Universidad De Alcalá

Universidad Nacional Autónoma de Nicaragua - León

Facultad de Ciencias y Tecnología
Departamento de Computación
Maestría en Computación

Tesis de Maestría

DESARROLLO DE UN SIMULADOR EDUCATIVO CON ANIMACIÓN PARA LOS PROTOCOLOS DE PARADA Y ESPERA, RECHAZO SIMPLE Y RECHAZO SELECTIVO.

Raúl José Palma Mendoza

León, Nicaragua
Julio de 2011

Contenido

Agradecimientos y Dedicatorias

Este proyecto está dedicado a Dios Padre, Hijo y Espíritu Santo porque simplemente sin si gran amor y misericordia no hubiese podido escribir ni una sola línea de código, ni un solo renglón. Además agradezco a mi Madre María Reina del Cielo que me ha acompañado durante todo el proceso de la maestría y ha estado intercediendo ante el Padre por mí.

En segundo lugar quiero dedicar este trabajo a mi difunto padre Raúl Ovidio Palma quien murió durante el transcurso de la maestría, porque su amor de padre, su trabajo y buen ejemplo son elementos que forman parte esencial en mi vida. Agradezco a mi madre “Gina” porque ha hecho suyos todos los momentos alegres o difíciles que me ha tocado pasar, a mis tías Alba Luz y Guadalupe por su apoyo incondicional, a mis tíos Saúl y Santiago por su interés, a mi abuela Angelina por sus cuidados y buenos almuerzos, a mi novia Aneliza por soportarme y esperarme tanto durante cada encuentro de la maestría y a toda mi familia por recibirme siempre con amor. Quiero también agradecer a todos mis hermanos de la Comunidad de Jóvenes Siervos del Señor porque sus oraciones y ayuda permitieron que llegara a este punto.

Agradezco a mis compañeros de maestría que con su cálida amistad y apoyo hicieron que mi estadía en León fuese mucho más agradable. También agradezco a todos los que fueron nuestros maestros porque sin excepción se esforzaron en compartir sus conocimientos y experiencias para nuestro bien.

También quiero agradecer a mis compañeros de trabajo en el Departamento de Ingeniería en Sistemas de la Universidad Nacional Autónoma de Honduras, por su apoyo e interés desde el inicio de este proyecto hasta la fecha de hoy.

Finalmente agradezco a mi tutor de tesis el profesor Javier de Pedro Carracedos porque a pesar de la distancia siempre estuvo pendiente de mis consultas por correo y no le faltaron palabras para animarme a seguir adelante.

Capítulo I. Introducción

Resumen

El proyecto descrito en este documento tiene como propósito el desarrollo de un simulador educativo con animación para los protocolos de parada y espera, rechazo simple y rechazo selectivo. La razón para el desarrollo de esta aplicación fue debido a que se encontró un vacío en esta área pues aunque hay muchos simuladores algunos son muy complejos y otros tienen pocas capacidades gráficas y de animación. Este proyecto se inspiró sobretodo en el simulador educativo programado por el Dr. Jim Kurose de la Universidad de Massachusetts, Amherst y usado para realizar dos proyectos de programación durante el curso Protocolos de Comunicación impartido por el Dr. José Manuel Arco de la Universidad de Alcalá, España con el apoyo del Msc. Aldo Martínez de la Unán-León.

El actual documento está dividido en cinco capítulos que describimos brevemente:

Capítulo I. Introducción: Es el actual capítulo, aquí se enuncian algunos antecedentes del proyecto, su justificación y alcance, se define el problema a resolver y se plantean los objetivos del mismo.

Capítulo II. Marco Referencial: El marco referencial hace un breve repaso bibliográfico de temas importantes para el desarrollo del proyecto, no se intenta definir un marco exhaustivo sino más bien tocar puntos claves que dan paso al desarrollo del proyecto.

Capítulo III. Desarrollo de la Aplicación: En este capítulo abordamos los pasos tomados para desarrollar la aplicación, se sigue un modelo de desarrollo de software iterativo, y se describe cada una de las iteraciones llevadas a cabo que incrementalmente fueron definiendo la aplicación. Los objetos de software que conforman la aplicación se introducen gradualmente tratando de seguir el orden cronológico en el que se fueron incluyendo en la aplicación.

Capítulo IV. Implementación de los Protocolos: Este capítulo aborda las implementaciones para los protocolos que están incluidos en la aplicación: parada y espera, rechazo simple y selectivo. Se presentan diagramas de flujo de los métodos desarrollados para cada protocolo.

Capítulo V. Conclusiones y Líneas de Trabajo Futuro: Aquí simplemente se enuncian algunas conclusiones obtenidas a partir del proyecto realizado y de la evaluación de los objetivos planteados. A parte de mencionar algunas ideas de trabajo futuro que permitan expandir aún más la funcionalidad de la aplicación.

Antecedentes

Existen en la actualidad varias herramientas desarrolladas a nivel mundial que podemos clasificar como antecedentes de este proyecto, unas de ellas son pagadas y otras son de uso público, unas muy complejas como los simuladores de red profesionales y otras más sencillas en forma de applets de Java y animaciones en Flash, unas llevan años de desarrollo y otras se realizaron en cuestión de días o meses. El principal motivo que nos llevó a pensar en este proyecto es el espacio existente entre las diferentes herramientas, unas muy complejas y otras muy simples.

Nuestro objetivo es facilitar el aprendizaje y evaluación de los protocolos de bit alternante, rechazo simple y rechazo selectivo mediante una herramienta que permita la definición, simulación y animación de los mismos pero sin tener que provocar un cambio significativo en la planificación de la asignatura que desee incluirla. A continuación enumeramos algunos de los simuladores de red más usados, sus características y las desventajas que tienen entorno al cumplimiento de nuestro objetivo.

El simulador x-Sim. Es una herramienta poco usada con fines educativos, está basada en el lenguaje C/C++ y en un emulador llamado x-Kernel que ofrece un buen framework para simulaciones basadas en TCP (Hutchitson & Peterson, 1991). Además posee pocas capacidades gráficas y es muy avanzado para principiantes (Crescenzi, Gambosi G. & Innocenti, 2005).

El simulador cnet. Al igual que x-Sim, está basado en C/C++, es muy usado en educación y está enfocado para estudiantes de pregrado. Permite la simulación y experimentación con una variedad de protocolos de capa de enlace a datos, red y transporte en redes formadas por cualquier combinación de enlaces WAN, LAN o WLAN. El cnet tiene muy buenas capacidades de visualización de los protocolos pero tiene la desventaja de que no funciona sobre plataformas Windows (Crescenzi et al. 2005).

El simulador ns-3. A pesar de que ns-2 fue el estándar de-facto para investigación académica en métodos de comunicación y protocolos de red durante varios años, ns-3 surgió como proyecto en el 2005 para reemplazar al ns-2. El principal objetivo fue volver al ns-3 un simulador más realista que el ns-2, al hacer que los modelos fuesen más parecidos en implementación al software que representan (Wehrle, Güne & Gross , 2010). El ns-3 es capaz de generar archivos específicamente diseñados para facilitar la visualización del flujo de

paquetes en la simulación. Además ns-3 es software libre y corre en Linux, Unix, Mac OS X y en Windows via Cygwin. La desventaja de ns-3 es que debido a su complejidad requiere invertir un tiempo significativo en su aprendizaje y sus capacidades de animación son limitadas.

GNS3 es un simulador gráfico de redes de computadoras, su objetivo principal es el de crear un ambiente virtual para la ejecución del sistema operativo de Cisco (Cisco IOS). GNS3 permite crear topologías de red complejas usando routers y firewalls ejecutando el Cisco IOS (Fuszner). Corre en plataformas Windows y Linux, pero no está orientado a la definición ni animación de protocolos.

OMNeT++ es una librería y framework de simulación, extensible, modular y basada en componentes, fue diseñada para simular principalmente redes de computadoras de gran tamaño, y facilitar la trazabilidad y depuración de las simulaciones. Las simulaciones pueden ejecutarse bajo una interfaz gráfica de usuario (Graphical User Interface) o GUI por sus siglas en inglés, que muestra gráficos de la red, animaciones del flujo de los mensajes, y permite ver detalles internos de los componentes y variables del modelo. El código de OMNeT++ es abierto y puede ser usado bajo la licencia GNU GPL para académicos, de investigación y/o comerciales (Varga A.). OMNet++ es otro de ejemplo de un simulador complejo y con muchas funcionalidades que requiere una inversión significativa de tiempo para su aprendizaje.

OPNET es un simulador comercial especializado en el desarrollo e investigación de redes fue desarrollado originalmente en el MIT. Puede ser usado para el estudio de redes de comunicación, dispositivos, protocolos y aplicaciones (Pan, 2008). Por el hecho de ser comercial es de entre todos los anteriores es el que ofrece un mayor soporte visual para sus usuarios, esto también lo hace ser económicamente costoso, aunque ofrece una versión denominada OPNET IT Guru Academic Edition que puede ser descargada gratuitamente desde su sitio web[1], dicha versión concentra sus capacidades de visualización en la definición de topologías y la presentación de gráficas estadísticas, la animación que se se puede observar sigue siendo limitada en comparación con la siguiente categoría de herramientas.

Simulador de Jim Kurose v1.1, es un pequeño simulador de red que está orientado específicamente a los protocolos de bit alternante y el de rechazo simple. Es de código abierto y se incluye como herramienta de apoyo en Kurose & Ross (2010)[2], está desarrollado en C, su gran

1 Accesible desde: http://www.opnet.com/itguru-academic/download.html
2 Disponible en:

ventaja para nuestro objetivo es la facilidad que proporciona para que el estudiante pueda definir (escribiendo código en C) y probar sus protocolos, su desventaja principal es que no presenta una visualización gráfica de los protocolos sino únicamente una salida de texto y por cuestiones de diseño no permite la simulación del protocolo de rechazo selectivo.

Además de los ejemplos anteriores, existe otro tipo de herramientas que no son simuladores, sino que son únicamente animaciones educativas de los protocolos, a continuación mencionamos tres artículos que presentan herramientas de este tipo:

White (2001). presenta un conjunto de once herramientas de apoyo a la educación en redes de ordenadores todas escritas usando Macromedia Director, estas herramientas muestran de forma detallada el flujo de datos en la red en varias situaciones como: encapsulación de datos, detección de errores, DNS, etc.

Holliday (2003), muestra un grupo de tres herramientas que apoyan la educación en los temas de encapsulación y fragmentación, control de errores y acceso al medio, las herramientas son todas applets de Java.

Hsin (2010) igual que los anteriores presenta dos herramientas, estas también fueron desarrolladas como applets de Java, son usadas a nivel global y están incluidos como apoyo al libro de Kurose & Ross (2010). Los primeros dos applets son animaciones de los protocolos de rechazo simple y selectivo, el segundo muestra el control de flujo TCP.

Después de todo lo anterior se consideró conveniente crear un simulador que por su sencillez y especificidad es fácil de usar, que permite definir los protocolos usando un lenguaje de programación conocido, que es capaz de presentar de forma animada los protocolos en cuestión, que facilita aún más su aprendizaje y evaluación, que corre sobre plataformas Windows, UNIX y Linux y que es software libre y de acceso público.

http://wps.aw.com/aw_kurose_network_5/111/28536/7305312.cw/index.html

Planteamiento del Problema

En el ambiente académico y estudio de las redes de computadores, enseñar y mostrar el funcionamiento de los protocolos de red, hasta los más básicos, siempre es un reto que hay que enfrentar. Existen casos en los que el docente cuenta sólo con una pizarra y un marcador, pero aunque aún queda mucho por recorrer, existen varias iniciativas de países latinoamericanos que impulsan proyectos la tecnologías de la información y comunicación en la educación (Ávila, 2002) y los docentes encargados de enseñar en el área de redes de computadores siempre deben estar abiertos al uso de herramientas didácticas como las presentaciones de diapositivas por computadora. Esto abre el campo para saltar del texto e imágenes tradicionales al uso de animaciones para enseñar los protocolos de datos.

Se han realizado estudios sobre cómo las tecnologías de información y comunicación tienen un impacto positivo en el aprendizaje de los estudiantes (Zahorec, Haskova & Munk, 2010). Además Naps et al. (2002) mencionan el impacto positivo de las tecnologías de visualización en el aprendizaje de estudiantes específicamente en el área de educación en las ciencias de la computación. A pesar de esto White (2001) dice que el campo de las redes de computadoras se ha quedado corto en cuanto a la parte de visualización, con respecto a otros campos de las ciencias de la computación como por ejemplo el de los algoritmos.

Dentro del campo de las redes de computadoras está el estudio de los protocolos de computadoras, rama en el cual creemos es que de mucho provecho el uso de visualización y animación. Actualmente, como se mencionó en los Antecedentes hay varias herramientas que sirven como visualizaciones y animaciones de protocolos que pueden ser utilizadas libremente como apoyo en el aprendizaje, entendiendo como visualizaciones a aquellas aplicaciones que muestran de forma gráfica el comportamiento de un protocolo y animaciones a aquellas que hacen que los gráficos varíen en el tiempo, algunos ejemplos de tecnologías usadas para esto son los applets de Java y las animaciones en Flash. También existe otro tipo de herramientas más complejas, los simuladores de red, que son abstracciones de diferentes elementos de la red para poder analizar su comportamiento en un ambiente controlado, algunas de éstas incluyen características de visualización y animación de elementos de las redes que modelan, incluyendo dentro de estos elementos a los protocolos de red.

Cabe recalcar que como lo mencionan Hsin (2010) y White (2001) las visualizaciones son particularmente útiles cuando son parametrizables, es decir cuando facilitan la interacción del estudiante al permitirle modificar los

parámetros de entrada para observar diferentes comportamientos del protocolo.

Partiendo de lo anterior podemos ir más allá de la creación de una herramienta parametrizable que presente de forma gráfica un protocolo y considerar una herramienta didáctica que permita al estudiante definir un protocolo mediante código de programación de forma similar a cómo se haría en la realidad, y que además permita luego observar una animación parametrizable del protocolo definido. Podríamos clasificar este tipo de herramientas como simuladores educativos pues para animar el protocolo definido es necesario que exista un ambiente que literalmente simule un comportamiento específico de la red, sin llegar a tener la complejidad de los simuladores de red profesionales.

Hacemos especial mención al simulador de red realizado por Jim Kurose para los protocolos de bit alternante y rechazo simple, éste es un buen ejemplo de un simulador educativo, que tiene la desventaja de que no presenta una visualización ni animación del protocolo, sino simplemente una salida de texto que lista una serie de eventos que ocurrirían según el protocolo programado por el estudiante, que puede volverse muy extensa y llegar a ocultar con mayor facilidad errores en la programación.

Se vio entonces la oportunidad de crear una herramienta capaz no sólo de simular un ambiente de red sino también de generar una animación parametrizable de los protocolos de bit alternante y rechazo simple y además del protocolo de rechazo selectivo. Frente a esta oportunidad surgió la siguiente pregunta general y otras preguntas específicas:

¿Es posible desarrollar una aplicación educativa que permita definir, simular y animar los protocolos de parada y espera, rechazo simple y rechazo selectivo y que además se aceptada ampliamente?

> ¿Cómo deberá estar diseñada una interfaz gráfica que permita observar claramente los detalles de la comunicación entre dos hosts que usan los protocolos de red en cuestión?
>
> ¿Cuál sería un diseño interno adecuado para el software simulador-animador?
>
> ¿Qué métodos se pueden utilizar para que la aplicación sea utilizada ampliamente?

Justificación y Alcance

Es fácil observar como en los últimos años las tecnologías de comunicación de datos y las redes de computadoras han tomado gran importancia como campo de estudio e investigación, unido a esto el crecimiento del Internet y el acceso a nivel global de sus aplicaciones populares como la consulta de páginas web y el correo electrónico, hace de la enseñanza de la teoría de las redes de computadoras un elemento importante y es impensable que existan universidades con grados orientados a las ciencias de la computación que no incluyan asignaturas relacionadas con éstos temas.

Schaible & Gotzhein (2002) introducen la definición de animación de protocolos, como una serie de técnicas para la representación de aspectos específicos de los protocolos, como su simulación, ejecución y visualización; además mencionan que éstas técnicas están ganando importancia en su uso tanto en el desarrollo de protocolos como en su investigación y enseñanza.

En el planteamiento del problema observamos ejemplos de dos tipos de herramientas: las aplicaciones sencillas de animación de protocolos creadas como applets de Java o proyectos de Macromedia Director fáciles de aprender y usar; y los simuladores de red profesionales muy complejos que requieren de un esfuerzo e inversión de tiempo previo para poder ser capaces de usarlos. Se hizo especial mención al simulador de Jim Kurose por ser meramente educativo y especializado en los protocolos que nos interesan pero por su salida basada en texto, no se puede considerar como una animación de dichos protocolos según la definición dada por Schaible y Gotzhein (2002) .

Debido a lo anterior consideramos en este proyecto el desarrollo de un simulador meramente educativo desarrollado, con el fin de que funcione en múltiples plataformas, en el lenguaje de programación Java. La aplicación tiene un diseño interno que simula de forma básica el comportamiento de una red y expone al los estudiantes una interfaz de programación (métodos de una clase) que les permite programar un protocolo de red ya sea de parada y espera, rechazo simple o selectivo e incluso permite definir un protocolo personalizado con tal cumpla con la interfaz proporcionada. Una vez programado el protocolo, la aplicación puede correr una simulación del mismo entre dos hosts y mostrar de forma animada los detalles de la comunicación, incluyendo los paquetes enviados, corruptos, perdidos y recibidos.

Se espera que el uso de la aplicación tenga un efecto facilitador, que ayude a que el estudiante pueda comprender el comportamiento de los protocolos en

cuestión. Además que reduzca el trabajo al docente encargado de evaluar los protocolos mediante el estudio inicial de la animación, y el código realizado por el alumno.

El mayor impacto pensado para la aplicación sería que sus beneficios se extendiesen a nivel internacional, pues los antecedentes actuales nos muestran que el simulador basado en texto de Jim Kurose es muy usado o al menos muy conocido debido a su libro (Kurose & Ross, 2010). Un hito importante para lograr este impacto sería que la aplicación fuese aceptada por algún autor reconocido internacionalmente y sea sugerida como herramienta de apoyo a la parte específica que trata. Para lo cual, una vez completada, se realizaron acciones básicas para su promoción.

Objetivos

Objetivo General

Desarrollar un simulador de red para los protocolos de parada y espera, rechazo simple y rechazo selectivo que muestre mediante una animación los detalles de la comunicación entre dos hosts y sea aceptado ampliamente como herramienta de apoyo al estudio de dichos protocolos.

Objetivos Específicos

Diseñar una interfaz gráfica que permita observar claramente los detalles de la comunicación entre dos hosts que usan los protocolos de red en cuestión.

Diseñar según el modelo orientado a objetos el software simulador-animador.

Determinar y realizar actividades básicas que permitan que la aplicación pueda ser utilizada ampliamente.

Capítulo II. Marco Referencial

Introducción a la Simulación de Redes de Datos

Wehrle K., Güne M. & Gross J (Ed.), 2010

En este capítulo abordaremos los principales elementos teóricos necesarios para lograr el desarrollo del simulador-animador. Tratando de seguir el orden cronológico del desarrollo de la aplicación iniciamos con una breve introducción a la simulación de redes de datos, luego abordamos muy brevemente el tema de los protocolos de comunicación y finalmente terminamos con la creación de animaciones en Java.

En general existen tres diferentes técnicas para evaluar el rendimiento de sistemas y redes: el análisis matemático, las mediciones del sistema real y la simulación por computadora. Cada una de éstas tiene sus fortalezas y debilidades, por ejemplo conforme la complejidad de un sistema aumenta se vuelve imposible hacer un análisis matemático detallado del mismo y las mediciones en el sistema real requieren que el sistema exista lo que puede llegar a ser muy costoso, en estos casos la simulación o el análisis por computador de un modelo del sistema puede ser una buena opción.

Las simulaciones por computadora son aplicadas en varios campos y hay varios tipos de simulaciones, por ejemplo: la simulación de eventos de discretos, la simulación continua, simulación Monte Carlo, simulación de hoja de cálculo, la simulación dirigida por trazas, etc. A pesar de esto en el campo de la redes de computadoras la técnica dominante es la simulación de eventos discretos, cuya propiedad principal es que el estado del modelo de simulación puede cambiar sólo en puntos discretos en el tiempo. A éstos puntos discretos se les conoce como eventos (Banks, Carson, Nelson & Nicol, 2005). Según Wehrle et al. (2010) la simulación de eventos discretos provee una forma simple y flexible para evaluar la redes de computadoras en todas sus capas y estudiar su comportamiento en diferentes condiciones. Un aspecto importante de las simulaciones por computadora es su capacidad para evaluar diferentes diseños bajo los mismos parámetros de entorno.

Simulación de Eventos Discretos

Iniciamos definiendo una serie términos y componentes que son comunes en todos los sistemas simuladores de eventos discretos, desafortunadamente no hay un grupo de términos estándar y por eso los nombres varían en la literatura.

> Una entidad es una abstracción de un objeto de interés, que está descrita por sus atributos, por ejemplo un paquete con número de

secuencia, carga de datos, número de asentimiento, etc.

Un sistema es un grupo de entidades y sus relaciones que cumplen un cierto propósito, por ejemplo la transmisión de datos.

Un sistema discreto es un sistema cuyo estado definido como el estado de todas las entidades que lo forman, dicho estado cambia solamente en puntos discretos en el tiempo, este cambio es provocado por la ocurrencia de un evento. La definición de qué es exactamente un evento depende sobretodo del sistema y el objetivo de estudio, por ejemplo pueden ser eventos: la recepción de un paquete, el envío de un mensaje, el vencimiento de un temporizador.

Comúnmente el sistema que interesa es muy complejo y para poder evaluarlo a través de una simulación por computadora se crea un modelo de simulación. Un modelo es una abstracción del sistema, consiste únicamente de entidades y relaciones selectas que son de interés para el fin de la simulación.

La idea central de un simulador de eventos discretos es saltar de un evento al siguiente, donde la ocurrencia de un evento puede provocar cambios en el estado del sistema así como crear nuevas notificaciones de eventos en el futuro. Los eventos se almacenan como notificaciones de eventos en una lista llamada “lista de eventos futuros”, que es un estructura de datos apropiada para manejar los eventos de la simulación. Una notificación de evento está compuesta por al menos dos datos: un tiempo que indica cuándo ocurrirá el evento y un tipo que indica la categoría del evento que ocurrirá. Algo interesante de notar es que los eventos no tienen porqué estar separados una distancia igual en el tiempo unos de otros y éstos se procesan en orden según su tiempo de ocurrencia como lo muestra la Ilustración 1, tomada de Wehrle et al. (2010).

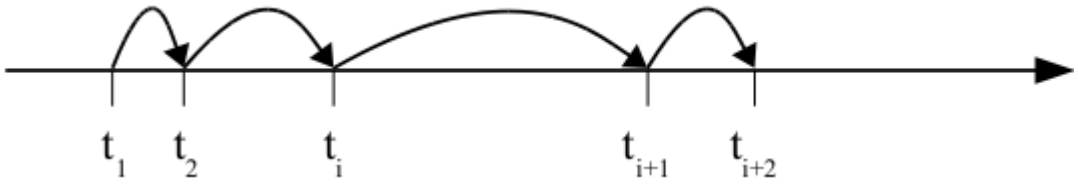


Ilustración 1: Eventos en el tiempo en una simulación de eventos discretos

Todos los simuladores de eventos discretos comparten los siguiente componentes:

El estado del sistema: Es un grupo de variables cuyos valores determinan el estado de cada uno de las entidades que conforman el

sistema.

Reloj: El reloj nos proporciona el tiempo actual durante la simulación.

Lista de eventos futuros: Contiene todas las notificaciones de eventos a ocurrir.

Contadores estadísticos: Un grupo de variables que contienen información estadística del rendimiento de la simulación.

Rutina de inicialización: Una rutina que inicializa el modelo de la simulación y establece el tiempo inicial en 0.

Rutina de temporización: Una rutina encargada de obtener el siguiente evento de la lista y avanzar el reloj hasta el tiempo de ocurrencia del evento.

Rutina manejadora de eventos: Una o varias rutinas que se llaman cuando un evento de determinado tipo ocurre.

El Algoritmo Planificador de Eventos y Manejador del Tiempo

Un simulador de eventos discretos tienen un algoritmo central que llamamos Algoritmo Planificador de Eventos y Manejador de Tiempo. Durante la simulación el estado del sistema evoluciona con el tiempo así que hay un reloj que proporciona el tiempo actual en la simulación. La lista de eventos futuros contiene todas las notificaciones de eventos ordenadas según su tiempo de ocurrencia.

En la Ilustración 2 (Wehrle et al.) mostramos el Diagrama de Flujo Algoritmo Central de un Simulador de Eventos Discretos, el proceso del simulador se organiza en tres etapas: en la primera se ejecuta la rutina de inicialización que como recién mencionamos pone el tiempo en cero y se le asigna valores iniciales a las variables de estado y en entidades en general, además creará al menos una notificación de evento inicial y la agregará a la lista. En la segunda parte el simulador entra en un ciclo en el cual se toma el siguiente evento de la lista de eventos futuros y se procesa por una rutina manejadora para el evento del tipo adecuado; esta rutina puede hacer cambios a las variables de estado, a los contadores estadísticos y también puede crear nuevas notificaciones de eventos en la lista. Por último, al cumplirse la condición de término de la simulación, el simulador entra en la tercera etapa en la que se imprimen los resultados estadísticos y probablemente sean salvados en ficheros del computador.

