Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Laboratorio de Comunicación y Redes

Propuesta de un Conjunto de Benchmarks para Evaluar el Desempeño de Simuladores de Red en el Área de Redes Vehiculares

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central de Venezuela
por los Bachilleres:

Larry R. Acosta Z.
V-18329344
larryacost@gmail.com

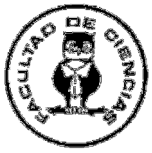
Darwing A. Hernández G.
V-19.226.819
darwing.her69@gmail.com

para optar al título de Licenciado en Computación

Tutor**:** Prof. Eric Gamess

Caracas, Mayo 2013

Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Laboratorio de Comunicación y Redes



Acta del Veredicto

Quienes suscriben, Miembros del Jurado designados por el Consejo de Escuela de Computación, para examinar el Trabajo Especial de Grado, presentado por los Bachilleres Larry Rafael Acosta. C.I.: 18.329.344 y Darwing Alejandro Hernández C.I.: 19.226.819, con el título "**Propuesta de un Conjunto de Benchmarks para Evaluar el Desempeño de Simuladores de Red en el Área de Redes Vehiculares**", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído como fue dicho trabajo por cada uno de los Miembros del Jurado, se fijó el día 30 de Abril de 2013, a las 2:00 PM, para que sus autores lo defiendan en forma pública, en Sala PA-III de la Escuela de Computación, mediante la exposición oral de su contenido, y luego de la cual respondieron satisfactoriamente a las preguntas que le fueron formuladas por el Jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió aprobarlo.

En fe de lo cual se levanta la presente Acta, en Caracas a los treinta días del mes de Abril del año dos mil trece, dejándose también constancia de que actuó como Coordinador del Jurado el Profesor Tutor Eric Gamess.

Prof. Eric Gamess
(Tutor)

Prof. Francisco Sans
(Jurado Principal)

Prof. Karima Velásquez
(Jurado Principal)

Agradecimientos

Este Trabajo Especial de Grado y todo el esfuerzo que implicó su desarrollo se lo quiero dedicar a todas aquellas personas que contribuyeron en su realización y a todos aquellos que siempre me inspiraron a lograr mis metas durante la carrera universitaria.

A mi madre Marlene Coromoto Zambrano, gracias por cuidarme siempre y por lo mucho que te has esforzado para convertirme en el hombre que soy. Gracias madre, ¡siempre te voy a amar!

A mi padre Ángel Antonio Acosta, de quien aprendí el valor de la educación, la constancia, responsabilidad y el deporte. Gracias padre, cada una de tus enseñanzas son invaluables. También a ti padre, ¡siempre te amaré!

A mis hermanas Maraitza y Arnelis Acosta, quienes son un gran apoyo y siempre han estado allí cuando se necesitan. ¡Gracias por todo lo que me han dado hermanas!

A mi familia en general, quienes siempre han creído en mí y aportaron su granito de arena para ayudarme a construir un gran futuro. Nunca olvidaré cuando era un niño y desde entonces me decían que estaban frente a un futuro licenciado o ingeniero, gracias por siempre creer en mí.

A todos mis compañeros y amigos, entre los cuales nos apoyamos, reímos, lloramos y luchamos para seguir adelante. Gracias Johana, Arthur, Fernando, Alexis, gracias a todos por esos momentos que compartimos y por todos los que están por venir.

A mi tutor Eric Gamess, quien me brindó una gran ayuda y orientación a través de sus conocimientos permitiéndome realizar un gran Trabajo Especial de Grado. Muchas gracias por todo su aporte profesor.

Especialmente quiero agradecer a mi compañero de tesis Darwing Hernández, con el cual fue un gran placer haber desarrollado este trabajo tan complejo. Gracias Darwing por todo el esfuerzo y la dedicación.

A Karina Pedrique, gran persona y gran apoyo. Gracias por siempre estar presente y por toda la ayuda que me has brindado. Siempre te estaré agradecido.

A Karima Velásquez, persona de gran corazón, gracias por todos los concejos útiles y por los divertidos momentos. ¡Ah y por los chocolates!

A la UCV, gracias por todo lo que me diste y por ayudarme a vencer tantas sombras en el camino. ¡Es un orgullo representarte y que tú me representes!

A mi Venezuela querida, tierra de grandeza, majestuosidad y de mucho coraje. Gracias por ser una tierra tan hermosa y por todo lo que me inspiras para demostrarle al mundo lo mucho que valemos y lo grande que somos.

Quiero agradecer al CDCH (Consejo de Desarrollo Científico y Humanístico) de la UCV por haber aportado muchos de los equipos que fueron utilizados durante la realización de este trabajo a través del proyecto PG 03-8066-2011/1 titulado “Propuesta y Simulación de Algoritmos para Detectar y Contar Vehículos que Emplean Tecnologías de Redes Vehiculares en Diferentes Escenarios”.

