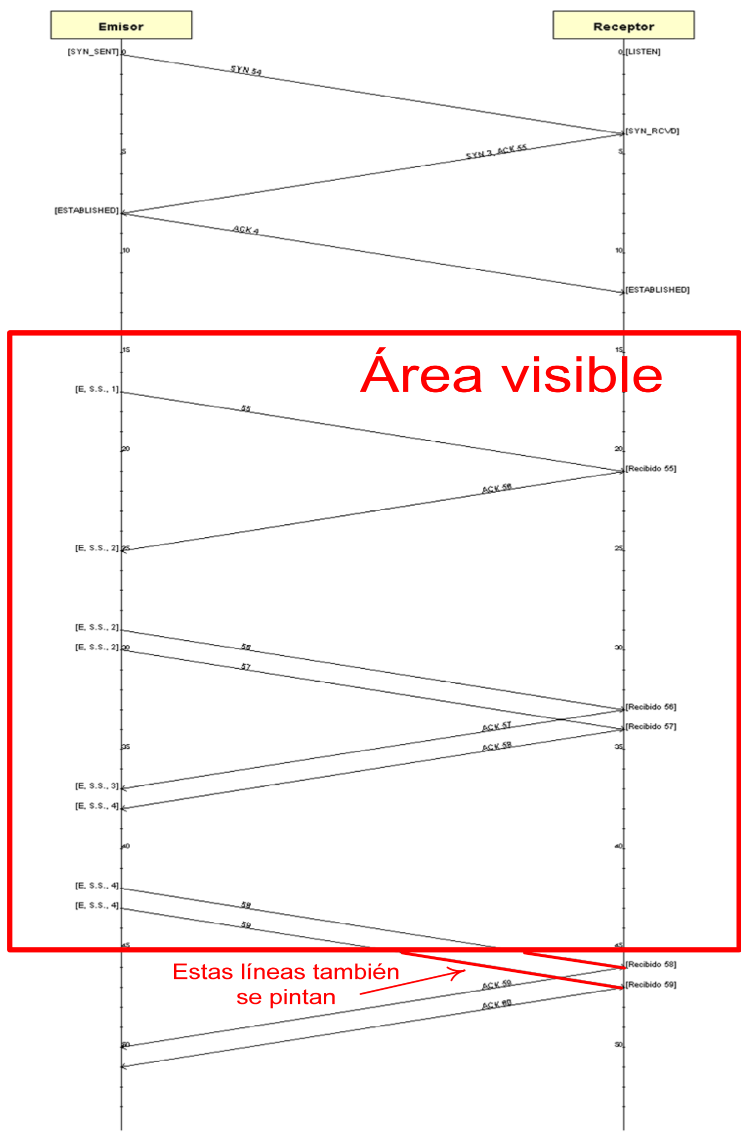


Figura 4.7 Área de dibujo sobre el objeto *JCanvas*

Una vez implementada la manera correcta de pintar sobre el objeto se debía comenzar a introducir elementos. Tres de ellos son objetos de tipo *ArrayList* en donde introduciremos:

1. Objetos de tipo *Punto* que representan puntos interesantes para la presentación de *ToolTips*.
2. Objetos de tipo *Lineas* que serán dibujadas entre emisor y receptor.
3. Objetos de tipo *Texto* que representan tanto el texto en las líneas que representan comunicación como en los extremos.

Cada vez que se realiza un *repaint* automático por parte del sistema se llama a la función **paint()** heredada e implementada por nuestro componente. La primera vez que se llame a dicho método se adquirirán los datos relativos a los parámetros de tamaño y aspecto, calculándose además un parámetro realmente importante, el ángulo, que se calculará en función de la separación entre emisor-receptor y el tiempo de propagación del canal, ya que cuantos más mensajes simultáneos tengamos, más inclinación necesitaremos dado que más slots temporales deberán estar en el canal.

Nuestra clase debe implementar dos interfaces muy importantes. La primera de ella, necesaria a la hora de poder imprimir a través de impresora, es la interface *Printable*, de la que deberemos implementar su método **print(),** encargado de la impresión.

La impresión de un elemento gráfico no es tan simple como parece, ya que inicialmente se debe copiar el elemento gráfico a imprimir sobre un objeto de tipo *BufferedImage*, acción realmente simple. Sin embargo el problema fundamental reside en los tamaños, dado que mi buffer tendrá un alto y ancho idéntico al del objeto *JCanvas* sobre el que realizamos el dibujo, pero diferente a los de impresión (ver **Figura 4.8**).

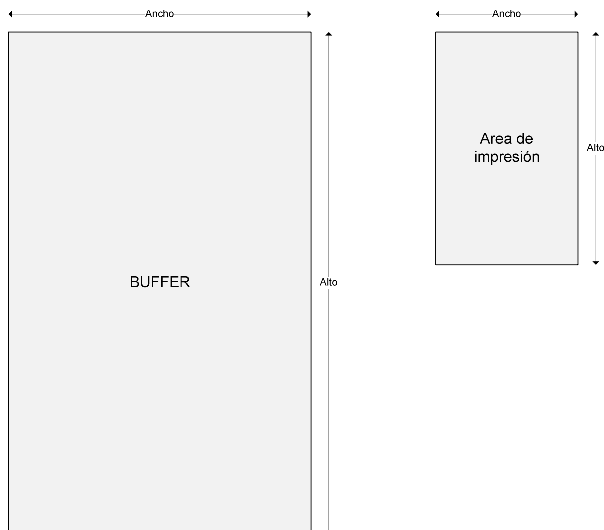


Figura 4.8 Áreas de *buffer* e impresión

Es por esto que al copiar el dibujo alojado en nuestro *JCanvas* sobre el *buffer* de imágenes deberemos realizar un escalado horizontal, buscando que el ancho de nuestro buffer coincida con el ancho de nuestra área de impresión. Para que no se deforme la imagen es necesario escalar en la misma proporción verticalmente.

```
double factorEscala = pf.getImageableWidth()/getWidth();
```

Además es necesario conocer cuantas páginas son equivalentes al dibujo, ya que este método se llama tanta veces como páginas tengamos.

```
int TotalPages = (int)Math.ceil(imagen.getHeight() * factorEscala / pf.getImageableHeight());
```

Después de transformar mediante una translación y una escala en el dibujo ya estamos en disposición de imprimir. Pero a la hora de imprimir debemos generar una subimagen del dibujo quedándonos únicamente con el área de interés, es decir, el área relativa a la página a imprimir.

```
if(pi<(TotalPages - 1))
{
        g2d.drawImage(imagen.getSubimage(0,
        (int)(pi*pf.getImageableHeight()/factorEscala), imagen.getWidth(),
        (int)(pf.getImageableHeight()/factorEscala)),0,0,this);
}
else
{
        g2d.drawImage(imagen.getSubimage(0,
        (int)(pi*pf.getImageableHeight()/factorEscala), imagen.getWidth(),
        (int)(imagen.getHeight()-pi*pf.getImageableHeight()/factorEscala)),0,0,this);
}
```

Figura 4.9 Código para imprimir cada página

Además de permitir la impresión, la aplicación permitirá la creación de una imagen en formato *png* de la simulación en uso. Para ello se dispone de un método que captura todo el contenido gráfico del componente y lo pasa a un *buffer* con la finalidad de guardarlo.

Anteriormente comentamos la problemática con la memoria de intercambio a la hora de trabajar con objetos de tipo *BufferedImage*. Dado que la impresión y la generación de un fichero gráfico necesitan de dicho objeto, será necesaria la inclusión de la directiva en la ejecución del sistema para evitar excepciones a raíz de la falta de memoria de intercambio.

La segunda interface que implementa la clase es la utilizada por el patrón observador, y es la llamada al método **update()** para la actualización del *JCanvas*, es decir, este método se llama para añadir líneas y texto al mismo.

Llegados a este punto es interesante ver el funcionamiento de los *tooltips*. Estos elementos son cajas de texto que aparecen mostrando información al pasar el ratón sobre un determinado punto. Añadir uno sobre un componente es casi inmediato, pero en nuestro caso no deseamos añadir el *tooltip* sobre todo el componente, sino que deberemos diferenciar puntos significativos para poder tratar dichos puntos y mostrar o no los diferentes *tooltips*.

```
public String getToolTipText(MouseEvent event)
{
int x = event.getX();
        int y = event.getY();
        for(int i = 0; i < listaPtos.size(); i++)
{
                if((listaPtos.get(i).getInicial_X()< x && listaPtos.get(i).getFinal_X() >
x) && (listaPtos.get(i).getInicial_Y()< y && listaPtos.get(i).getFinal_Y() > y))
{
                return listaPtos.get(i).getTexto();
        }
}
        return null;
}
```

Figura 4.10 Método que recupera el texto del *tooltip*

Para ello guardamos en una lista de puntos (X, Y) los textos significativos, que recuperaremos mediante el método sobrescrito de la API *tooltip* del *JComponent*. En este método simplemente recuperamos el punto X e Y donde está situado el ratón y lo buscamos en nuestra lista de *Puntos*. Si no está en la lista se devuelve un *String null*, mientras que si está se devuelve el texto en formato *HTML* que nos dará como resultado un *tooltip* con formato.

```
anchoTexto = tamaño.stringWidth(texto);
int rectificacionY = (int)Math.ceil(Math.sin(angulo_texto)* anchoTexto / 2);
int rectificacionX = (int)Math.ceil(Math.cos(angulo_texto)* anchoTexto / 2);

if(direccion == Texto.DIRECCION_EMISOR_RECEPTOR)
{

               pos_X =  (getWidth()- 2 * lineaEmisorX) / 4 - rectificacionX +
lineaEmisorX;
               pos_Y = inicioSlots + (slot) * TAMAÑOSLOTS + incremento_MSG_Y / 4 -
rectificacionY;
               g2.translate(pos_X, pos_Y);
               g2.rotate(angulo_texto);
               g2.drawString(texto, 0, 0);
               g2.rotate(-angulo_texto);
               g2.translate(-pos_X, -pos_Y);
       }
       else if(direccion == Texto.DIRECCION_RECEPTOR_EMISOR)
{
              …
       }
```

Figura 4.11 Código para realizar la corrección de la posición del texto

A la hora de añadir texto en un punto determinado debemos considerar si el mismo está inclinado, ya que si este aparece con cierta inclinación deberemos realizar la corrección adecuada a la hora de pintarlo. Los factores a tener en cuenta en esta corrección son la longitud del *String* a escribir y la inclinación con la que se escribe, de tal manera que deberemos corregir según el código mostrado en la **Figura 4.11**.

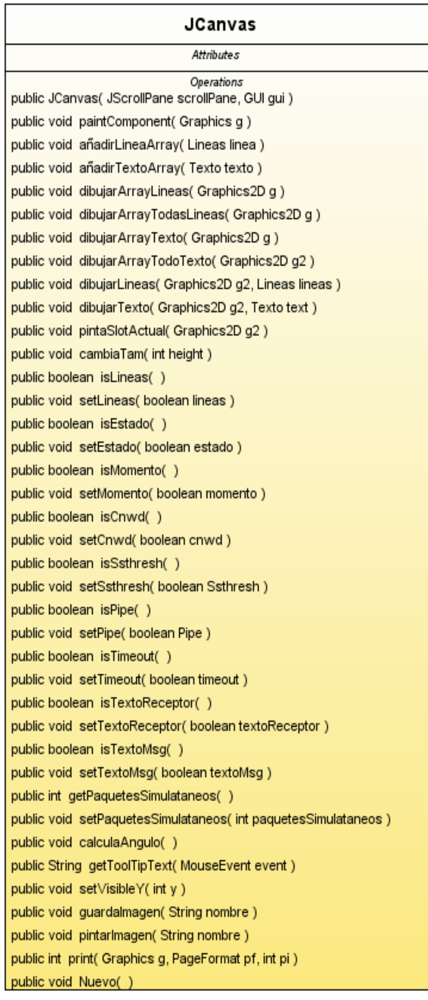


Figura 4.12 Diagrama de clase de la clase *JCanvas.java*

Una vez calculada la corrección trasladáremos al punto calculado donde debemos escribir, rotamos para escribir con el ángulo dado y pintaremos en el punto (0,0), ya que nos encontramos ya en el punto adecuado. Una vez pintado dicho texto deberemos deshacer la translación realizada en el objeto gráfico para poder pintar de manera normal.

4.2.3 *JCanvasEstados.java*

Estamos ante un *JComponent* que realiza la labor de mostrar los estados de nuestro emisor y receptor. En él podemos ver como en la parte superior se definen los estados del emisor, mientras que en la parte inferior se visualizan los del receptor. Dichos estados siguen el siguiente esquema de colores:

- **Gris**: Estado anterior.
- **Rojo**: Estado actual.
- **Verde**: Posibles estados posteriores.

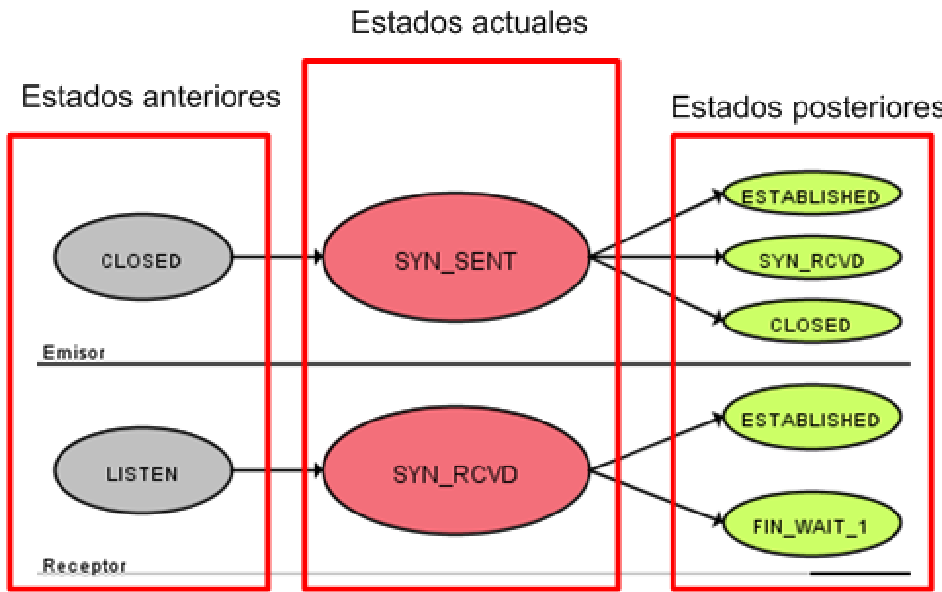


Figura 4.13 Visualización del *JCanvasEstados*

La filosofía de pintado es idéntica a la comentada en el punto anterior, por lo que deberemos añadir el código necesario en su método **paint()**.

La idea fundamental para conocer cuáles son los estados posteriores a los que puede acceder un determinado sujeto, se basa en la toma de decisiones a través de un *switch*.

Este objeto implementa el patrón observador, lo que es utilizado en este caso para conocer cuál es el estado actual en cada actualización del objeto *Estado*, tanto del emisor como del receptor.

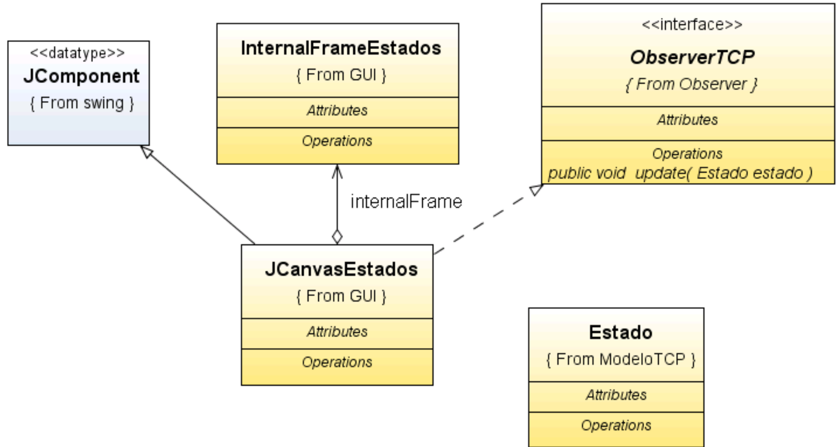


Figura 4.14 Diagrama de dependencias de la clase *JCanvasEstados.java*

Si observamos la **Figura 4.14** podemos ver que además tenemos un objeto de la clase *InternalFrameEstados*, que no es más que un *JInternalFrame* que contiene un objeto *JCanvasEstadosExtendido*, siendo éste una representación completa del diagrama de estados que estudiaremos en el siguiente punto.