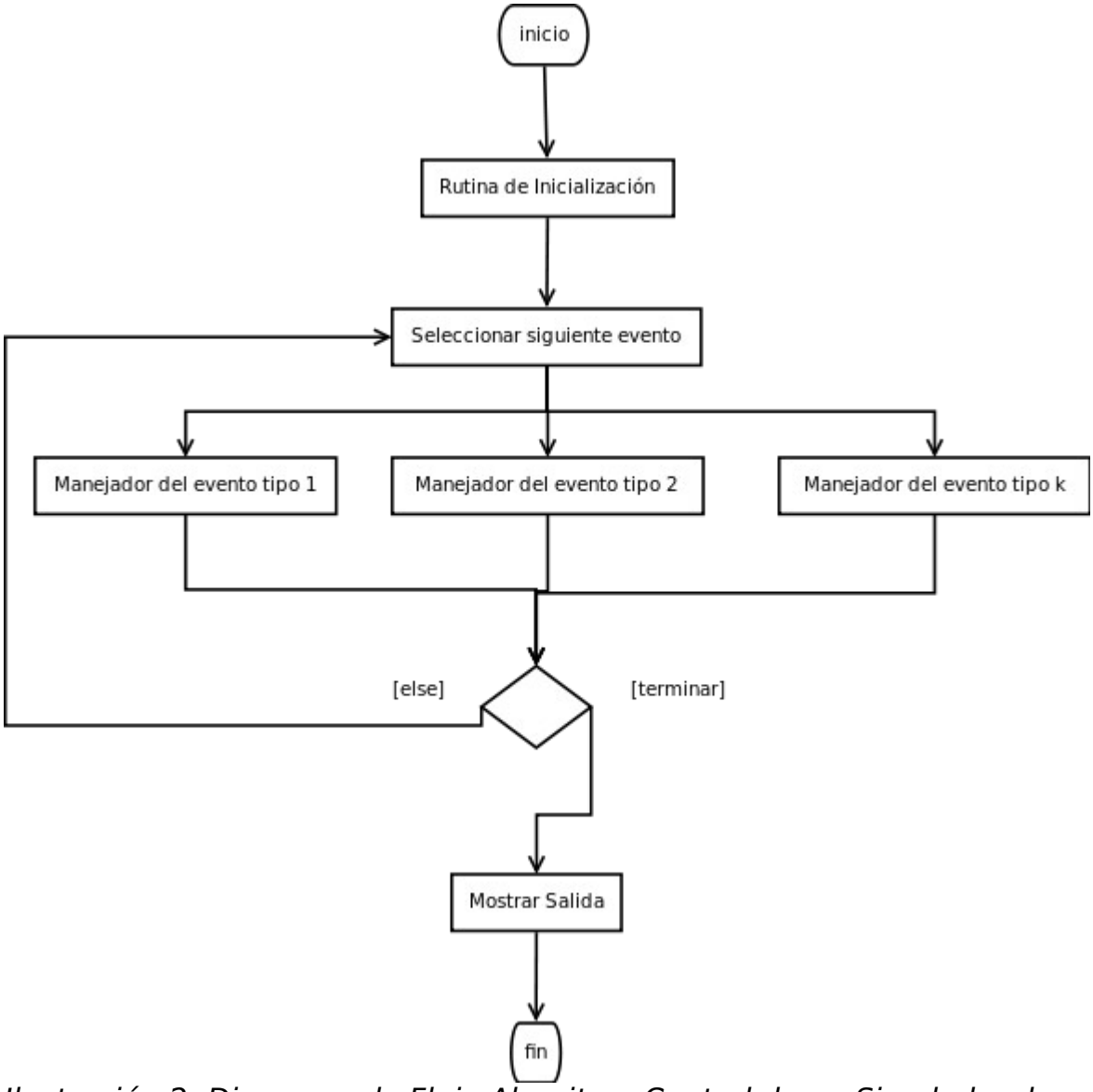


Ilustración 2: Diagrama de Flujo Algoritmo Central de un Simulador de Eventos Discretos

Típicamente la finalización de una simulación está basada en:

La ocurrencia de un evento especial, por ejemplo un paquete que se retrasó más de 500ms, la pérdida de 30 paquetes, el envío de una cantidad máxima, etc.

El cumplimiento de un tiempo máximo de simulación.

Hasta que la lista de eventos futuros quede vacía.

Protocolos de Red

Kurose & Ross (2010)

Toda la actividad en una red de computadoras que implica la comunicación entre dos o más entidades está gobernada por un protocolo. Por ejemplo, existen protocolos implementados en la tarjetas de interfaz de red de dos computadoras conectadas físicamente que controlan el flujo de bits en el cable entre las dos interfaces, también hay protocolos de control de congestión implementados en el software que controlan el ritmo al cual se transmiten paquetes entre un emisor y un receptor, además hay protocolos en los enrutadores que determinan el camino de un paquete desde su origen hasta su destino.

Podemos decir que un protocolo es un conjunto de reglas que *definen el formato y el orden de los mensajes intercambiados entre dos o más entidades que se comunican, así como las acciones tomadas en la emisión o recepción de un mensaje o evento* (Kurose y Ross, 2010).

Debido a la complejidad que presentan las redes de computadoras y para proveer una estructura al diseño de los mismos, los diseñadores de red organizan los protocolos en capas. Cada protocolo pertenece a una capa y cada capa presta un servicio a su capa inmediatamente superior. Una capa puede estar implementada en software, en hardware o en una combinación de ambos. Los protocolos de la capa de Aplicación como HTTP o SMTP y los protocolos de capa Transporte son usualmente implementados en software, los protocolos de capa Física y de Enlace a Datos son típicamente implementados en hardware en la tarjeta de interfaz de red. Los protocolos de red generalmente tiene una mezcla entre implementación en hardware e implementación en software.

Transferencia de Datos Fiable

El problema de implementar una transferencia de datos fiable se encuentra a nivel de capa de Aplicación, de capa de Transporte y de capa de Enlace a Datos, es un tema central en el área de redes de computadoras. El problema consiste en prestar el servicio de un canal fiable a las entidades de la capa superior, el trabajo se complica porque la capa inferior a la capa que proporciona el servicio de transferencia de datos fiable puede no ser fiable. Por ejemplo el protocolo TCP es un protocolo de capa de Transporte que es fiable pero está implementado sobre un protocolo de capa de Red no fiable como lo es el protocolo IP.

La responsabilidad de un canal fiable es que ningún bit transmitido sea corrupto (pase de 1 a 0 o viceversa) o se pierda y todos sean enviados en el orden en que se recibieron. Pasamos a describir ahora protocolos concretos.

Protocolos de Parada y Espera

La transmisión de datos fiable a través de un protocolos de parada y espera es la técnica más simple. Un protocolo de este tipo transmite un paquete de datos y luego espera una respuesta. El receptor recibe este paquete y envía un paquete de asentimiento (ACK) si los datos se recibieron correctamente y si no se recibe correctamente podría enviar un asentimiento negativo (NACK) o esperar a que un temporizador venza en el emisor para que el paquete sea retransmitido. Además el temporizador se usa para recuperarse de la condición que se da si el receptor no responde (posiblemente debido a la pérdida del paquete de datos o de su asentimiento).

En condiciones normales el emisor recibe un asentimiento para el paquete y luego inicia la transmisión del siguiente, en un enlace con mucho retardo el emisor deberá esperar una considerable cantidad de tiempo para recibir su respuesta y poder enviar más paquetes. Lo que puede hacer que este tipo de protocolos tengan un bajo rendimiento general.

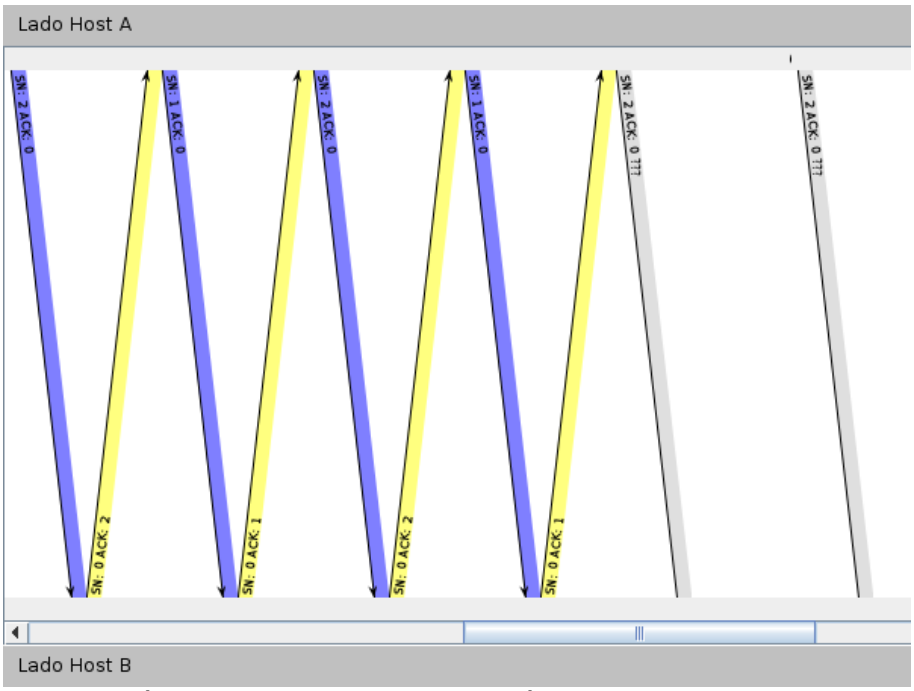


Ilustración 3: Ejemplo Transmisión de Datos con un Protocolo de Parada y Espera

Protocolos de Rechazo Simple

Los protocolos de rechazo simple tratan de mejorar el rendimiento general de los protocolos de parada y espera, llenando el medio de paquetes, esto se logra permitiendo al emisor enviar una cantidad N de paquetes sin recibir un asentimiento. Para soportar el rechazo simple un protocolo debe cumplir con lo siguiente:

> Se deben usar números de secuencia para identificar cada paquete que se envía. El rango de éstos números debe ser suficiente como para que el tiempo que pase antes de volver a tener que usar el mismo número para un paquete sea suficiente para que el paquete que llevaba este número antes ya haya dejado la red.
>
> El emisor deber mantener un buffer con todos los paquetes que se han enviado pero no se han asentido, pues si uno llega mal o se pierde debe ser capaz de retransmitirlo.
>
> El receptor debe conocer el número de secuencia más alto que ha recibido.

El funcionamiento básico de este protocolo consiste en que el emisor manda un cantidad N de paquetes, cantidad que se conoce como tamaño de ventana y que hace que a éstos protocolos se les conozca también como protocolos de ventana deslizante. Cada vez que el receptor recibe un paquete éste envía un asentimiento de dicho paquete, los asentimientos son acumulativos, por ejemplo, si el asentimiento del paquete i se pierde pero llega al emisor el asentimiento del paquete i + 3, entonces se entiende que se han recibido correctamente todos los paquetes hasta el i + 3.

El nombre del protocolo de rechazo simple se debe a la forma que el receptor manejan los paquetes corruptos, cuando llega un paquete corrupto éste y el resto de los paquetes que lleguen (aunque no estén corruptos) se descartan. Justo después de recibir el paquete corrupto, en algunas implementaciones, el receptor enviará un asentimiento negativo que le indicará el emisor que debe reenviar todos los paquete enviados. En las implementaciones en las que el receptor no envía asentimientos negativos, cuando un temporizador en el emisor se venza se deberá reenviar todos los paquete enviados sin asentir.

Como podemos observar el problema de rendimiento de los protocolos de rechazo selectivo radica en que el receptor ignora los paquetes fuera de orden aunque lleguen bien y esto causa que el emisor deba reenviar muchos paquetes innecesarios que ya habían llegado bien. Este problema se incrementa aún más cuando el tamaño de ventana crece.

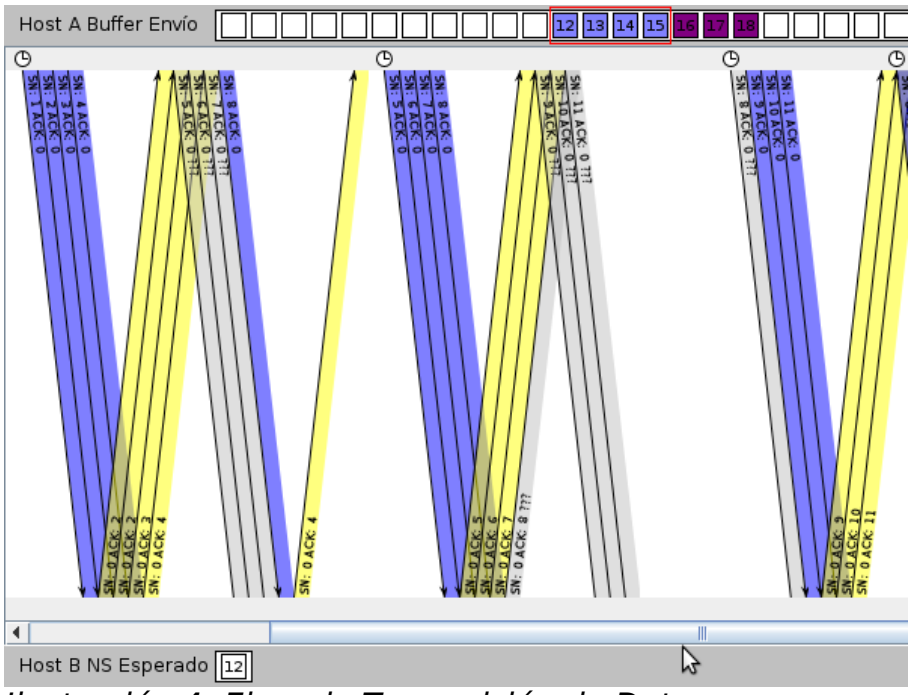


Ilustración 4: Ejemplo Transmisión de Datos con un Protocolo de Rechazo Simple

Protocolos de Rechazo Selectivo

Los protocolos de rechazo selectivo solucionan el problema del reenvío de paquetes innecesarios que tienen los de rechazo simple, haciendo que el emisor reenvíe únicamente los paquetes que sospecha se perdieron o se recibieron corruptos. Esta retransmisión individual requiere que el receptor asienta de forma individual y no acumulativa los paquetes recibidos y que se utilice un temporizador para cada paquete, se usa siempre un tamaño de ventana N para limitar el número de paquetes enviados y sin asentir en el canal. Sin embargo a diferencia de los protocolos de rechazo simple, cuando no se haya asentido el primer paquete de la ventana el emisor podrá tener en su ventana de transmisión ciertos paquetes ya asentidos.

En los protocolos de rechazo selectivo además de que se deben cumplir los requisitos necesarios para el protocolo de rechazo simple también es necesario que el receptor mantenga un buffer de los paquetes que se han recibido en desorden y no se han entregado a la capa superior. De esta forma el receptor puede asentir un paquete correctamente recibido esté orden o no.

Cuando se recibe un paquete corrupto en el receptor, éste se descarta, pero acepta cualquier otro paquete recibido correctamente y se envía su

asentimiento, en algunas implementaciones se envía un asentimiento negativo para informar el emisor del paquete corrupto, en otras, se espera que el temporizador individual del paquete corrupto se venza en el emisor para hacer retransmisión de dicho paquete. Lo anterior mejora la eficiencia en el uso del medio de este tipo de protocolos con la desventaja de que ahora se impone mayor carga en el emisor y receptor por el manejo de los buffers y de muchos temporizadores (emisor).

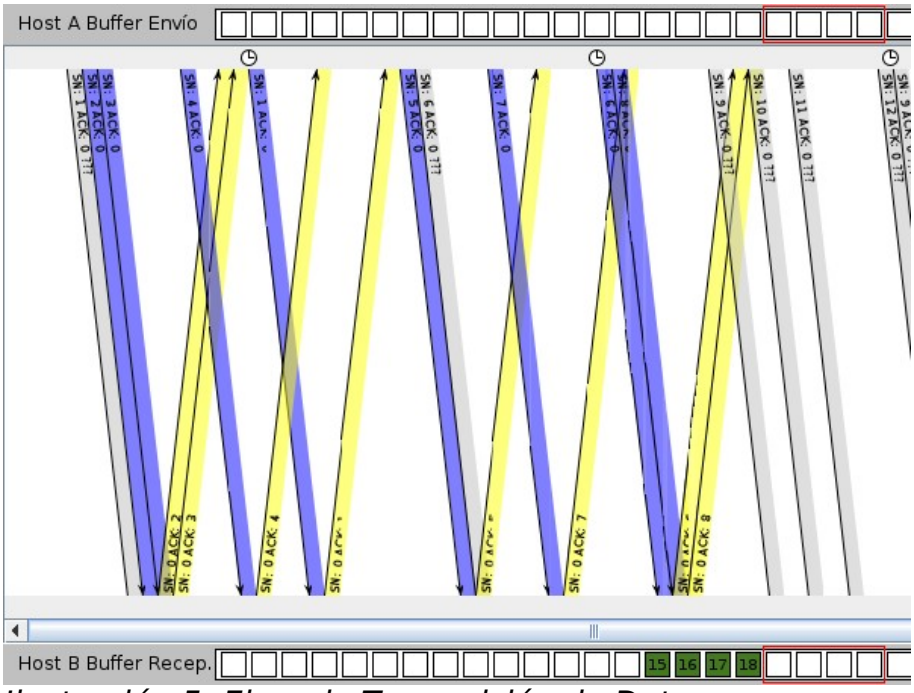


Ilustración 5: Ejemplo Transmisión de Datos con un Protocolo de Rechazo Selectivo

Fundamentos de Renderizado en Swing y Java2D

Haase & Guy (2008)

Swing

Swing es la principal y más usada librería para proveer de una GUI a las aplicaciones de escritorio de Java. Swing está compuesto por un conjunto de objetos diseñados para formar parte de una GUI que usualmente se les conoce como widgets. Algo importante de los widgets de Swing es que son ligeros, esto significa que no dependen directamente de los controles del sistema nativo sino que están implementados sobre otras librerías de Java, específicamente: Java2D y AWT. Las razones más importantes para usar Swing según Horstmann y Cornell (2008) son:

> Swing proporciona un conjunto rico y conveniente de elementos de interfaz de usuario.
>
> Swing tiene pocas dependencias en el sistema nativo, por tanto es menos propenso a errores específicos del sistema nativo.
>
> Swing proporciona un experiencia de usuario consistente entre todas las plataformas.

Eventos

Los eventos en Java pueden venir del sistema nativo, por ejemplo una ventana cuando se hace visible o una tecla presionada en el teclado, también pueden venir del mismo Java, por ejemplo el cambio del valor de una propiedad en un objeto. Todos esto eventos son colocados en una cola de eventos, el objeto *java.awt.EventQueue* tiene la responsabilidad de sacar uno a uno los eventos de esta cola y procesarlos de forma apropiada, el algoritmo que maneja esta cola ejecuta en un sólo hilo llamado el Hilo de Despacho de Eventos o EDT (Event Dispatch Thread) por sus siglas en inglés.

Es muy importante para toda aplicación interactuar eficientemente con este sistema de eventos, pues todo el trabajo relacionado con la GUI como el proceso de pintado que se aborda a continuación debe ocurrir en el EDT. Los eventos del sistema relacionados con la GUI son agregados en la cola de eventos y procesados en el EDT. Cualquier trabajo que una aplicación quiera realizar que implique modificar la GUI debe ser también procesado en el EDT lo que implica que para hacerlo habrá que agregar un evento al objeto *EventQueue.*

Proceso de Pintado en Swing

El pintado en Swing es el proceso por el cual una aplicación actualiza su apariencia, este proceso puede involucrar código personalizado si se desea tener algún pintado especial o sólo puede involucrar código interno de Swing que sabe como pintar los componentes estándar. El proceso inicia con una solicitud de pintado que se coloca en la cola de eventos y resulta en una llamada al método *paint* y *paintComponent* en el EDT para cada componente Swing afectado.

Una solicitud de pintado se puede originar de dos maneras: la librería Swing o AWT podrían colocar la solicitud o la propia aplicación puede hacerlo. Swing y AWT colocan una solicitud en respuesta a algún otro evento en el sistema nativo o en los componentes de la GUI, por ejemplo cuando una ventana aparece por primera vez o cuando se modifica su tamaño AWT recibe un evento del sistema nativo y coloca un evento Java para que la ventana se pinte. Similarmente, cuando un botón es presionado Swing emite una solicitud de pintado para asegurarse que el botón se muestre en el estado presionado. Una aplicación también puede emitir una solicitud de pintado a Swing directamente, este tipo de solicitud se hace en situaciones en las que el código de la aplicación en base a algún cambio en el estado interno sabe que la apariencia debería de cambiar.

En general el pintado ocurre automáticamente, Swing detecta que cuando el contenido de un componente ha cambiado de forma que necesita ser repintado. Existen varios métodos en los componentes Swing que permiten hacer solicitudes de pintado, pero acá sólo mencionaremos uno de los más usados.

Método repaint

El método repaint es bastante fácil de usar, éste le indica a Swing que todo el componente debe ser repintado. Es importante mencionar que las solicitudes de repintado se combinan por razones de eficiencia, por ejemplo si ya hay una solicitud de repintado en la cola de eventos y llega otra para el mismo objetos, entonces ésta última se ignora, pues al procesar la primera la necesidad de repintado será satisfecha.

Proceso de Renderizado en Swing

El modelo de renderizado en Swing es directo y se centra en el proceso de pintado de un sólo hilo que ya se mencionó. Primero se envía una solicitud de pintado que hace que se cree un nuevo evento en la cola. Luego en el EDT el

evento es despachado al objeto Swing *RepaintManager* que llama al método *paint* en el objeto apropiado. La llamada a *paint* provoca que el componente pinte primero su contenido, luego su borde y finalmente se pinten todos los componentes contenidos en él (componentes hijos). Este proceso se muestra en la Ilustración 6, tomada de Haase & Guy (2008).

De la forma descrita antes es como una jerarquía de componentes desde un *JFrame* hasta el último botón es renderizado. Hay que notar que este es un método de atrás hacia adelante donde el componente que se encuentra más al fondo es renderizado primero.

Una aplicación que desea conectarse a este modelo para pintar componentes personalizados necesita hacerse cargo de tres elementos:

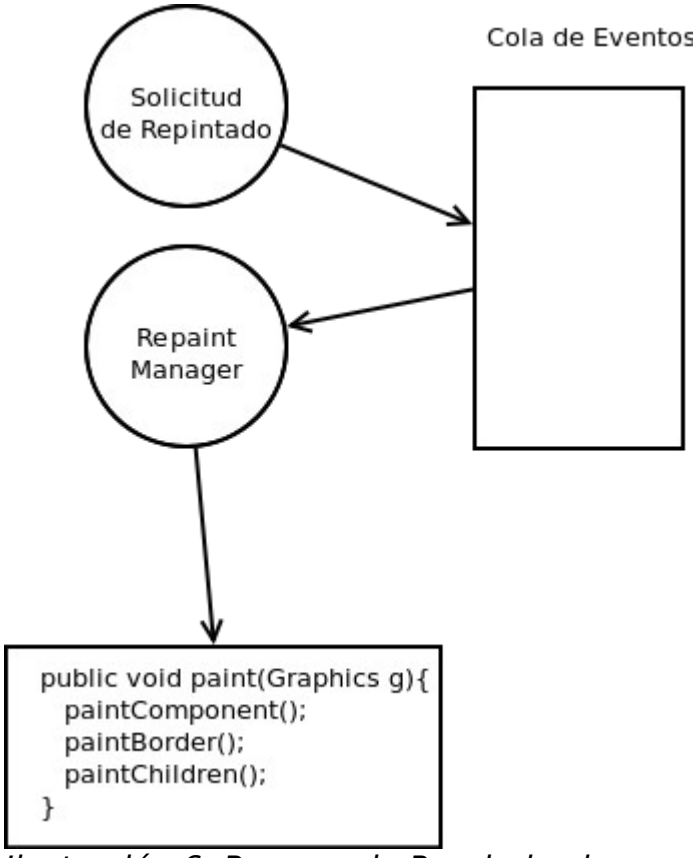


Ilustración 6: Proceso de Renderizado en Swing

JComponent.paintComponent(Graphics). Las aplicaciones que tienen componentes con renderizado personalizado necesitan sobreescribir este método para lograrlo. Éste renderizado incluye operaciones como

el dibujo de objetos , definir un gradiente para pintar el fondo, etc.

Component.paint(Graphics). La mayor parte de las veces las aplicaciones Swing no necesitan sobreescribir este método como ocurría con las aplicaciones anteriores basadas en AWT. Sin embargo, hay algunas situaciones en las que puede ser importante sobreescribir el método *paint* porque de esta manera podemos alterar el estado gráfico que será usado para pintar un componente y sus hijos.

JComponent.setOpaque(boolean). Las aplicaciones pueden en algún momento necesitar invocar al método *setOpaque* con el parámetro *false* en un componente que cuyos límites rectangulares no son completamente opacos para que Swing pueda renderizar correctamente los componentes que están atrás del actual.

Proceso de Renderizado en Java 2D

Java2D es una Interfaz de Programación o API (Application Programming Interface) por sus siglas en inglés, está formada por un conjunto de clases que permite renderizar imágenes y gráficos en 2D, cubre un amplio conjunto de operaciones, manipulación de imágenes, texto e impresión. Cualquier aplicación basada en Swing también está basada en Java2D, esto es porque Swing depende de Java2D para renderizar sus componentes.

Hemos visto que para poder personalizar la apariencia gráfica de un componente es necesario modificar su método *paintComponent*, formalmente el proceso de renderizado en Java2D tiene tres pasos:

1. Obtener un objeto de tipo Graphics o Graphics2D
2. Establecer atributos en el objeto Graphics.
3. Invocar primitivas gráficas de dibujo con el objeto Graphics.

Un código de ejemplo para un método *paintComponent* de un componente Swing que dibuja un recuadro de color azul sería:

```
protected void paintComponent(Graphics g) {
        g.setColor(Color.BLUE);
        g.fillRect(0, 0, getWidth(), getHeight());
}
```

Una vez conociendo el proceso central, el siguiente paso es estudiar cada una de la primitivas que se pueden usar con el objeto Graphics o Graphics2D, una amplia discusión sobre las mismas se puede encontrar en Haase & Guy (2008) y también el la documentación de Java.

Creación de Animaciones en Java

Haase & Guy (2008)

La Animación es un tema muy amplio y diverso, abarcando operaciones simples como copiar una imagen en distintos lugares de la pantalla hasta operaciones muy complejas como el último videojuego de disparos en primera persona o la realización de una película animada. Para nuestro propósito definimos animación como *la alteración basada en el tiempo de objetos gráficos haciéndolos pasar a través de diferentes estados, ubicaciones, tamaños y orientaciones* (Haase & Guy, 2008). La alteración de objetos se refiere a la modificación de la forma en que éstos se pintan o dibujan en la pantalla y el hecho de que sea basada en el tiempo se refiere que se debe definir cómo éstos objetos deberán ser alterados en el tiempo y que una vez alterados deberán de mostrarse sus cambios en la pantalla.

Animación Basada en Marcos (Frames)

En la animación por computadora como en las películas, el ser humano ve los cambios en dicha animación como una serie de imágenes sin movimiento que su mente junta como un flujo continuo. Una animación basada en marcos consiste precisamente en esto, en mostrar una serie de marcos o imágenes sin movimiento lo suficientemente rápido como para que la mente las junte y las haga ver como un movimiento fluido.

Frecuencia de Marco o Frame Rate

Cada vista separada de una animación es llamada un marco o frame y una animación es una sucesión de estos marcos. La frecuencia a la cual éstos marcos son mostrados es la frecuencia de marco o frame rate. Esta frecuencia está determinada por la frecuencia a la cual se necesita mostrar los marcos para que la animación se vea fluida y la frecuencia máxima que la computadora puede mostrar según su rendimiento y la complejidad de cada marco.

Movimiento basado en el tiempo

Es posible crear un animación en la cual se indiquen al computador todos los estados gráficos por los cuales debe pasar un objeto y el orden en que deben hacerse, el problema es que una vez iniciada la animación, el computador mostraría todos estos marcos tan rápido como pudiese, es decir que en un computador con altas prestaciones la animación pasaría muy rápido y en un

computador de menor capacidad la animación iría más lento. Para resolver este problema la animación debe estar basada en el tiempo. Esto es, podemos definir cuánto alterar una propiedad como el color o la posición en un periodo de tiempo t. Luego durante la animación podemos calcular cuál debería de ser el valor esta propiedad dado un tiempo específico. De esta forma no importa cuán rápido o lento sea un sistema específico el objeto animado siempre está en el estado gráfico correcto en el tiempo correcto.