Larry Acosta.

Agradecimientos

Todo el esfuerzo y dedicación hecho en este Trabajo Especial de Grado y en toda mi carrera universitaria fue profundamente apoyado y aconsejado por una gran cantidad de personas que quisiera agradecer por su apoyo incondicional, comprensión y cariño para lograr mis metas y la culminación exitosa de este trabajo.

Principalmente agradezco a mi madre María Carolina por su inmenso amor y esfuerzo, por impulsarme y apoyarme en cada una de las metas que me propuse durante toda mi carrera. Por todas tus atenciones, preocupaciones, buenos deseos y muchas cosas infinitas más que me formaron durante todo este tiempo ¡Te amo infinitamente mamá!

A mi papá Carlos por todo el apoyo brindado en los momentos más difíciles, por tu incondicional cariño y grandes consejos, sencillamente por brindar ese apoyo de un padre, tanto a mí como a mis hermanos, gracias por estar siempre allí durante todo este tiempo, te quiero inmensamente.

A mis hermanos Daylin y Carlitos que siempre me sacaron una sonrisa cada día y me dieron luz y esperanza en aquellos días difíciles. Gracias por considerarme un ejemplo a seguir y quererme tanto como yo los quiero a ustedes.

A mi tío José Gregorio por apoyarme tanto, especialmente al principio de la carrera que me ayudó en los momentos más difíciles y me aconsejó en muchos aspectos de la vida que hoy en día los tengo muy presente. Te quiero tío.

A Michael Royero por su gran amistad, por escucharme y aconsejarme en los tiempos difíciles. De verdad gracias por tu inmenso apoyo en todos los sueños que me he propuesto.

Especialmente quiero agradecer a mi tutor Eric Gamess por su inmenso apoyo y enseñanza durante todo este tiempo. Gracias por compartir y ser parte del cumplimiento de nuestra meta. Muchísimas gracias por todos los consejos y ser nuestro guía durante todo el proceso de tesis.

También quiero dar un profundo agradecimiento a Larry por ser mi compañero en la culminación de esta meta. Gracias por compartir el mismo sueño, por tu amistad y comprensión, por tu gran esfuerzo y dedicación en esta tarea que nos ha enriquecido enormemente.

Agradezco a Karima Velásquez por todo el trabajo realizado durante un largo tiempo y la formación que esa experiencia me dejó. Muchas gracias por todos los consejos y amistad.

Muchísimas gracias a mi querida UCV, por la gran oportunidad de representarte, por la formación recibida de todos los profesores en esta casa de estudios que me brindaron sus enseñanzas y consejos para lograr ser mejor persona cada día.

Agradezco enormemente a todas esas personas que llegaron a mi vida y fui conociendo en todo el camino dado durante mi carrera. A todos esos compañeros de clases y amigos que brindaron su amistad, sonrisas y buenos momentos, especialmente a Estrella, Frank, Fernando, Audel, Freddy, Irena, Mercedes, Anabel, Víctor, Milagros, Enrique, y Alejandro Sans. Los quiero a todos.

Darwing Hernández.

Resumen

TÍTULO
Propuesta de un Conjunto de Benchmarks para Evaluar el Desempeño de Simuladores de Red en el Área de Redes Vehiculares.

AUTORES
Larry R. Acosta Z. y Darwing A. Hernández G.

TUTOR
Prof. Eric Gamess

Resumen

En este Trabajo Especial de Grado, se propone el desarrollo de un conjunto de benchmarks que sirvan como plataforma para realizar un análisis de desempeño de los simuladores de red ns-3, OMNeT++ y JiST/SWANS en el área de las redes vehiculares.

Las redes vehiculares o VANETs (Vehicular Ad Hoc Networks) son cada día más importantes debido a los servicios que estas pueden proveer a conductores y pasajeros de vehículos. Sus funcionalidades abarcan tanto aspectos de seguridad física y asistencia al conductor como aplicaciones de entretenimiento y acceso al Internet. Las redes vehiculares son en la actualidad un área fascinante y prometedora para el desarrollo de sistemas de transporte del mañana. Es por ello que se hace necesaria la implementación de nuevos estándares, protocolos y aplicaciones que sirvan de soporte en estas redes. En este proceso de desarrollo, la simulación representa una solución viable y poco costosa para la experimentación sobre VANETs. Por esta razón, es importante evaluar el desempeño de los simuladores de red para esta área. Para el TEG, se estudiaron y seleccionaron los simuladores más utilizados por la comunidad de investigadores (ns-3, OMNet++, y JiST/SWANS).

ns-3 (Network Simulator 3) es un simulador de red basado en eventos discretos. Permite estudiar los protocolos del Internet y sistemas a gran escala en un ambiente controlado. Está escrito en C++ y posee un entorno gráfico limitado mediante el uso de NetAnim (Network Animator). OMNeT++ es un entorno de simulación basado en eventos discretos. Fue diseñado desde un principio para soportar simulaciones de redes a gran escala. Está escrito en C++ y posee interfaz gráfica muy poderosa. JiST (Java in Simulation Time) es un motor de simulación de eventos discretos de alto rendimiento para la simulación de eventos discretos escrito en Java. SWANS (Scalable Wireless Network Simulator) es un simulador de redes inalámbricas construido sobre la plataforma JiST. No posee interfaz gráfica.