El dibujo realizado es mediante la adquisición del objeto gráfico del *JComponent* y pintando sobre él, según el número de estados, las diferentes líneas, óvalos y texto.

JCanvasEstados

Attributes

Operations

public JCanvasEstados(JDesktopPane desktop)
public void inicializar( )
public void paintComponent(Graphics g)
public void dibujaEstados(Graphics2D g2, int x, int y, int width, int height, int ente)
public void setEstadosEmisor(int estadoAnterior, int estadoActual, int estadoPosterior1, int estadoPosterior2, int estadoPosterior3)
public void setEstadosReceptor(int estadoAnterior, int estadoActual, int estadoPosterior1, int estadoPosterior2, int estadoPosterior3)
public void fijaEstadosSgtes(int estadoActual, int tipo)
public void update(Estado estado)
public InternalFrameEstados getInternalFrame( )
public void Nuevo( )

Figura 4.15 Diagrama de clase de la clase *JCanvasEstados.java*

4.2.4 *JCanvasEstadosExtendido.java*

Al realizar un doble *click* sobre el *JCanvasEstados* aparece un *JInternalFrame* que tiene en su interior un objeto de este tipo. Se trata de un objeto que tiene una representación gráfica completa del diagrama de estados TCP.

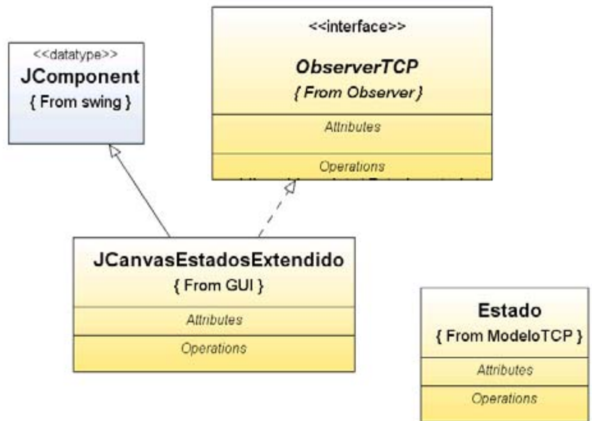


Figura 4.16 Diagrama de dependencias de la clase *JCanvasEstadosExtendido.java*

Además mantiene el estado actual tanto del emisor como del receptor, indicandonoslo mediante el coloreado del correspondiente óvalo. Como podemos observar en la **Figura 4.17** se dispone de los nombres de los estados, así como de las acciones a realizar en una transición, introduciendo además cual es el evento que hace que salte dicha transición.

Para la situación de los óvalos, así como de las flechas, se realizó un mallado obteniendo puntos que son guardados en *arrays* para su posterior utilización a la hora del pintado. Estas posiciones son relativas al tamaño del objeto, por lo que no aparecerían problemas al aumentar o reducir las dimensiones del área de pintado.

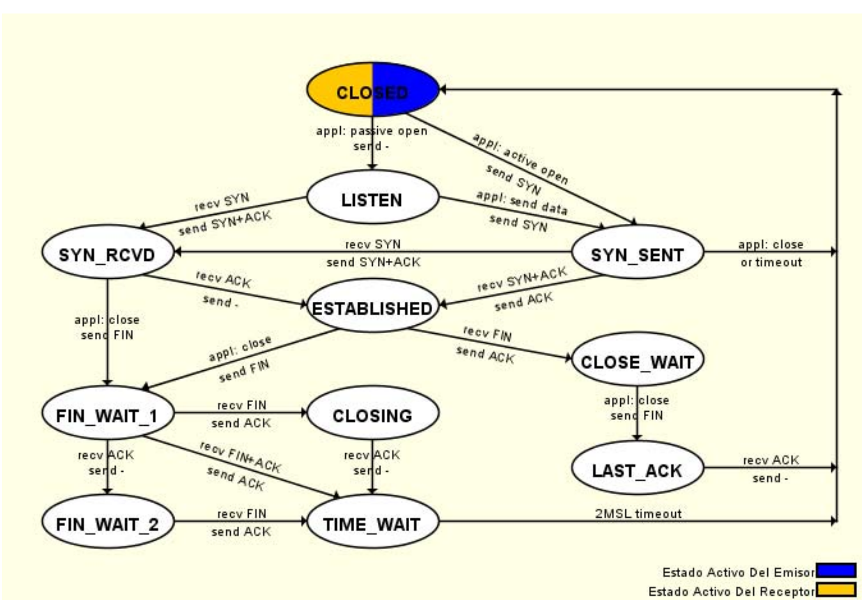


Figura 4.17 Visualización del *JCanvasEstadosExtendido*

De igual manera se tiene un *array* de objetos *String* con los nombres de los estados, que se corresponden en índice con los del modelo TCP. A la hora de realizar la actualización de los estados actuales de emisor y receptor, se le pasa únicamente el índice o entero que determina el estado en su método **update()**.

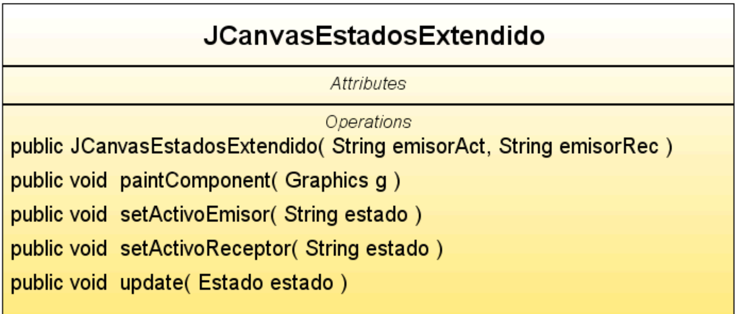


Figura 4.18 Diagrama de clase de la clase *JCanvasEstadosExtendido.java*

4.2.5 *JCanvasVentana.java*

Esta clase muestra la información de la ventana deslizante, tanto los números de secuencia, como los que están enviados a la espera del reconocimiento, como los que pueden ser enviados todavía.

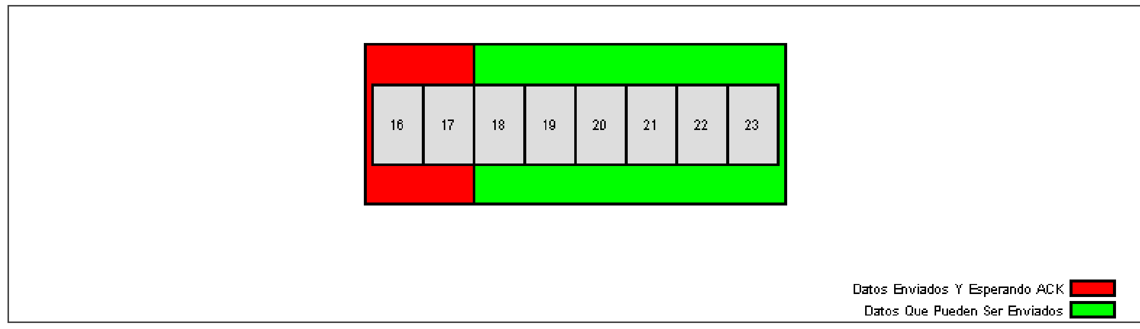


Figura 4.19 Visualización del *JCanvasVentana*

Gracias a este elemento podremos observar el deslizamiento e incremento/decremento de la ventana relativa a la simulación en ejecución. Para la correcta presentación de la ventana es necesario el uso de tres parámetros relevantes, que son:

- *tamaño*: Tamaño total de la ventana deslizante.
- *noReconocidos*: Cantidad de los segmentos que no están reconocidos del tamaño total, siendo menor o igual que el tamaño total de la ventana.
- *inicio*: Número de secuencia del primer segmento de la ventana deslizante.

De igual manera que las clases anteriores estamos ante una clase que implementa la interface *ObserverTCP*, utilizando dicho interface para estar informado sobre el tamaño de la ventana actual. Un apunte importante es que el diagrama de ventana representado es únicamente del emisor, que es el sujeto sobre el que podemos realizar decisiones y que realmente es nuestro ámbito de estudio.

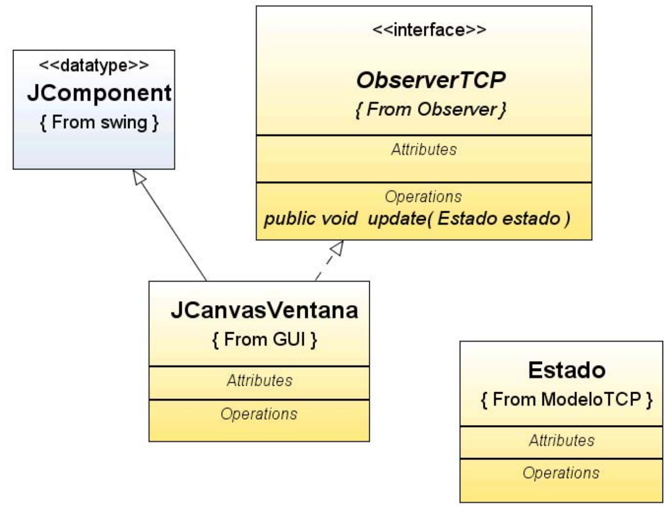


Figura 4.20 Diagrama de dependencias de la clase *JCanvasVentana.java*

Como podemos ver en la **Figura 4.21**, es una clase muy sencilla en cuanto a número de métodos, centrando toda la dificultad en la implementación del método **paint()**. Esta dificultad radica en los diferentes estados en los que puede estar, por lo que al tener que crear un método de dibujo genérico se presentan dificultades, ya que por ejemplo el espacio está limitado a veintidós segmentos, por lo que tenemos que buscar una estrategia a la hora de mostrarlos a todos. Cuando nuestro tamaño sea superior al anteriormente especificado se pintará el principio y fin de los segmentos no reconocidos y de los que se pueden enviar, introduciendo en la mitad puntos suspensivos que representarán los segmentos intermedios.

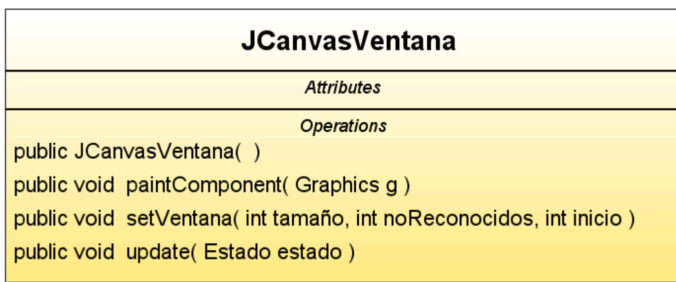


Figura 4.21 Diagrama de clase de la clase *JCanvasVentana.java*

4.2.6 Graficos.java

Estamos ante una clase que tiene como objetivo ayudar a la hora de pintar elementos que en el trascurso de la simulación se van a pintar de forma repetitiva. Comentar que son métodos estáticos, por lo que no es necesaria la generación de un objeto de dicha clase.

En esta clase se han definido cuatro métodos, que pasamos a nombrar y a definir su funcionalidad:

```
public static void drawFlecha(Graphics2D g2, int x1, int y1, int x2, int y2){…}
```

Dicho método tiene como finalidad el pintado de una flecha en un objeto gráfico pasado como argumento empezando en (x1, y1) y terminando en (x2, y2).

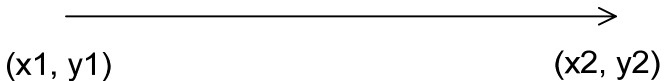


Figura 4.22 Resultado del empleo del método drawFlecha()

```
public static void drawFlechaRellena(Graphics2D g2, int x1, int y1, int x2, int y2){…}
```

Este método es prácticamente idéntico al anterior, teniendo como diferencia la cabeza de la flecha, siendo la misma rellena como podemos ver en la **Figura 4.23**.

(x1, y1) (x2, y2)

Figura 4.23 Resultado del empleo del método drawFlechaRellena()

En este caso dibujamos lo que representaría una pérdida de mensaje, es decir, una flecha pero con cabeza terminada en cruz.

```
public static void drawX(Graphics2D g2, int x1, int y1, int x2, int y2){…}
```

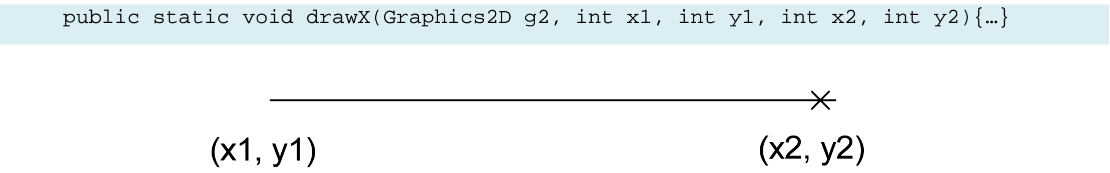


Figura 4.24 Resultado del empleo del método drawX()

En todos estos métodos simplemente debemos introducir los puntos iniciales y finales, calculándose internamente los ángulos necesarios para la correcta visualización de las cabezas de las flechas.

Además disponemos de un método denominado **drawInicio()** que pinta el fondo y las líneas verticales temporales, siendo el tamaño de las mismas dependiente del alto del *JComponent* donde está alojado. Asimismo se realizan separaciones horizontales que definen los slots temporales (ver **Figura 4.25**).

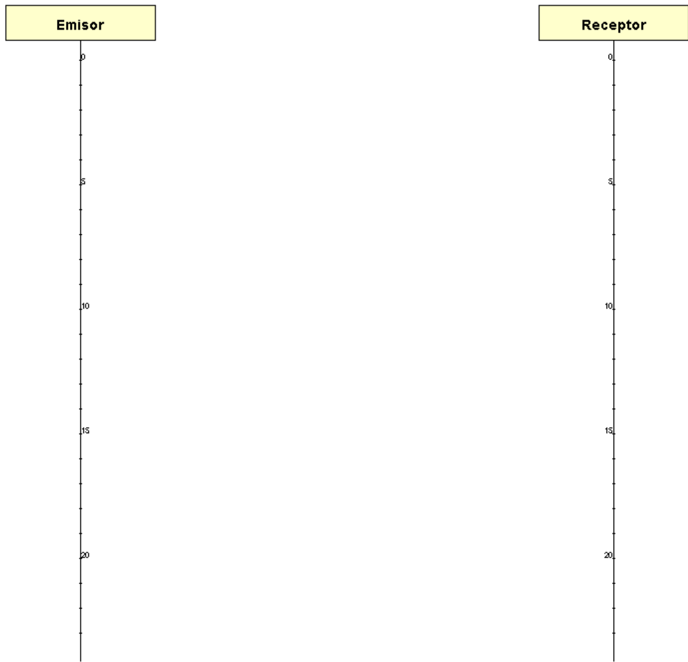


Figura 4.25 Resultado del empleo del método drawInicio()

En la **Figura 4.26** podemos observar el diagrama de clase UML con los métodos comentados anteriormente, y que nos serán de gran utilidad a la hora de implementar el apartado gráfico de nuestro simulador.