La forma más simple de basar una animación en el tiempo es usando una interpolación lineal entre el estado inicial deseado y el estado final en función del tiempo transcurrido en la animación. Para lograr esto se puede usar la siguiente ecuación:

$$x = x_0 + t(x_1 - x_0)$$

donde

x es el valor que se quiere calcular en base al tiempo,

x_0 es el valor inicial de x,

x_1 es el valor final de x, y

t es la fracción transcurrida de la duración total de la animación que va de 0 a 1.

Capítulo III. Desarrollo de la Aplicación

Capítulo III.Desarrollo de la Aplicación

El presente proyecto es un proyecto de desarrollo de software, por tanto en esta sección nos centraremos en describir el proceso que se siguió para desarrollar el software simulador. En la producción de aplicaciones educativas basadas en computador, se ha considerado importante la aportación de la orientación a objetos desde el punto de vista del desarrollo de software (Armstrong y Loane, 1994), y dado que es indiscutiblemente el enfoque más seguido en la actualidad se usó también en el desarrollo del proyecto.

Un proceso de desarrollo de software describe una aproximación para construir, desarrollar y posiblemente mantener un software (Larman, 2004). Esto nos lleva a pensar que hay varias formas de llevar este proceso, el UP (Unified Process) (Jacobson, Booch, & Rumbaugh, 1999) es un tipo de proceso de desarrollo que ha cobrado mucha popularidad para el desarrollo de sistemas orientados a objetos, sus características más importantes, que a la vez son sus ventajas son:

> Es un modelo iterativo e incremental: el proyecto se organiza en una serie de miniproyectos cortos de duración fija (de dos a seis semanas) llamadas iteraciones, que elige un conjunto reducido de requerimientos, los diseña, implementa y prueba. El resultado de cada iteración es un sistema que puede ser probado, integrado y ejecutado. La salida es un subconjunto con calidad de producción final.
>
> Ofrece una rápida retroalimentación y asimilación de los cambios, posibilitada por el tamaño limitado de lo realizado en cada iteración. Se abordan, resuelven y prueban primero las decisiones de diseño críticas o de alto riesgo.

Por tanto para la realización de este proyecto se siguieron las fases y actividades propuestas por el Proceso Unificado de desarrollo de software:

- Inicio
- Elaboración
- Construcción
- Transición

En el UP se le llama artefactos a todo los resultados obtenidos como producto de una actividad, diagramas, códigos, documentos, etc. Debido a la naturaleza iterativa del UP, los artefactos también se desarrollan de forma iterativa, se inician en una fase y se refinan en las siguientes. Incluidos dentro de los artefactos del proyecto se encuentran varios tipos de diagramas UML (Unified Modeling Language)[3].

3 La última especificación de UML está disponible desde: http://www.omg.org/spec/UML/2.3/

Fase de Inicio

La fase de inicio según el UP, es una fase de factibilidad, se determina si vale la pena continuar con el proyecto, cuáles son sus objetivos, su visión, es una fase corta no debe durar más de un par de semanas. La actividad básica a realizar fue el análisis de los requerimientos de más alto nivel de la aplicación y la planificación de la primera iteración.

Consideramos que esta fase se realizó en su mayor parte con la elaboración del capítulo introductorio de este documento, pues en él determinamos la factibilidad del proyecto, sus objetivos y requerimientos de alto nivel.

Las actividades a realizadas fueron:

- Determinar requerimientos que presentan mayor riesgos.
- Inicio de la escritura de los casos de uso, haciendo énfasis en los centrales que presentan mayor riesgo.
- Inicio de elaboración de los diagramas de casos de uso. La mayoría de los diagramas se elaboró usando el software de diagramado Dia[4].

En lo que respecta al uso de tecnologías, ya se ha mencionado que la aplicación se desarrolló en el lenguaje de programación Java, la selección de este lenguaje de programación se realizó por los siguientes motivos:

- En la sección Objetivos, se mencionó que se desea que la aplicación sea ampliamente utilizada, por lo que debido a su indiscutible popularidad[5] Java resulta ser una buena opción.
- Java además contribuye al objetivo del amplio uso debido a que es un lenguaje multiplataforma y permite que la aplicación se ejecute en sistemas operativos Windows y Linux sin tener que hacer modificaciones al código.
- Las características orientadas a objetos de Java y el uso de paquetes permiten encapsular y ocultar al alumno todo el código que no es necesario que edite y mostrarle únicamente los métodos que debe programar.
- Las capacidades gráficas proporcionas por Java a través de su API Java 2D y las clases Swing, se ajustan muy bien a las necesidades de presentación de gráficos animados de la aplicación.

4 Sitio web oficial en: http://live.gnome.org/Dia

5 Se puede consultar: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html para ver estadísticas de popularidad de varios lenguajes de programación

- La abundante documentación disponible sobre el lenguaje permite que aunque no se tenga mucha experiencia con el mismo se pueda explotar todas sus facilidades.

Siguiendo con la selección de tecnologías, se usó el IDE (Integrated Development Environment) Netbeans[6] corriendo sobre el sistema operativo Linux, sobre todo por dos razones: su amplio uso a nivel industrial y académico y debido a que se puede usar de forma gratuita, elemento importante dado el bajo presupuesto del proyecto.

En este punto se tomó la decisión de ponerle un nombre a la aplicación y se escogió llamarla SW2PA (Stop and Wait and Sliding Window Protocol Animator) un nombre en inglés pensando siempre en su amplia aceptación y que resume el funcionamiento de la misma. La duración aproximada de la fase de inicio fue de aproximadamente dos semanas.

Modelo de Casos de Uso

Los casos de uso son un mecanismo ampliamente usado para descubrir y registrar los requisitos de una aplicación, en el UP el modelo de casos de uso no es más que el conjunto de todos los casos de uso de la aplicación y como el resto de sus artefactos se elabora iterativamente sobre todo en la Fase de Inicio y en la Fase de Elaboración. A pesar que este modelo no se creó completamente en esta etapa, decidimos mostrarlo aquí pues es un buena forma que empezar a ilustrar la aplicación. A continuación mostramos el modelo de casos de uso.

Caso de Uso 001: Definir Protocolo

Actor principal: Alumno.

Precondiciones: Ninguna.

Postcondiciones: El o los ficheros correspondientes al protocolo se crean o actualizan.

Escenario principal de éxito:

1. El Alumno obtiene una copia del código fuente de la Aplicación proporcionado por el Profesor.

2. El Alumno escribe el código necesario en los métodos vacíos de una de las clases derivadas de la clase Protocolo, que se le presentan en el directorio “student” de la Aplicación, cada fichero corresponde a un protocolo ya sea de parada y espera, rechazo simple o rechazo

6 Sitio web oficial en: http://netbeans.org/

selectivo en el Emisor (Host A) o el Receptor (Host B). El protocolo de parada y espera puede ser bidireccional para que se ejecute en ambos hosts A y B.

2.1 El Alumno define cualquier clase de apoyo a su protocolo que considere conveniente incluyendo sus propiedades y métodos.

Extensiones:

2a. El Alumno decide crear un protocolo personalizado que no corresponde a ninguno de los tres considerados inicialmente.

2a.1 El Alumno crea una nueva clase y por tanto un nuevo fichero que hereda de la clase Protocolo e implementa los métodos necesarios. El formato para el nombre de la nueva clase deber ser:

HostA<nombreProtocolo>Protocol.java, para el protocolo que se ejecutará en el emisor.

HostB<nombreProtocolo>Protocol.java, para el protocolo que se ejecutará en el receptor.

<nombreProtocolo>BidirectionalProcotol.java para un protocolo bidireccional que se ejecutará el mismo en ambos hosts A y B.

2a.2 El Alumno define cualquier clase de apoyo a su protocolo que considere conveniente incluyendo sus propiedades y métodos.

Caso de Uso 002: Simular Protocolo

Actor principal: Alumno.

Precondiciones: El Caso de Uso 001: Definir Protocolo se ha realizado con éxito para el protocolo que se desea simular. No existe ninguna simulación abierta en la Aplicación.

Postcondiciones: La Aplicación genera la lista de eventos que conforman la simulación del protocolo y calcula todas las variables de estado y estadísticas, además se generan la listas de eventos que conforman la animación. Se considera que la simulación ha sido abierta.

Escenario principal de éxito:

1. El Alumno indica a la Aplicación que desea ejecutar una nueva simulación.

2. La Aplicación muestra en una lista de opciones los protocolos definidos.

3. El Alumno selecciona el protocolo que desea observar: ya sea rechazo simple o rechazo selectivo.

4. El Alumno ingresa la cadena de texto que se enviará entre los hosts.

5. El Alumno indica los parámetros de la red para la ejecución: probabilidad de error, probabilidad de pérdida, tiempo promedio entre envíos, número máximo de paquetes a enviar.

6. El Alumno indica que ha terminado de ingresar todos los parámetros solicitados.

7. La Aplicación realiza la simulación del protocolo, generando las listas de eventos que conforman la animación del protocolo y estableciendo valores para todas las variables estadísticas de la simulación.

Extensiones:

3a. El Alumno selecciona el protocolo de parada y espera.

3b. El Alumno selecciona un protocolo genérico e introduce el nombre de la clase que lo define.

3a-b.1 El Alumno puede seleccionar si la comunicación será bidireccional, (para los demás protocolos esta opción no está disponible).

Caso de Uso 003: Animar Protocolo

Actor principal: Alumno.

Precondiciones: El Caso de Uso 002: Simular Protocolo se ha realizado con éxito para el protocolo que se desea analizar.

Escenario principal de éxito:

1. El Alumno inicia la animación del protocolo.
   1.1. En cualquier momento, el Alumno puede pausar, y continuar la animación del protocolo.

2. Conforme en la animación llegan paquetes al receptor la Aplicación muestra los datos que se han entregado a la capa superior.

3. En cualquier momento, el Alumno puede seleccionar un paquete enviado y la Aplicación deberá mostrar información detallada del mismo como: número de secuencia, número de asentimiento, suma de comprobación, carga de datos y tiempo en que salió.

4. Si el protocolo maneja buffers de envío y/o recepción usando los objetos *Manager* proporcionados, la Aplicación mostrará también el estado de esto buffers en el tiempo.

Caso de Uso 004: Cerrar Simulación Protocolo

Actor principal: Alumno.

Precondiciones: El Caso de Uso 002: Simular Protocolo se ha realizado con éxito.

Escenario principal de éxito:

1. El Alumno indica a la Aplicación que desea cerrar la simulación actual.

2. La Aplicación limpia la pantalla y deja de mostrar cualquier animación que se esté mostrando.

3. La Aplicación queda lista para iniciar una nueva simulación.

Caso de Uso 005: Cambiar Idioma

Actor principal: Alumno.

Precondiciones: Ninguna.

Escenario principal de éxito:

1. El Alumno indica a la Aplicación que desea cambiar el idioma que muestra actualmente al interfaz gráfica y selecciona uno diferente.

2. La Aplicación muestra en el idioma seleccionado todas las etiquetas y mensajes visibles y las que no son visibles se muestran en el idioma seleccionado en su debido momento.

Diagrama General de Casos de Uso

El Diagrama General de Casos de Uso para la Aplicación es bastante sencillo , lo podemos apreciar en la Ilustración 7.

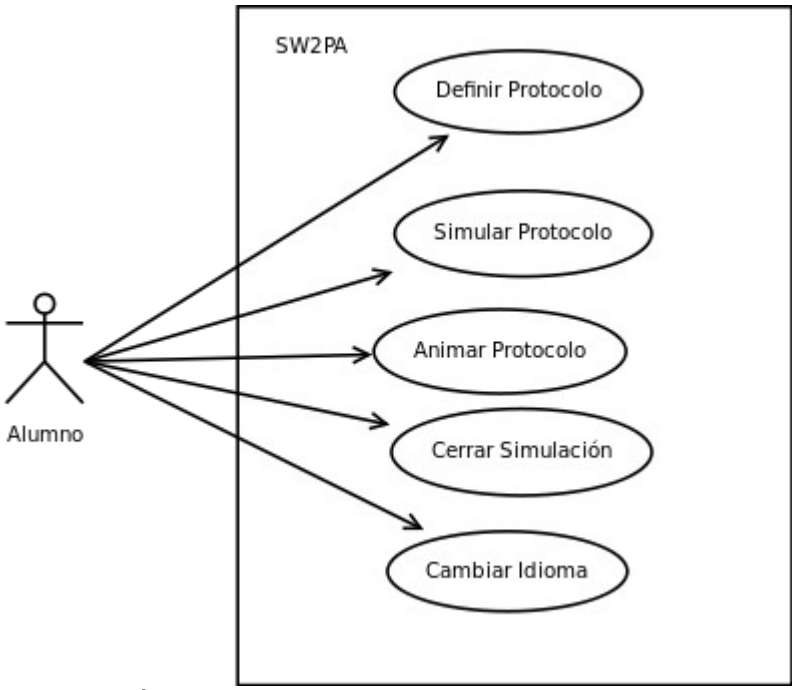


Ilustración 7: Diagrama General de Casos de Uso

Fase de Elaboración

La fase de elaboración es la serie inicial de iteraciones donde se desarrolla la arquitectura central de la aplicación, se definen y clarifican la mayoría de los requerimientos y se mitigan todos los riesgos mayores del proyecto. El UP propone que esta fase debería estar formada por dos o más iteraciones de dos a seis semanas de duración cada una. La duración de las iteraciones depende en gran medida del grupo de trabajo que desarrollará el proyecto, en este caso estuvo formado por una sola persona por lo cual se prefirió iteraciones cortas de dos semanas de duración. Esta fase se dividió en dos iteraciones, organizadas de la siguientes manera:

Iteración Nº1: Núcleo Básico del Simulador, objeto Simulator

Esta se centró en el Caso de Uso 001: Definir Protocolo y en el Caso de Uso 002: Simular Protocolo. Los requisitos cumplidos durante esta etapa fueron los siguientes:

1. Creación del modelo de dominio y diagramas de casos de uso.
2. Diseñar e Implementar el núcleo básico del simulador de eventos discretos.
3. Definir un protocolo de parada y espera para poder probar el simulador.
4. Hacer pruebas al simulador verificando la lista de eventos que genera.
5. Documentar en digital diagramas y algoritmos.

Iniciamos nuestra descripción de la primera iteración con la Ilustración 8: Diagrama de Clases de Diseño, donde observamos todas las clases que conforman la aplicación. Por supuesto todas estas clases no fueron creadas en esta primer iteración pero las mostramos desde ya para ir mejorando la visión global que se va formando de la aplicación.

Durante la actual iteración se inició el desarrollo de las clases *Simulator*, *SimulationEvent, SimulationEventList, SimulationEventType,* la clase abstracta *Protocol,* las clases que heredan de la misma y definen el protocolo de parada y espera: *HostAStopAndWaitProtocol, HostBStopAndWaitProcotol* y *StopAndWaitBidirectionalProtocol;* y la clase *Packet* que define cómo son los paquetes en la aplicación. La creación de las anteriores clases fue muy inspirada en el código del simulador de Jim Kurose mencionado en la sección “Justificación y Alcance“. A continuación revisaremos de nuevo cada caso de uso abordado y comentaremos las decisiones de diseño que se tomaron.

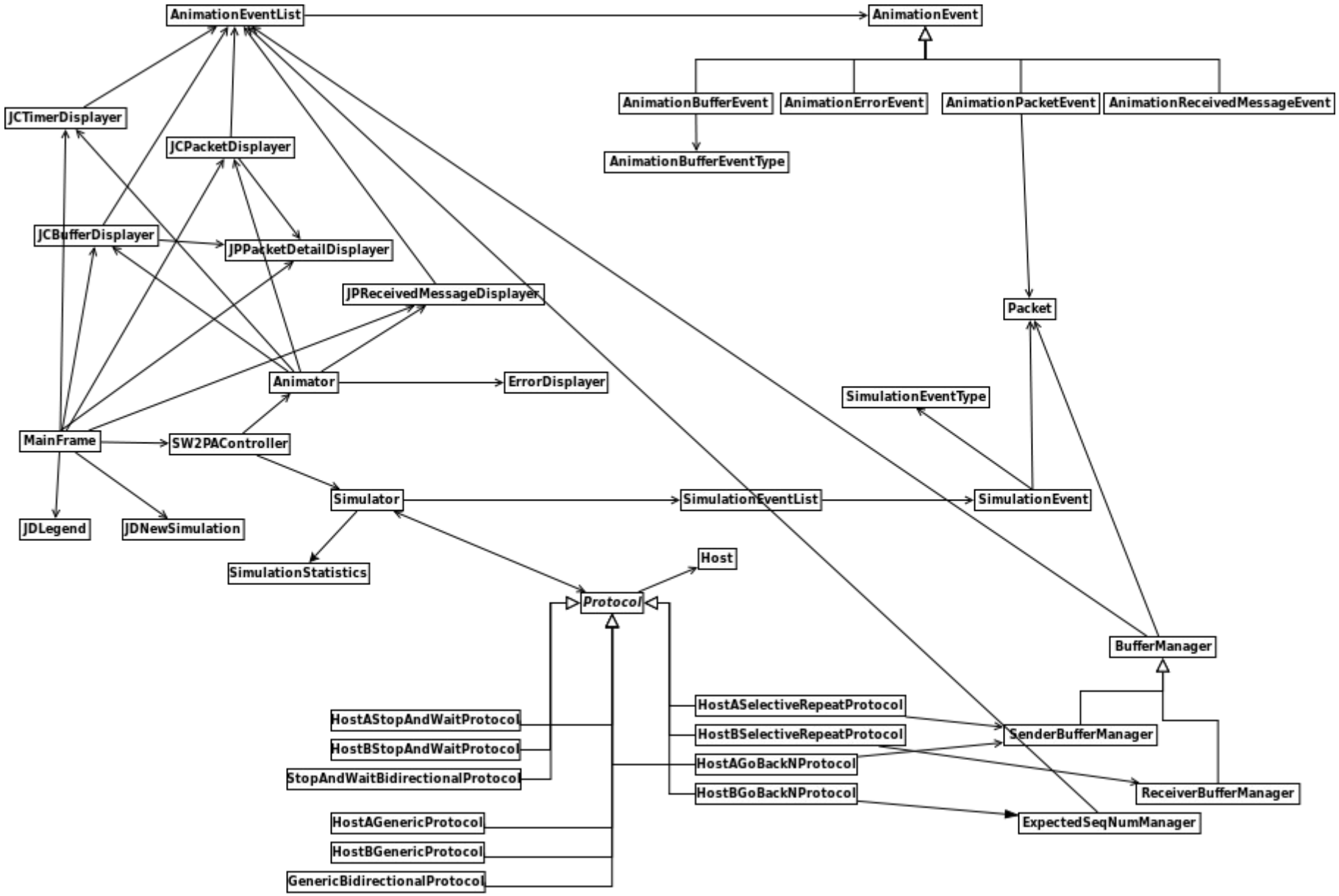


Ilustración 8: Diagrama de Clases de Diseño

Caso de Uso 001: Definir Protocolo

Cualquier protocolo que se quiera definir en la aplicación deberá implementar los métodos abstractos de la clase *Protocol* y deberá invocar al constructor de dicha clase proporcionándole como parámetro una instancia de la clase *Simulator* cuyos métodos públicos le servirán para acceder a la capa inferior y a la superior y manejar los temporizadores. Además un objeto *Protocol* tiene una propiedad *Host* que le permite saber en qué host se está ejecutando, ya sea el host A o el host B. Los métodos a implementar en todo protocolo se presentan a continuación, cabe mencionar que los nombres de los tres métodos están inspirados en los nombres de los eventos de las máquinas de estado finito presentadas para los protocolos en cuestión en Kurose & Ross (2010).

public void rdt_rcv (Packet packet). Este método es invocado cada vez que se recibe un paquete desde la capa inferior, el protocolo deberá decidir qué hacer con el paquete recibido.

public void rdt_send (char[] data). Este método es invocado cuando hay un grupo de datos que la capa superior solicita enviar usando el protocolo.

public void timeout (int timer). Este método se llama cuando se vence un temporizador en el host actual, dicho temporizador está identificado mediante un entero recibido como parámetro.

Como se mencionó antes en esta etapa se definió el Protocolo de Parada y Espera para poder hacer pruebas al simulador, los detalles de la implementación de este protocolo y los otros se muestran en el Capítulo IV Implementación de los Protocolos.

Caso de Uso 002: Simular Protocolo

Este caso de uso es central en la aplicación, el principal objeto que lleva la responsabilidad del mismo es una instancia de la clase *Simulator*, misma instancia a la cual tienen acceso los protocolos en ambos hosts. La clase *Simulator* define un simulador de eventos discretos como el que se describió en el Marco Referencial. El simulador maneja tres tipos de eventos, definidos en la clase tipo enumeración *SimulationEventType*:

FROM_UPPER_LAYER. Este evento es lanzado cuando la capa superior desea enviar datos al protocolo para que los transmita, provoca directamente un llamado al método rdt_send del host correspondiente.

FROM_LOWER_LAYER. Este evento ocurre cuando llega un paquete

desde la capa inferior, esto provoca un llamado al método rdt_rcv del protocolo en el host al cual le llega el paquete.

TIMER_INTERRUPT. Ocurre este evento cuando un temporizador se vence en un host, esto provoca una llamada directa al método timeout indicando el temporizador que se venció.

Todos los eventos son instancias de la clase *SimulationEvent* y todos tienen la propiedad *host* que indica el lugar de su ocurrencia, además se manejan en una lista: *SimulationEventList* se procesan uno a uno según su tiempo de ocurrencia hasta que la lista queda vacía o se alcanza el número máximo de paquetes a enviar definido por el usuario y enviado como parámetro al objeto *Simulator*.

La Ilustración 9 corresponde al Diagrama de Secuencia para el Caso de Uso 002 Simular Protocolo, aquí podemos observar todos los objetos involucrados en este caso de uso y los mensajes que se envían entre los mismos. Vemos que el mensaje que da inicio al caso de uso es *createNewSimulation* el cual es recibido por el objeto único *SW2PAController*, que actúa como controlador de fachada de la aplicación sirviendo de interfaz entre el resto de los objetos y la GUI. Este primer mensaje recibe de la GUI todos los parámetros necesarios para ejecutar una simulación como son: el nombre del protocolo a ejecutar, si es bidireccional o no, el mensaje a enviar, la probabilidad de error y de pérdida, el tiempo promedio entre mensajes enviados de la capa superior al host A y al host B (en caso de que el protocolo sea bidireccional), el tamaño de ventana (para los protocolos de ventana deslizante) y el número máximo de paquetes a enviar, para poder interrumpir la simulación cuando se cumpla. Al recibir el primer mensaje el controlador crea un objeto *Simulator* que a la vez crea la lista de eventos necesaria para ejecutar la simulación: *SimulationEventList* y todas las listas de eventos necesarias para ejecutar la animación, pues a pesar de que en este caso de uso no se ve aprecia la animación, todos los eventos de la animación se crean durante la simulación.

En el caso de los protocolos de ventana deslizante el simulador también crea dos objetos cuyo nombre de clase finaliza en la palabra *Manager* uno para el emisor y otro para el receptor, estos objetos facilitan al alumno el manejo de los buffers y/o el número de secuencia esperado y a la vez crean los eventos de animación necesarios para mostrar de forma dinámica el comportamiento de los buffers y/o el número de secuencia esperado durante la animación. El objeto *Simulator* finalmente crea dos instancias del protocolo solicitado utilizando las clases definidas por el alumno en el caso de uso anterior.