En este trabajo se realizó una implementación de un conjunto de benchmarks donde se puede: (1) simular redes vehiculares en ns-3, OMNeT++ y JiST/SWANS con una topología circular (circle benchmark) o una topología basada en una red de carreteras real (city bechmark) utilizando trazas ns-2 generadas a partir de mapas reales, (2) configurar la simulación mediante el uso de archivos donde se definen una gran cantidad de parámetros de entrada (número de nodos, protocolo de enrutamiento, bitrate, etc.), (3) almacenar los resultados en un archivo de salida donde se reportan tanto estadísticas acerca del comportamiento de la red (delay, PDR, NRL, etc.) así como también estadísticas de benchmarking (tiempo de ejecución, consumo de CPU y memoria). Finalmente con los resultados obtenidos, se realizó una comparación analítica del desempeño de los simuladores escogidos.

Palabras Claves: Simulador de red, Benchmarks, Redes Vehiculares, VANET.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

1. Introducción

Las redes de comunicaciones son cada día más complejas y costosas, es por ello que la experimentación basada en la simulación se ha convertido en una práctica esencial. A través de la simulación se puede enseñar las claves de las tecnologías en las redes de comunicaciones a estudiantes y profesionales, pero también provee un entorno con las especificaciones necesarias para modelar sistemas de redes particulares como es el caso de las redes vehiculares.

En los últimos años las redes vehiculares han sido objeto de estudio y desarrollo por parte de la academia y la industria, convirtiéndose en una nueva área de investigación. En particular, este interés se fundamenta en la importancia que tienen las aplicaciones relacionadas con la seguridad vial, el control del tráfico y el entretenimiento. Gracias a la simulación se logra experimentar con redes vehiculares a bajo costo y tiempo en comparación con la experimentación en la vida real. A pesar de esto, la simulación no es una práctica sencilla, es importante utilizar las herramientas adecuadas para desarrollar modelos que se ajusten a los escenarios deseados y que arrojen resultados que no se alejen de la realidad.

Existe una gran variedad de simuladores de red de propósito general o específico disponibles para su libre utilización e investigación. Cada uno de estos simuladores cuenta con sus propias características y pueden ser bastante diferentes entre sí. Entre esta diversa gama de simuladores de red se tienen ns-3, OMNeT++ y JiST/SWANS que representan soluciones potentes y muy usadas por la comunidad de investigadores. Además de esto, han sido utilizados como plataformas para simular redes vehiculares porque cuentan con las herramientas y la capacidad de integración con otros sistemas que permiten el modelado de las mismas.

Cada día son más y más los trabajos de redes vehiculares que se desarrollan basados en la simulación usando ns-3, OMNeT++ y JiST/SWANS. Por esta razón, realizar un estudio a profundidad de las capacidades de estos simuladores con respecto a la simulación en el área de las redes vehiculares genera gran expectativa. La utilización de las herramientas adecuadas y que más se ajusten a los escenarios a modelar en este tipo de sistema de comunicación es de suma importancia para lograr resultados significativos.

Este trabajo tiene la siguiente organización:

Capítulo 2: En este capítulo se presenta la descripción del problema.

Capítulo 3: Contiene una breve introducción a las técnicas de evaluación de desempeño, sus características, tipos y limitaciones.

Capítulo 4: Este capítulo está centrado en analizar el funcionamiento y las características de ns-3.

Capítulo 5: Este capítulo se basa en el análisis de las características y del funcionamiento de OMNeT++.

Capítulo 6: Este capítulo está enfocado en analizar el funcionamiento y las características de JiST/SWANS.

Capítulo 7: Muestra un estudio en profundidad relacionado al área de las redes vehiculares, tecnologías implicadas y la descripción de los protocolos que intervienen en este tipo de sistema de comunicación.

Capítulo 8: En este capítulo se evalúan diferentes herramientas para la generación de trazas vehiculares.

Capítulo 9: Presenta los trabajos relacionados recopilados durante la investigación.

Capítulo 10: Describe el Marco Metodológico.

Capítulo 11: Plantea el Marco Aplicativo.