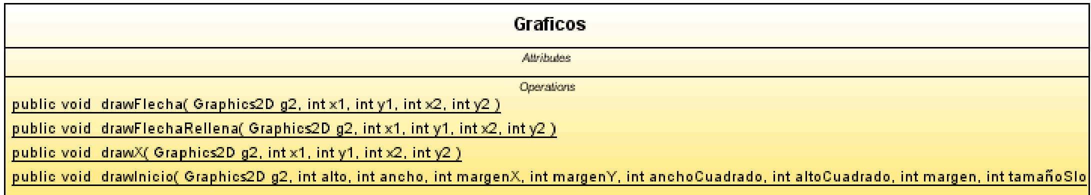


Figura 4.26 Diagrama de clase de la clase *Graficos.java*

4.2.7 *JPanelBotones.java*

Esta clase es la encargada de añadir en la GUI los botones necesarios para realizar las interacciones entre el usuario y el simulador. Como se puede observar en la siguiente figura disponemos de nueve botones para la realización de las mismas.

Figura 4.27 Visualización del *JPanelBotones*

Tal y como podemos ver en la **Figura 4.27** no todas las acciones están habilitadas en todo momentos, sino que solamente estarán activos aquellos botones cuyas acciones están permitidas en dicho slot temporal.

Es en el método **update()** donde se centra la lógica para conocer qué botones estarán activos y cuales no. Este método es resultado de utilizar, al igual que en las clases anteriores, el patrón observador mediante la implementación de la interface *ObserverTCP* (ver **Figura 4.28**). De dicha figura podemos ver que la clase aquí explicada además implementa otra interface llamada *ActionListener*, que es necesaria para dotar a los botones de funcionalidad cuando se realiza una pulsación sobre ellos.

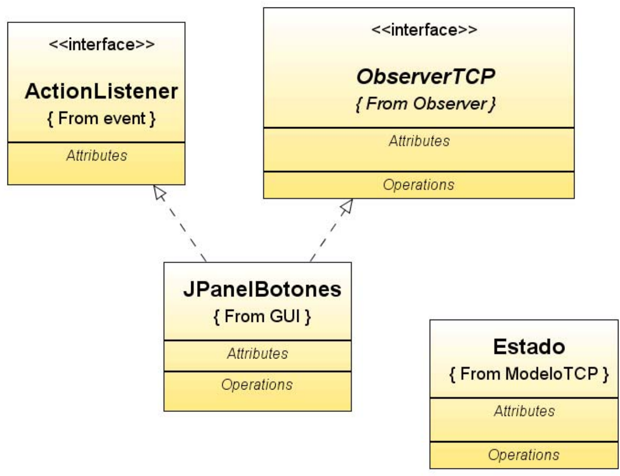


Figura 4.28 Diagrama de dependencias de la clase *JPanelBotones.java*

Esta interface se basa en que los botones añaden a la propia clase *JPanelBotones* como escuchador de sus eventos. Cada vez que se produzca una pulsación en un determinado botón se llamará al método, que debe estar implementado en la clase, llamado ***actionPerformed*()**.

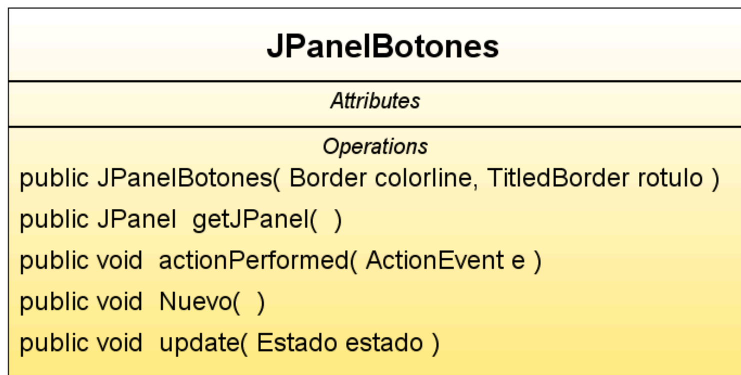
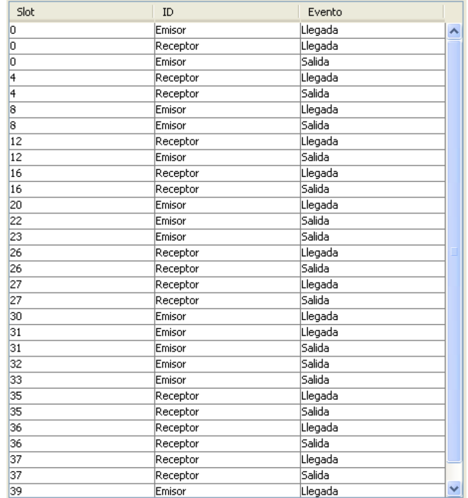


Figura 4.29 Diagrama de clase de la clase *JPanelBotones.java*

4.2.8 *miJTable.java*

Esta clase tiene como objetivo crear una tabla donde se identifican los eventos para tener un registro de todo lo ocurrido en el sistema. Dicha tabla se añade a la GUI y está siempre visible para el usuario. Se pueden identificar tres columnas que definen el slot donde se ha desatado el evento, el sujeto que lo ha producido (emisor o receptor) y el evento consecuente (llegada o salida).



Slot	ID	Evento
0	Emisor	Llegada
0	Receptor	Llegada
0	Emisor	Salida
4	Receptor	Llegada
4	Receptor	Salida
8	Emisor	Llegada
8	Emisor	Salida
12	Receptor	Llegada
12	Emisor	Salida
16	Receptor	Llegada
16	Receptor	Salida
20	Emisor	Llegada
22	Emisor	Salida
23	Emisor	Salida
26	Receptor	Llegada
26	Receptor	Salida
27	Receptor	Llegada
27	Receptor	Salida
30	Emisor	Llegada
31	Emisor	Llegada
31	Emisor	Salida
32	Emisor	Salida
33	Emisor	Salida
35	Receptor	Llegada
35	Receptor	Salida
36	Receptor	Llegada
36	Receptor	Salida
37	Receptor	Llegada
37	Receptor	Salida
39	Emisor	Llegada

Figura 4.30 Visualización de la *miJTable.java*

Para la creación de este objeto es necesaria la intervención de otras tres clases que están relacionadas y de las que hablaremos en los siguientes puntos. Estas clases son:

- *MiTabla.java*
- *CustomRenderer.java*
- *listaObjetos.java*

Esta es una clase que, como podemos ver en el diagrama de dependencia, extiende de un objeto de tipo *JTable*, implementado además la interface *ObserverTCP* para la actualización de la tabla.

Cabe destacar que es necesario clonar el objeto *Estado,* que es un objeto que nos describe el estado de cada sujeto de comunicaciones, utilizado para mostrar el mismo en la tabla para guardarlo, ya que es un objeto único compartido con el *modeloTCP* que se sobrescribe en cada slot temporal.

Crear un modelo de tabla personalizado es tan sencillo como usar un objeto *DefaultTableModel*, que requiere muy poca codificación adicional. Podemos implementar un modelo de tabla implementando un método que devuelva el número de entradas del modelo, que recupere un elemento en una posición específica de ese modelo, etc. Por ejemplo, el modelo *JTable* puede ser implementado desde *AbstractTableModel* mediante la implementación de los métodos que se quieran redefinir.

La clase *JTable* tiene asociada un objeto *DefaultTableModel* que internamente usa un vector para almacenar datos. Los datos de cada fila son almacenados en un objeto *Vector* para su representación en la tabla, teniendo dicho objeto distintos constructores que nos permiten crear entradas de diferente manera.

Así que al constructor de esta clase le pasamos como parámetro un objeto de tipo *MiTabla* que es un objeto de tipo *AbstractTableModel*, que asociaremos al propio objeto *miJTable* (ver **Figura 4.31**).

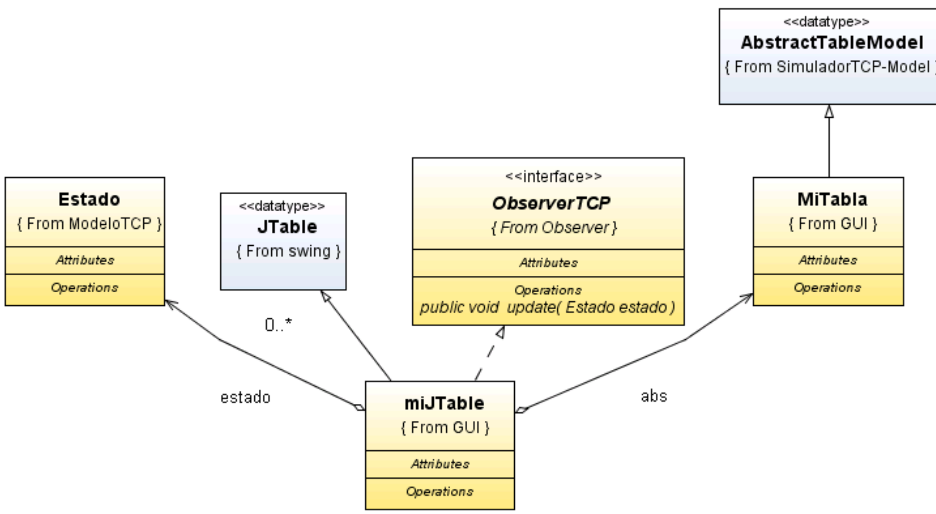


Figura 4.31 Diagrama de dependencias de la clase *miJTable.java*

Esta clase posee un método que es utilizado para obtener un objeto *Estado* de una fila determinada, cuya finalidad será mostrar en una ventana interna el estado completo del emisor o receptor. Además debemos dotar a la tabla de una funcionalidad para poder eliminar todos sus elementos, dada por su método **Nuevo()**.

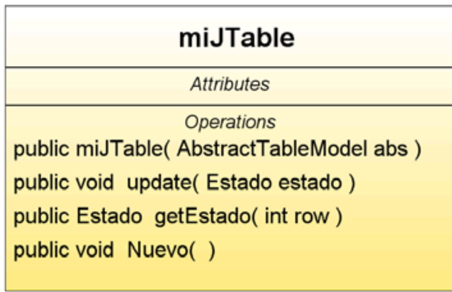


Figura 4.32 Diagrama de clase de la clase *miJTable.java*

4.2.9 *MiTabla.java*

Como hemos comentado en el punto anterior esta clase se define para dotar de la funcionalidad deseada a la tabla implementada. Como podemos ver en el diagrama de dependencias se tiene una clase llamada *listaObjetos*, que no es más que un *array* de objetos. Esta lista define los tres elementos comentados anteriormente:

- Slot.
- Id.
- Evento.

Al constructor le debemos pasar un *array* de *String* con los nombre de la cabecera de la tabla. Podemos crear una tabla editable añadiendo el método de verificación **isCellEditable()**, que es usado por el editor de celda por defecto.

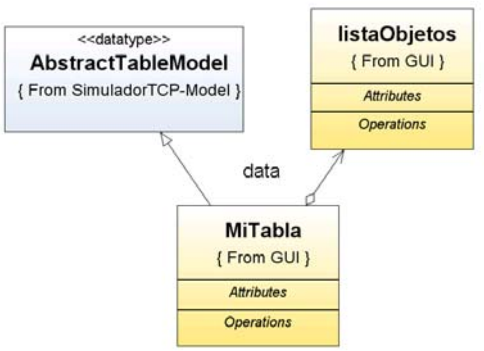


Figura 4.33 Diagrama de dependencias de la clase *MiTabla.java*

Además dotamos a la tabla de métodos para devolver el número de filas, columnas, el nombre de las mismas, y un valor de una fila y columna dada. Además se puede fijar el valor de una columna y fila determinada, añadir y borrar filas, y eliminar todos los elementos de la tabla.

MiTabla

Attributes

Operations

```
public MiTabla( String cabecera[0..*] )
public int  getColumnCount(  )
public int  getRowCount(  )
public String  getColumnName( int col )
public Object  getValueAt( int row, int col )
public boolean  isCellEditable( int row, int col )
public void  setValueAt( Object value, int row, int col )
public void  addRow( Object vector[0..*] )
public void  delRow( int row )
public void  clear(  )
public void  setEditable( boolean valor )
```

Figura 4.34 Diagrama de clases de la clase *MiTabla.java*

4.2.10 *CustomRenderer.java*

Un *CustomRenderer* es un objeto *CellRenderer* determinado, que nos permite dotar a cada fila de un determinado formato. En nuestro caso sólo deseamos modificar el color cuando está seleccionado, seleccionando un color azul para el estado seleccionado y blanco cuando no está seleccionado.

Para ello debemos extender de la clase *DefaultTableCellRenderer* y redefinir el siguiente método:

```
getTableCellRendererComponent(JTable table,Object value, boolean isSelected, boolean
                              hasFocus, int row, int column)
```

Después de realizar lo comentado al método el uso de esta clase sería como mostramos:

```
        String[] cabecera = {"Slot", "ID", "Evento"};
        MiTabla tabla = new MiTabla(cabecera);
        miJTable  table = new miJTable(tabla);
        CustomRenderer miRender = new CustomRenderer();
        table.setDefaultRenderer(Integer.class, miRender);
        table.setDefaultRenderer(String.class, miRender);
```

Como podemos ver hemos utilizado las tres clases explicadas para la generación de la tabla con la funcionalidad que nosotros le deseamos dotar.

4.2.11 *listaObjetos.java*

Ya hemos hablado de esta clase, simplemente es un vector de *array* de objetos que nos sirve para introducir los datos de la tabla. Como podemos ver en la **Figura 4.35** se han desarrollado los métodos necesarios para añadir y eliminar objetos, así como para obtener y definir los mismos.

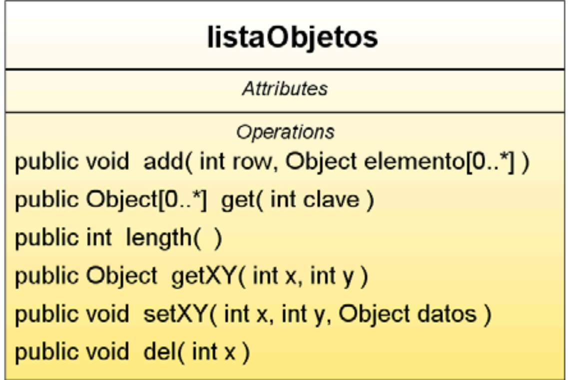


Figura 4.35 Diagrama de clase de la clase *listaObjetos*

4.2.12 *InternalFrameEstados.java*

Comenzamos con los objetos de tipo *JInternalFrame*, que son utilizamos como *frames* internos, es decir, ventanas dentro de la GUI dependientes a la misma. En este caso se utiliza para introducir en ella el *JCanvasEstadosExtendido*.

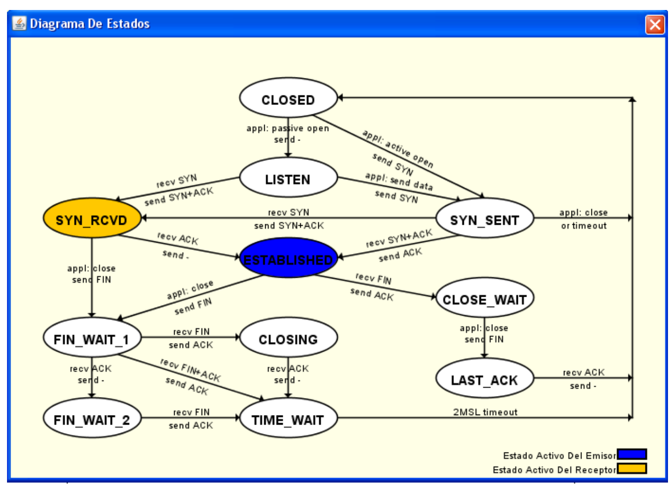


Figura 4.36 Visualización del *InternalFrameEstados*

Para la creación del objeto simplemente le debemos decir la localización y tamaño del mismo, especificándole además que se oculte cuando se presione el botón de cerrar. Esto lo hacemos para no estar realizando nuevas instancias al objeto, ya que estará siempre creado pero no visible.