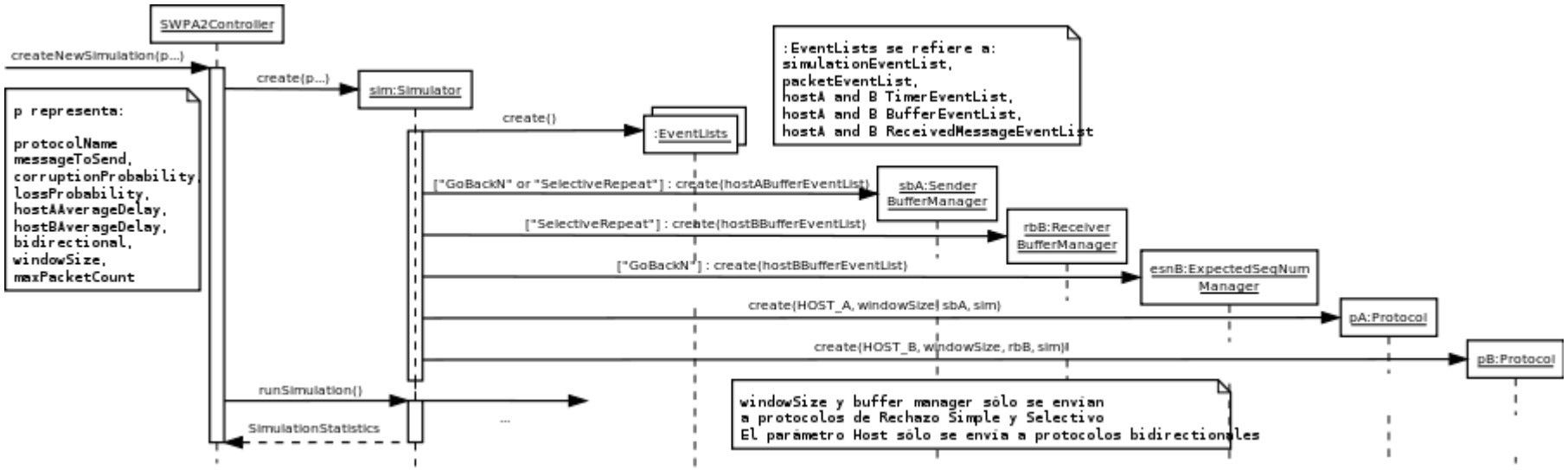


Ilustración 9: Diagrama de Secuencia para el Caso de Uso 002 Simular Protocolo

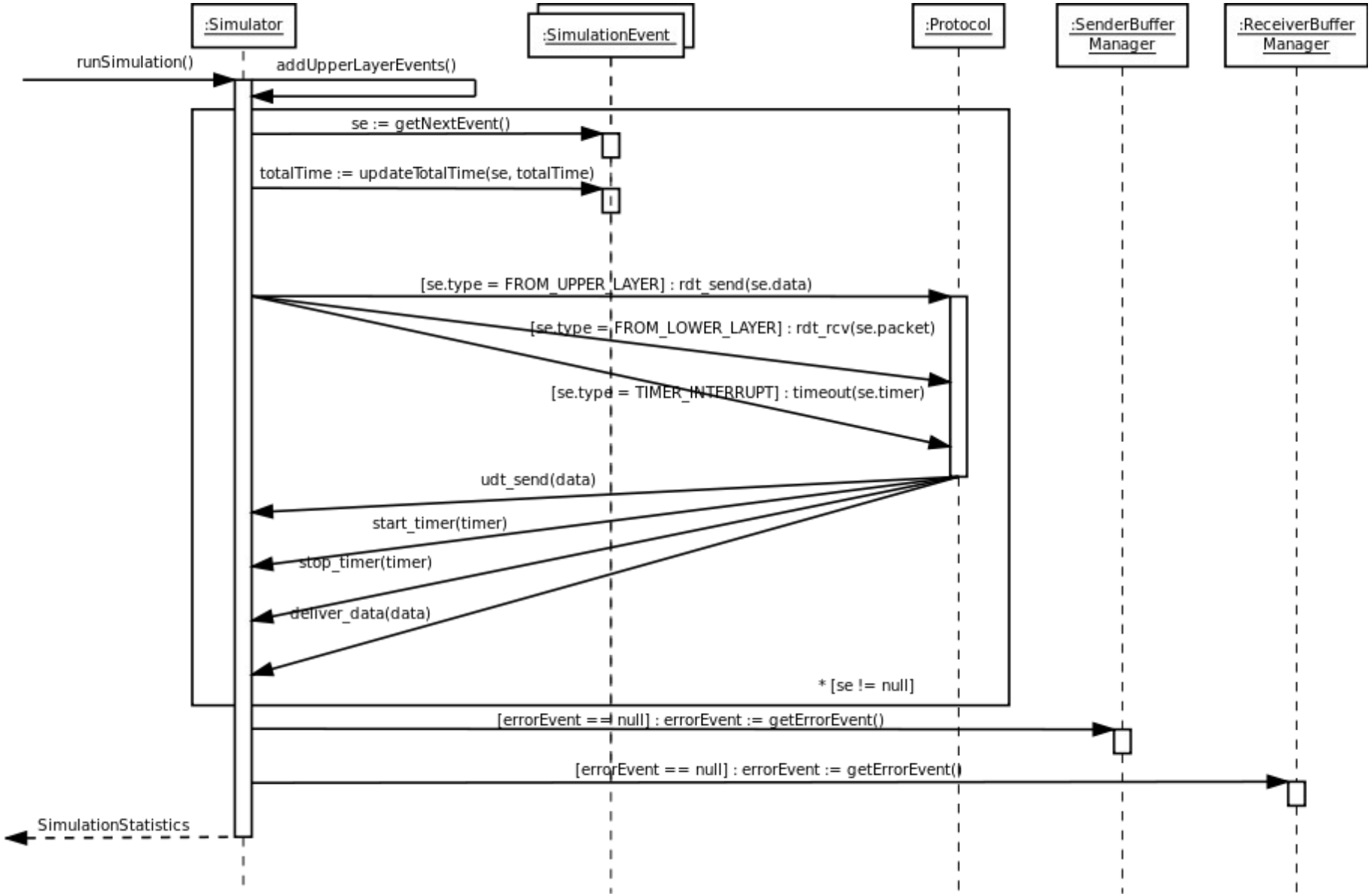


Ilustración 10: Diagrama de Secuencia para el Método runSimulation

Método runSimulation

Una vez que el objeto *Simulator* ha sido creado, el controlador le envía el mensaje *runSimulation* que es el que realmente inicia la simulación y cuyo Diagrama de Secuencia se muestra en la Ilustración 10. Como observamos este diagrama sigue el algoritmo de un simulador de eventos discretos que va procesando uno a uno los eventos *SimulationEvent* de la lista de eventos. El primer mensaje en ejecutarse es *addUpperLayerEvents* que como su nombre lo indica agrega todos los eventos del tipo FROM_UPPER_LAYER separándolos en el tiempo según el tiempo promedio de envío de mensajes enviado como parámetro al objeto *Simulator*. Como antes se mencionó un evento FROM_UPPER_LAYER provoca que se llame al método *rdt_send* del protocolo y desde este método o cualquier otro el protocolo puede invocar a cualquiera de los cuatros métodos públicos del simulador:

public void udt_send (Host host, Packet packet). Este método se usa para intentar enviar un paquete *packet* desde el host indicado hacia el otro, es aquí donde se puede corromper o perder los paquetes. Este método también es responsable de crear un evento FROM_LOWER_LAYER en el host destinatario del paquete. Puede consultarse el Diagrama de Flujo de este método en la Ilustración 11.

public void start_timer (Host host, int timer, int delay). Este método se usa para iniciar un temporizador en el host indicado que vence dentro de delay unidades de tiempo. Se crea un evento TIMER_INTERRUPT en el host indicado o se genera un evento de error si el temporizador ya existe. La Ilustración 12 muestra el Diagrama de Flujo de este método.

public void stop_timer (Host host, int timer). Este método detiene un temporizador existente en el host indicado, o se genera un evento de error si el temporizador no existe. Se busca en *SimulationEventList* y se elimina el evento TIMER_INTERRUPT correspondiente. Puede consultar la Ilustración 12 para ver el Diagrama de Flujo de este método.

public void deliver_data (Host host, char[] data). Este método se usa para entregar datos a la capa superior en el host indicado, no genera un evento de simulación. La Ilustración 13 muestra un Diagrama de Flujo para este método.

Hay que mencionar que los eventos no tienen porque agregarse en orden a la lista de simulación, y es la lista la responsable de mantener el orden según el tiempo de ocurrencia los eventos que se le agregan, para dar un ejemplo: supongamos que el primer evento de la lista en el tiempo 2 es un evento FROM_UPPER_LAYER, y el segundo en el tiempo 5 otro evento del mismo tipo

y el tercero en el tiempo 8 del mismo tipo, al procesar el primer evento el protocolo invoca al método udt_send y éste hace que se cree un evento del tipo FROM_LOWER_LAYER en el host opuesto y este paquete llegará en el tiempo 6, por tanto al agregar este evento la lista de simulación colocaría este evento después del que tiene tiempo 5 y antes del que tiene tiempo 8.

En el diagrama de la Ilustración 10 se observa que antes de que cada evento se procese el tiempo total o actual de la simulación se actualiza de forma que cualquier nuevo evento que se genere se realizará con respecto a este, por ejemplo si el tiempo actual es 120 y se crea un temporizador que vence en 12 unidades de tiempo, entonces el tiempo para este nuevo evento TIMER_INTERRUPT será 132.

Finalmente el método *runSimulation* devuelve un objeto del tipo *SimulationStatistics* que recoge estadísticas básicas de la simulación como son el tiempo total, el número de paquetes enviados, el número de paquetes perdidos, corruptos y el número de mensajes entregados al protocolo y recibidos en su destino.

En las siguientes páginas mostramos los diagramas de flujo para los cuatro métodos públicos del objeto *Simulator,* recalcando que no todos los pasos fueron programados durante la primera iteración y por tanto algunos de estos pasos se seguirán describiendo en las posteriores iteraciones.

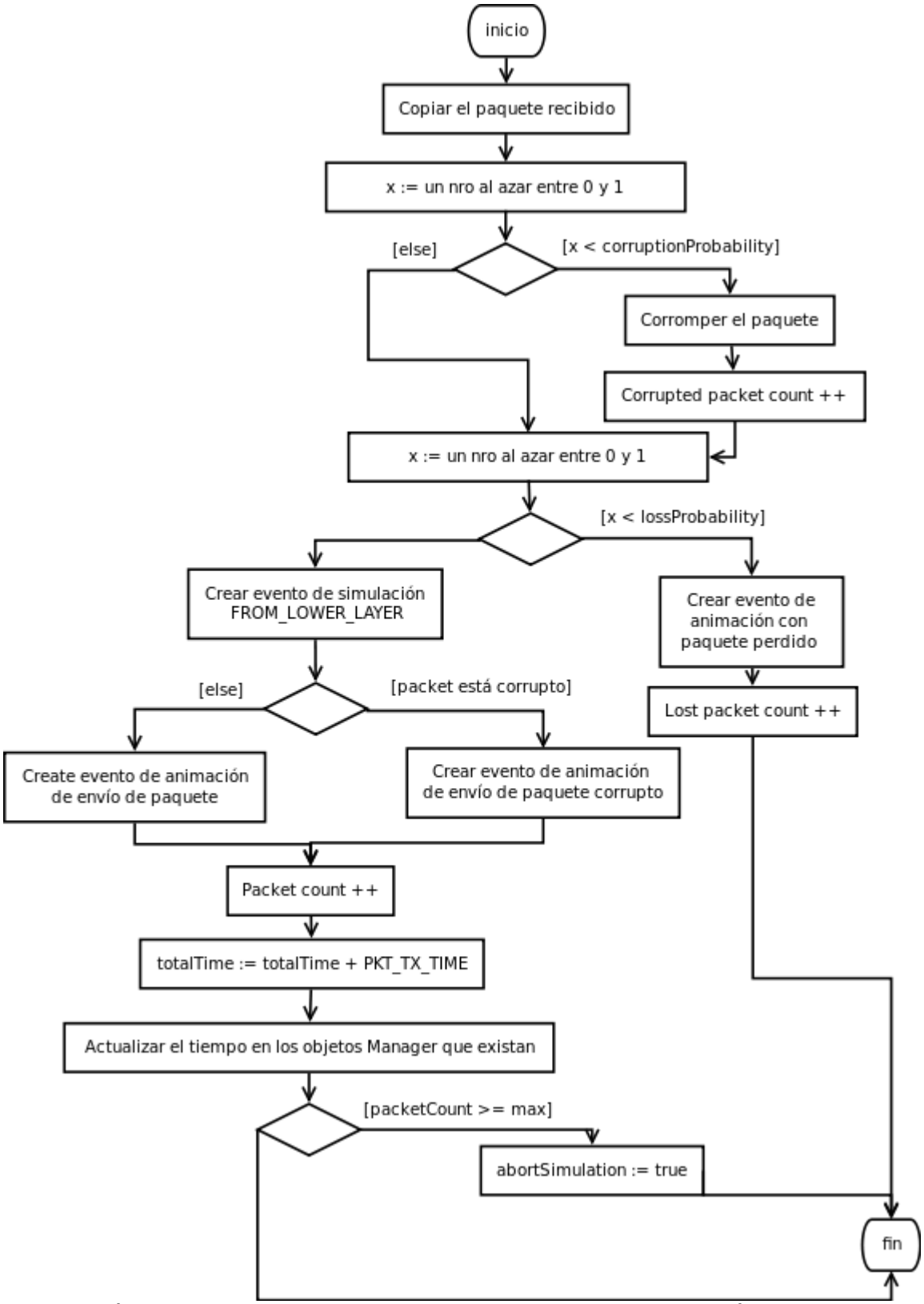
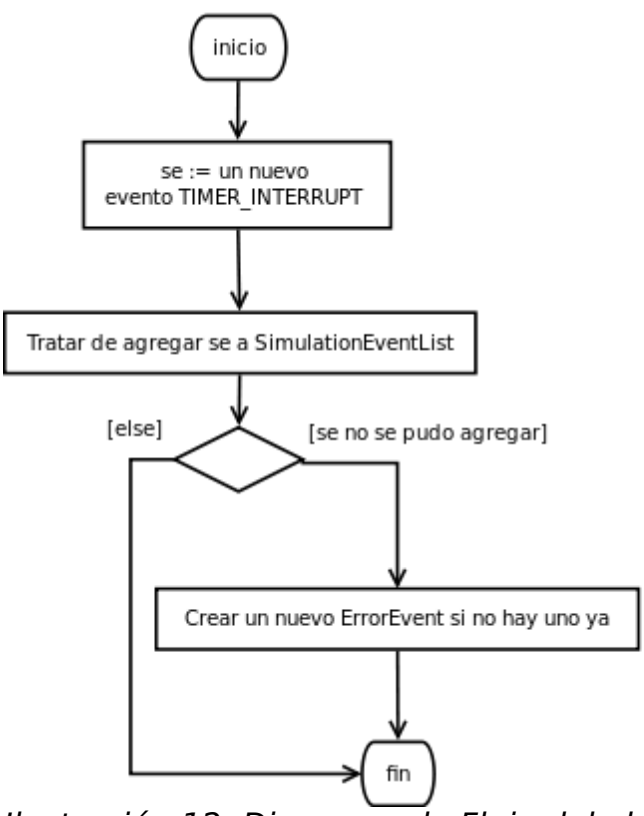


Ilustración 11: Diagrama de Flujo del objeto Simulator Método udt_send



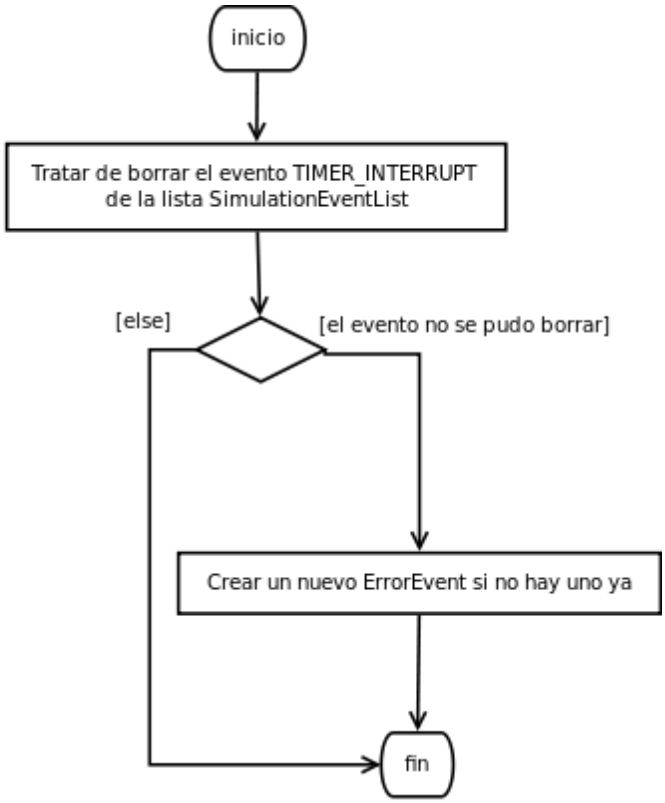


Ilustración 12: Diagrama de Flujo del objeto Simulator Métodos start_timer y stop_timer

Ilustración 13: Diagrama de Flujo del objeto Simulator Método deliver_data

Iteración Nº2: Núcleo Básico de la Animación, objetos Animator y JCPacketDisplayer

Esta etapa se centró en el Caso de Uso 003: Animar Protocolo. Los requisitos cumplidos durante esta etapa fueron los siguientes:

1. Continuar el diseño e implementación del núcleo básico del simulador de eventos discretos para que genere los eventos de la animación, necesarios para animar el protocolo de parada y espera.
2. Diseñar e Implementar el núcleo básico del animador que procese los eventos de la animación y muestre gráficamente el envío de paquetes a través de un *JComponent*.
3. Hacer pruebas comparando la animación que se genera con la lista de eventos en texto de la simulación.
4. Documentar en digital diagramas y algoritmos.

El primer requisito de esta iteración nos llevó a refinar el Caso de Uso 002: Simular Protocolo y nos recuerda que a pesar de su nombre es durante este caso de uso donde se generan todos los eventos de la animación. Es por esto, que el objeto *Simulator* no tiene únicamente la responsabilidad de ejecutar la simulación sino también llenar varias listas de eventos del tipo *AnimationEventList*. Un objeto de la clase *AnimationEventList* contiene objetos del tipo *AnimationEvent*, existen, según se ve en la Ilustración 8: Diagrama de Clases de Diseño, cinco tipos de eventos de la animación que describimos a continuación, todos los eventos heredan de la clase *AnimationEvent* y por tanto todos tienen la propiedad *time* para indicar el tiempo en que ocurren:

AnimationBufferEvent. Este evento indica que ocurrió algo en uno de los buffers, se manejan dos listas *AnimationEventList* una para los eventos en el buffer del host A en los protocolos de ventana deslizante y otra para los del buffer en el host B (sólo en el protocolo de rechazo selectivo). Existen varios tipos de eventos en los buffers, los tipos están definidos por la clase enumeración *AnimationBufferEventType* y son los siguientes:

SNDR_SET_BASE_INDEX. Indica un cambio en el índice de la base de la ventana en un buffer emisor.

SNDR_SET_NEXT_PKT_INDEX. Indica un cambio en el índice que apunta hacia la posición donde está o estará el siguiente paquete a ser enviado en el emisor.

SNDR_ADD_PACKET. Indica que se agregó un paquete al buffer del emisor que se maneja como una lista circular primero en entrar primero en salir.

RCVR_SET_EXPECTED_SEQ_NUM. Indica un cambio en el siguiente número de secuencia esperado por el receptor.

RCVR_SET_BASE_INDEX. Indica un cambio en el índice de la base de la ventana en un buffer receptor.

RCVR_ADD_PACKET. Indica que se agregó un paquete al buffer del receptor

RCVR_REMOVE_PACKET. Indica que se borró un paquete del buffer del receptor, los paquetes en el buffer emisor no se borran pues se considera que al mover el índice de la base de la ventana los paquete que quedan tras la ventana ya no forman parte del buffer.

AnimationPacketEvent. Este evento indica el envío de un paquete desde un host a otro en un tiempo determinado, además indica si el paquete que se envía está corrupto o es un paquete que se perdió. Como se aprecia en el Diagrama de Flujo de la Ilustración 11 es el método *udt_send* el que crea estos eventos.

AnimationReceivedMessageEvent. Este evento indica la entrega de de un mensaje de datos a la capa superior en uno de los hosts.

AnimationErrorEvent. Este evento indica un número de error ocurrido en cierto tiempo. Este es el único evento que por simplicidad no se almacena en una lista, debido a esto sólo se almacena un evento de error: el primero que ocurra.

AnimationEvent. Esta es la clase padre de todas las anteriores y también se utilizan instancias de la misma para indicar el vencimiento de un temporizador, los eventos de este tipo son agregados por el objeto *Simulator* como parte del proceso de un evento de simulación de tipo TIMER_INTERRUPT. Gráficamente estos eventos se muestran como pequeños relojes en el objeto GUI *JCTimerDisplayer.*

Los eventos anteriores se almacenan en las siguientes listas, cuyos nombres son bastante explícitos para explicar qué tipo de eventos almacenan.

packetEventList,

hostATimerEventList,

hostBTimerEventList,

hostABufferEventList,

hostBBufferEventList,

hostAReceivedMessageEventList y

hostBReceivedMessageEventList.

Antes de continuar es importante observar el Diagrama de Secuencia para el Caso de Uso 003 Animar Protocolo de la Ilustración 14. Una vez que ha ejecutado el método *runSimulation* el usuario puede solicitar a través de la GUI al controlador para que inicie la animación, esto hace que se invoque al método *startAnimation* del controlador que comienza enviando datos de inicialización a una serie de objetos que cumplen un rol vital en la animación, pues todos son miembros de la interfaz gráfica de la aplicación y son capaces de mostrar diferentes estados gráficos según el paso del tiempo. En las posteriores iteraciones se explicarán uno a uno, por el momento nos concentraremos en el objeto que sirve para mostrar los paquetes que van y vienen entre el host A y el host B, el *JCPacketDisplayer.* Continuando la explicación del Diagrama de Secuencia para el Caso de Uso 003 Animar Protocolo, vemos que el controlador después de enviar los datos de inicialización a los objetos mencionados previamente, crea un objeto de la clase *Animator,* este objeto a diferencia del *Simulator* que tenía toda la responsabilidad de la simulación, se encarga únicamente de coordinar la animación, su componente principal es un objeto de la clase *javax.swing.Timer* que se crea cuando el controlador invoca el método *runAnimation* del *Animator* y que aproximadamente cada 50 ms emite una interrupción invocando al método *actionPerformed* que es desde donde el objeto *Animator* coordina la animación.

Coordinar la animación por parte del *Animator* consiste simplemente en invocar diferentes métodos en los objetos de la GUI que los llevan a conocer el tiempo actual y determinar según la lista de eventos de animación que cada uno posee si necesitan cambiar su estado visual o no, también consiste en que cuando se cumpla el tiempo total de la animación (es mismo tiempo total de la simulación) o el usuario provoque la invocación del método *pauseAnimation* el *Timer* debe detenerse y por tanto la animación también. Finalmente cuando el usuario a través del control *JSlider* (barra de deslizamiento) y el método *playAnimationFrom* solicite iniciar la animación desde otro punto, el *Animator* puede calcular el tiempo transcurrido a este punto, iniciar el *Timer* sino está corriendo y actualizar a los objetos de la GUI con el tiempo calculado.

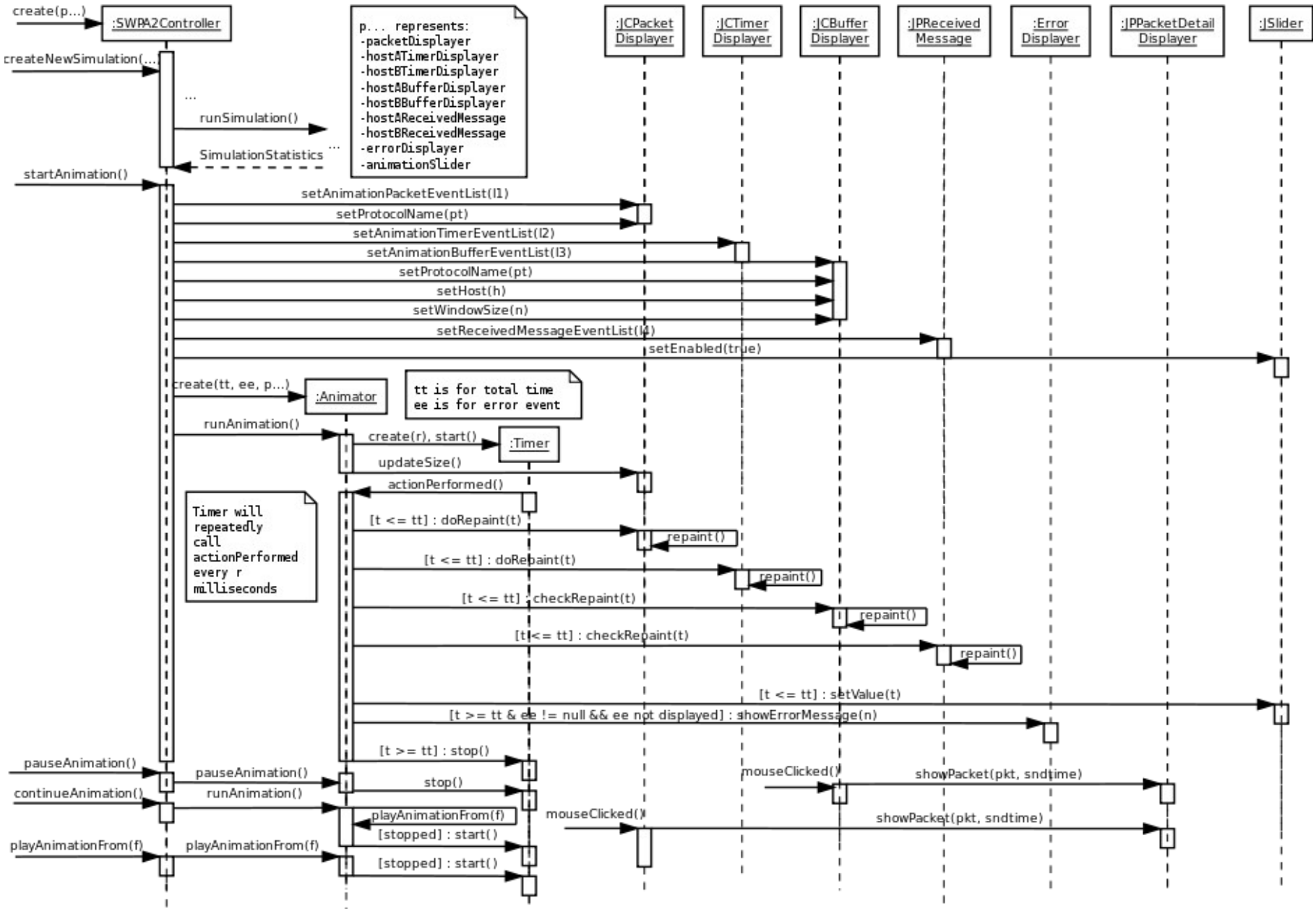


Ilustración 14: Diagrama de Secuencia para el Caso de Uso 003 Animar Protocolo

JCPacketDisplayer

Como acabamos de mencionar, el *JCPacketDisplayer* es el objeto que se encarga de mostrar los paquetes que han viajado o están viajando desde un host a otro durante la animación, esto se logra a través de un diagrama como el que muestra la Ilustración 15.

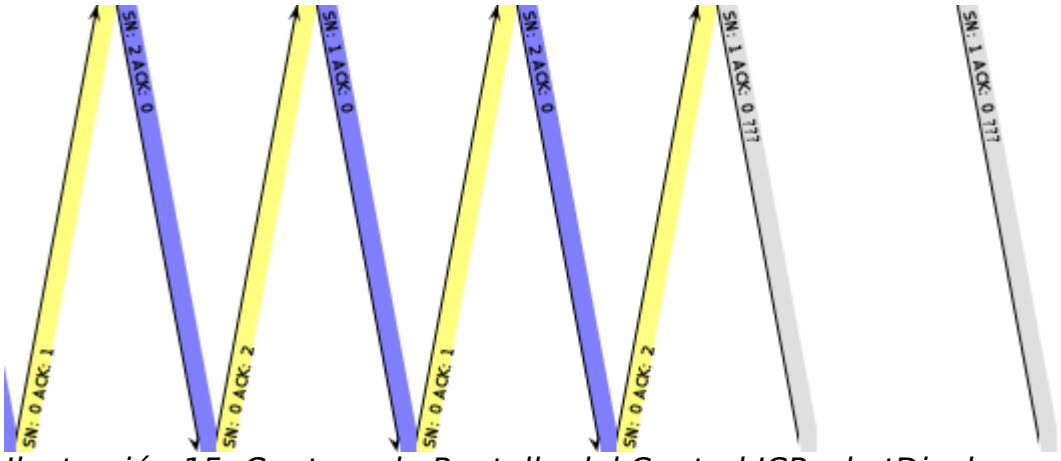


Ilustración 15: Captura de Pantalla del Control JCPacketDisplayer

Para poder modificar el aspecto visual de un *JComponent* es necesario sobreescribir su método *paintComponent* e indicar cómo se va a mostrar el componente, pero para que este método puede dibujar correctamente necesita conocer el tiempo actual. Es por esto que cada vez que el tiempo cambia el *Animator* invoca al método *doRepaint* del *JCPacketDisplayer,* éste se encarga principalmente de actualizar la propiedad *time* del componente, y llamar al método *repaint* que hace que se llame al deseado método *paintComponent*. En la Ilustración 18 vemos el Diagrama de Flujo del objeto JCPacketDisplayer Método paintComponent, dentro de este diagrama llama la atención el paso en el cual se dibuja una parte de un paquete en un área delimitada. La razón para que se tenga que hacer esto es que se desea dar la apariencia de que el paquete va avanzando poco a poco en el tiempo, además debido a que todas la operaciones gráficas se realizan primero en una imagen de tipo *BufferedImage* y luego se dibuja la imagen en el objeto, no es necesario dibujar una parte del paquete que ya ha sido dibujada[7].