Capítulo 12: Muestra las especificaciones técnicas de los benchmarks desarrollados.

Capítulo 13: Se discuten las conclusiones obtenidas producto de la investigación realizada y los trabajos futuros.

2. El Problema

En este capítulo se exponen los argumentos que justifican el diseño y desarrollo de un conjunto de benchmarks ideados para simular redes vehiculares y evaluar el desempeño de los simuladores de red, así como el planteamiento de nuevos objetivos y alcances para obtener una solución eficaz.

2.1 Planteamiento del Problema

En la actualidad, las redes vehiculares van teniendo una importancia cada vez mayor debido a su gran impacto en la sociedad. Con este fin, se han creado consorcios que involucran a fabricantes de automóviles, organizaciones gubernamentales y academias que desarrollan protocolos y tecnologías para dar soporte a esta área emergente.

Hoy día, los investigadores de las redes vehiculares utilizan la simulación para verificar el funcionamiento de sus algoritmos. Existen muchas razones que motivan el uso de la simulación, entre las cuales se tienen: (1) los altos costos de los dispositivos WAVE (Wireless Access in Vehicular Environments), (2) las restricciones sobre los experimentos (disponibilidad de vehículos y conductores), (3) la carencia a nivel mundial de test-tracks, (4) la dificultad para llevar a cabo experimentos en carreteras abiertas al tráfico vial, (5) la seguridad física de los conductores durante los experimentos, y (6) la casi imposibilidad de repetir un experimento cambiando una variable. En los próximos años es seguro que van a disminuir en forma importante los costos de los dispositivos WAVE, pero los demás factores que dificultan la realización de los experimentos no van a mejorar significativamente sino que más bien deberían empeorarse. En consecuencia, es muy probable que la simulación se mantenga como una de las únicas alternativas para validar los algoritmos propuestos por la comunidad de investigadores en el área de las redes vehiculares.

La selección de un simulador adecuado para la simulación de redes vehiculares es primordial. Es importante que el simulador soporte los protocolos del área (IEEE 802.11p y la familia de los IEEE 1609.X). Ningún simulador actual tiene esta capacidad, algunos implementan solo algunas características de estos protocolos mientras que en otros no se tiene implementación alguna. Eso ha llevado a los investigadores a evaluar sus algoritmos con modelos del IEEE 802.11. Otro factor muy importante que debe tener el simulador elegido es una alta escalabilidad. El vehículo se ha vuelto el medio de transporte más usado. En las horas pico se pueden ver centenas de vehículos en una porción de carretera o de autopista. Validar un algoritmo de enrutamiento no se debería hacer con apenas una centena de vehículos, ya que limitaría su aprobación por la comunidad científica. Así que utilizar un simulador que permita ejecutar un experimento con centenas (o en su posibilidad miles) de vehículos en tiempo razonable es de primera importancia.

2.2 Justificación del Problema

En el área de las redes vehiculares la simulación es una práctica de uso extendido porque permite modelar y obtener resultados a menor costo y tiempo. Existe una gran cantidad de estudios realizados con respecto al análisis de desempeño de los simuladores ns-3, OMNeT++ y JiST/SWANS. La mayoría de estos análisis evalúan al simulador mediante la propuesta de experimentos relacionados con redes MANETs, donde la velocidad de los nodos es lenta o moderada. Además, el número de nodos involucrados en las simulaciones realizadas es relativamente bajo. Resulta interesante hacer un análisis de desempeño de estos simuladores complementando así la carencia de estudios y proponiendo un conjunto de benchmarks que podrían ser la base del análisis de desempeño y escalabilidad para simuladores en el área de redes vehiculares.

2.3 Objetivos

En esta sección se definen los objetivos que se quieren conseguir con la realización del trabajo propuesto.

2.3.1 Objetivo General

Proponer un conjunto de benchmarks para evaluar el desempeño de los simuladores ns-3, OMNeT++ y JiST/SWANS en el área de las redes vehiculares.

2.3.2 Objetivos Específicos

- Estudiar el área de la simulación de redes y las redes vehiculares.
- Aprender a utilizar los simuladores ns-3, OMNeT++ y JiST/SWANS.
- Generar y depurar trazas para redes vehiculares con herramientas especializadas.
- Desarrollar una herramienta que permita realizar un análisis exhaustivo a las trazas de movilidad generadas para los benchmarks.
- Proponer un conjunto de benchmarks para evaluar a los simuladores en el área de redes vehiculares.
- Implementar los benchmarks propuestos en los simuladores ns-3, OMNeT++ y JiST/SWANS.
- Buscar mapas digitales de Caracas (u otras ciudades venezolanas) para proponer un benchmark realista en una ciudad del territorio nacional.
- Probar cada benchmark con parámetros distintos y almacenar los resultados.
- Realizar el análisis de los resultados obtenidos. Concluir sobre el realismo de los resultados.