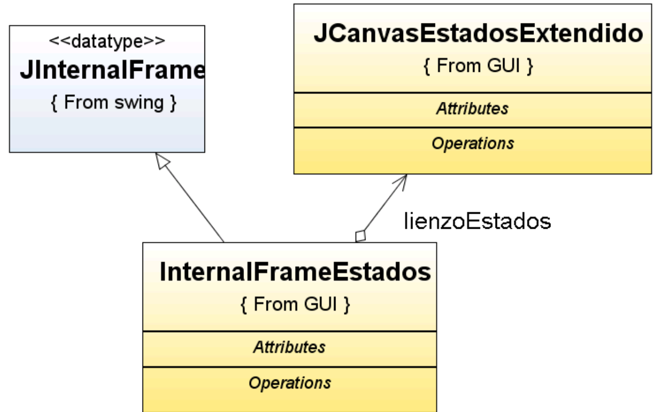


Figura 4.37 Diagrama de dependencias de la clase *InternalFrameEstados.java*

4.2.13 *InternalFrameLog.java*

En este caso tenemos un objeto *JInternalFrame* sobre el que se encuentra un *JPanel* que tiene en su interior un *JTextArea*. Dicho *JTextArea* es donde escribimos los datos de estado de cada evento guardado en el registro del sistema.

Figura 4.38 Visualización de *InternalFrameLog*

Cuando se realiza un doble *click* sobre una línea del registro se nos abre el *JInternalFrame* que nos mostrará toda la información existente sobre ese evento (ver **Figura 4.38**).

4.2.14 *InternalFrameNuevo.java*

Esta es la clase encargada de realizar una nueva simulación, para ello se generará un nuevo *JInternalFrame* en donde podemos ver las diferentes características con las que puede trabajar nuestro modelo.

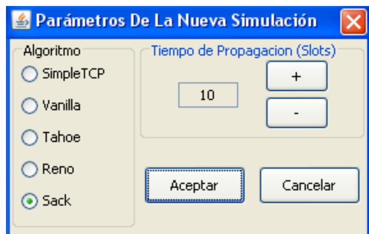


Figura 4.39 Visualización de *InternalFrameNuevo*

Como podemos ver en la **Figura 4.39** tenemos distintos *JButtons*, *JRadioButton* y *JTextField*. Podemos seleccionar los distintos algoritmos TCP, así como los slots que podemos coexistir simultáneamente en el canal.

Según el diagrama de dependencias de la **Figura 4.40**, el objeto *InternalFrameNuevo* se comunica con el controlador, es decir, con el objeto *ModeloTCP*, guardando además esa configuración mediante el uso del objeto *Configuracion*.

Además notifica la creación de una nueva simulación a todos los elementos de la GUI mediante el uso del método **Nuevo()** de cada una de las clases gráficas.

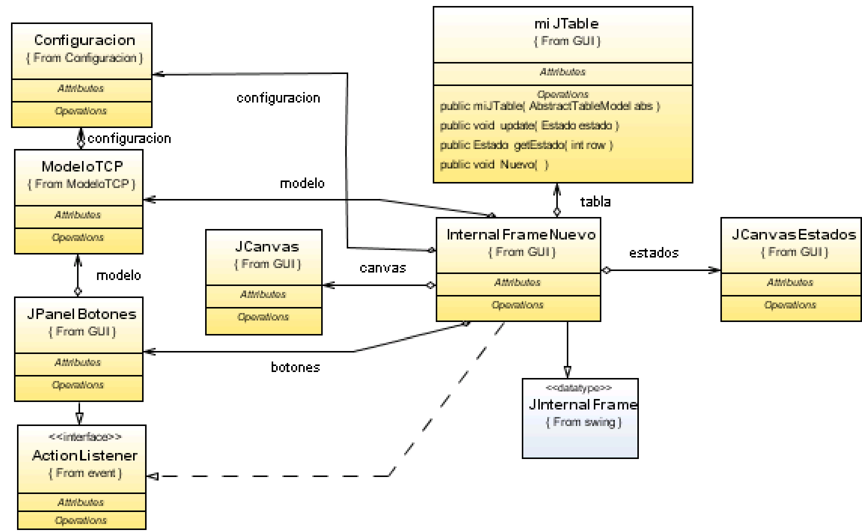


Figura 4.40 Diagrama de dependencias de la clase *InternalFrameNuevo.java*

4.2.15 *InternalFrameOpciones.java*

Es en este *JInternalFrame* donde se pueden seleccionar las opciones de visualización de la simulación. Todas son relativas a lo que se puede ver en el *JCanvas*, como líneas, texto en el receptor, etc.

Figura 4.41 Visualización de *InternalFrameOpciones.java*

El funcionamiento es simple, este *JInternalFrame* tiene la referencia del *JCanvas*, cuando se pulsa el botón *Aceptar* se pasa el estado de todos los *checkbox* al *JCanvas* mediante el uso de sus diferentes métodos de establecimiento.

4.2.16 *InternalFrameAyuda.java*

En este *frame* tenemos la ayuda del simulador, donde encontramos todo lo necesario para utilizar y comprender el simulador. En su interior tenemos un *JPanel* que tiene en su interior un objeto de tipo *JEditorPane*, es la base para los componentes de texto formateado de *Swing*.

Ayuda Del Sistema

Ayuda del 'Simulador de Algoritmos de TCP'

1. Introducción.
2. Teoría TCP.
   2.1. Características.
   2.2. TCP proporcionando confiabilidad.
   2.3. La ventana deslizante.
   2.4. TCP → Protocolo de control de transmisión.
   2.5. Aperturas pasivas y activas.
   2.6. Segmentos, flujos y números de secuencia.
   2.7. Tamaño de la ventana y el control de flujo.
   2.8. Algoritmos de transmisión.
      2.8.1. Slow-Start.
      2.8.2. Congestion Avoidance.
      2.8.3. Fast Retransmit y Fast Recovery.
      2.8.4. SACK.
      2.8.5. Implementaciones.
      2.8.6. Conclusiones.
      2.8.7. Medida RTT (Algoritmo de Karn).
      2.8.8. Máquina de estados TCP.
3. Uso y configuración del simulador.
   3.1. Introducción.
   3.2. Partes y funciones del simulador.

Figura 4.42 Visualización del *InternalFrameAyuda*

Este *JEditorPane* lo utilizamos para abrir en él un fichero de formato HTML, donde se encuentra la ayuda que ha sido generada con anterioridad. Y para finalizar a dicho *JEditorPane* le añadimos un *JScrollPane*.

4.2.17 *JScrollPaneListener.java*

Este es un elemento básico a la hora del correcto funcionamiento del elemento de mayor peso en el simulador, el visualizador de intercambio de mensajes, *JCanvas*. En el *JCanvas* tenemos un *JScrollPane* asociado, pero dicho elemento necesita de más funcionalidad a parte de desplazarse a través del elemento.

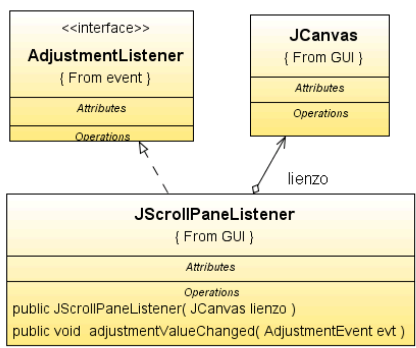


Figura 4.43 Diagrama de dependencias de la clase *JScrollPaneListener.java*

Como podemos ver en nuestra figura debemos implementar la interface *AdjustmentListener*, teniendo que sobrescribir el método **adjustmentValueChanged()** en donde debemos indicar al objeto *JCanvas* el valor de posición de la barra de desplazamiento, procediendo a repintarlo. Gracias a esto es posible pintar sólo la zona que es visible, por lo que mejoramos la carga computacional.

4.2.18 *Lineas.java*

Esta clase representa un objeto de tipo *Linea*, el cual puede ser de tres tipos:

- TIPO_FLECHA_NORMAL: Es una flecha de tipo normal, es decir, con cabeza no rellena.
- TIPO_FLECHA_RELLENA: Es una flecha con cabeza rellena.
- TIPO_FLECHA_X: Es una flecha a medias que termina con una X, representando un mensaje perdido.

Además puede tener dos direcciones, del emisor al receptor y viceversa. Todo esto son tipos como podemos ver en el diagrama de clase, donde además se aprecian los métodos de establecimiento y obtención.

Una flecha queda definida totalmente por tres parámetros, slot, tipo y dirección.

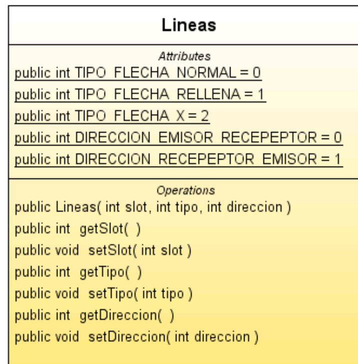


Figura 4.44 Diagrama de clase de la clase *Lineas.java*

4.2.19 *Punto.java*

Para la localización de los *tooltips* es necesario fijar puntos de interés. Un punto de interés, o zona de interés mejor dicho, se fija en un rectángulo según cuatro puntos, dos en el eje X y dos en el eje Y.

No será necesario tener toda la lista de puntos importantes, sólo aquellos que son visibles, por lo que la *Vector* de *Puntos* se limpia en cada repintado y se genera según se pintan.

Punto
Attributes
Operations
public Punto(int x1, int y1, int x2, int y2, String texto)
public String getTexto( )
public int getInicial_X( )
public int getInicial_Y( )
public int getFinal_X( )
public int getFinal_Y( )

Figura 4.45 Diagrama de clase de la clase *Punto.java*

4.2.20 *Texto.java*

Este objeto representa al texto que se pinta tanto en los extremos del receptor y del emisor, como sobre los mensajes intercambiados por los mismos.

Como podemos ver en el diagrama de clase tenemos cinco variables de tipo final, es decir, que son constantes, que representan la posición donde se debe situar el texto, así como la dirección del mismo en caso de ir sobre el mensaje.

Existen diversos constructores de este objeto según las necesidades del mismo, como los distintos métodos de obtención y establecimiento de las variables necesarias que complementan el funcionamiento de la clase.

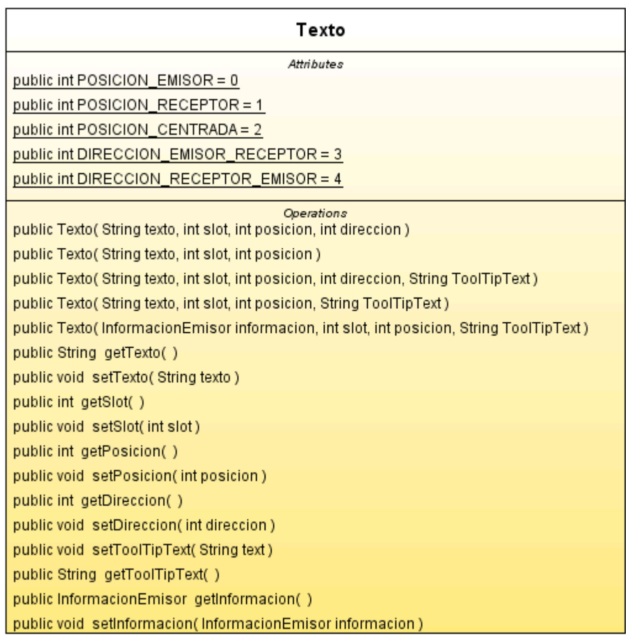


Figura 4.46 Diagrama de clase de la clase *Texto.java*

4.2.21 *FiltroFicherosSIM.java*

Esta clase se utiliza dentro de cuadros de diálogo de selección de ficheros que se generan directamente en la GUI. La utilidad principal es el filtrado de los ficheros que se muestran en estos cuadros de diálogo.

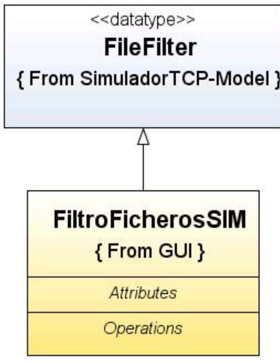


Figura 4.47 Diagrama de dependencias de la clase *FiltroFicherosSIM.java*

Como podemos ver debemos extender de la clase *FileFilter* sobrescribiendo dos métodos:

- *public boolean accept(File f)*
- *public String getDescription()*

En esta clase se hace por tanto un filtrado de ficheros con extensión '*.sim*', que modela ficheros de simulación propia de nuestra aplicación.

4.2.22 *FiltroFicherosPNG.java*

Clase prácticamente idéntica que la anterior, con la única diferencia en que los ficheros filtrados no son de tipo '*.sim*', sino que son ficheros gráficos con extensión '*.png*'.

4.3 Paquete *ModeloTCP*

Hasta ahora hemos mostrado todo lo relacionado con la vista de nuestro simulador y tal y como comentamos en la introducción del capítulo tenemos otro modulo aislado que se correspondería con el modelo. Es en este paquete donde tenemos dicho modelo que ejemplariza una conexión TCP.

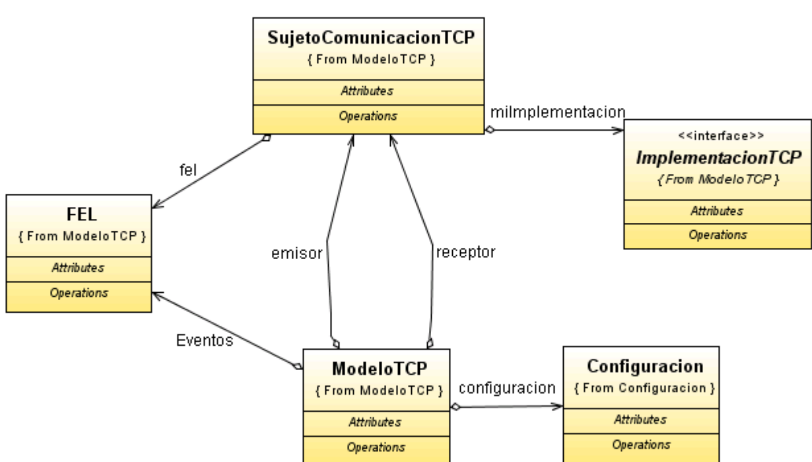


Figura 4.48 Diagrama de dependencias del *ModeloTCP*

Como se puede ver en la **Figura 4.48**, partiendo del objeto *ModeloTCP*, objeto controlador de la simulación, podemos comprobar que éste tiene un objeto de tipo *Configuración*, el cual le fija el tipo de implementación TCP y el tiempo de propagación del canal. Además tiene una cola de eventos discretos donde se introducirán tanto los mensajes provenientes de nuestro simulador a través de la GUI, como los eventos producidos internamente por el intercambio de segmentos TCP, y dos objetos de tipo *SujetoComunicacionTCP* que referencian al emisor y al receptor. A su vez el *SujetoComunicacionTCP* tiene un objeto *ImplementacionTCP*, que como veremos no es más que una interface que define los métodos y parámetros necesarios en todo protocolo TCP.

En la **Figura 4.49** tenemos la dependencia entre los distintos algoritmos que podemos utilizar en el simulador (*SimpleTCP*, *Tahoe*, *Reno*, *Vanilla*, *SACK*). El esquema seguido es el siguiente: *SimpleTCP* es el algoritmo básico, el cual implementa la interface que deben seguir todos los algoritmos de TCP, a su vez los otros cuatro algoritmos extienden del *SimpleTCP*, rescribiendo simplemente aquellos métodos que sean necesarios y añadiendo otros que complementen a los anteriores.

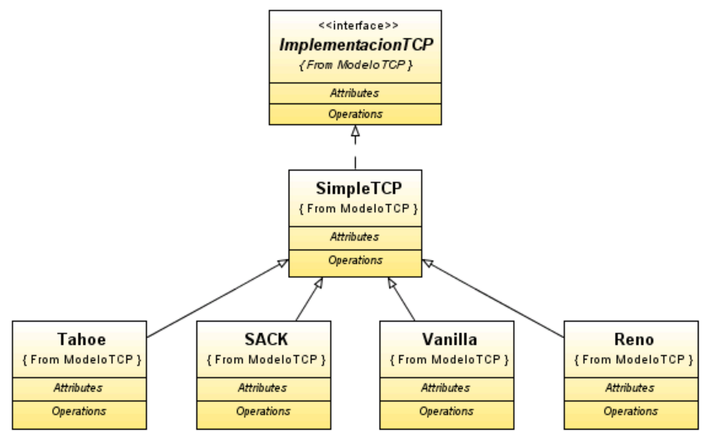


Figura 4.49 Diagrama de dependencias de los algoritmos TCP

Si especificamos ahora la cola de eventos discretos, que partir de ahora llamaremos *FEL*, podemos ver que está formada por objetos de tipo *Evento*, pudiendo estar compuesto este objeto *Evento* otros dos objetos, *SegmentoTCP* o *MesajeAplicacion*.