El paso en el cual se dibuja un paquete es en realidad una llamada al método *drawPacket* cuyo Diagrama de Flujo se muestra en la Ilustración 19, en esta Ilustración los primeros dos pasos son un poco complejos y por ello hacemos

7 Durante la programación se encontró dificultades propias el Antialising de Java que hicieron necesario pintar partes ya dibujadas aumentando considerablemente el área delimitada pues el pintar de áreas nuevas las afectaba indirectamente.

referencia a ellos a continuación.

1. Determinar los cuatro puntos que forman el cuerpo del paquete dentro del área solicitada. Cada evento de tipo *AnimationPacketEvent* tiene un tiempo de envío que se tomó como el tiempo en el cual se empieza a enviar el primer bit del paquete y un tiempo de llegada que es el tiempo en el cual llega el último bit del paquete a su destino. Definiendo una constante de conversión podemos convertir estos tiempos en pixeles y tener un punto de inicio a partir del tiempo de envío y un punto de fin a partir del tiempo de llegada. De esta forma podemos representar un paquete completo mediante un paralelogramo cuyas rectas paralelas más largas se pueden representar mediante ecuaciones lineales. Usando la forma punto pendiente, tomando un punto en el eje Y tenemos:

   $y = m(x - x_1)$, y definiendo

   $$m0 = \frac{altura\ del\ componente}{punto\ de\ fin - ancho\ del\ paquete - punto\ de\ inicio}$$

   Entonces, para un paquete que va desde el host A hasta el B su pendiente sería *m=m0,* en la primera recta x_1*=punto de inicio*, en la segunda recta x_1*=punto de inicio + ancho del paquete*. Y para un paquete que va desde del host B al host A su pendiente sería negativa, *m=-m0,* en la primera recta x_1*=punto de fin - ancho del paquete* y en la segunda recta x_1*=punto de fin*. Lo anterior lo podemos ver más claramente en la Ilustración 16 que muestra un paquete que va del host A al B, tomando en consideración que el eje Y aumenta hacia abajo.

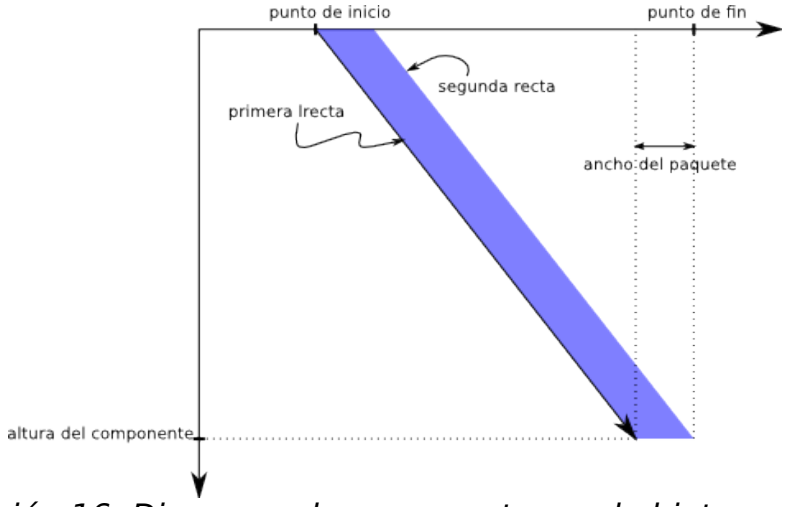


Ilustración 16: Diagrama de un paquete en el objeto JCPacketDisplayer

Cuando el área solicitada incluye toda el área del paquete determinar los cuatro puntos es directo, el p1 es el punto de inicio, p2 es el punto de fin menos el ancho del paquete a la altura del componente, p3 el punto de fin a la altura del paquete y p4 es el punto de inicio más el ancho del paquete. Pero se puede dar el caso que al área solicitada no incluya completamente el paquete y éste no termine y/o no inicie dentro del área como lo muestra la Ilustración 17.

En este caso es muy importante conocer las ecuaciones de las rectas para poder determinar las componentes en Y de los puntos que se encuentran al borde del área solicitada.

2. Determinar el color del cuerpo del paquete. Se decidió que los paquetes corruptos no importando de que host vengan serían de color gris oscuro, los paquetes perdidos de color gris claro, los paquetes que sean asentimientos serían de color amarillo y el resto de paquetes si vienen desde el host A serán azules y si vienen del host B serán rojo claro. Es importante mencionar que un paquete se considera como asentimiento si el protocolo que se ejecuta es uno de los tres definidos en la aplicación, es decir si no es un protocolo personalizado y si el número que va en su campo ACK es diferente de cero.

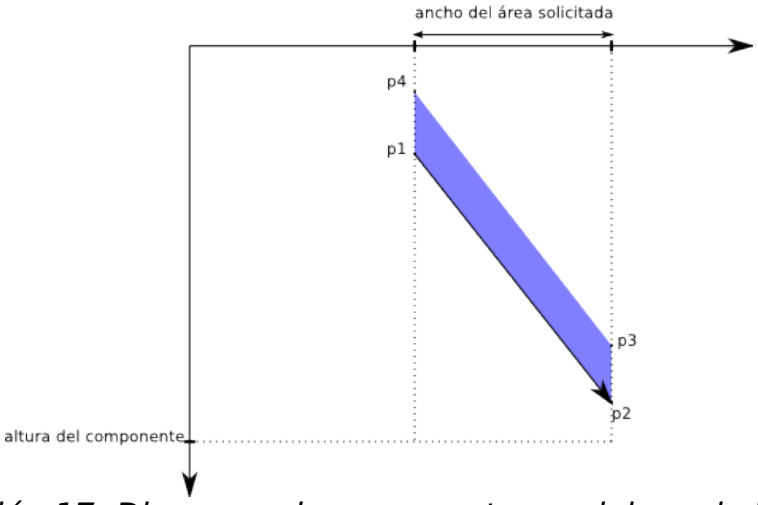


Ilustración 17: Diagrama de un paquete parcial en el objeto JCPacketDisplayer

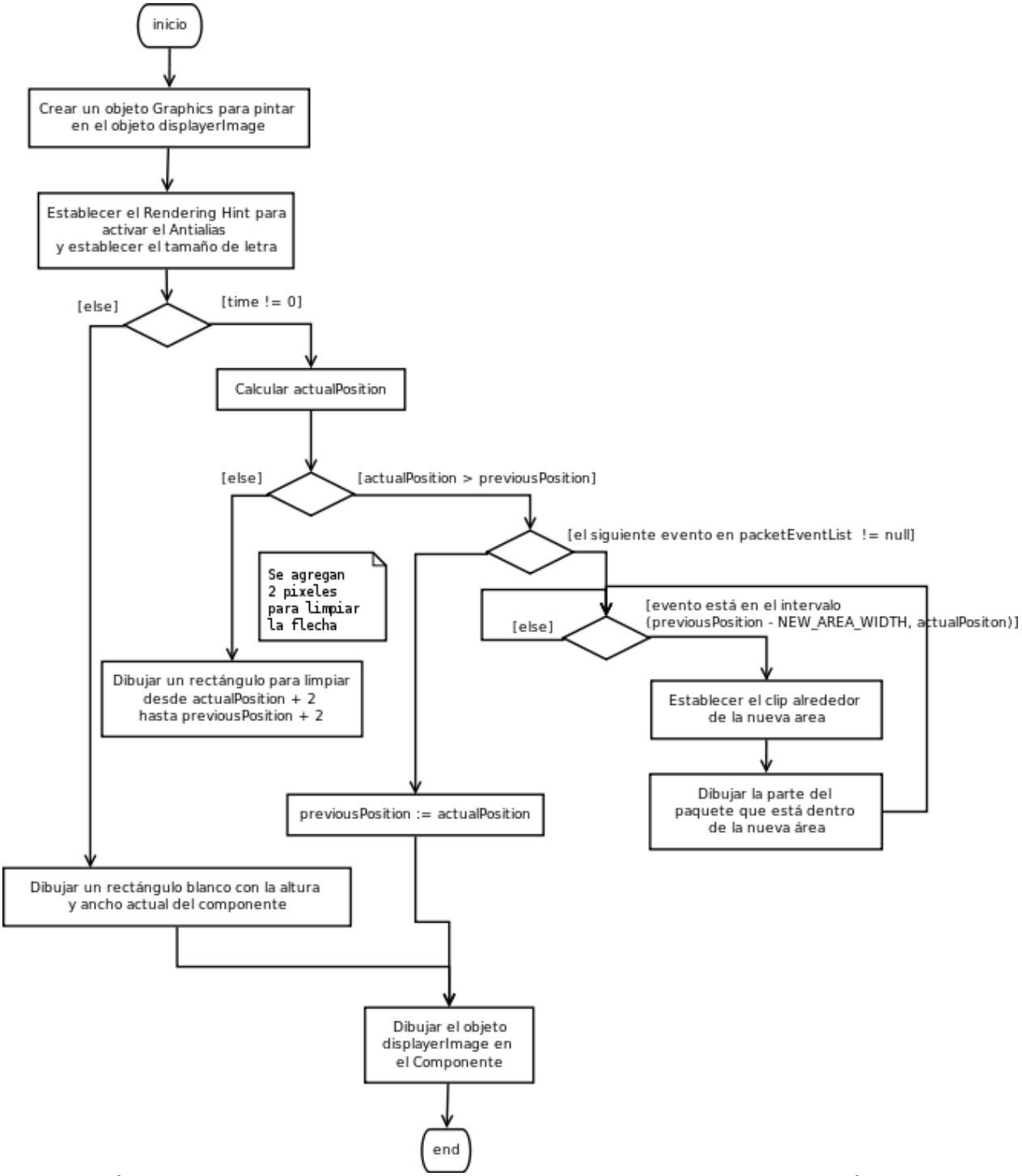


Ilustración 18: Diagrama de Flujo del objeto JCPacketDisplayer Método paintComponent

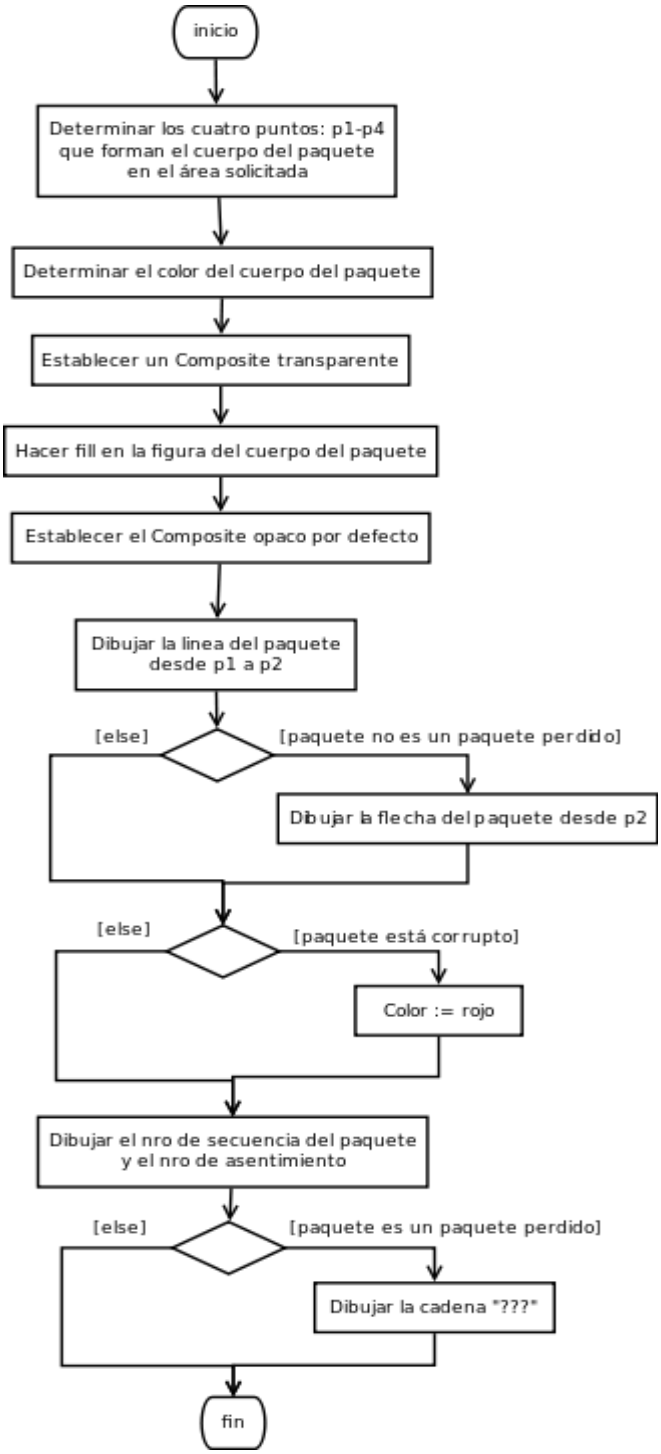


Ilustración 19: Diagrama de Flujo del objeto JCPacketDisplayer Método drawPacket

Fase de Construcción

En la fase de construcción ya se cuenta con un núcleo central estable de la aplicación, se procede de forma iterativa al desarrollo del resto de los elementos que también son importantes, pero que implican un menor riesgo en el proyecto. Esta fase se desarrolló en cuatro iteraciones de aproximadamente dos semanas cada una. Al final de cada iteración se debía tener un prototipo ejecutable de la aplicación.

Iteración Nº3: Continuando la Animación, objetos JCTimerDisplayer, JPReceivedMessageDisplayer y JCPacketDetailDisplayer

Esta iteración da inicio a la fase de Construcción y se centró nuevamente en el Caso de Uso 003: Animar Protocolo. Los requisitos a cumplidos en esta etapa fueron los siguientes:

1. Diseñar e implementar los controles de la interfaz de usuario para mostrar los temporizadores vencidos, el detalle de un paquete seleccionado y los mensajes recibidos en la capa superior en cada host.
2. Hacer pruebas verificando la animación que se genera con la lista de eventos en texto.
3. Documentar en digital diagramas y algoritmos.

Comenzamos la descripción de esta iteración recordando el Diagrama de Secuencia para el Caso de Uso 003 Animar Protocolo presentado en la Ilustración 14, donde se veía cómo el objeto *Animator* envía datos de inicialización a una serie de objetos de la GUI antes de ejecutar la animación, en esta iteración se elaboró dos de estos objetos y un tercero que no necesita datos de inicialización y al cual el objeto *Animator* no accede directamente, el objeto *JPPacketDetailDisplayer*, a continuación comentamos con detalle cada uno de estos tres objetos.

JCTimerDisplayer

La interfaz gráfica de la aplicación cuenta con dos instancias de esta clase, cada una sirve para mostrar un pequeño reloj justo en el momento y ubicación que corresponde a cada vencimiento de temporizador ya sea en el host A o en el host B. Para mostrar los eventos en cada host existen los objetos *hostATimerDisplayer* y *hostBTimerDisplayer* respectivamente. La ubicación de estos dos objetos es justo sobre los bordes superior en inferior

del objeto *JCPacketDisplayer*, como lo muestra la Ilustración 20. El nombre este objeto así el como el del *JCPacketDisplayer* inicia con las letras JC debido a su clase padre: *JComponent.*

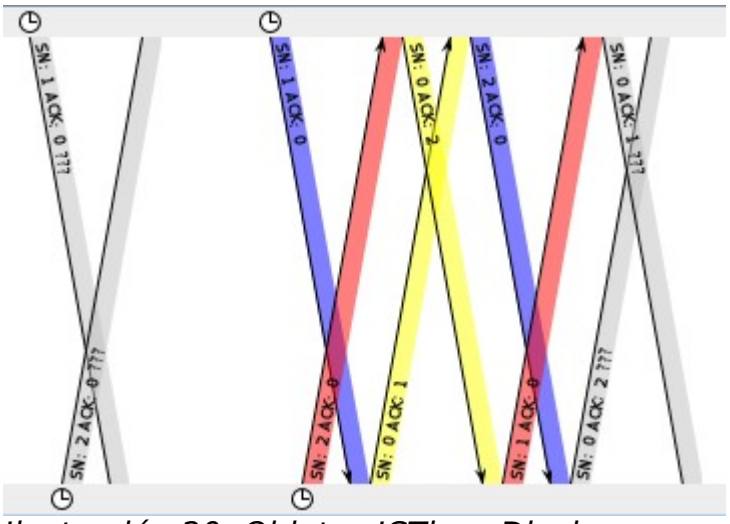


Ilustración 20: Objetos JCTimerDisplayer

Cada objeto *JCTimerDisplayer* cuenta con su propia lista de eventos *AnimationEventList* que almacena eventos del tipo *AnimationEvent*, un evento de este tipo cuenta sólo con un valor numérico para indicar el tiempo en el que se venció el temporizador. La lista de eventos es el único dato de inicialización que necesita un *JCTimerDisplayer* para mostrar su animación.

A diferencia del objeto *JCPacketDisplayer* una instancia de la clase *JCTimerDisplayer* no necesita actualizar su estado visual siempre que el tiempo cambie. Por esta razón el objeto *Animator* no invoca a un método llamado *doRepaint* sino al método *checkRepaint* en el *JCTimerDisplayer* que se encarga de verificar si dado el tiempo actual es necesario repintar el objeto o no.

En la Ilustración 21 vemos el Diagrama del Flujo del objeto JCTimerDisplayer Método checkRepaint. Hacemos especial mención al segundo paso en el algoritmo que nos lleva a mover un índice *currentIndex* hasta que apunte al último evento que debió haber sido pintado según el tiempo actual, en muchas ocasiones cuando se actualice el tiempo este evento será el mismo y *currentIndex* será igual *previousIndex* por lo que no habrá necesidad de pintar de nuevo el objeto, pero cuando se reciba un tiempo que haga que *currentIndex* cambie entonces significa que el estado visual debe cambiar y por tanto se invoca al método *repaint* para que se llame al método *paintComponent* cuyo Diagrama de Flujo observamos en la Ilustración 22.

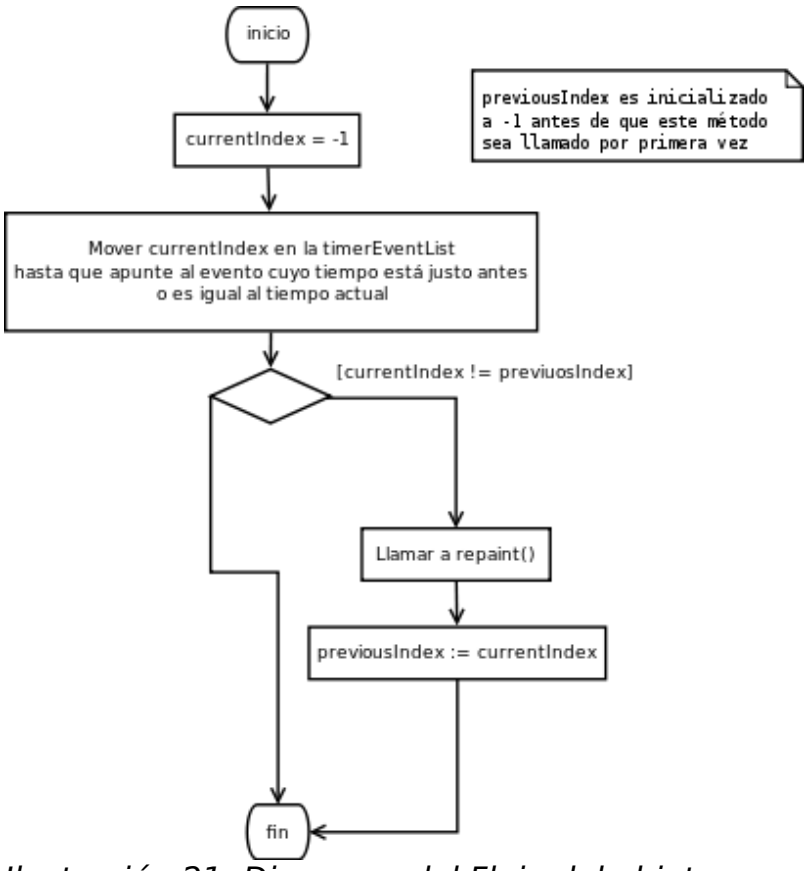


Ilustración 21: Diagrama del Flujo del objeto JCTimerDisplayer Método checkRepaint

Con respecto al método *paintComponent* del objeto *JCTimerDisplayer* observamos primero que no se pinta nada mientras el *currentIndex* sea igual a -1. Este valor se mantendrá hasta que inicie la animación y el método *checkRepaint* lo cambie. Además vemos que como los gráficos a dibujar son sencillos, cada vez que se pinta, lo que se hace es pintar todos los relojes (temporizadores vencidos) que sean visibles hasta el tiempo actual, es decir que estén dentro del Clip. Cuando se retrocede la animación bruscamente usando el *animationSlider s*e puede dar el caso que el tiempo actual tenga una posición fuera de los límites del Clip, para este caso se usa la última condición que interrumpe el ciclo si se encuentra con una posición a la derecha fuera del Clip.

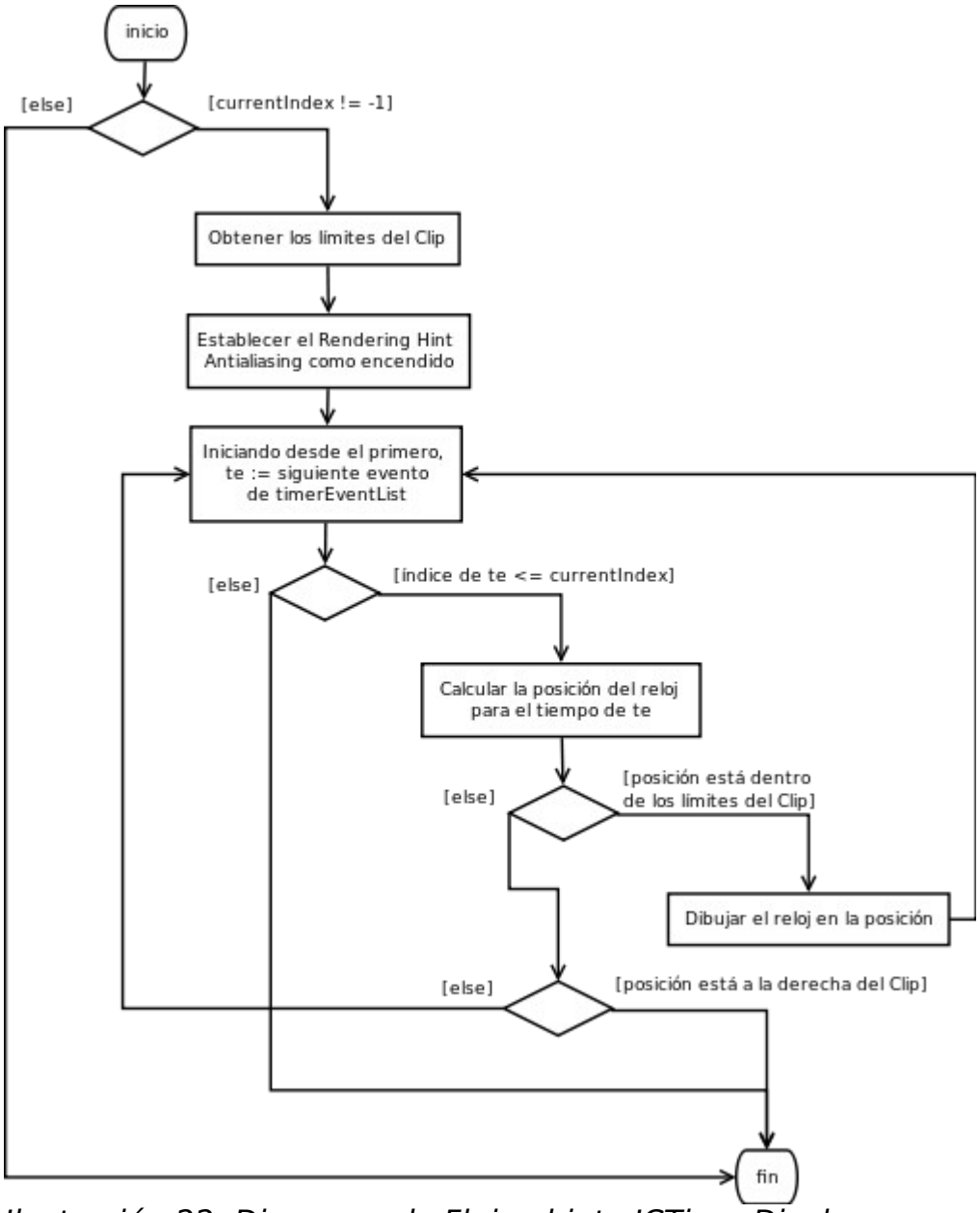


Ilustración 22: Diagrama de Flujo objeto JCTimerDisplayer Método paintComponent

JPReceivedMessageDisplayer

Tal cual su nombre lo indica un objeto *JPReceivedMessageDisplayer* se utiliza para mostrar en la animación los mensajes recibidos en la capa superior en cualquiera de los dos hosts, por esto de la misma forma que el *JCTimerDisplayer,* la interfaz gráfica usa dos instancias este objeto, en este caso este objeto hereda de la clase *JPanel* pues contiene dentro de sí mismo un *JScrollPane* que a su vez contiene un *JTextArea.* Además como dato de inicialización para poder mostrar la parte de la animación que le corresponde necesita una lista de eventos de animación con objetos del tipo *AnimationReceivedMessageEvent* que está formados por dos valores: el tiempo de ocurrencia y una cadena con los datos entregados en dicho tiempo. Los objetos *JPReceivedMessageDisplayer* se puede encontrar en la parte inferior derecha de la ventana principal y su apariencia se ve en la Ilustración 23.

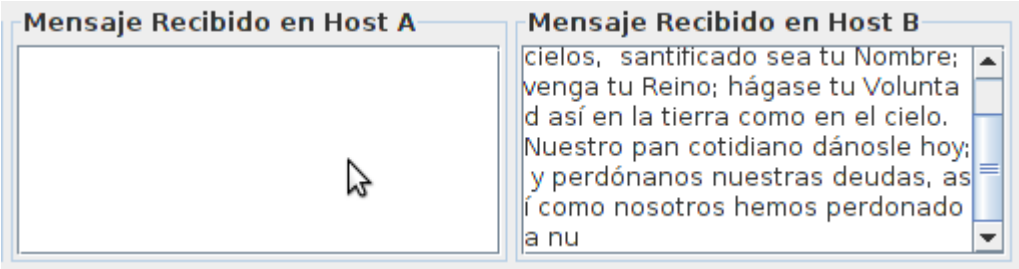


Ilustración 23: Objetos JPReceivedMessageDisplayer

Al igual que ocurre con los objetos *JCTimerDisplayer* los objetos *JPReceivedMessageDisplayer* no necesitan actualizarse cada vez que cambia el tiempo, para manejar esto estos objetos también cuentan con un método *checkRepaint* casi idéntico al mostrado en la Ilustración 21, con la salvedad de que ahora como no heredamos directamente del objeto *JComponent* no se llama al método *repaint* sino a un método definido en el objeto mismo cuyo nombre es *updateDisplayedMessage* y que claramente se ve que sirve para actualizar el mensaje mostrado. En detalla la función de este método consiste en limpiar el mensaje actual y luego recorrer en la lista de eventos hasta el que está justo antes del tiempo actual o tiene un tiempo igual al actual, cada vez que se pasa por un evento sus datos se añaden al mensaje mostrado por el objeto *JTextArea*.