2.4 Alcances

Los benchmarks propuestos deberán generar resultados significativos para poder realizar el análisis de desempeño fundamentado tanto en la medición del tiempo de ejecución como en la cantidad de memoria consumida por los simuladores. Adicionalmente, se hará un estudio de algunos parámetros intrínsecos a las redes en general para la validación de los benchmarks.

3. Técnicas de Evaluación de Desempeño

El modelado de sistemas se refiere a la acción de representar un sistema real de una o más formas simples. Es muy importante en el diseño y desarrollo de un sistema, ya que da una idea de cómo se comportaría el sistema en la realidad. Con el modelado, los parámetros del sistema se pueden cambiar para analizar sus efectos en posteriores estudios. Más importante aún, si se maneja y se usa adecuadamente, el modelado de sistemas puede ahorrar costos en el desarrollo del sistema ya que permite la planificación. Para modelar un sistema, algunas aproximaciones son requeridas [11]. Es importante tener en cuenta que demasiadas aproximaciones podrían simplificar el modelo y puede dar lugar a una representación inexacta del sistema.

El desempeño es un factor importante para cualquier sistema ya que permite determinar qué tan eficiente o rentable está siendo. La evaluación de desempeño se puede dividir en tres áreas: (1) modelado de desempeño en forma analítica o matemática, (2) simulación y (3) benchmarks [9][11]. En la evaluación de desempeño se utilizan técnicas, hipótesis y procesos analíticos cuyo objetivo es la eficiente estimación de alguna medida de desempeño. El desempeño de un sistema de comunicación depende de las partes que lo conforman, de la red de interconexión y de las aplicaciones que ejecuta. En consecuencia, la evaluación de desempeño de un sistema puede resultar un proceso complejo. Frecuentemente, cada técnica de evaluación de desempeño normalmente tiene que ser diseñada sobre una base caso por caso dado el sistema en cuestión. El proceso de definición de una técnica de evaluación consiste en [11]:

- Simplificaciones: de las propiedades de los elementos atómicos dentro de un sistema y la lógica subyacente de su evolución a través de interacciones.
- Aproximaciones estadísticas: sobre las propiedades de aspectos no deterministas del sistema.
- Técnicas estadísticas: para la agregación de las medidas de rendimiento estimadas obtenidas a partir de simulaciones o modelados de diferentes configuraciones del sistema.

3.1 Modelos Analíticos o Matemáticos

El concepto general del enfoque de los modelos analíticos o matemáticos plantea una manera de describir un sistema matemáticamente con la ayuda de herramientas matemáticas aplicadas, tales como teorías de la probabilidad y colas, así como otros modelos matemáticos en específico, y luego aplicar métodos numéricos para obtener conocimientos del nuevo modelo desarrollado. Cuando el sistema es sencillo y relativamente pequeño, aplicar modelos analíticos sería preferible para emular el sistema y realizar el análisis correspondiente [11]. Las soluciones numéricas de este tipo de modelo requieren ligeros esfuerzos computacionales, aunque puede que no sea así si el modelo es muy complicado.

Durante las fases de diseño y desarrollo de un sistema, el modelado puede ser utilizado para estimar las variables de rendimiento que se obtendrán cuando el sistema esté implementado. El modelado matemático no afecta el sistema medido como sucede en la técnica de medición directa. Sin embargo, el modelado sufre de los siguientes problemas [11]:

- Abstracción del sistema: esto puede llevar a la generación de un modelo que no representa el sistema real bajo evaluación.
- Representación de la carga de trabajo del sistema.
- Obtención de medidas de desempeño para el modelo y la asignación de los resultados de vuelta al sistema real.

Como resumen, un modelo analítico de un sistema trata de deducir los parámetros a través de la derivación matemática, por lo que lleva al dominio de las teorías matemáticas, es decir, no recurre a la ejecución de programas, sino que está basado en análisis matemáticos que se apoyan generalmente en técnicas como la teoría de colas, redes de Petri, álgebra de procesos, métodos numéricos, diagramas de estado, etc. Es un método interesante ya que una formulación matemática da a los usuarios información sobre el desempeño del sistema [9]. Sin embargo, el análisis se vuelve a menudo complejo y la búsqueda del modelo matemático es difícil, en particular cuando intervienen muchas variables o no se realiza la abstracción adecuada. Por estas razones, es usado para estudiar el desempeño de sistemas sencillos o simplificados. Si se emplea correctamente, el modelado matemático puede proporcionar una visión abstracta de los componentes que interactúan entre sí en el sistema. Sin embargo, si muchas simplificaciones son hechas en el sistema durante el proceso de modelado, los modelos analíticos podrían no dar una adecuada representación del sistema real.