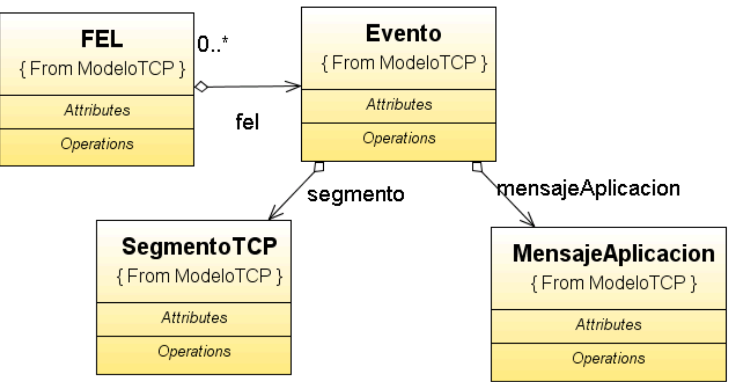


Figura 4.50 Diagrama de dependencias de la *FEL*

Después de realizar una primera aproximación al paquete de forma generalista para darle al lector una visión de conjunto, pasamos a explicar cada una de las clases que forman el modelo. Comenzaremos introduciendo cada uno de los elementos de la **Fig. 4.50**.

4.3.1 *ImplementacionTCP.java*

Estamos ante la interface, que no es más que una colección de declaraciones de métodos y constantes, sujeta a todos los tipos de protocolo TCP, es decir, que deben cumplir todos los algoritmos implementados en el modelo. Todos métodos de esta interface deben ser implementados por cada una de los algoritmos y todas las contantes son visibles a los mismos.

El uso de interfaces nos aporta más que una simple herencia, ya que si tenemos distintas clases que implementan cierta interface podemos tener un sujeto genérico que mediante la implementación de dicha interface pueda, mediante la propiedad del polimorfismo, decidir a qué método debe llamar dada la ligadura dinámica existente en tiempo de ejecución.

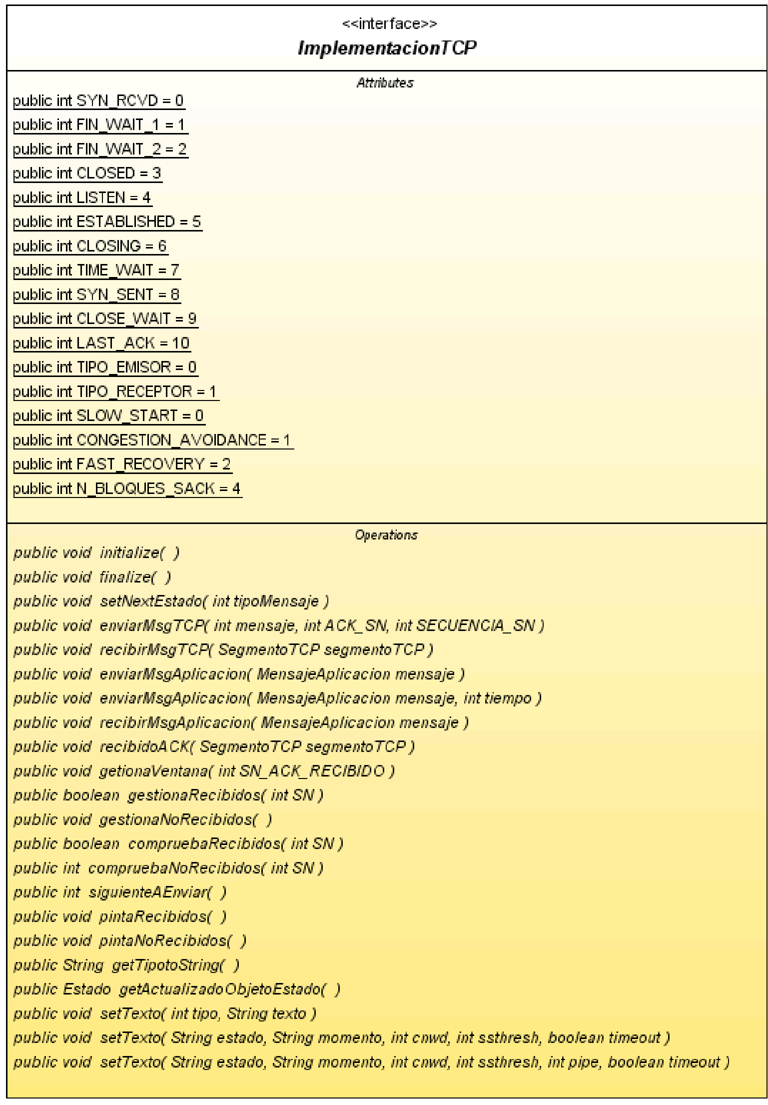


Figura 4.51 Diagrama de clase de la interface *ImplementacionTCP.java*

En nuestro caso la clase que implementa directamente la interface en este punto descrita es *SimpleTCP*, pero los demás algoritmos extienden de esta última, por lo que también la implementan aunque sea indirectamente.

En la **Figura 4.51** mostramos el diagrama de clase. Si nos fijamos en las constantes tenemos aquellas que nos indican el estado en el que está el algoritmo (*SYN_RCVD*, *FIN_WAIT_1*, *FIN_WAIT_2*,…, *LAST_ACK*), también podemos identificar aquella que nos indica la naturaleza del sujeto (emisor o receptor), aquellas que nos revelan el momento del mismo (S*low Start*, *Congestion Avoidance* o *Fast Recovery*) y por último el número máximo de bloques SACK.

Vamos a especificar qué métodos existen en esta interface y que como mínimo debe implementar cada algoritmo TCP. Para comenzar tenemos los métodos **initialize()** y **finalize()**, que como su propio nombre indican son los utilizados cuando se crea una nueva simulación y cuando se termina la misma.

Para pasar de un estado a otro se debe llamar, siempre que se reciba un mensaje ya sea de aplicación o un segmento TCP, a un método que escogerá según el estado actual y el mensaje recibido, el estado al que debe pasar. Este método es **setNextEstado()**.

Pasamos ahora a ver los métodos para el envío y recepción de mensajes de aplicación, un segmento TCP. Como se puede ver en el diagrama de clase existen dos métodos para enviar un mensaje de aplicación, uno al que sólo se le pasa el mensaje que se desea enviar y otro al que también se le pasa un parámetro llamado tiempo. El método sin parámetro tiempo introduce el dato en la FEL para que se envíe en el siguiente slot. En cambio, cuando se quiere programar un mensaje con unos ciertos slots de retardo se debe realizar con este método.

Un método de suma importancia es aquella que gestiona la ventana deslizante. A partir de las secuencias recibidas, el tamaño de la ventana, etc, es posible calcular el tamaño de la ventana, el número de enviados no reconocidos, y los que se pueden enviar todavía.

Además tenemos cuatro métodos que gestionan y comprueban los segmentos que tenemos y aquellos que nos faltan, son **gestionaRecibidos()**, **gestionaNoRecibidos()**, **compruebaRecibidos()** y **compruebaNoRecibidos()**.

Tenemos dos métodos de depuración para ver las secuencias recibidas y las que están perdidas, son **pintaRecibidos()** y **pintaNoRecibidos()**.

Un método muy importante es aquel que nos devuelve el objeto *Estado* actualizado, es decir, nos dice como se encuentran las variables importantes para que sean representadas en el simulador mediante el uso del patrón observador pasando como argumento el propio objeto *Estado*.

Por último tenemos tres métodos sobrecargados para fijar el texto tanto en el emisor, como en el receptor.

4.3.2 *SimpleTCP.java*

Estamos ante la clase que implementa directamente la interface comentada en el punto anterior. Sería la implementación más básica de TCP, donde únicamente implementamos el mecanismo S*low Start*, es decir, que incrementamos la ventana en uno por cada segmento que se reconozca, el único mecanismo que tenemos para efectuar una retransmisión es que salte un *timeout*.

Aquí implementamos todos los métodos especificados en la interface. Cuando se crea un nuevo objeto de este tipo, y de las otras implementaciones, se genera aleatoriamente el número de secuencia, que será utilizada a la hora de enviar los segmentos TCP.

Cuando deseamos enviar un segmento TCP debemos añadirlo en la cola FEL dependiendo si es el emisor, o el receptor, por lo que debemos generar un segmento TCP y comprobar si el sujeto es emisor o receptor. De igual manera será el proceso cuando se quiere enviar un mensaje de la aplicación, pero simplemente el cambio será que no se genera un objeto *segmentoTCP*, sino un *mensajeAplicacion*.

Quizás el método con mayor peso sea **recibirMsgTCP()**, el cual hace funcionar el algoritmo cuando se recibe un segmento. En nuestro simulador el emisor recibe únicamente ACKs, mientras que el receptor recibe los datos. Estos son los dos caminos que se pueden ver en el siguiente diagrama de flujo.

El método que se llama cuando se recibe un ACK, **recibidoACK()**, comprueba la secuencia recibida para ver si es la mayor recibida y gestiona posteriormente la ventana deslizante, actualizando el objeto *Estado*.

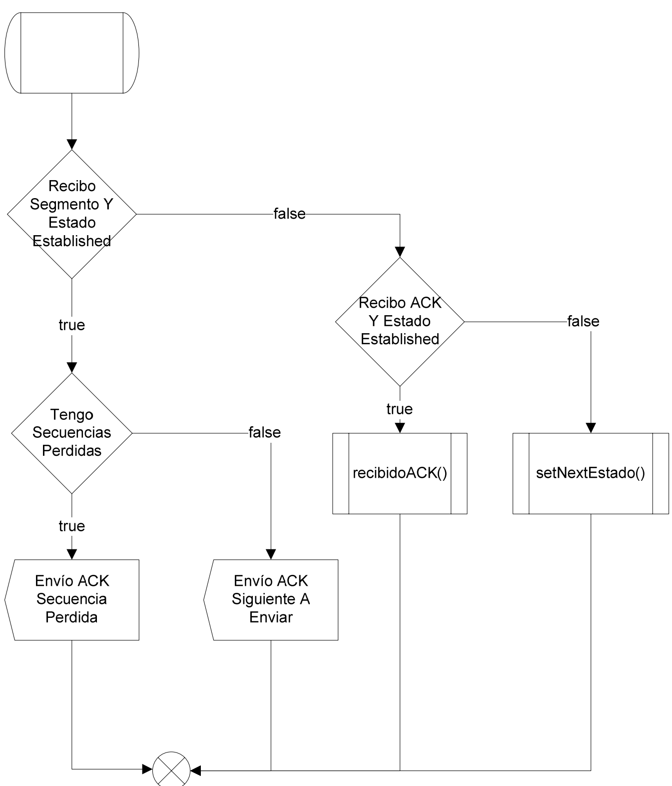


Figura 4.52 Diagrama de flujo del método recibirMsgTCP()

4.3.3 *Vanilla.java*

A partir de este punto no implementamos directamente la interfaz, sino como hemos visto en la **Fig 4.49**, extendemos de la clase *SimpleTCP*. Con esto conseguimos seguir la interface, pero sólo deberemos modificar aquellos métodos en donde cambie su comportamiento, teniendo todos los demás por herencia de la clase padre.

Las únicas modificaciones de código con respecto al *SimpleTCP* son dos métodos, **recibirMsgAplicacion()** y **gestionaVentana()**. Como podemos ver ha sido realmente simple hacer crecer el algoritmo reutilizando el código de la clase anterior.

4.3.4 *Tahoe.java*

Quizás sería posible pensar en que deberíamos reutilizar el código del *Vanilla*, ya que *Tahoe* simplemente añade *Fast Retransmit* a lo ya conseguido con *Vanilla*, pero nos pareció más interesante trabajar sobre el código básico del *SimpleTCP*, para tener un código más limpio y claro.

EL único cambio con respecto a la versión simple es en la gestión de la ventana, al recibir mensajes de aplicación, y solamente en el caso de que sea un *timeout*, ya que debemos guardar el valor de la ventana actual, tal y como debemos hacer en le *Vanilla*, y al recibir segmentos TCP, ya que al tercer segmento repetido, entiéndase que nos referimos a segmentos de reconocimiento (ACKs), debemos enviar el segmento pedido.

4.3.5 *Reno.java*

Esta sería la versión simple más complicada, ya que sigue añadiendo cosas en la misma dirección que las versiones anteriores. En este caso tenemos *Fast Recovery*, que mediante un cálculo de segmentos en el canal nos permite introducir más segmentos y recuperarnos más rápido.

De igual manera que en los otros algoritmos sólo debemos modificar los mismos métodos que hemos comentado. Si no entramos en profundidad en el funcionamiento de los algoritmos es debido a que simplemente es implementar los algoritmos explicados en el capítulo de teoría, por lo que no merece la pena, además el esquema de colaboración y dependencias ha sido expuesto en la introducción de este punto.

4.3.6 *SACK.java*

Estamos ante la versión más completa y con mejores resultados a la hora de recuperarse de errores. De igual manera que en la teoría es la versión más compleja, en la práctica también es así, debiéndose implementar nuevos métodos que nos ayuden a realizar el algoritmo y redefinir de forma más compleja aquellos que ya teníamos.

Como mención comentar que ha sido necesario redefinir:

- **enviarMsgTCP()**
- **getionaVentana()**
- **recibidoACK()**
- **recibirMsgAplicacion()**
- **recibirMsgTCP()**

En cuanto a los nuevos métodos tenemos:

- **generaSACKmsg()**
- **generaSegmentoSACK()**

4.3.7 *SujetoComunicacionTCP.java*

Estamos ante la clase que encapsula a los dos sujetos que componen la comunicación. En ella se generarán los objetos de comunicación dependiendo del algoritmo escogido teniendo un objeto genérico de *ImplementacionTCP*. Tal y como se

puede ver tenemos constantes que definen tanto el algoritmo a escoger, como el tipo de sujeto que será, emisor o receptor.

Es en esta clase donde tenemos los métodos para añadir los objetos que observarán los cambios producidos en los dos sujetos de comunicaciones, así como los métodos para notificar los cambios y eliminar los objetos que observan.

```
SujetoComunicacionTCP
Attributes
public int ALGORITMO_SIMPLE_TCP = 0
public int ALGORITMO_TAHOE = 2
public int ALGORITMO_RENO = 3
public int ALGORITMO_SACK = 4
public int TIPO_EMISOR = 0
public int TIPO_RECEPTOR = 1
Operations
public SujetoComunicacionTCP( int algoritmo, int tipo, FEL fel )
public void setTiempo( int tiempo )
public int getTiempo( )
public void añadirEvento( Evento evento )
public void addObserver( ObserverTCP obs )
public void notifyObserver( )
public void removeObserver( ObserverTCP obs )
public ImplementacionTCP getMiImplementacion( )
public void setIncremento( int incremento )
public int getIncremento( )
public String getTipotoString( )
public void setNuevo( int algoritmo, int tipo, FEL fel )
public String getAlgoritmo( )
```

Figura 4.53 Diagrama de clase de la clase *SujetoComunicacionTCP.java*

Como podemos ver tenemos métodos para la obtención y establecimiento del tiempo de simulación actual, es decir, el slot actual de simulación. Además disponemos de un método para cuando se debe definir un par de nuevos sujetos, ya sea porque son distintos algoritmos, o porque se quiere reiniciar la simulación con distintos parámetros.