JPPacketDetailDisplayer

El objeto *JPPacketDetailDisplayer* es bastante sencillo no necesita datos de inicialización se utiliza únicamente para mostrar de forma detallada el

contenido de un paquete mostrado gráficamente por el objeto *JCPacketDisplayer*. Cuenta con dos métodos muy simples, el primero llamado *showPacket* recibe un paquete y su tiempo de envío como parámetro y usa estos valores para mostrarlos en las etiquetas que lo conforman, la apariencia de este control se muestra en la Ilustración 24.

Ilustración 24: Objeto JPPacketDetailDisplayer

Como los métodos del *JPPacketDetailDisplayer* son bastante simples no mostraremos diagramas para los mismos, pero para que este objeto pueda recibir el paquete que necesita y el tiempo de envío es necesario que el objeto *JCPacketDisplayer* sea capaz de reconocer los clicks que se dan dentro de su área y buscar si ese click fue dado dentro del área de un paquete para poder invocar al método *showPacket* del objeto *JPPacketDetailDisplayer*, quien hace este trabajo en el *JCPacketDisplayer* es el método *findClickedPacketEvent* cuyo Diagrama de Flujo se muestra en la Ilustración 25. Éste diagrama muestra que básicamente se hace una búsqueda en todos los paquetes para ver si alguno contiene en su área el click recién dado, si se encuentra un *AnimationPacketEvent* que cumpla entonces se devuelve sino se retorna *null*. En el paso en el cual se verifica si un evento contiene el click lo que en realidad se hace es una llamada al método *packetEventAreaContains* que determina los cuatro puntos que forman el paralelogramo del paquete y responde si dentro de éste está el punto solicitado.

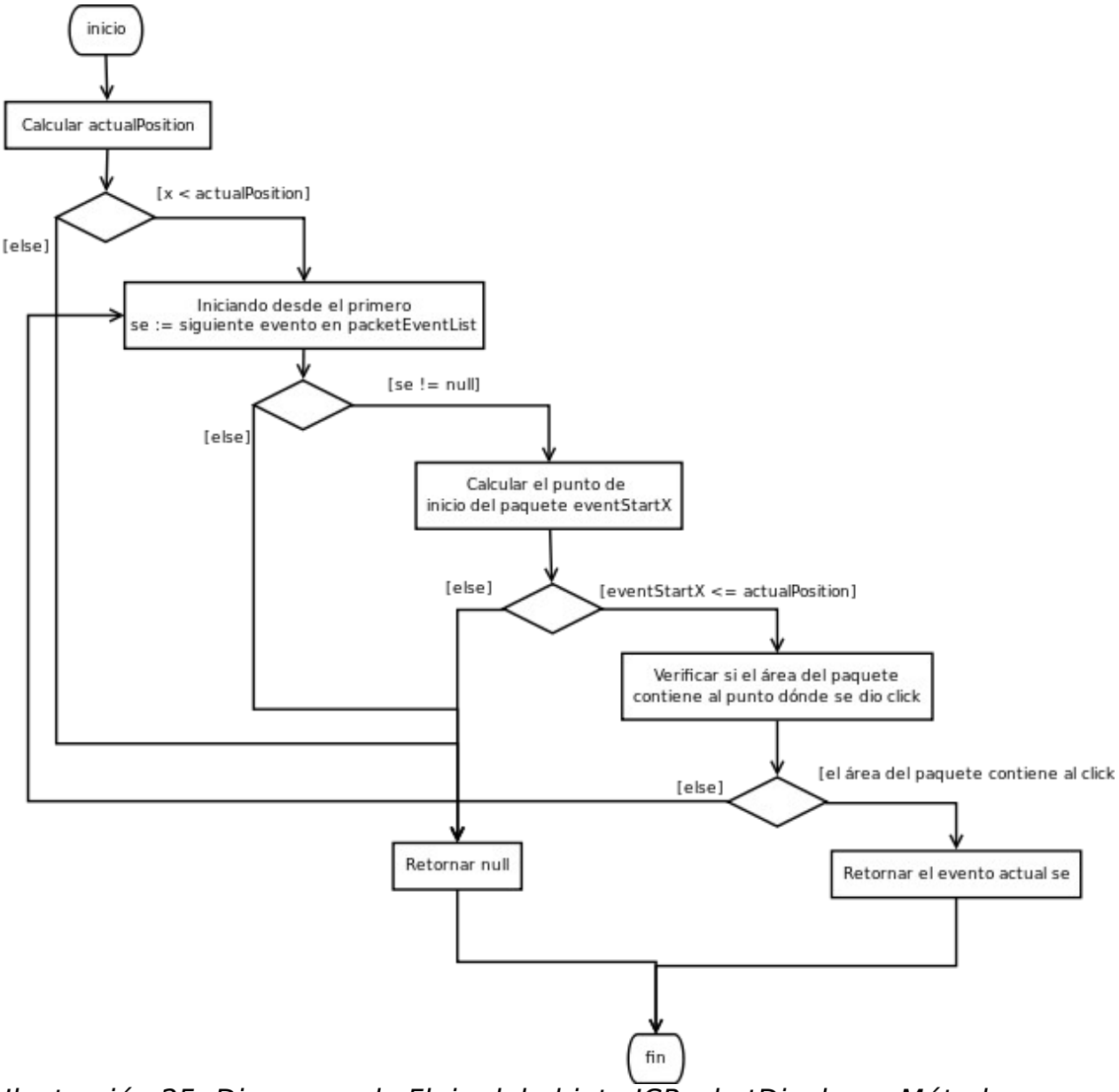


Ilustración 25: Diagrama de Flujo del objeto JCPacketDisplayer Método findClickedPacketEvent

Iteración Nº4: Cerrar Simulación y Protocolo de Rechazo Simple

Esta iteración se centró en el Caso de Uso 004: Cerrar Simulación Protocolo y continuó el Caso de Uso 003: Animar Protocolo esta vez con el protocolo de Rechazo Simple. Los requisitos cumplidos en esta iteración fueron:

1. Diseñar e implementar completamente el Caso de Uso 004: Cerrar Simulación Protocolo.
2. Diseñar e implementar las clases *HostAGoBackNProtocol* y *HostBGoBackNProtocol* que definen el emisor y el receptor del Protocolo de Rechazo Simple.
3. Probar el funcionamiento de los controles de la animacón ya implementados con el Protocolo de Rechazo Simple.
4. Documentar diagramas en digital.

La implementación del Caso de Uso 004: Cerrar Simulación Protocolo, requiere la interacción del objeto *Simulator,* del objeto *Animator* y de todos los objetos que muestran la animación y que forman parte de la GUI de la aplicación, la mejor forma de mostrar esta interacción es a través de un Diagrama de Secuencia para el Caso de Uso el cual se presenta en la Ilustración 26. En esta ilustración vemos que la solicitud de cerrar la simulación provoca también que se cierre la animación, esto se hace notificando a todos los objetos que forman parte de la animación que el tiempo actual es 0. Además hay que decir que es posible que se solicite cerrar la simulación aún cuando no se haya iniciado nunca la animación y por tanto no exista un objeto *Animator* y todos los objetos de la animación tengan su tiempo en 0 pues no ha sido modificado.

El resto de la iteración se dedicó a la implementación del Protocolo de Rechazo Simple, los detalles de la implementación de éste y los demás protocolos se pueden encontrar en el Capítulo IV. Implementación de los Protocolos.

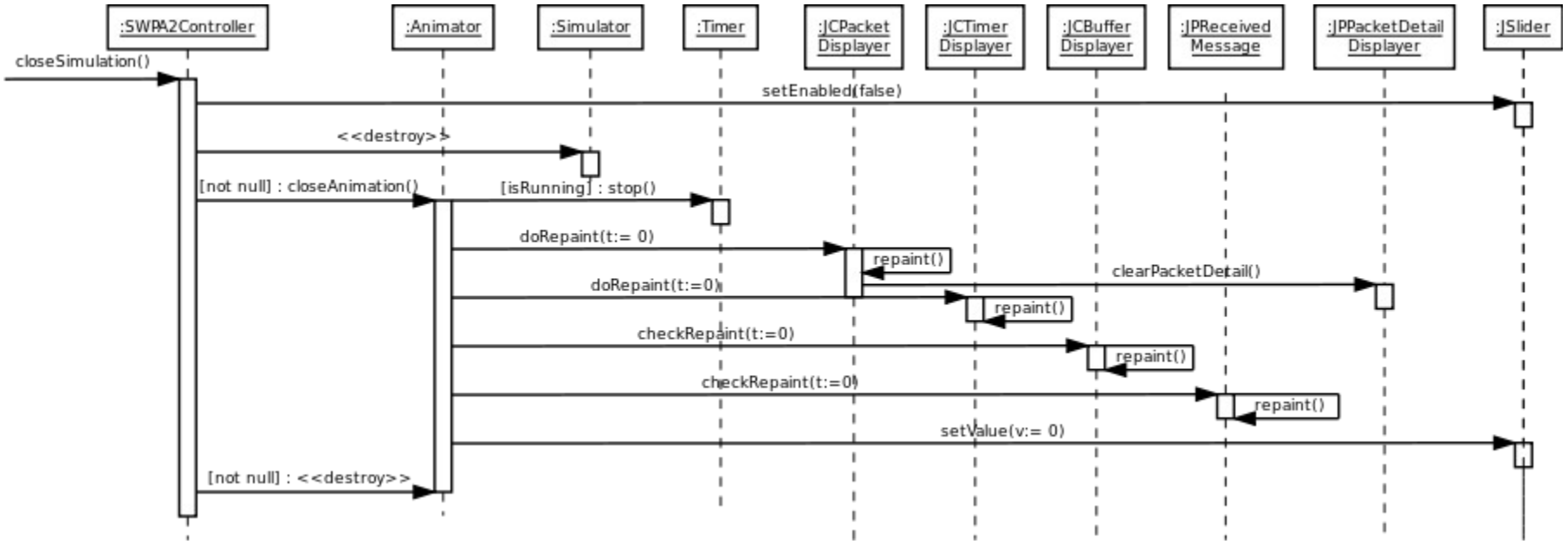


Ilustración 26: Diagrama de Secuencia Caso de Uso Cerrar Simulación Protocolo

Iteración Nº5: Buffers del P. de Rechazo Simple y Cambiar Idioma

En esta iteración se continuó con el Caso de Uso 003: Animar Protocolo culminado el protocolo de rechazo simple y se realizó el Caso de Uso 005: Cambiar Idioma. Los requisitos cumplidos en esta iteración fueron:

1. Diseñar e implementar el objeto manejador del buffer de envío para el Protocolo de Rechazo Simple.
2. Diseñar e implementa el objeto manejador del número de secuencia esperado por el receptor en el Protocolo de Rechazo Simple.
3. Diseñar e implementar completamente el Caso de Uso 005: Cambiar Idioma.
4. Hacer pruebas y documentar.

Para poder mostrar el estado visual de los buffers en el tiempo, es necesario que uno o varios objetos tengan la responsabilidad de crear los eventos del tipo *AnimationBufferEvent*. En esta iteración se implementó la clase *SenderBufferManager*, esta forma parte de la clase *HostAGoBackNProtocol* y le sirve al emisor del protocolo para manejar el buffer de envío, pero sobretodo su fin es el de crear los *AnimationBufferEvent* necesarios para que el objeto *JCBufferDisplayer* pueda mostrar de forma animada el comportamiento de dicho buffer en el tiempo.

La clase *SenderBufferManager* maneja un arreglo de paquetes como una cola circular en la que todos los paquetes nuevos se agregan en la parte frontal y los más antiguos se eliminan de la parte trasera, para esta cola define tres índices que explicamos a continuación:

baseIndex. Este representa la posición de inicio de la ventana de transmisión en el buffer. El protocolo puede modificarla en cualquier momento.

nextPacketIndex. Representa la posición donde debe estar el siguiente paquete que se enviará. El protocolo puede modificarla en cualquier momento.

top. Representa la posición del último paquete agregado al buffer. No puede ser modificada directamente por el protocolo.

La Ilustración 27 nos muestra dos ejemplos de la apariencia del buffer de envío mostrada por el objeto *JCBufferDisplayer*, en cada ejemplo podemos apreciar los valores de los tres índices mencionados y en específicamente en el segundo se aprecia la naturaleza circular del buffer.

Una consecuencia notable de que el buffer se maneje de forma circular es que para hacer que un índice se mueva al siguiente elemento no se puede simplemente sumarle uno, sino que se debe revisar si no está apuntando al último elemento del arreglo, pues el siguiente a éste sería el primero, por esta razón la clase *SenderBufferManager* hereda de la clase *BufferManager* los métodos *addOne*, *addN*, *subtractOne* y *subtractN* para hacer estas operaciones con los índices.

Además de mostrar el buffer emisor, la aplicación cuando ejecuta el Protocolo de Rechazo Simple muestra el número de secuencia esperado en el lado del receptor, la clase *ExpectedSeqNumManager* se encarga de crear los eventos *AnimationBufferEvent* del tipo RCVR_SET_EXPECTED_SEQ_NUM, para que el control *JCBufferDisplayer* del host B sea capaz de mostrar la animación de este valor. La clase *HostBGoBackNProtocol* cuenta con una instancia de la clase *ExpectedSeqNumManager* para que el alumno a través de ella modifique el número de secuencia esperado.

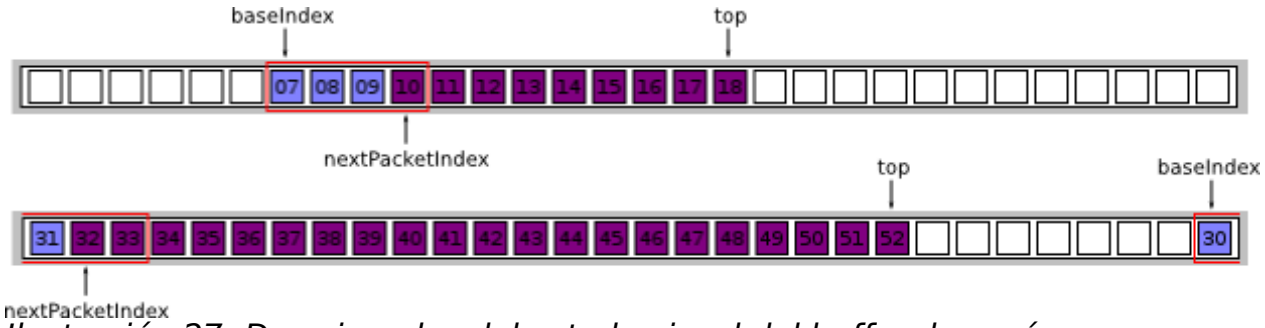


Ilustración 27: Dos ejemplos del estado visual del buffer de envío

En lo que respecta al Caso de Uso 005: Cambiar Idioma, éste junto al primer caso de uso son los únicos que no son manejados por el controlador *SW2PAController,* al Caso de Uso 005: Cambiar Idioma lo maneja la ventana principal directamente, pues consiste únicamente en cargar las etiquetas en el idioma seleccionado y actualizarlas en todos los objetos de la GUI que muestran texto, como es la ventana principal quien tiene acceso a todos estos objetos se le asignó a ella esta responsabilidad.

En la Ilustración 28 se aprecia el Diagrama de Secuencia para el Caso de Uso 005 Cambiar Idioma, observamos que el primer paso en el diagrama es la obtención de un objeto de la clase *java.util.ResourceBundle* éste objeto a la vez obtiene todas la etiquetas correspondientes al objeto *Locale* que se le envía como parámetro al método *changeLanguage.* Las etiquetas están almacenadas en los ficheros con nombres labels_en.properties, labels_es.properties y labels.properties del directorio /sw2pa/labels.

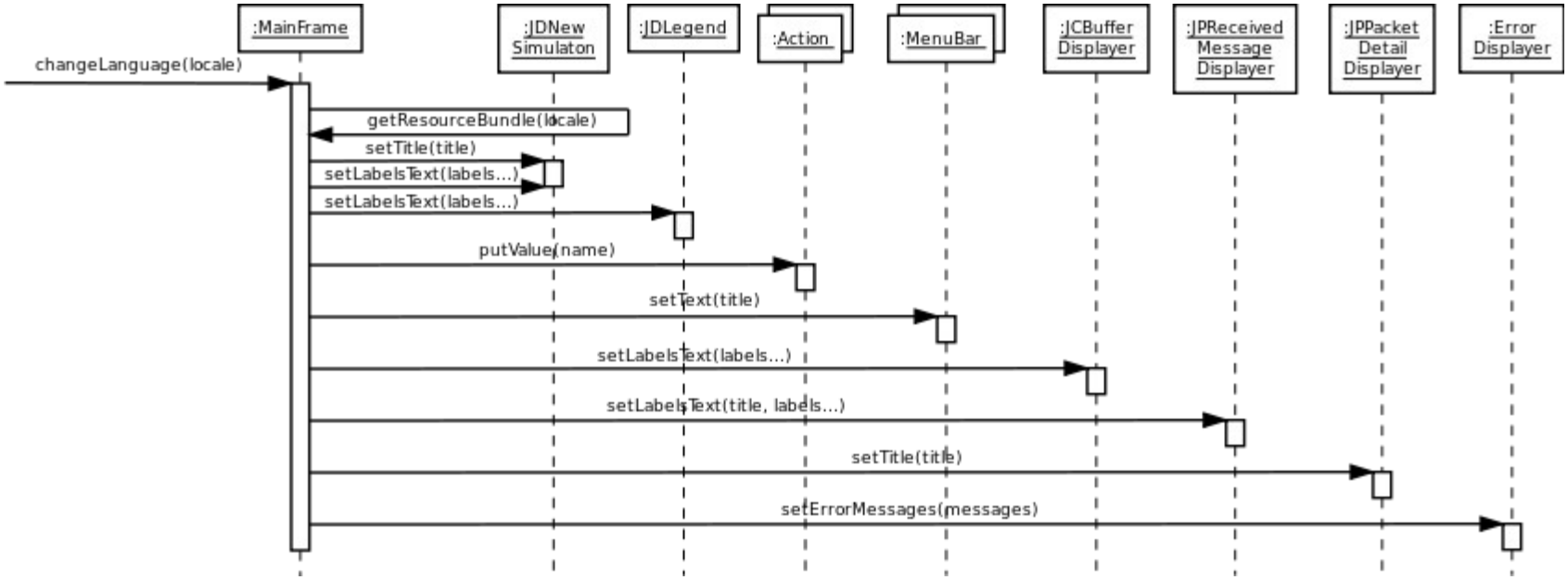


Ilustración 28: Diagrama de Secuencia para el Caso de Uso 005 Cambiar Idioma

Iteración Nº6: Buffers del P. de Rechazo Selectivo y ventanas JDNewSimulation y JDLegend

Esta última iteración finalizó el Caso de Uso 003: Animar Protocolo, específicamente el protocolo de rechazo selectivo. El requisito a cumplir fue:

1. Diseñar e implementar un objeto manejador del buffer de recepción para el Protocolo de Rechazo Selectivo.
2. Continuar el diseño e implementación del objeto *SenderBufferManager* para que se adapte también al Protocolo de Rechazo Selectivo.
3. Diseñar e implementar la ventana que solicita los datos de una nueva simulación: *JDNewSimulation.*
4. Diseñar en implementar la ventana que muestra la leyenda de los colores usados en los paquetes *JDLegend*.
5. Hacer pruebas del funcionamiento general de la aplicación.

Con el fin de cumplir el primer requisito en esta última iteración de la Fase de Construcción, se creó el objeto *ReceiverBufferManager* que permite manejar el buffer de recepción del Protocolo de Rechazo Selectivo pero sobretodo, de forma análoga al *SenderBufferManager* se encarga de crear los eventos *AnimationBufferEvent* que logran que el *JCBufferDisplayer* pueda mostrar de forma animada el comportamiento en el tiempo del buffer.

El objeto *ReceiverBufferManager* maneja el buffer de recepción usando un arreglo de paquetes pero éste ya no se gestiona como una cola como se hacía en el buffer de envío, esto porque el receptor en el Protocolo de Rechazo Selectivo es capaz de recibir los paquetes en desorden con tal su número de secuencia entre en la ventana de recepción. El único índice que se maneja es el *baseIndex* que de nuevo representa la posición de inicio de la ventana de recepción, otros valores que es necesario manejar como el siguiente número de secuencia esperado en la base de la ventana deben ser manejados por el alumno. El *ReceiverBufferManager* permite que se agregue un paquete en cualquier lugar del arreglo y también permite eliminar de cualquier lugar, pues (al igual que el *SenderBufferManager)* no conoce el tamaño de ventana, valor que interesa sólo al *JCBufferDisplayer*. La Ilustración 29, nos muestra un ejemplo de cómo se ve el buffer de recepción gestionado por el *ReceiverBufferManager* y pintado en el *JCBufferDisplayer.*

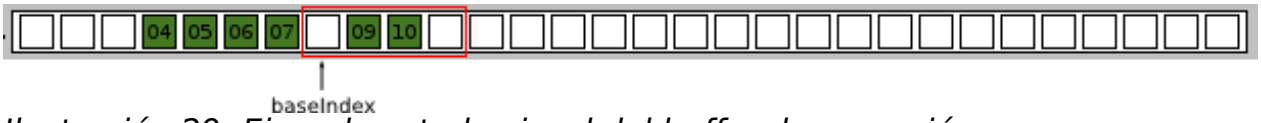


Ilustración 29: Ejemplo estado visual del buffer de recepción

Por otra parte, para que el objeto *SenderBufferManager* se adaptara también al Protocolo de Rechazo Selectivo, es necesario que almacene la información de si se ha recibido o no un asentimiento para cada paquete en el buffer, para esto se agregó a la clase *SenderBufferManager* los métodos *setACKReceivedAt* y *isACKReceivedAt,* ambos reciben el índice del paquete en el buffer donde se quiere establecer o consultar si el paquete ha sido asentido.

Finalmente se diseñó e implementó las clases *JDNewSimulation* y *JDLegend*, ambas heredan de la clase *JDialog*, la primera se elaboró con el fin de recoger los parámetros de la simulación que se desea ejecutar y la segunda para mostrar la leyenda de los colores usados en lo paquetes tanto en el objeto *JCPacketDisplayer* como en el *JCBufferDisplayer*. La Ilustración 30 y la Ilustración 31 nos muestran capturas de pantalla de estos objetos.

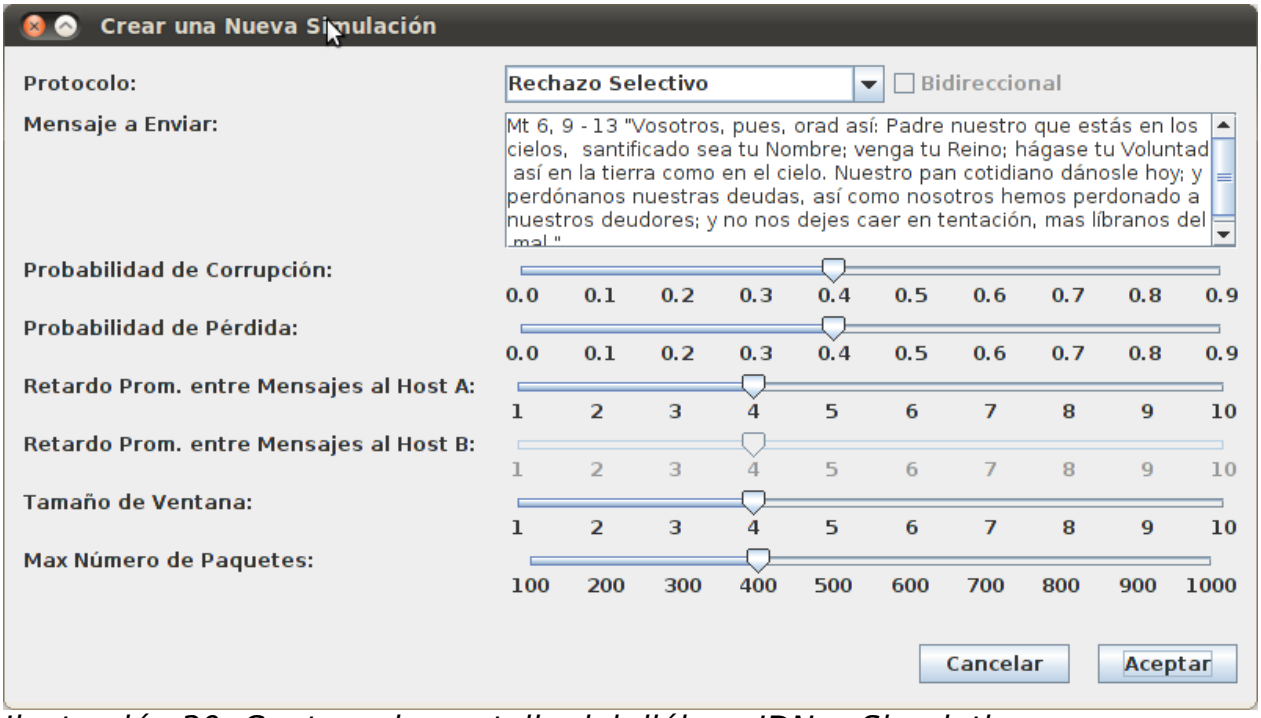


Ilustración 30: Captura de pantalla del diálogo JDNewSimulation

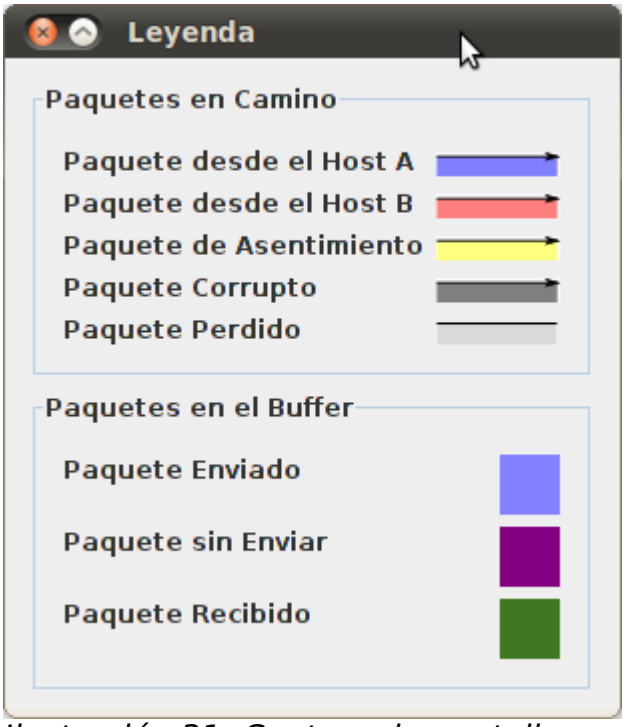


Ilustración 31: Captura de pantalla del diálogo JDLegend

Fase de Transición

La fase final del proyecto se llama transición, su nombre se debe a que una aplicación nunca está definitivamente terminada, siempre es posible mejorarla. En esta fase la aplicación es desplegada a los usuarios finales. Probablemente se despliegue una versión beta (versión de prueba) que sirva para recibir retroalimentación de los usuarios y así hacer las mejoras que se consideren apropiadas. En nuestro caso incluimos en esta fase las actividades necesarias para cumplir nuestro último objetivo específico: que la aplicación sea usada ampliamente.