3.2 Simulación

En la simulación se trata de reemplazar los modelos por un programa computacional que intenta simular un sistema particular. Los modelos computacionales [11] pueden ser clasificados acorde a su criterio binario ortogonal:

- Estado continuo o estado discreto: si las variables de estado del sistema pueden asumir cualquier valor, entonces el sistema se modela con un modelo de estado continuo. Por otro lado, un modelo en el que las variables de estado sólo pueden asumir valores discretos se llama un modelo de estado discreto.
- Modelos de tiempo continuo o de tiempo discreto: si las variables de estado del sistema pueden cambiar sus valores en cualquier instante de tiempo, entonces el sistema puede ser modelado por un modelo de tiempo continuo. Por otro lado, un modelo en el que las variables de estado pueden cambiar sus valores sólo en instantes discretos de tiempo se denomina modelo de tiempo discreto.

La simulación es la imitación de un sistema del mundo real a través de una reconstrucción computacional de su comportamiento de acuerdo a las reglas descritas en modelos matemáticos, pero esto no quiere decir que use estrictamente modelos matemáticos para cumplir su función [11][38]. La simulación es ampliamente utilizada en el modelado de sistemas para aplicaciones que incluyen la investigación en ingeniería, el análisis de negocios, la planificación de la fabricación, la experimentación de las ciencias biológicas, y los estudios de protocolos de redes, sólo para nombrar unos pocos. El hecho de simular un sistema general implica considerar un número limitado de características y comportamientos del sistema real de interés, y así evitarse la complejidad traída por un sin número de detalles. Una simulación permite examinar el comportamiento de un sistema bajo diferentes escenarios que son evaluados por la recreación en un mundo virtual informático [11]. La simulación puede ser utilizada, entre otras cosas, para identificar los cuellos de botella en un proceso, proporcionar un ambiente seguro y relativamente más económico (en términos de costos y tiempo), establecer un banco de pruebas para evaluar la correlación de variables y optimizar el rendimiento del sistema, todos se pueden realizar antes del establecimiento de esos sistemas en el mundo real.

En comparación con los modelos analíticos, la simulación por lo general requiere menos abstracción en el modelo, es decir, menos simplificaciones, ya que casi todos los detalles posibles de las especificaciones del sistema se pueden colocar en la simulación del modelo para describir mejor el sistema actual. Cuando el sistema es bastante grande y complejo, una simple formulación matemática puede no ser factible. En este caso, el enfoque de simulación es por lo general preferible al enfoque analítico.

En común con el modelo analítico, la simulación puede dejar de lado algunos detalles, ya que los detalles en exceso pueden dar lugar a una simulación difícil de manejar y un esfuerzo computacional considerable [11]. Es importante considerar cuidadosamente una medida en estudio y no incluir detalles irrelevantes en la simulación.

Aspectos claves en la simulación incluyen la adquisición de información válida sobre lo que se quiere simular, la selección de características y comportamientos relevantes, el uso de aproximaciones, la fidelidad y la validez de los resultados de la simulación. Permite experimentación orientada a los sistemas dinámicos, es decir, sistemas con un comportamiento dependiente del tiempo. Se ha convertido en una importante tecnología y es ampliamente utilizada en muchas áreas de investigación científica [11].

Además, el modelado y la simulación son etapas esenciales en el diseño de ingeniería y de resolución de problemas de procesos, y se llevan a cabo antes de que un prototipo físico esté construido. Los ingenieros utilizan computadores para diseñar las estructuras físicas y para hacer modelos que simulan el funcionamiento de un dispositivo o técnica. Las fases de modelado y la simulación son a menudo la parte más larga del proceso de diseño en ingeniería.

Las fases de modelado y simulación por lo general pasan por varias iteraciones como pruebas de ingeniería de varios diseños para crear el mejor producto o la mejor solución a un problema. La simulación proporciona un método para comprobar la comprensión de todo el mundo y ayuda a producir mejores resultados más rápidamente. Se puede utilizar la simulación cuando la solución analítica no existe o es demasiado complicada. La simulación [11] no se debe utilizar en los siguientes casos:

- La simulación requiere un largo tiempo de cómputo: en este escenario es probable que no sea posible realizar simulaciones a menos que se implementen mecanismos para la distribución de las tareas de la simulación y se acorte el tiempo de la misma.
- La solución analítica existe y es sencilla: es más fácil utilizar la solución analítica para resolver el problema en lugar de usar la simulación, a menos que se requiera comparar la solución analítica con la simulación.