4.3.8 ModeloTCP.java

Estamos ante lo que sería el controlador del programa. En la clase *ModeloTCP* se crea un objeto cola de eventos discretos (*FEL*), los dos sujetos que intervienen en el sistema (emisor y receptor TCP) que son objetos de tipo *SujetoComunicacionTCP*. Además tiene la referencia al objeto *Configuracion* que utiliza para establecer el valor de los sujetos.

Si observamos el diagrama de clase vemos los métodos de que dispone esta clase, pudiendo ver con claridad que tenemos aquí los métodos que hacen las acciones en nuestro simulador, es decir, enviar un mensaje, perder un mensaje, saltar un *timeout*, iniciar una comunicación, etc.

Un método muy importante en esta clase es la gestión de la FEL, que es donde vamos aumentando el tiempo de simulación y comprobando los eventos que tenemos por ejecutar en dicho tiempo.

Además tenemos tres métodos distintos utilizados para guardar las simulaciones que se han ejecutado y cargar las mismas. La filosofía es simple, yo simplemente guardo las acciones en cuanto botones que pulso, al mismo tiempo guardamos en el fichero la configuración del protocolo. El fichero es comprobado antes de cargarlo, ya que podría tener parámetros no validos por una modificación del fichero de forma manual, ya que es un simple fichero de texto, produciendo fallos en la simulación.

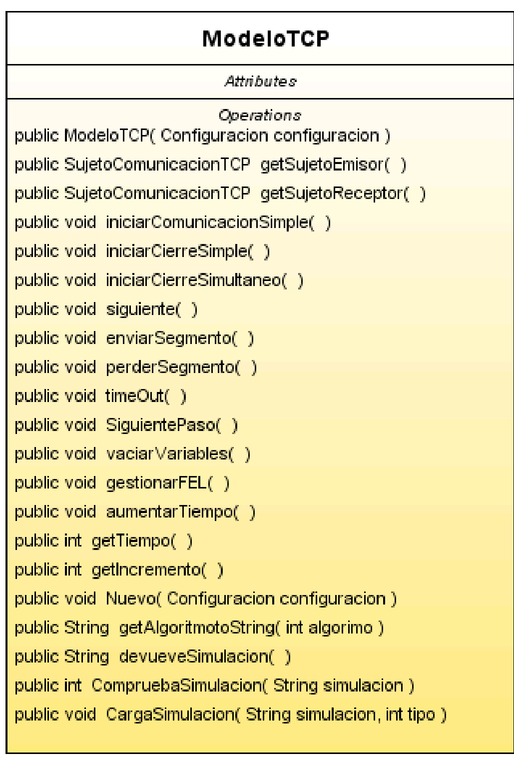


Figura 4.54 Diagrama de clase de la clase *ModeloTCP*

4.3.9 *Estado.java*

Este es el objeto que identifica al estado del emisor y receptor, y que es pasado por referencia en cada llamada al método **update()** usado en el patrón observador. Como podemos ver en el diagrama de clase tenemos métodos de obtención y establecimiento para todas y cada una de las variables importantes para la definición del estado de los sujetos de comunicación.

No es más que una clase sin inteligencia, ya que guardas los valores que le son otorgados para que sean recuperados posteriormente por aquellas clases que necesiten conocer su valor, que en este caso son las clases del paquete *GUI*.

Un método interesante es el **toString()**, que nos devuelve un objeto de tipo *String* con formato HTML, que es usado para la generación de los *tooltips* formateados.

Puede parecer que es una clase demasiado complicada, dada la cantidad de métodos de establecimiento y obtención, pero es necesaria tal cantidad de métodos, ya que existe una gran cantidad de variables. Esto es dado por la gran cantidad de objetos que deben observar el comportamiento del protocolo, que tienen distintas necesidades para su funcionamiento.

Sólo existe una referencia a este objeto, ya que no es necesario crear un objeto distinto para cada instante de simulación, que además repercutiría en la cantidad de memoria que debería usar el programa. Por ello es útil hacer copias de objetos más pequeños para ir guardándolos, obteniendo de esta clase únicamente los parámetros que son importantes para cada sujeto.

Estado

Attributes

Operations

```
public int getEstado( )
public void setEstado( int estado )
public int getInicioVentana( )
public void setInicioVentana( int inicioVentana )
public int getNoReconocidos( )
public void setNoReconocidos( int noReconocidos )
public int getSlot( )
public void setSlot( int slot )
public int getTamañoVentana( )
public void setTamañoVentana( int tamañoVentana )
public int getTipo( )
public void setTipo( int tipo )
public boolean isEvento( )
public void setEvento( boolean evento )
public String getTextoCentrado( )
public void setTextoCentrado( String textoCentrado )
public String getTextoReceptor( )
public void setTextoReceptor( String textoReceptor )
public boolean isLog( )
public void setLog( boolean log )
public String getLlegadaSalida( )
public void setLlegadaSalida( String llegadaSalida )
public boolean isMsg( )
public void setMsg( boolean msg )
public boolean isFinish( )
public void setFinish( boolean finish )
public boolean isVentana( )
public void setVentana( boolean ventana )
public boolean isPerder( )
public void setPerder( boolean perder )
public boolean isEnCierre( )
public void setEnCierre( boolean enCierre )
public String toString( )
public String getToolTipText( )
private String estadoToString( int estado )
public String momentoToString( int momento )
public String getTipotoString( )
public Object clone( )
public boolean isCarga( )
public void setCarga( boolean carga )
public int getMomento( )
public void setMomento( int momento )
public boolean isMomentoBool( )
public void setMomentoBool( boolean momentoBool )
public InformacionEmisor getInformacion( )
public void setInformacion( InformacionEmisor informacion )
public void setSsthresh( int ssthresh )
public void setPipe( int pipe )
public boolean isEmisorBloqueado( )
public void setEmisorBloqueado( boolean emisorBloqueado )
public boolean isMsgDUP( )
public void setMsgDUP( boolean msgDUP )
public void setSACK( Integer sack[0..*] )
public int getPipe( )
public boolean isSACKBool( )
public void setSACKBool( boolean bool )
```

Figura 4.55 Diagrama de clase de la clase *Estado.java*

4.3.10 *Evento.java*

Tal y como se muestra en la **Fig. 4.50** un objeto de la clase Evento puede ser un mensaje de la aplicación que nos da órdenes ejecutadas a través de la interface grafica, objeto *MensajeAplicación*, o un segmento TCP que ha viajado por el canal entre emisor y receptor o viceversa, objeto *SegmentoTCP*.

Evento

Attributes
public int EMISOR_LLEGADA = 0
public int EMISOR_SALIDA = 1
public int RECEPTOR_LLEGADA = 2
public int RECEPTOR_SALIDA = 3
public int EMISOR_PERDER = 4

Operations
public Evento(int tiempo, SegmentoTCP segmento, int tipo)
public Evento(int tiempo, MensajeAplicacion mensajeAplicacion, int tipo)
public int getTiempoEjec( )
public int getTipo( )
public SegmentoTCP getSegmento( )
public MensajeAplicacion getMensajeAplicacion( )
public boolean isTCPEvent( )
public void setTiempo(int tiempo)

Figura 4.56 Diagrama de clase de la clase *Evento.java*

Como podemos ver en el diagrama de clase tenemos cinco tipos de eventos, dos del emisor, ya sean de llegada al mismo o salida, otros dos del mismo tipo en el receptor, y otro que simboliza un mensaje perdido en el canal dirección emisor hacia receptor.

De igual manera comprobamos que la clase nos otorga dos constructores, uno para los segmentos TCP, y otro para los mensajes de la aplicación. Estos métodos, a parte del tipo que son los comentados anteriormente, tiene un parámetro básico, el tiempo, que no es más que el slot temporal en el que deberá ocurrir el evento dado.

El resto de métodos son para la obtención de los objetos que contiene el evento para poder extraer los datos relevantes del mismo.

4.3.11 *FEL.java*

Esta es la clase que ejemplariza a una cola de eventos discretos para su posterior procesado conforma aumenta el tiempo. El funcionamiento es simple, tenemos un objeto *Vector* que es donde vamos añadiendo y eliminando los eventos.

Tenemos métodos para añadir eventos. El evento no se introducirá en la última posición, sino que según el tiempo en el que se debe ejecutar se introducirá en la posición correcta. Con esto tendremos la cola ordenada, por lo que la búsqueda se reducirá a los primeros elementos de la cola, sin tener que recorrerlos todos.

FEL

Attributes

Operations
public FEL( )
public void añadirEvento(Evento ev)
public Evento getEvento( )
public Evento getEvento(int i)
public Evento getAndRemoveEvento( )
public Evento getAndRemoveEvento(int i)
public int size( )
public void pintaFEL( )

Figura 4.57 Diagrama de clase de la clase *FEL.java*

También tenemos métodos para la hora de extraer un evento, el cual eliminará al mismo tiempo el evento extraído.

4.3.12 *InformacionEmisor.java*

Este objeto enmarca todas las propiedades que puede tener el elemento principal a seguir, el emisor. Este objeto se introduce dentro del global visto anteriormente y denominado *Estado*.

Tenemos tres constructores, que son suficientes, según la información que se tiene en ese momento, y que nos otorgará el estado completo del mismo para poder fijar el momento en el que se encuentra.

El resto de métodos son de obtención, los cuales nos devuelven cada uno de los parámetros de información relevantes.

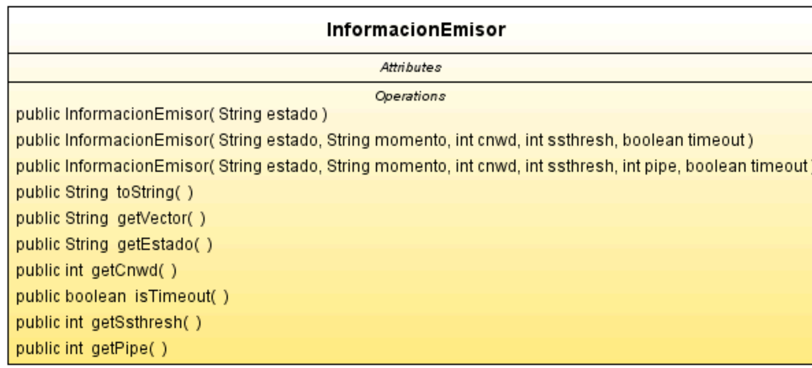


Figura 4.58 Diagrama de clase de la clase *InformacionEmisor.java*

4.3.13 *MensajeAplicacion.java*

Este objeto ejemplariza las acciones que puede hacer un usuario del algoritmo TCP. Tal y como podemos ver en el diagrama de clase tenemos distintas constantes que nos definen el tipo de acciones que se pueden realizar.

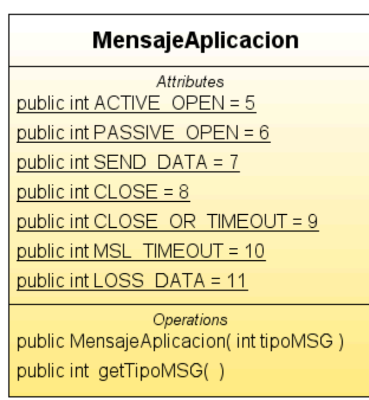


Figura 4.59 Diagrama de clase de la clase *MensajeAplicacion.java*

Tenemos desde las aperturas y cierres de conexión, tanto activa como pasiva, como el *timeout* que salta cuando se cierra una conexión, como la pérdida de un segmento.

En cuanto a los métodos sólo tenemos uno, que es el encargado de devolvernos el tipo del mensaje, y un constructor, que establece lo anterior.

4.3.14 *SegmentoTCP.java*

Clase parecida a la anterior, pero en vez de referenciar a un mensaje por parte de la aplicación lo hace a un segmento TCP. De igual forma tenemos constantes que nos especifican el tipo de mensaje que es, que son todos los tipos que define el protocolo. Como podemos ver tenemos los tipos para el establecimiento de la conexión (SYN), como para terminar la misma (FIN), además de la ya comentada usada para el reconocimiento de segmentos (ACK).

Este objeto *SegmentoTCP* sería el equivalente al mensaje que transportaría el canal hacia y desde el emisor y receptor. Además del tipo de mensaje tenemos otros campos, como el número de secuencia y de reconocimiento, en el caso que sea un ACK. Además de un *Vector* en donde guardamos la información SACK en caso de que se utilice dicho campo.

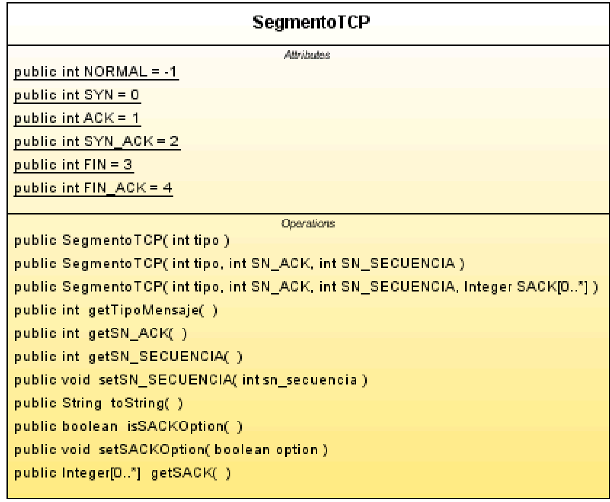


Figura 4.60 Diagrama de clase de la clase SegmentoTCP.java

4.4 Paquete *Observer*

Al principio del capítulo se hizo la introducción al patrón observador, que tal como mostramos en los siguientes dos puntos lo hemos dividido en dos clases.

4.4.1 *ObserverTCP.java*

Como vemos en el diagrama simplemente es una interface con un solo método a implementar denominado **update()**, que se le pasa como parámetro un objeto de *Estado* para la actualización de los observadores.

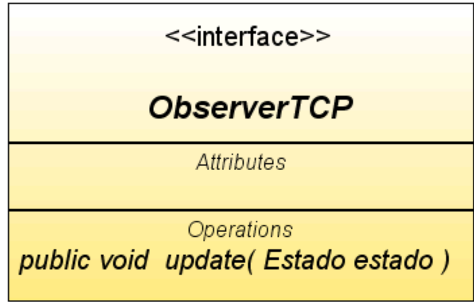


Figura 4.61 Diagrama de clase de la interface *ObserverTCP.java*

4.4.2 *SujetoObservable.java*

Esta interface define tres métodos que deberán rellenar las clases que decidan implementar dicha interface. Estos tres métodos son para añadir, eliminar y notificar a la lista de observadores que tenga definido cada sujeto que decida ser observable a través de esta interfaz.

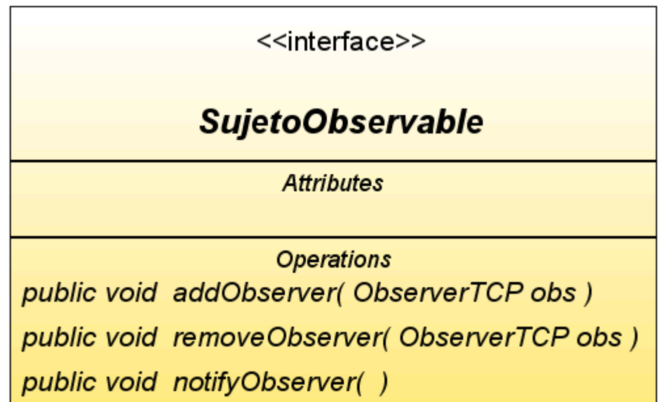


Figura 4.62 Diagrama de clase de la clase *SujetoObservable.java*

4.5 Paquete *Configuracion*

Este paquete tiene una única clase donde se guarda la configuración del simulador.

4.5.1 *Configuracion.java*

Esta clase definirá un objeto que será único en el sistema, y que define la configuración del simulador como su nombre indica, entendiendo por configuración los siguientes parámetros:

- Algoritmo
- Slots simultáneos en el canal entre emisor y receptor.