- La aplicación se envió al tutor del proyecto para que fuese evaluada y proporcione la información necesaria para hacer las correcciones pertinentes.
- Se escribió el manual de aplicación en español e inglés para fortalecer su amplio uso.
- Se publicó la aplicación en el sitio sourceforge.net como proyecto personal.
- Se envió por correo la aplicación a los autores del libro Computer Networking A Top-Down Approach: Jim Kurose y Keith Ross ofreciéndola gratuitamente como herramienta de apoyo educativo.

Capítulo IV. Implementación de los Protocolos

Protocolo de Parada y Espera

Este capítulo está dedicado a discutir los detalles de implementación de los tres protocolos que incluye la aplicación. Como se comentó en el Caso de Uso 001: Definir Protocolo, la aplicación no está limitada a mostrar estos protocolos en el objeto *JCPacketDisplayer*, y será capaz de mostrar detalles en los buffers siempre y cuando se use correctamente los objetos *Manager* discutidos en el capítulo anterior. Para implementar los siguientes protocolos se usó como guía la excelente explicación dada en Kurose y Ross (2010), pero no se hizo una implementación exactamente igual a la propuesta en el material pues los autores no consideraron en sus algoritmos de ejemplo el almacenamiento de los mensajes enviados por la capa superior en el emisor que no se pueden enviar en el momento en que se reciben. Además con respecto al uso de los números de secuencia se tomó la convención en todos los protocolos que el primer número de secuencia fuese el número 1 y no el 0, pues para poder diferenciar los asentimientos de los paquetes normales se considera como asentimientos a todos aquellos paquetes que tienen un número distinto a 0 en su campo ACK.

Iniciamos el capítulo con el protocolo más sencillo, el Protocolo de Parada y Espera. La Ilustración 32 nos muestra el Diagrama de Flujo del Protocolo de Parada y Espera en el Emisor Método rdt_rcv. Este protocolo se implementó considerando que el emisor está continuamente pasando a través de cuatro estados que se observan el diagrama, además es importante mencionar que el estado inicial para el emisor es WAIT_FOR_CALL_FROM_ABOVE_1 y que cuando se recibe un asentimiento se consulta el buffer para ver si hay paquetes pendientes que enviar y si los hay enviar el siguiente.

En la Ilustración 33 vemos el Diagrama de Flujo del Protocolo de Parada y Espera en el Emisor Método rdt_send, acá cuando se recibe un paquete en un estado de espera por asentimiento, como no es posible enviarlo se almacena en un buffer, pero cuando se va almacenando los paquetes el número de secuencia de los mismos debe ir alternándose para lo cual se mantiene una referencia al último paquete ingresado al buffer: *lastBufferedPacket*.

El vencimiento de un temporizador invoca al método timeout cuyo diagrama se muestra en la Ilustración 34, vemos que simplemente se vuelve a enviar el último paquete enviado y se reinicia el temporizador. Finalmente la Ilustración 35 nos muestra el diagrama del método rdt_rcv en el receptor, se utiliza la variable *oncethru* para saber si al menos se ha enviado un asentimiento, para que si se recibe un paquete con número de secuencia inesperado se pueda reenviar el último asentimiento enviado.

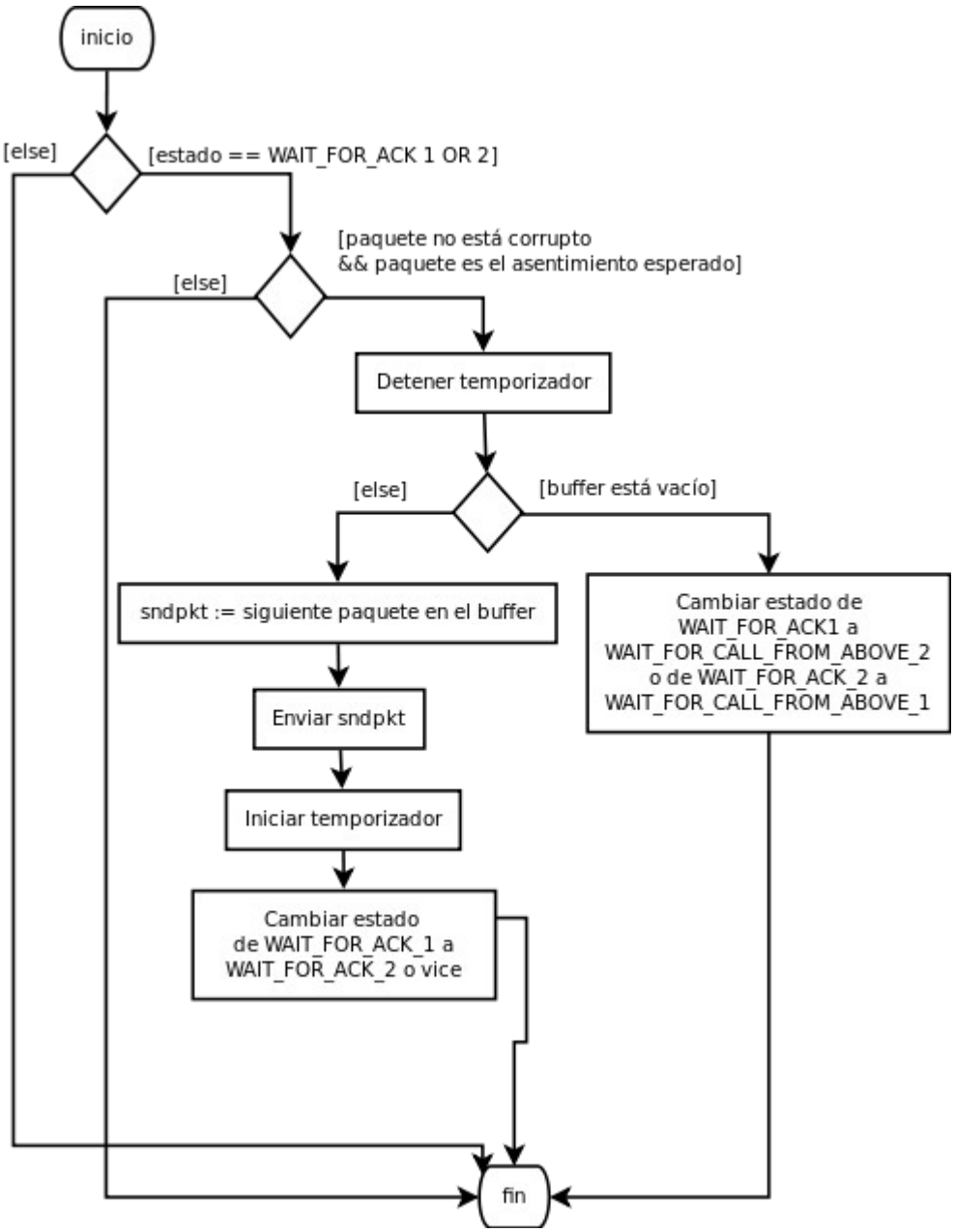


Ilustración 32: Diagrama de Flujo del Protocolo de Parada y Espera en el Emisor Método rdt_rcv

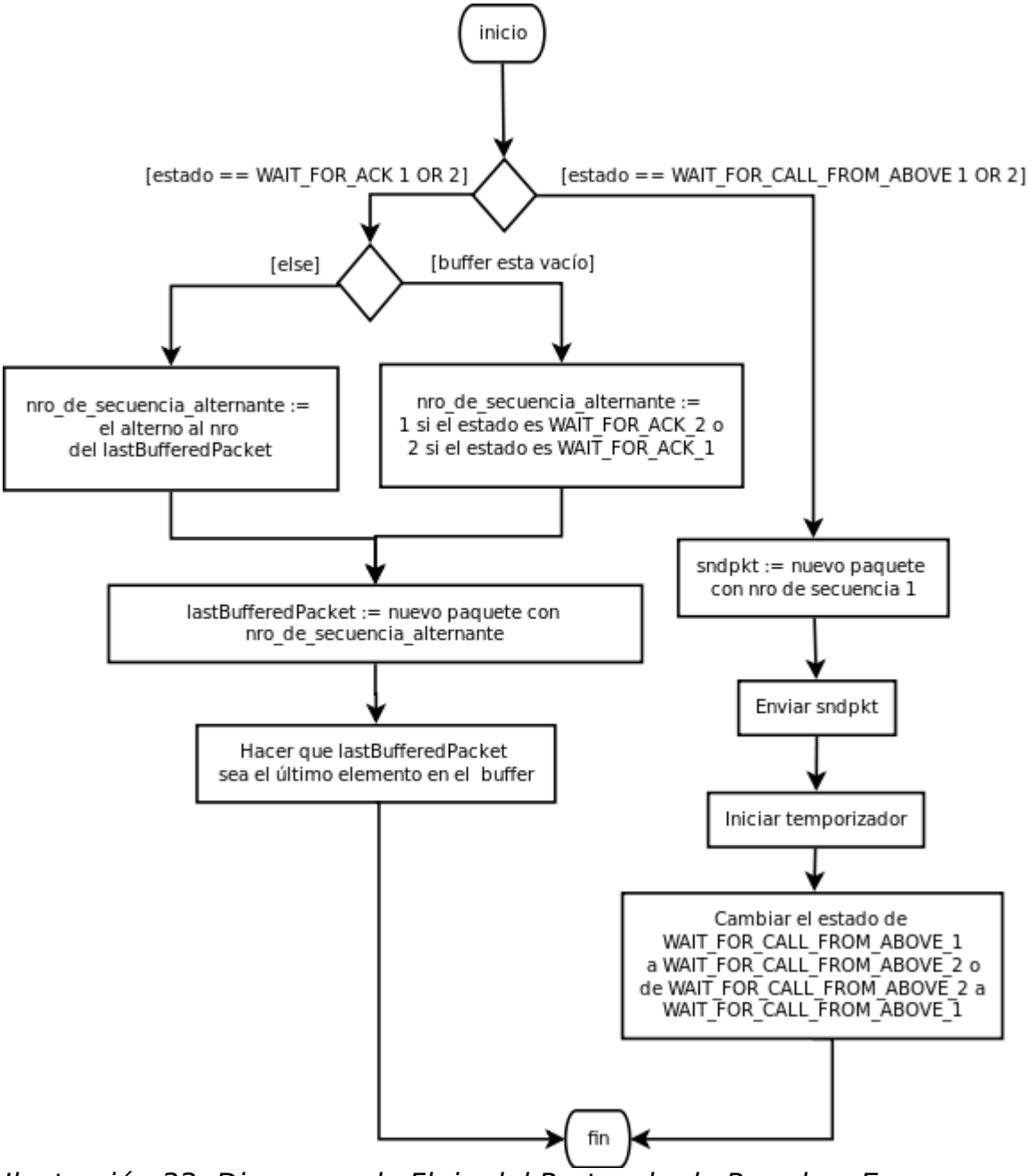


Ilustración 33: Diagrama de Flujo del Protocolo de Parada y Espera en el Emisor Método rdt_send

Ilustración 34: Diagrama de Flujo Protocolo de Parada y Espera Emisor Método timeout

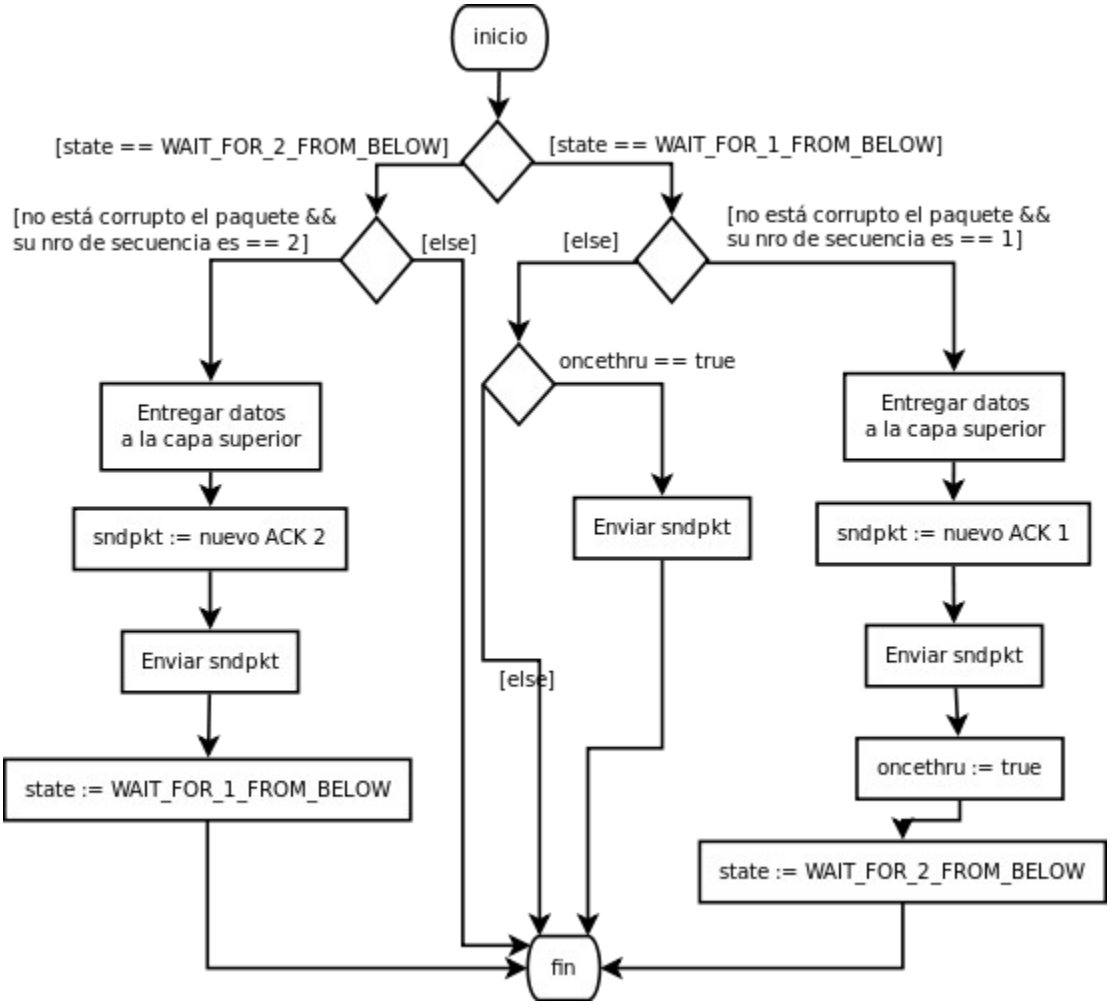


Ilustración 35: Diagrama de Flujo Protocolo de Parada y Espera Receptor Método rdt_rcv

Protocolo de Rechazo Simple

El primer protocolo de ventana deslizante que discutimos es el Protocolo de Rechazo Simple. En la Ilustración 36 apreciamos el comportamiento del emisor cuando recibe un paquete desde la capa inferior a través del método *rdt_rcv*. Se usan los índices *baseIndex y nextPacketIndex* proporcionados por el *SenderBufferManager* y descritos en la Iteración Nº5: Buffers del P. de Rechazo Simple y Cambiar Idioma. Lo que se hace en este método es ignorar todos los paquetes que no sean asentimientos, luego verificar si el asentimiento que se recibió corresponde a uno los paquetes en el buffer de envío que han sido enviados y no están asentidos, si corresponde a uno entonces hay que mover la ventana hasta el que está después de este, estos es, se considera los asentimientos como acumulativos. Luego si ya no quedan más paquetes por asentir se detiene el temporizador o si aún hay sólo se reinicia. Al final como probablemente la ventana se haya deslizado, es decir *baseIndex* haya avanzado, se trata de enviar todos los paquetes sin enviar que estén en el buffer y quepan en la ventana.

Ahora, en la Ilustración 37 vemos lo que ocurre cuando se recibe un mensaje de la capa superior a través del método *rdt_send,* lo primero es crear un paquete para el mensaje y tratar de agregarlo al buffer, decimos tratar porque existe la posibilidad de que el buffer esté lleno, si este es el caso, el *SenderBufferManager* no agregará el paquete y creará un evento de error que nos informará de esto durante la animación (por esto no es necesario verificar antes si el buffer está lleno). Luego de esto se comprueba que *nextPacketIndex* es distinto de *baseIndex + windowSize*, esto significa que si la ventana no está llena entonces se puede enviar inmediatamente el paquete recién creado, se utiliza el operador "!=" (distinto) en vez de decir que *nextPacketIndex < baseIndex + windowSize* porque como el buffer se maneja como una cola circular el operador "<" (menor que) no está definido. Después se revisa si el paquete que se acaba de enviar (en caso de que se haya enviado) es el primero en la ventana, si este es el caso entonces hay que iniciar el temporizador. Y para terminar, si se envió un paquete se aumenta *nextPacketIndex* en uno, recordando que para este por ser un índice se debe usar el método *addOne* que hereda el *SenderBufferManager* del *BufferManager.*

La Ilustración 38 nos muestra que cuando se vence un temporizador simplemente se inicia otra vez y se envían todos los paquetes sin asentir dentro de la ventana del emisor. Para terminar nuestra discusión del Protocolo de Rechazo Simple, observamos la Ilustración 39 que tiene el diagrama del receptor, antes se ha inicializado el número de secuencia esperado en 1 automáticamente por el objeto *ExpectedSeqNumManager* y se ha inicializado el paquete *sndpkt* como un asentimiento de este número de secuencia. Luego si se recibe un paquete con el número de secuencia esperado se entregan sus datos a la capa superior, se almacena un asentimiento para el paquete en *sndpkt* y se envía dicho asentimiento, si se recibe un paquete inesperado o corrupto simplemente se reenvía *sndpkt,* el asentimiento del último paquete que llegó correctamente.

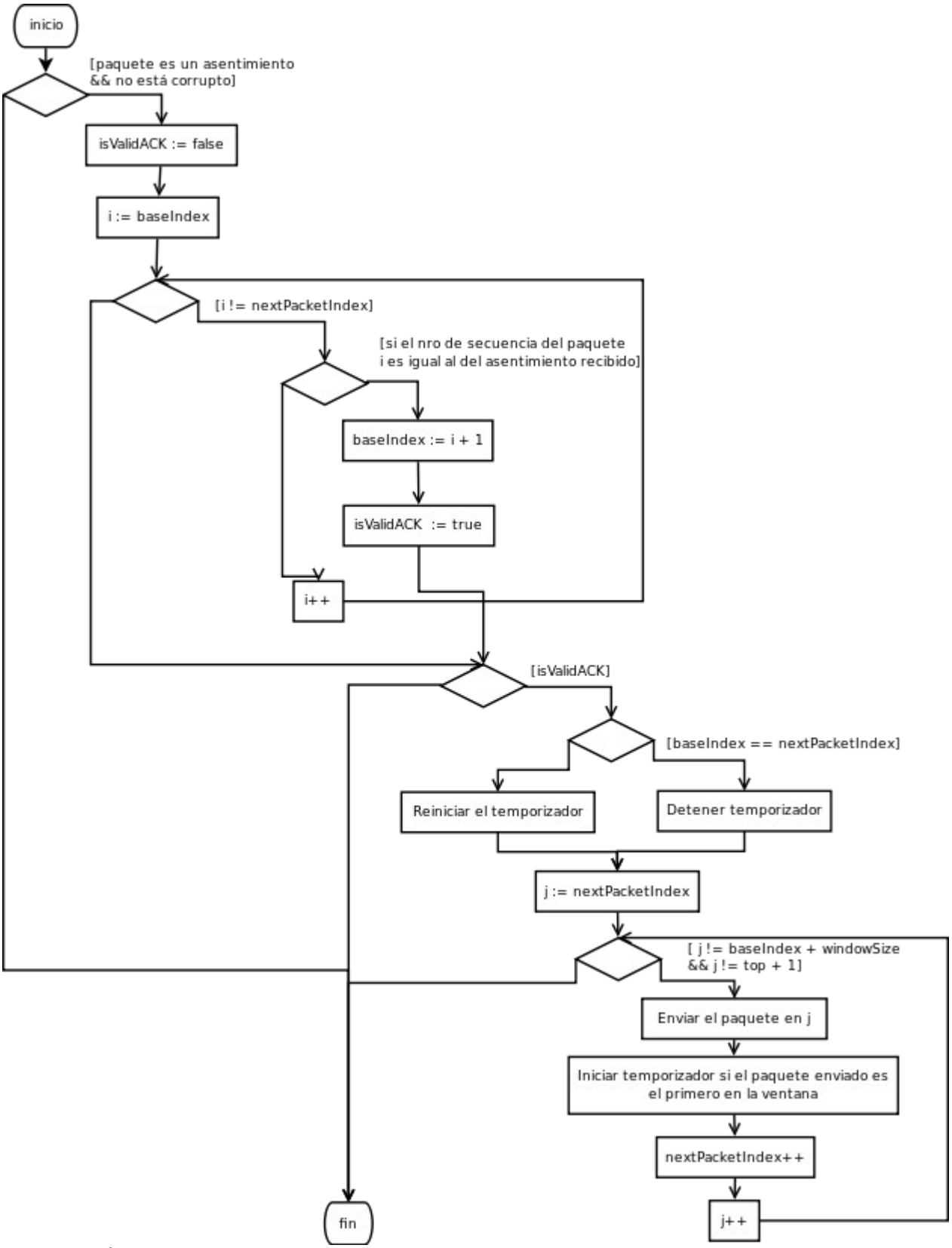


Ilustración 36: Diagrama de Flujo Protocolo de Rechazo Simple Emisor Método rdt_rcv

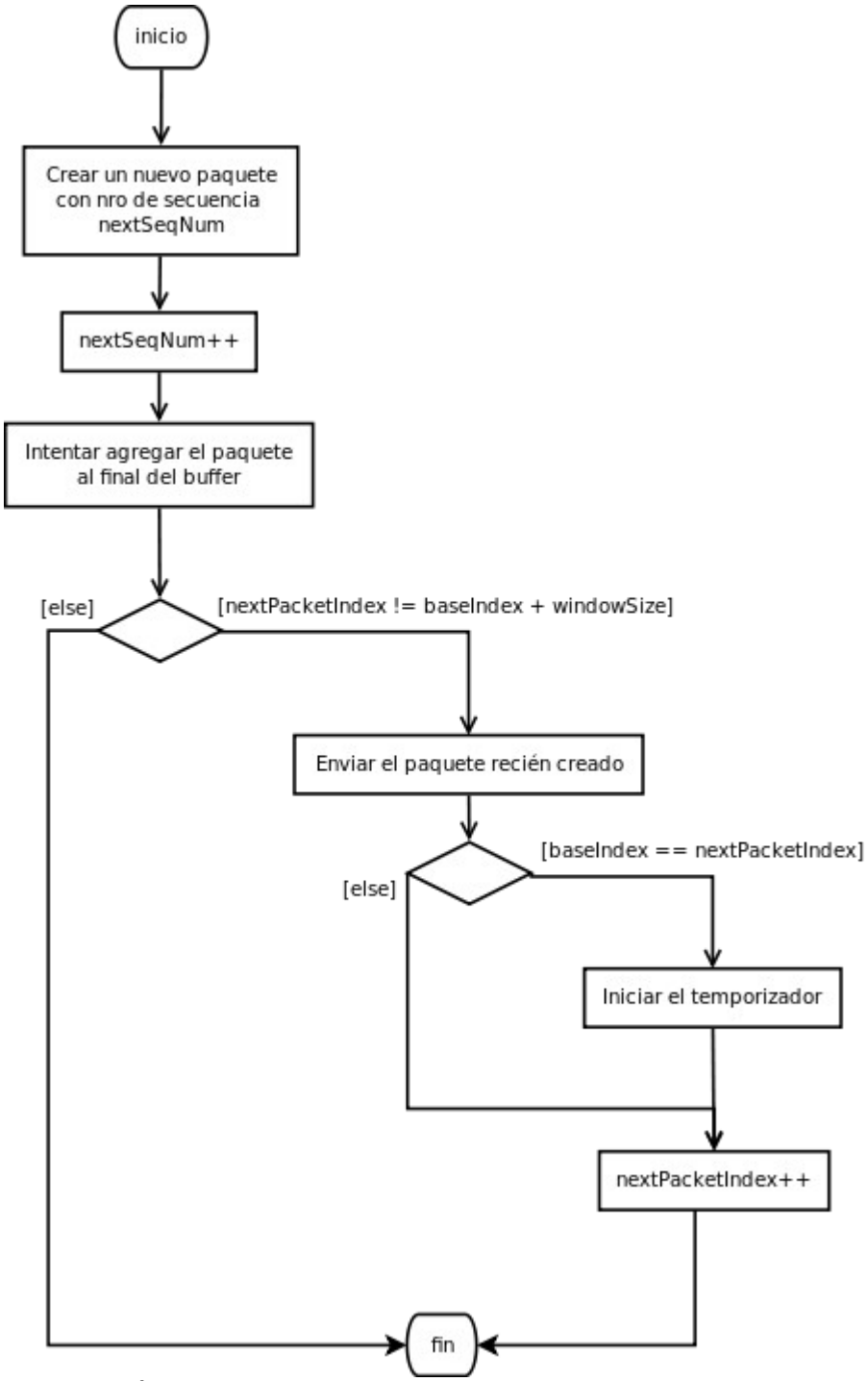


Ilustración 37: Diagrama de Flujo del Protocolo de Rechazo Simple en el Emisor Método rdt_send

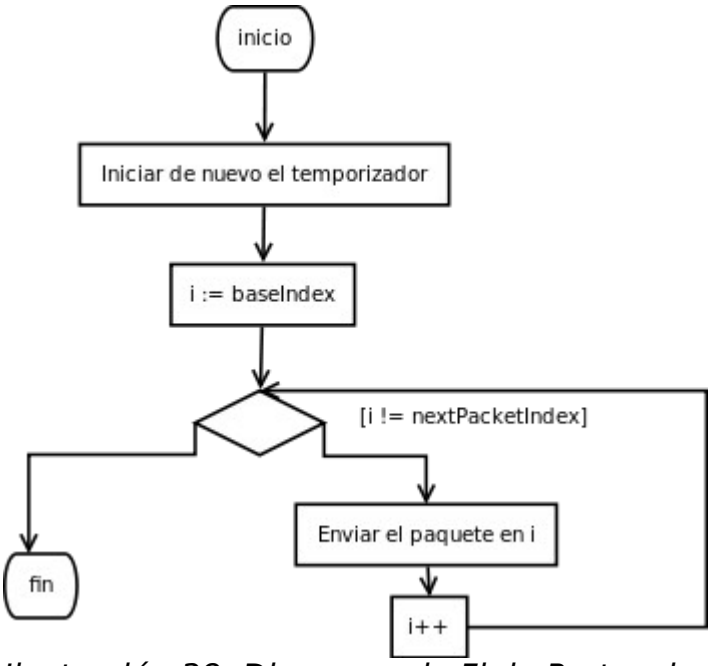


Ilustración 38: Diagrama de Flujo Protocolo de Rechazo Simple Emisor Método timeout

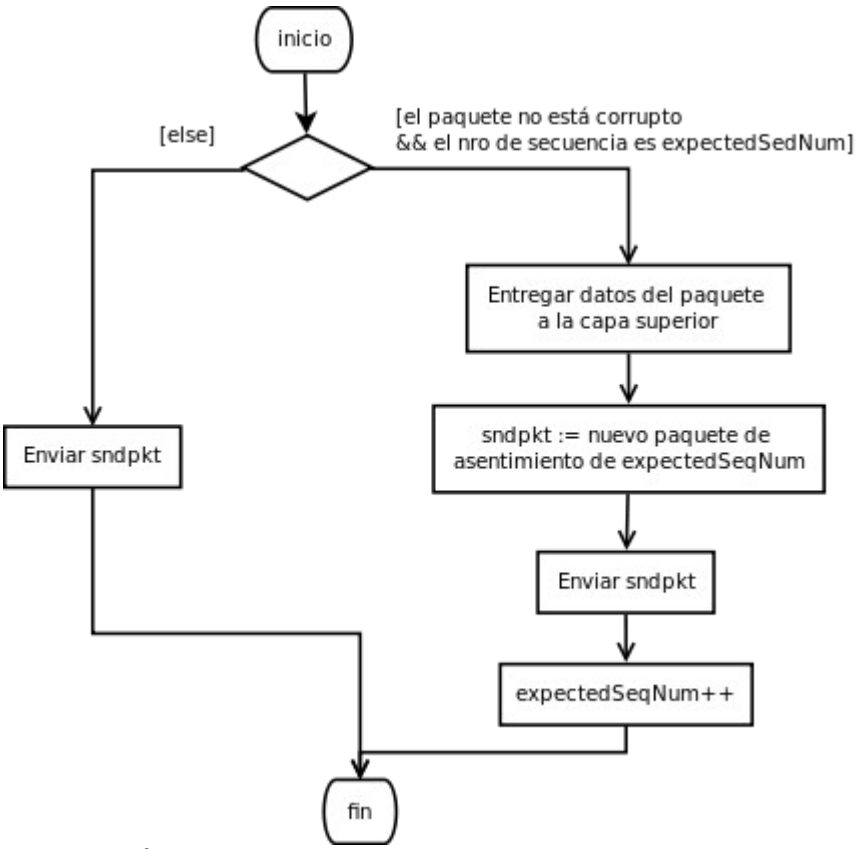


Ilustración 39: Diagrama de Flujo Protocolo de Rechazo Simple Receptor Método rdt_rcv

Protocolo de Rechazo Selectivo

Concluimos nuestra discusión sobre los protocolos con el Protocolo de Rechazo Selectivo cuya mayor diferencia con el protocolo anterior se debe al uso de temporizadores individuales y asentimientos individuales para los paquetes de modo que no es necesario reenviar todos los paquetes sin asentir al vencerse un temporizador sino únicamente el paquete al cual pertenece dicho temporizador.