En los modelos de simulación hay algunas técnicas de evaluación de desempeño que se pueden aplicar en general, aunque muchas de éstas se basan en modificaciones del método llamado Monte Carlo [11]. Se puede simular el sistema para estudiar su comportamiento de forma parcial o total y el modelo que simula el sistema toma valores dados del sistema a evaluar o algún otro sistema similar para arrojar los resultados para su posterior análisis [11], para poder realizar así evaluaciones de desempeño orientadas al sistema real. Se dice que por lo general los simuladores son lentos pero tienen la gran ventaja de disminuir costos, ya que no se requiere la existencia de los componentes del sistema que se quiere evaluar.

3.3 Benchmarking

Una técnica obvia para evaluar el rendimiento de un sistema es la medición y evaluación directa [19]. Sin embargo, existen limitaciones en las mediciones disponibles por las siguientes razones:

- La medición directa de un sistema sólo está disponible con aquellos que ya se encuentren totalmente operativos. La medición directa no es posible con los sistemas en fase de diseño y desarrollo.
- La medición directa del sistema puede afectar al sistema medido mientras se obtienen los datos requeridos. Esto puede llevar a falsas mediciones del sistema estudiado.

- Puede que no sea práctico medir directamente el nivel de rendimiento de extremo a extremo sostenido en todas las rutas del sistema. Por lo tanto, para conseguir los objetivos de rendimiento, los modelos analíticos tienen que ser establecidos para convertir las mediciones en bruto en las medidas de rendimiento significativas.
- En muchos casos, no está permitida la evaluación de desempeño en sistemas reales de uso crítico debido a las implicaciones posibles en caso de falla inducida por dicha medición.

En computación, un benchmark es el acto de ejecutar un programa computacional, un conjunto de programas, u otras operaciones, a fin de evaluar el rendimiento relativo de un objeto o sistema, normalmente mediante la ejecución de una serie de pruebas estándares [7]. La ventaja de utilizar un benchmark para evaluar un sistema es que permite evaluar el sistema de computación como un conjunto. La ejecución de un programa real evalúa todos los aspectos del sistema, incluyendo los aspectos relativos al hardware, la sobrecarga del sistema operativo, las optimizaciones de compilación, etc [9]. Por esta razón, el uso de benchmarks resulta la alternativa más práctica al momento de medir el desempeño de dispositivos, puesto que ofrecen una visión más realista del sistema, incluyendo en el estudio los factores externos que afecten su rendimiento.

El término *benchmarking* es usualmente asociado con la evaluación de las características de rendimiento del hardware, por ejemplo, el rendimiento de la operación punto flotante de un CPU, pero hay circunstancias en que la técnica también se aplica al software [7]. Un benchmark para software, por ejemplo, podría consistir en ejecutar tareas en los compiladores o sistemas de gestión de bases de datos para evaluar su desempeño.

Como la arquitectura de los computadores ha avanzado, se ha hecho más difícil comparar el rendimiento de diversos sistemas informáticos sólo con ver sus especificaciones [7]. Por lo tanto, se han desarrollado pruebas que permiten la comparación de diferentes arquitecturas. Los benchmarks están diseñados para imitar un determinado tipo de carga de trabajo de un componente o sistema. Los benchmarks sintéticos hacen esto a través de los programas especialmente creados que imponen la carga de trabajo en el componente.

Un buen benchmark [7] debe de poseer las siguientes propiedades:

- Linealidad: si el rendimiento de una máquina es tres veces mejor que otra máquina, el resultado del benchmark de la primera máquina debería ser tres veces más alto que aquel de la segunda máquina.
- Fiabilidad: si el resultado del benchmark para una máquina es más alto que el resultado del benchmark de otra máquina, el rendimiento de la primera máquina debería ser siempre más alta para la métrica referenciada.
- Repetición: el resultado del benchmark debería ser el mismo si el proceso es repetido.
- Facilidad de medición: cuando la medición se lleva a cabo por un proceso complejo, hay una alta probabilidad de errores en el propio benchmark.
- Consistencia: la definición de la métrica de rendimiento es válida en diferentes máquinas.
- Independencia: el benchmark no ha sido influenciado por las solicitudes de las manufactureras que desean colocar las mejores partes de sus productos.

La construcción de buenos benchmarks para los procesos de red es difícil, es de por sí dependiente del contenido de los paquetes procedentes de la red. Esto dificulta la construcción de un benchmark justo [7]. Los benchmarks existentes enfrentan este problema de diferentes maneras, por ejemplo, mediante la ejecución del benchmark con sólo ciertos tamaños de paquete o una cierta mezcla de

tamaños de paquetes. Lo ideal sería que el rendimiento de un procesador de red pudiese ser medido independientemente del tráfico de la red.

Hay básicamente dos formas prácticas para benchmarks [7]:

- Benchmark de aplicaciones típicas.
- Benchmark de solo un aspecto del sistema a la vez o *microbenchmark*.