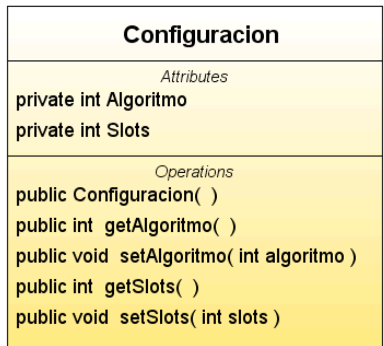


Figura 4.63 Diagrama de clase de la clase *Configuracion.java*

4.6 *Principal.java*

Estamos ante la clase principal, es decir, donde tenemos el método **main()** que crea e inicializa todo el sistema. Para ello y cómo podemos ver en el siguiente diagrama de dependencias, creamos las instancias necesarias, que en este caso son:

- Objeto de la clase *GUI* (interface gráfico).
- Objeto de la clase *ModeloTCP* (modelo del sistema).
- Objeto de la clase *Configuracion*.

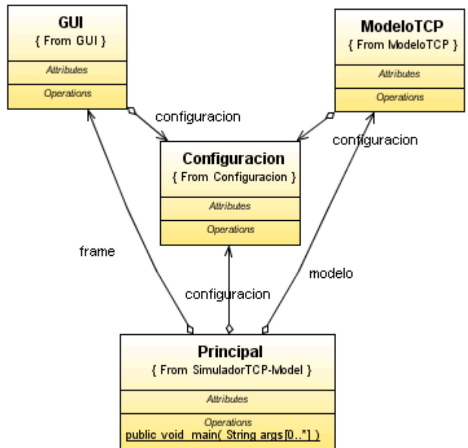


Figura 4.64 Diagrama de dependencias de la clase *Principal.java*

Aparte de crear las instancias a los objetos se define el '*Look and Feel*' del sistema, escogiendo un tema de visualización Windows que ha sido donde se ha generado dicho proyecto.

```
        LookAndFeel lf = UIManager.getLookAndFeel();
        try {
    UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        }
        catch (Exception e) {
            try{
                UIManager.setLookAndFeel(lf);
            }
            catch (Exception e2){}
        }
```

Además comprobamos la resolución de la pantalla, ya que como hemos comprobado el programa está diseñado para una resolución mínima de 1280x1024 pixeles, y en plataformas con resolución menor a la mencionada será imposible ejecutar el mismo.

```
Toolkit tk = Toolkit.getDefaultToolkit();
Dimension tamaño = tk.getScreenSize();
 if(tamaño.getWidth()<1280.0 || tamaño.getHeight()<1024.0)
{
        JFrame basico = new JFrame();
        System.out.println("Configuración No Valida De La Pantalla");
        System.err.println("La Resolución De La Pantalla Es De " +
tamaño.getWidth() + " x " + tamaño.getHeight() );
        System.err.println("La Resolución Deberia Ser De 1280x1024");
        JOptionPane.showMessageDialog(basico, "La Resolución Debe Ser Como
Mínimo 1280x1024",
                "Error",
                JOptionPane.ERROR_MESSAGE);
        System.exit(0);
}
else
{
        System.out.println("Configuración Valida De La Pantalla");
        System.out.println("La Resolución De La Pantalla Es De " +
tamaño.getWidth() + " x " + tamaño.getHeight() );
}
```

5. Uso y configuración del simulador

5.1 Introducción

Estamos ante un simulador de eventos discretos controlable por el usuario con fines docentes, es decir, el fin del sistema es hacer comprender de una forma amena y sencilla al alumno los diferentes algoritmos de TCP, así como el funcionamiento del mismo en general. Es por tanto, un elemento que demarcaríamos dentro de la tele-docencia, donde el alumno debería ser capaz de aprender con el único recurso del simulador, el funcionamiento de TCP.

5.2 Partes y funciones del simulador

Al ejecutar el simulador nos encontramos con cinco áreas gráficas bien diferenciadas, en donde podremos obtener toda la información necesaria del simulador. Además observamos dos tipos de menús en la parte superior de la pantalla, una pertenece al acceso a esta propia ayuda y otra es la de configuración y opciones del sistema. Por tanto, como podemos ver en la **Figura 5.1**, la ventana principal de la aplicación se compone de los siguientes elementos:

- ***Ventana de intercambio de segmentos***: Pantalla principal del sistema donde podremos ver el intercambio de segmentos entre el emisor y el receptor. En los extremos podremos ver la información de cada uno de los sujetos activos de la comunicación, y en la parte central aparecerá la información relativa al mensaje.
- ***Botones de acción***: Es en estos botones donde reside la capacidad del usuario de decidir las acciones del protocolo. Estas acciones son relativas al inicio, transcurso y cierre de la comunicación. Los botones activos variarán según los instantes de la simulación.
- ***Log del sistema***: Registro general donde podremos visualizar lo acontecido en un determinado instante.
- ***Ventana deslizante***: Ventana donde podremos visualizar el estado de la ventana deslizante del emisor en ese mismo instante, tanto el tamaño, como los paquetes enviados y no reconocidos, como los que todavía podemos enviar.
- ***Máquina de estados***: Gráfico donde se muestra el estado anterior, actual y posterior tanto del emisor como del receptor.
- ***Menús de configuración***: En este menú podremos configurar una nueva simulación y aquellos datos relativos a la simulación actual en cuanto a la información mostrada en la ventana principal de intercambio de segmentos. Además podremos cargar y guardar simulaciones, y realizar otros tipos de acciones.
- ***Ayuda***: Ventana con la ayuda del simulador y teoría relevante de TCP.

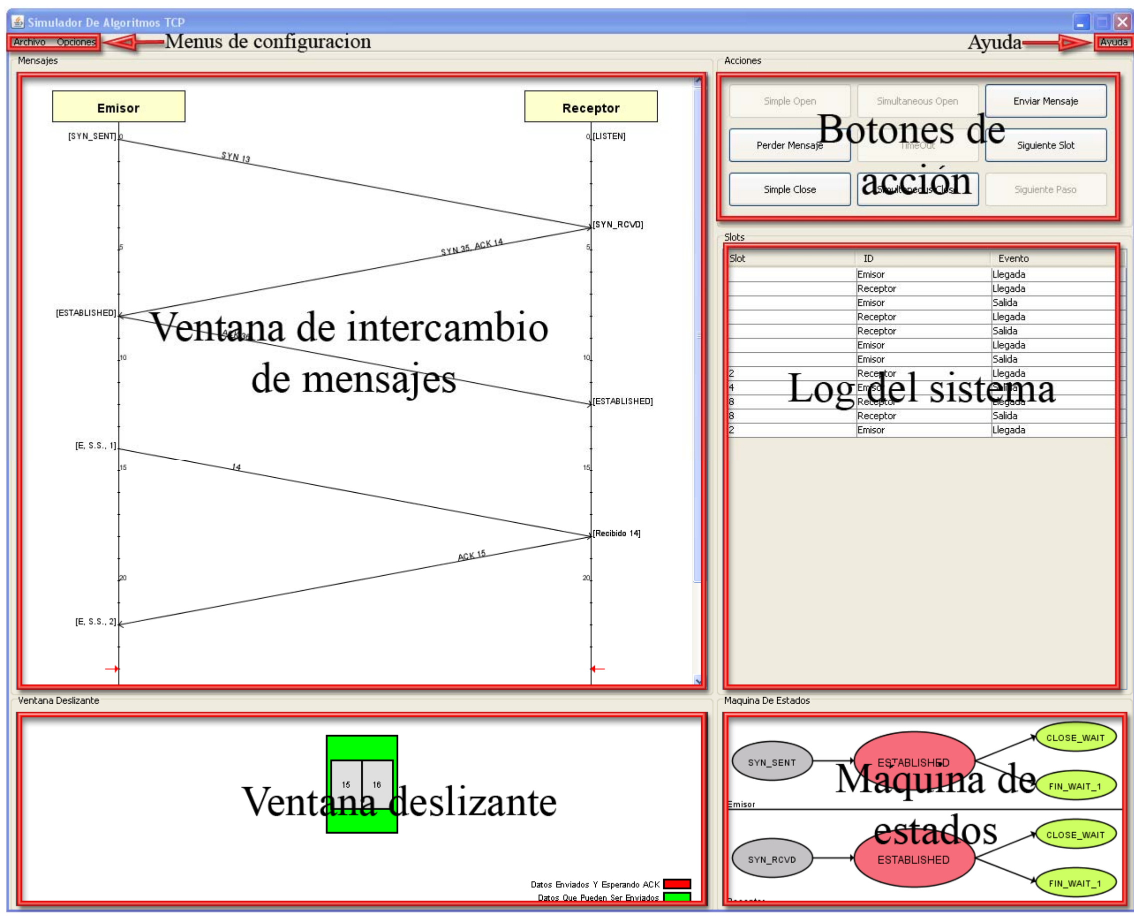


Figura 5.1 Ventana principal del sistema

A continuación se describirán cada uno de los elementos que componen la ventana principal del sistema.

5.2.1 Ventana de intercambio de segmentos

En esta pantalla podremos observar el intercambio de segmentos entre emisor y receptor. La evolución del intercambio de segmentos está acompañada por información relativa a cada elemento, información mostrada en la misma pantalla.

Además, cuando el usuario sitúa el ratón encima de la información del emisor o del receptor aparecerá un *tooltip* aumentando la cantidad de información que podemos obtener simplemente en la pantalla. La información visible en el emisor o en el receptor es modificable en el menú "*Opciones>Seleccionar*".

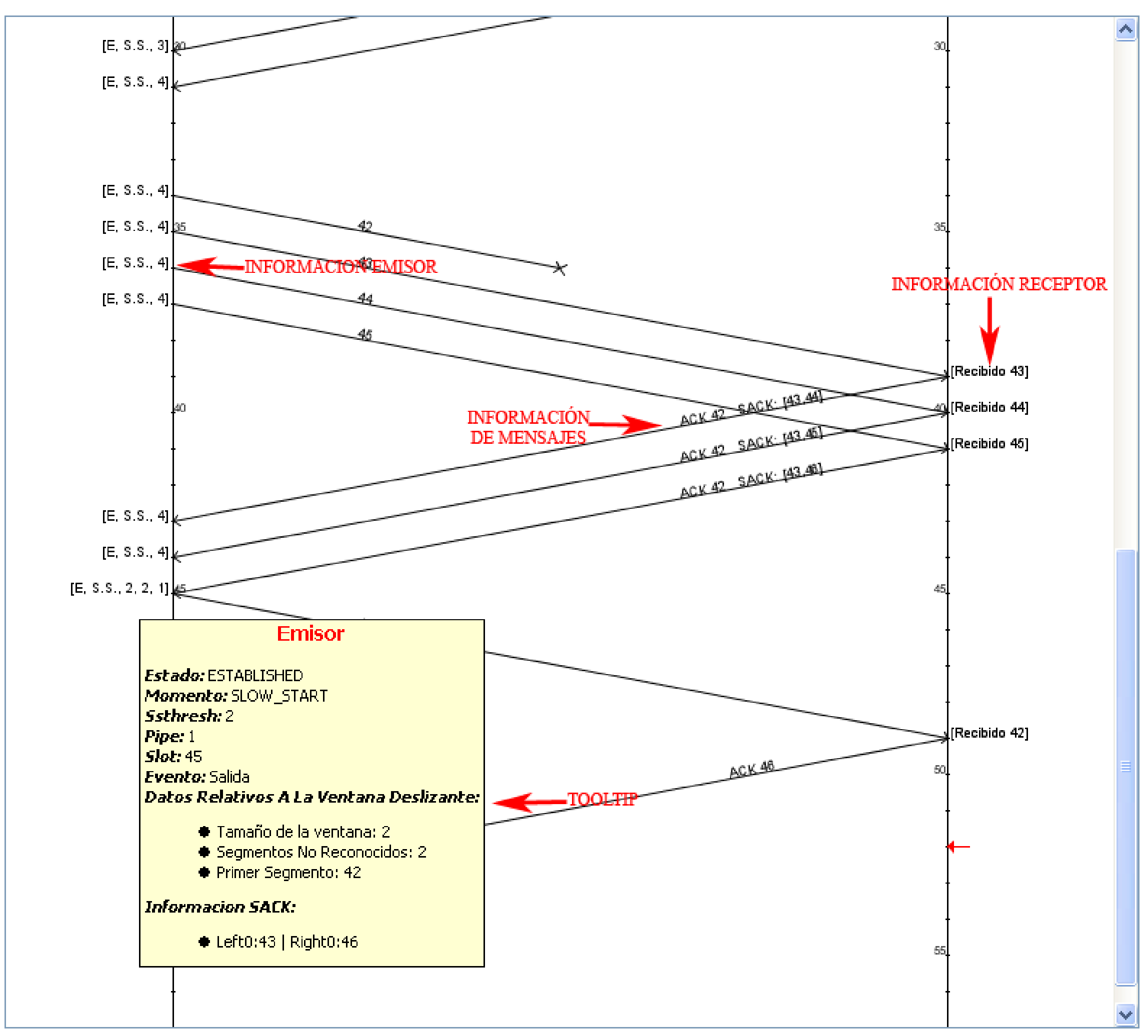


Figura 5.2 Ventana de intercambio de segmentos

A no ser que modifique la información a mostrar accediendo al menú “Opciones→Seleccionar”, la leyenda aparecida en el emisor indica:

[Estado, Momento, Cnwd, Ssthresh, Pipe, Timeout]

La leyenda en el receptor indica la secuencia recibida y si es duplicado o no, mientras que sobre el segmento mostraremos el número de secuencia en caso de ser un segmento de emisor a receptor, y de reconocimiento e información SACK en caso de ser un segmento de receptor a emisor.

Entendiéndose que no en todos los algoritmos se utilizarán todas las variables y por tanto no se mostrarán.

5.2.2 Botones de acción

Este elemento es de vital importancia, ya que permite la interoperación entre el usuario de la aplicación y la propia aplicación. En cada simulación que el usuario ejecute el comportamiento del algoritmo TCP bajo estudio será distinta, según las acciones que

el usuario seleccione mediante el uso de los botones activos. En cada instante de tiempo (o slot temporal), distintos botones se activarán/desactivarán, permitiéndonos realizar un determinado conjunto de acciones.

Como podemos ver en la **Figura 5.3** los botones que podrán estar o no activos en cada instante de tiempo son:

- ***Simple Open***: Ejecuta una apertura simple en el sistema, solamente ejecutable al comienzo de la simulación.
- ***Simultaneous Open***: Ejecuta una apertura simultanea en el sistema, solamente ejecutable al comienzo de la simulación.
- **Enviar Mensaje**: Envía un segmento TCP entre emisor receptor en el slot temporal que se encuentre.
- **Perder Mensaje**: Pierde un segmento TCP entre emisor receptor en el slot temporal que se encuentre.
- ***Timeout***: Hace saltar el temporizador cuando existen segmentos en el canal sin reconocer.
- **Siguiente Slot**: Incrementa la simulación en un slot temporal sin realizar ninguna acción en el emisor.
- ***Simple Close***: Realiza un cierre de conexión simple.
- ***Simultaneous Close***: Realiza un cierre de conexión simultánea.
- **Siguiente Paso**: Este botón solo se activará cuando carguemos una simulación guardada con el método de paso a paso. Iremos avanzando por cada pulsación un slot temporal.

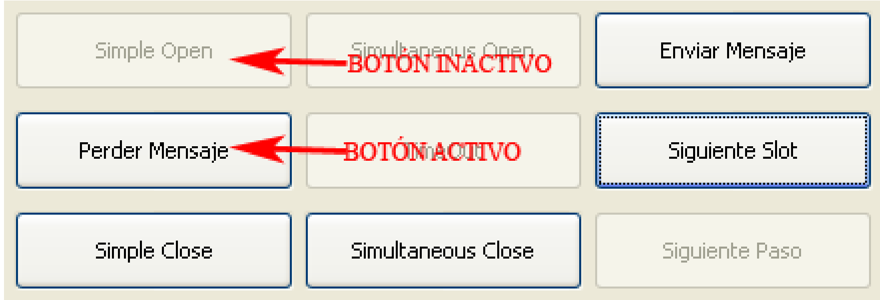


Figura 5.3 Ventana de acción

5.2.3 Log del sistema

En este apartado podremos acceder a los eventos ocurridos tanto en el emisor como en el receptor, ordenados temporalmente.