En la Ilustración 40 observamos el Diagrama de Flujo para el método rdt_rcv en el emisor, vemos que tiene una estructura general similar a la del Protocolo de Rechazo Simple. Iniciamos verificando que el paquete no esté corrupto y sea un asentimiento, luego revisamos si éste corresponde a uno de los paquetes que se han enviado pero no se han asentido, si es cierto marcamos el paquete como asentido, recordamos su ubicación en el buffer como *receivedPacketIndex* y detenemos su temporizador. Posteriormente comprobamos si su ubicación es la misma que el primer paquete de la ventana, si es así entonces tratamos de deslizar la ventana lo más que se pueda, siempre que después del actual haya un paquete enviado y asentido, para hacer efectivo el desplazamiento de la ventana asignamos su nueva posición inicial a *baseIndex.* Luego como la ventana se pudo haber deslizado tratamos de enviar todos los paquetes que ahora estén dentro de ella.

Ahora pasamos a revisar la Ilustración 41, que describe qué hace el emisor cuándo recibe un mensaje de la capa superior, vemos que el comportamiento es casi idéntico al del Protocolo de Rechazo Simple, con la única diferencia de que el temporizador se inicia con un identificador igual al número de secuencia del paquete enviado. La Ilustración 42 comprueba lo que recién se mencionó, al vencerse un temporizador se busca en los paquetes sin asentir a cuál pertenece y se reenvía sólo el paquete sin asentir iniciando nuevamente su temporizador.

Finalmente la Ilustración 43, nos muestra la acciones del receptor, sólo se procesan los paquetes que no están corruptos, primero se revisa si el número de secuencia del paquete que llegó esta en el rango de los esperados, si es así se almacena en el buffer de recepción (si no esta almacenado ya) y se envía su asentimiento. Después si el paquete tiene el primer número de secuencia esperado en la base de la ventana entonces la ventana se debe deslizar mientras el paquete siguiente al actual haya sido recibido, entregando los datos de los paquetes recibidos y eliminando del buffer los paquetes que están atrás de un tamaño de ventana *(windowSize)* antes que la actual base de la ventana (*baseIndex*). Esto es, en el buffer de recepción sólo permanecen los paquetes dentro de la ventana de recepción y *windowSize* paquetes antes de la ventana cuyos asentimientos se pudieron perder y puede que sea necesario enviarlos otra vez. Por último si el paquete no entra en la ventana de recepción pero está en el rango de los otros *windowSize* paquetes todavía almacenados en el buffer, reenviamos su asentimiento pues pudo haberse perdido.

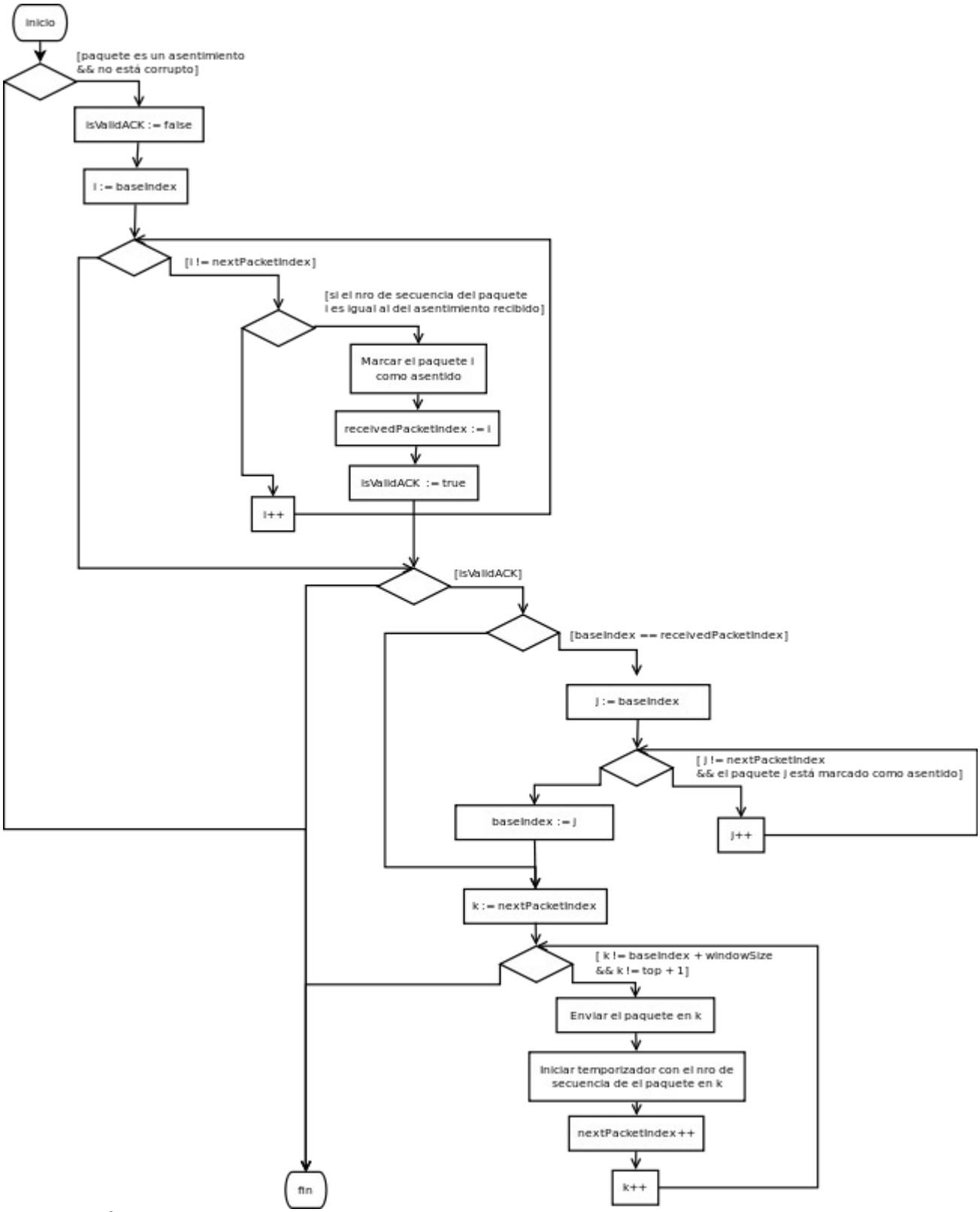


Ilustración 40: Diagrama de Flujo Protocolo de Rechazo Selectivo Emisor Método rdt_rcv

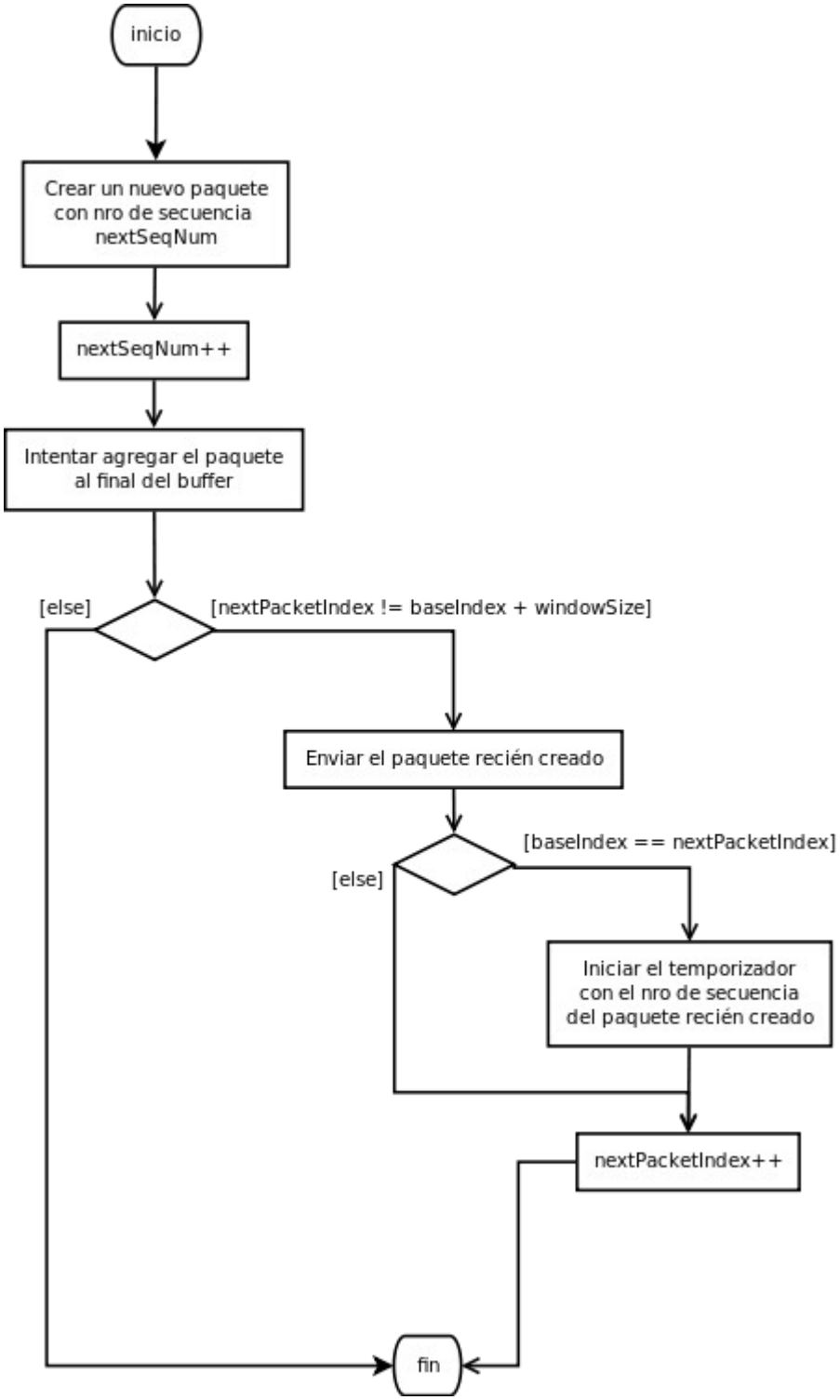


Ilustración 41: Diagrama de Flujo Protocolo de Rechazo Selectivo Emisor Método rdt_send

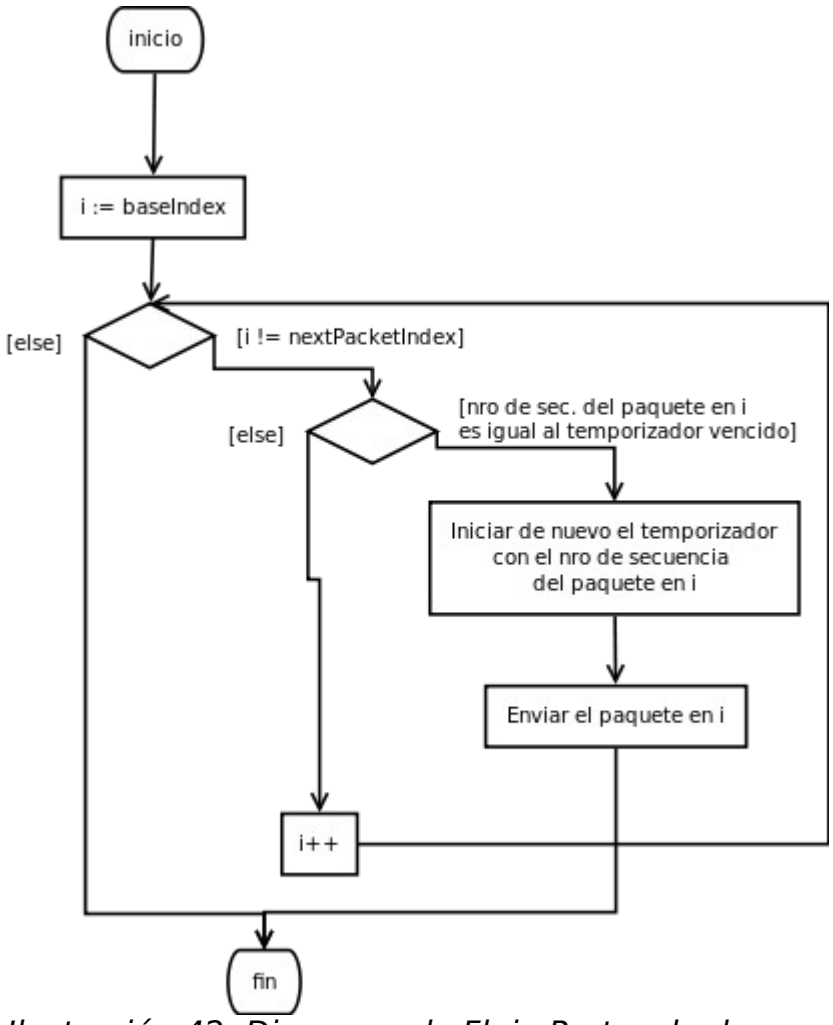


Ilustración 42: Diagrama de Flujo Protocolo de Rechazo Selectivo Emisor Método timeout

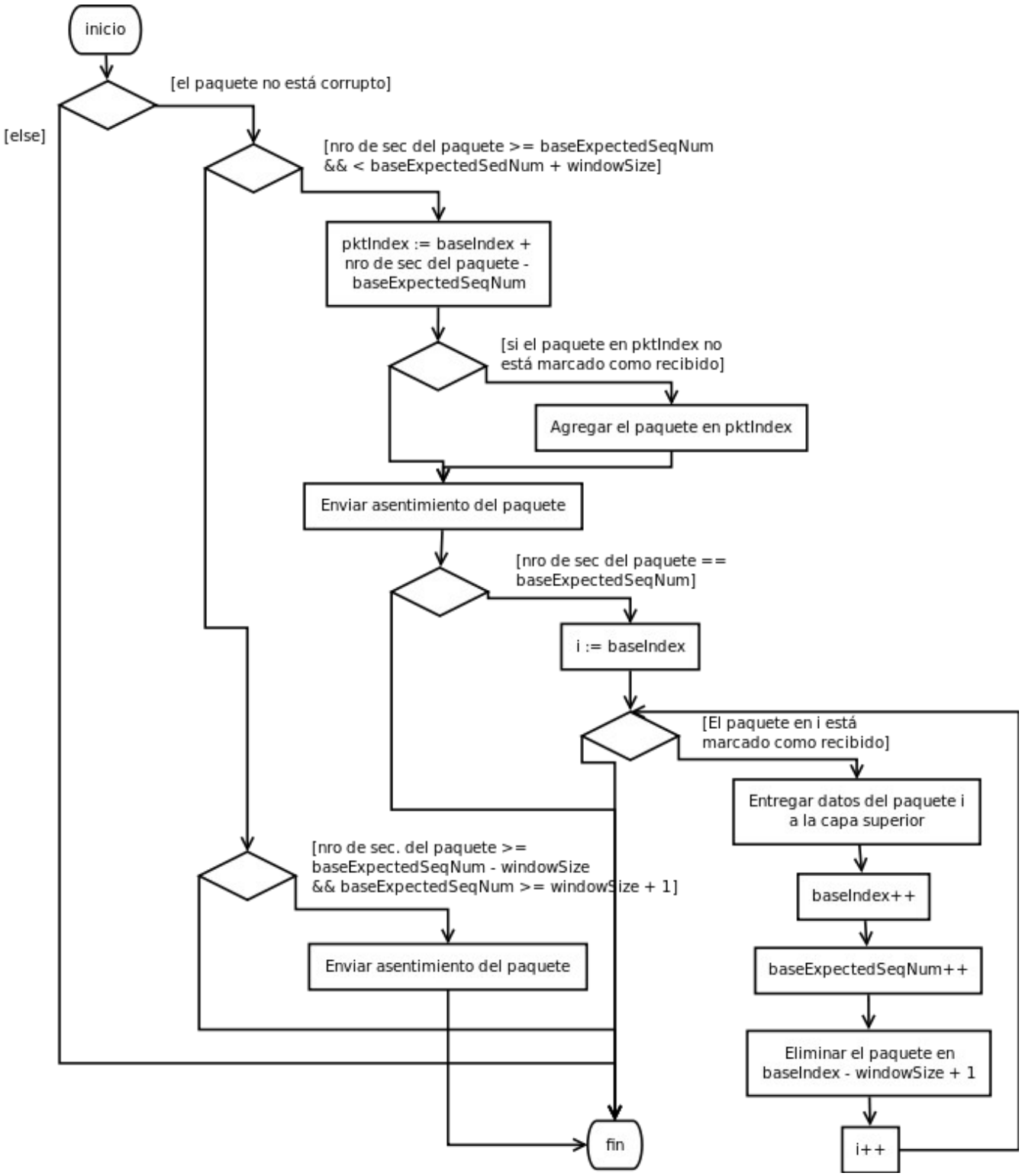


Ilustración 43: Diagrama de Flujo Protocolo de Rechazo Selectivo Receptor Método rdt_rcv

Protocolos Bidireccionales

Obtener la versión bidireccional de los protocolos anteriores es un proceso directo únicamente hay colocar ambas entidades Emisor y Receptor en un mismo host, los métodos *rdt_send* y *timeout* sólo son implementados por el emisor por esta razón no hay que hacer cambios en los mismos únicamente el método *rdt_rcv* es implementado necesita modificarse: hay que diferenciar cada paquete que se recibe, si es un paquete de datos se envía al método *rdt_rcv* del receptor y si es un asentimiento se envía al emisor.

Por supuesto al querer tomar el código unidireccional ya escrito del receptor y emisor y juntarlo en una sola clase existe la posibilidad de que algunas variables u objetos usados tengan el mismo nombre o identificador en ambas implementaciones razón por la cual es necesario modificarlos.

Capítulo V. Conclusiones y Líneas de Trabajo Futuro

Conclusiones

Habiendo finalizado el proceso de desarrollo de la aplicación *SW2PA* y regresando a las preguntas que nos planteamos en la sección titulada “Planteamiento del Problema“, podemos concluir que:

Sí fue posible desarrollar una aplicación educativa que permite definir, simular y animar los protocolos de parada y espera, rechazo simple y rechazo selectivo.

Además debido a que se pudo obtener un aplicación estable y con una interfaz gráfica muy rica para mostrar los detalles de la comunicación entre los hosts que usan los protocolos de red en cuestión, consideramos que la elección de tecnologías que se usó, esto es, el lenguaje de programación Java y el IDE Netbeans fueron valiosas herramientas para lograr este objetivo.

También fue valiosa la elección de la metodología de desarrollo orientada a objetos conocida como el UP o Proceso Unificado que con su enfoque de iteraciones fijas permitió ir entregando prototipos funcionales al tutor del proyecto y obtener rápida retroalimentación, y a la vez permitió entregar un producto de buena calidad en el tiempo deseado.

En lo que respecta al último objetivo planteado, que el *SW2PA* sea usado ampliamente, por su naturaleza deberá ser comprobado en el futuro, esperando que las medidas tomadas durante la Fase de Transición den sus resultados y tomando otras medidas que pueden surgir en el tiempo.

Líneas de Trabajo Futuras

El *SW2PA* tiene algunas capacidades que se incluyeron dentro del código pero aún se pueden completar:

Es posible agregar una funcionalidad más al objeto *JCBufferDisplayer* que como se ve en la Ilustración 8: Diagrama de Clases de Diseño también se pensó para que tuviese acceso al *JPPacketDetailDisplayer*, de forma que es posible hacer que al dar click sobre un paquete en el buffer también se aprecien los detalles del mismo igual que ocurre al dar click en un paquete que ha viajado o está viajando por el medio.

Como se aprecia en la Ilustración 9: Diagrama de Secuencia para el Caso de Uso 002 Simular Protocolo, cada vez que se invoca al método

runSimulation del objeto *Simulator* éste devuelve un objeto *SimulationStatistics* que recoge estadísticas básicas de la simulación como son el tiempo total, el número de paquetes enviados, el número de paquetes perdidos, corruptos y el número de mensajes entregados al protocolo y recibidos en su destino. Es posible entonces hacer un análisis básico del protocolo ejecutando varias simulaciones cambiando parámetros como por ejemplo el tamaño de ventana e ir recogiendo estas estadísticas de simulación para presentar gráficos y curvas sobre determinado protocolo.

La capacidad de incluir en la aplicación un protocolo genérico abre muchas posibilidades, entre ellas, implementar un protocolo bidireccional que utilice la técnica conocida como piggybacking.

Finalmente, la aplicación puede ser fácilmente traducida a otro idioma, agregándolo como opción en el menú de idiomas y sobretodo creando un nuevo fichero de etiquetas para el idioma objetivo, para más detalles sobre esto se puede consultar la Iteración Nº5: Buffers del P. de Rechazo Simple y Cambiar Idioma.

Referencias

Armstrong T. & Loane R. (1994). Educational Software: A developer's perspective. *TechTrends*, 39:20-22.

Ávila P. (2002). Tecnologías de Información y Comunicación en la Educación Proyectos Desarrollo en América Latina y El Caribe . *Revista Mexicana de Ciencias Políticas y Sociales, 45*(185) . 125-150

Banks J., Carson II J., Nelson B. & Nicol D. (2005). *Discrete-Event System Simulation* (4ª ed.), Prentice Hall.

Crescenzi P., Gambosi G. & Innocenti G. (2005). NetPrIDE An Integrated Environment for Developing and Visualizing Computer Network Protocols . *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education,* 306-310. doi=10.1145/1067445.1067529

Fuszner M (n.d.). *GNS3 Tutorial Oficial*. Recuperado desde http://downloads.sourceforge.net/gns-3/GNS3-0.5-tutorial.pdf?download el 8 de Febrero de 2011

Haase C. & Guy R. (2008). *Filthy Rich Clients: developing animated and graphical effects for desktop Java applications .* Boston, MA: Addison Wesley

Horstmann C. & Cornell G. (2008). *Core Java. Volume I, Fundamentals* (8ª ed.). Upper Saddle River, NJ.: Pearson Education

Hutchinson N. C. & Peterson L. L. (1991). The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 64–76.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Reading, MA.: Addison-Wesley.

Kurose, J. F. & Ross, K. (2010). *Computer Networking, a Top-Down approach featuring the Internet* (5ª ed), Addison Wesley.

Larman C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3ª de), Addison Wesley

Naps T. L., Robling G., Almstrum V., Dann W., Fleischer R., Korhonen A., … (2002). Exploring the Role of Visualization and Engagement in Computer Science Education. *ACM SIGCSE Bulletin 35*(2).131-152. doi=10.1145/782941.782998

Pan J. (2008) . *A Survey of Network Simulation Tools: Current Status and*

*Future Developments.*Washington University in St. Louis. Recuperado desde http://www.cse.wustl.edu/~jain/cse567-08/index.html

Schaible P. & Gotzhein R. (2002). View-based animation of communication protocols in design and in operation. *Computer Networks: The International Journal of Computer and Telecommunications Netwroking 40*(5). 621 – 638. doi=10.1016/S1389-1286(02)00355-9

Varga A (n.d). *The OMNeT++ Discrete Event Simulation System*. Budapest University of Technology and Economic . Recuperado desde: https://labo4g.enstb.fr/twiki/pub/Simulator/SimulatorReferences/esm2001-meth48.pdf el 2 de Febrero de 2011.

Wehrle K., Güne M. & Gross J (Ed.) (2010). *Modeling and Tools for Network Simulation.* Berlin, Alemania: Springer-Verlag. doi=10.1007/978-3-642-12331-3

Wong S. (1994). Quick prototyping of educational software: an object-oriented approach. *Journal of educational technology systems*, 22: 155-172.

Zahorec J., Haskova A. & Munk M. (2010). Impact of Electronic Teaching Materials on Process of Education – Results of an Experiment . *Informatics in Education, 9*(2), 261–281. Recuperado desde http://www.mii.lt/informatics_in_education/pdf/INFE165.pdf