En la información del registro simplemente tendremos información relativa a:

- Tiempo de suceso (Slot)
- Sujeto del suceso
- Acción (Llegada o Salida)

Si hacemos doble clic sobre cualquiera de estos eventos desplegaremos una ventana que nos da toda la información relativa a ese evento, así como el estado del sujeto que intervino en dicho evento. La información añadida mostrada en la ventana de registro extendido puede ser estado, momento del algoritmo, tamaño de la ventana, valores de la variables auxiliares, etc. (ver ejemplo en la **Figura 5.4**)

Figura 5.4 Registro del sistema

5.2.4 Ventana deslizante

Un punto importante para la comprensión de los algoritmos TCP es el conocimiento del funcionamiento de la ventana deslizante. Para hacer hincapié en el aprendizaje y comprensión del funcionamiento de la misma, el simulador está dotado de un área donde es posible ver su estado actual, como el tamaño de la ventana y los números de secuencia de los segmentos.

Como se puede observar en el ejemplo mostrado en la **Figura 5.5**, la ventana mostrará en color rojo los segmentos enviados que están a la espera de reconocimiento y en color verde los segmentos que el emisor podría enviar.

Conforme vayamos ejecutando una simulación veremos como se actualiza el estado de la ventana, incrementándose / decrementándose y deslizándose sobre los segmentos reconocidos.

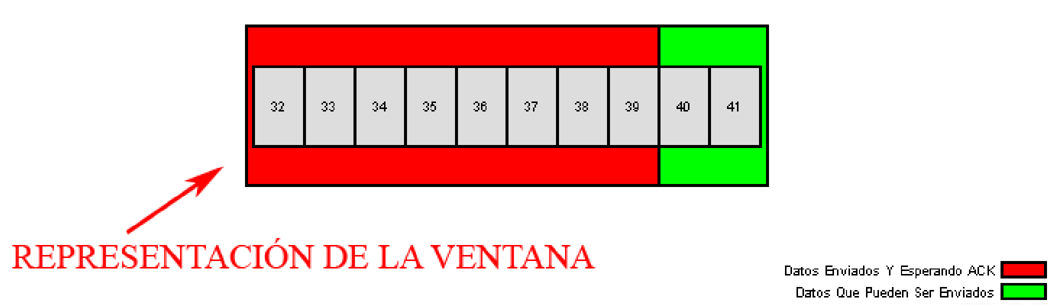


Figura 5.5 Registro del sistema

5.2.5 Máquina de estados

Como se vio en la teoría, el protocolo TCP usa una máquina de estados donde conmuta entre los mismos para establecer una conexión y cerrarla, manteniéndose durante la transacción de información en el estado "*Established*".

En la ventana de la aplicación denominada "*máquina de estados"* podemos ver de forma rápida y sencilla, el estado anterior, el actual y los posibles posteriores.

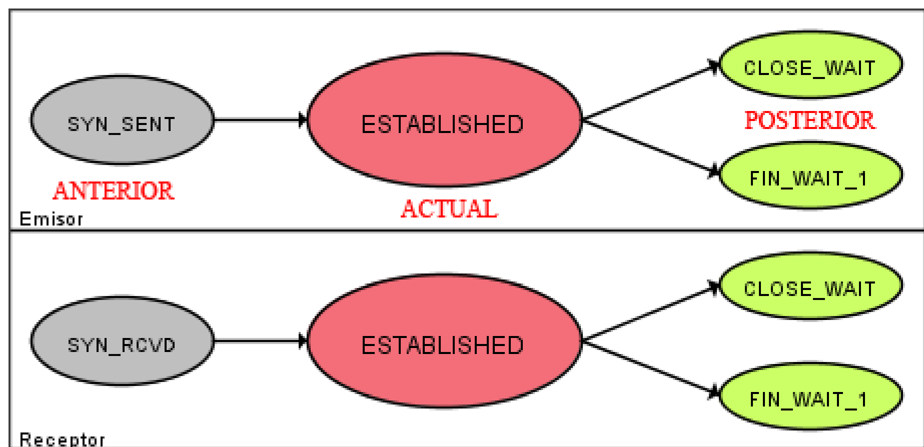


Figura 5.6 Zona de estados

Si además hacemos doble clic sobre esta ventana podremos observar una representación formal de la máquina de estados con las acciones y los eventos que desencadenan dichas acciones, así como los estados actuales del emisor y del receptor (ver **Figura 5.7**).

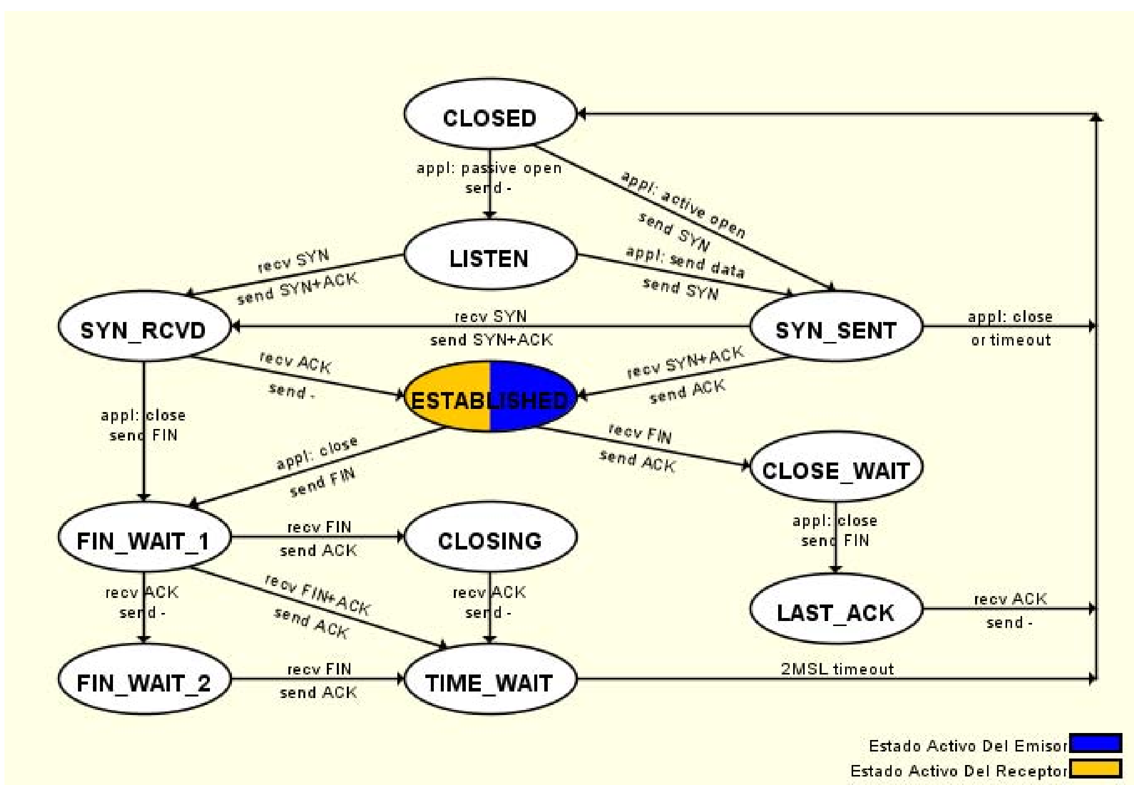


Figura 5.7 Diagrama de la maquina de estados completa

5.3 Menús de configuración

En este menú podremos acceder a la configuración del simulador. Pasamos a detallar cada uno de sus apartados.

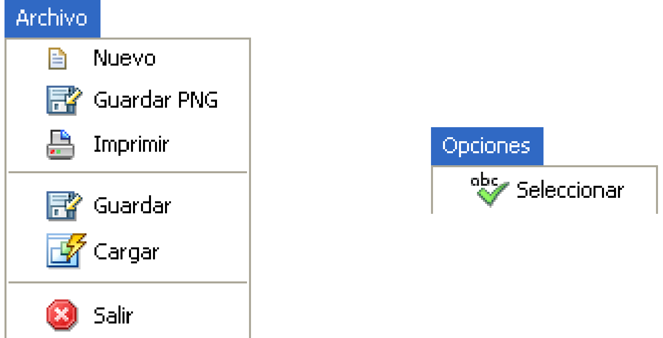


Figura 5.8 Menú Archivo y Opciones

5.3.1 Archivo

En este menú podemos encontrar seis acciones a realizar, donde la mayoría tienen clara su función. Hablaremos en primer lugar de la más compleja de ellas. "*Nuevo*" es la acción escogida si se desea generar una nueva simulación.

Mediante esta acción y utilizando la ventana mostrada en la **Figura 5.9** podremos indicar la configuración adecuada a la simulación que deseamos realizar. El usuario

deberá configurar dos parámetros. En primer lugar deberá especificar qué algoritmo TCP desea simular (simpleTCP, Vanilla, Tahoe, Reno o SACK).

En segundo lugar deberá especificar el tiempo de propagación en slots, es decir, el número de slots que son necesarios que transcurran para que un segmento enviado desde un sujeto de comunicaciones llegue al otro. Este parámetro debe estar comprendido entre uno y veinte, valores más que suficientes para realizar cualquier tipo de simulación.

Figura 5.9 Parámetros de la nueva simulación

Además de especificar los parámetros de una nueva simulación, el menú “Archivo” ofrece las siguientes opciones:

- *Guardar PNG*: Guarda la representación gráfica de la ventana de intercambio de segmentos en formato *'PNG'*.
- *Imprimir*: Imprime la representación gráfica de la ventana de intercambio de segmentos.
- *Guardar*: Guarda una simulación de manera completa que posteriormente podremos cargar.
- *Cargar*: Carga la simulación teniendo dos posibilidades de carga:
    - *Automática*: Carga hasta el final de la simulación guardada.
    - *Paso a paso*: Realiza paso a paso la carga de la simulación mediante el uso del botón de acción "Siguiente Paso".
- *Salir*: Termina el simulador.

5.3.2 Opciones

Mediante el menú "Opciones" el usuario de la aplicación podrá seleccionar qué información será mostrada en la ventana de intercambio de segmentos, pudiendo eliminar hasta las propias líneas de intercambio de segmentos, el texto en los mismos, en el receptor y las variables que aparecen en el emisor.

La información que puede mostrar relativa al emisor es:

- Estado: Indica el estado de la máquina TCP.
- Momento: Es cada uno de los posibles sub-estados dentro del algoritmo (*Slow Start*, *Congestion Avoidance*, etc.)
- Cnwd: Tamaño de la ventana de transmisión.
- *Timeout*: Indica si se ha producido un *timeout*.
- Ssthresh: Muestra el valor de la variable ssthresh en aquellos algoritmos que la utilizan.
- Pipe: Muestra el valor de la variable pipe en el algoritmo SACK.

En cuanto a los otros elementos que podemos modificar su visualización son:

- Texto Receptor: Como indica su nombre es el texto relativo a lo ocurrido en el receptor, normalmente recepción de números de secuencias.
- Líneas: Relativo a las líneas que representan los segmentos que viajan por el canal.
- Texto en mensaje: Texto importante que es transportado por los segmentos.

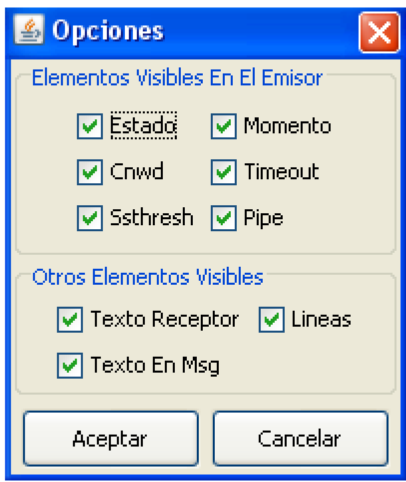


Figura 5.10 Opciones del simulador

5.4 Menú de ayuda

En este menú podemos acceder a esta ayuda en formato HTML que nos introduce en los aspectos teóricos necesarios de TCP, así como en el uso del simulador.

5.5 Nota importante para la ejecución del simulador

Como se ha comentado en el cuarto capítulo es necesario aumentar la memoria de intercambio para poder utilizar todas las funcionalidades del simulador. Debido a la necesidad de crear un tipo de buffers especiales para copiar el elemento gráfico de la ventana de segmentos para su impresión o guardado en PNG, deberemos ejecutar el comando con dicha modificación de tal forma que quedaría:

java -jar Simulador.jar –Xms<memoria>M

Donde <*memoria*> sería la cantidad de memoria que se quiere utilizar en la ejecución del sistema, siendo recomendable utilizar un mínimo de 250 Mbyte.

6. Conclusiones

6.1 Conclusiones

El objetivo principal que se fijó al inicio del desarrollo del simulador, consistía en brindar a los alumnos de esta Escuela una herramienta informática que les permitiera afianzar, entrenar e incluso autoevaluar, de una forma sencilla e intuitiva, sus conocimientos sobre el funcionamiento del protocolo TCP y sus algoritmos principales.

Una vez desarrollada nuestra aplicación, y analizando las características que ofrece la implementación final para poder valorar sus resultados, podemos distinguir dos tipos de cualidades que tiene la misma, externas e internas.

Sobre las cualidades externas podemos observar, mediante la ejecución del programa que destacan la funcionalidad (habilidad para realizar el trabajo para el que fue creado) y la fiabilidad (habilidad para mantenerse operativo en el tiempo sin pérdida de cualidades). Esta fiabilidad ha sido dada por el amplio trabajo realizado sobre el apartado gráfico de la aplicación, que ha sido costoso en cuanto a tiempo y conocimientos, pero que viendo los resultados obtenidos es del todo satisfactorio.

En cuanto a las internas podemos hablar de una gran modificabilidad, dada por el diseño modular de la aplicación que es fruto del empleo de patrones de diseño, así como el uso de interfaces y la aplicación de las propiedades de herencia proporcionadas por el lenguaje, que nos permiten adaptar el sistema actual a posibles ampliaciones.

En definitiva podemos calificar la aplicación desarrollada como un sistema orientado a objetos, modular y robusto, que nos ha presentado dificultades en cuanto al apartado gráfico del mismo, pero que han sido satisfechas con grandes resultados.

En cuanto al objetivo que se buscaba por parte del proyectista era la adquisición de conocimientos, resumiéndose los mismos en:

- Programación mediante el uso de patrones de diseño.
- Creación de interfaces de usuario (GUI).
- Conocimiento del dibujo 2D en Java.

Es por todo lo anterior que podemos concluir que los objetivos iníciales fijados, tanto para el simulador como para el proyectista, han sido ampliamente alcanzados con resultados satisfactorios.

6.2 Líneas de trabajo futuras

Dada la modularidad del sistema y el desarrollo mediante patrones de diseño estamos antes un simulador fácilmente ampliable para nuevas implementaciones TCP.

Es por tanto posible que éste no sea más que el inicio de una aplicación docente que pueda crecer, aportando a los alumnos nuevos conocimientos mediante el uso del simulador, así como la posibilidad de que los mismos se involucren en él desarrollando las nuevas modificaciones.

7. Bibliografía

Comer, D. E. (1996). TCP/IP principios básicos, protocolos y arquitectura.

Eclipse Foundation. (s.f.). *eclipse.org home*. Recuperado el 4 de Septiembre de 2007, de http://www.eclipse.org/

IETF. (s.f.). *RFC-Editor Webpage*. Recuperado el 2007 de Septiembre de 8, de http://www.rfc-editor.org/

Java Microsystems. (s.f.). *Descarga de software Java*. Recuperado el 7 de Septiembre de 2007, de http://java.com/es/

March Hare Pty Ltd & CVSNT Project. (s.f.). *CVS Soporte Profesional*. Recuperado el 4 de Septiembre de 2007, de http://www.march-hare.com/cvspro/es.asp

Sun Microsystems, Inc. (2003-2007). *The Java Tutorials: 2D Graphics*. Recuperado el 15 de Noviembre de 2006, de http://java.sun.com/docs/books/tutorial/2d/index.html