Una aplicación es algo que es útil en sí misma, mientras que una tarea es algo que no es útil en sí misma, sino más bien una parte específica de una aplicación más grande. El enrutamiento es una aplicación típica del procesamiento de la red, mientras que la gestión de colas en un router es una tarea. El benchmark de aplicaciones típicas es ideal si la aplicación que interesa es similar a la aplicación a ser referenciada. Pero los resultados no son muy útiles si la aplicación prevista no es similar a la aplicación referenciada. Un gran inconveniente de este enfoque es que consume bastante tiempo para colocar en práctica un gran número de aplicaciones optimizadas como sea posible en diferentes sistemas.

Los *microbenchmarks* representan otra forma de medir el desempeño en la identificación de las tareas típicas y medir el rendimiento de éstos en *microbenchmarks*. Los *microbenchmarks* fueron concebidos para los usuarios que son conscientes de las tareas implicadas en una determinada aplicación. Los *microbenchmarks* son útiles en este caso, no se decide de una vez, pero se indica un número de procesos que son de interés, es decir, una investigación más a fondo acerca de qué procesos de red pueden ser tomados en cuenta para un mayor rendimiento. El principal inconveniente de este tipo de benchmark es, por supuesto, que es difícil de encontrar tareas pequeñas representativas. Otro inconveniente es que es fácil llegar a conclusiones erróneas a partir de *microbenchmarks* si la aplicación prevista no se entiende bien. La principal ventaja es que es razonable para comparar una amplia variedad de sistemas porque los *microbenchmarks* no son tan difíciles de crear.

4. Network Simulator 3

En este capítulo se presenta un estudio acerca del simulador de red ns-3, resaltando sus principales características, ventajas y desventajas. Adicionalmente, se analizan todas las tecnologías que soporta y las herramientas que pueden ser utilizadas e integradas al simulador para facilitar y potencializar sus características.

4.1 Introducción

ns-3 [18] es un simulador de red basado en eventos discretos distribuido bajo la licencia GNU GPLv2[1]. La simulación de red basada en eventos discretos es una poderosa herramienta de investigación para estudiar el diseño de protocolos, las interacciones entre protocolos y el rendimiento a gran escala. Aunque la simulación no es la única herramienta utilizada para la investigación y obtención de datos acerca de un sistema de redes, es extremadamente utilizada porque a menudo permite que los prototipos e interrogantes a resolver sean explorados a menor costo y tiempo, a diferencia de las implementaciones reales. La simulación es también un enfoque muy efectivo en la educación, debido a que los conceptos claves pueden ser estudiados y resaltados claramente en forma aislada de otros elementos del sistema.

Por muchos años, la herramienta de simulación de red ns-2 [23] (predecesor de ns-3) fue el estándar de facto para los estudios académicos sobre los protocolos de red y métodos de comunicación. Innumerables documentos fueron escritos reportando resultados obtenidos usando ns-2 [38], y miles de nuevos modelos fueron desarrollados y contribuyeron con el código base de ns-2. A pesar de la popularidad de ns-2 y de la gran cantidad de simuladores de red existentes, se decidió diseñar un nuevo simulador que reemplazara a ns-2 para estudiar a los sistemas de redes. La decisión de desarrollar un nuevo simulador fue motivada por la visión de complementar de la mejor manera posible la forma en la cual se modelan las redes con la forma en la cual éstas son estudiadas. La experiencia colectiva en el uso y mantenimiento de ns-2 también influyeron en el desarrollo de ns-3 [38].

Una de las metas fundamentales del diseño de ns-3 fue mejorar el realismo de los modelos [18]. Diferentes herramientas de simulación toman diferentes enfoques de modelado, incluyendo el uso de lenguajes específicos de modelado, las herramientas de generación de código y los paradigmas de programación. Aunque los lenguajes de modelado de alto nivel y los paradigmas de programación específicos para simulación tienen cierta ventaja, al momento de modelar implementaciones reales no necesariamente son ventajosas [38]. Muchas veces, tener altos niveles de abstracción pueden causar que los resultados obtenidos de la simulación diverjan mucho de los resultados experimentales [38]. ns-3 se enfoca menos en estos niveles de abstracción y se centra más en el realismo.

ns-3 cuenta con unas herramientas poderosas y completas para simular sistemas de redes, e incluso también cuenta con la capacidad de emulación. Esta capacidad de emulación ya ha sido probada y utilizada con dispositivos y aplicaciones reales [17], logrando reducir un poco la brecha que existe entre la simulación y la experimentación.

4.2 Historia

Las siglas ns (network simulator) conforman el nombre que se le da a una serie de simuladores de red, específicamente ns, ns-2 y ns-3. Estos simuladores soportan protocolos populares de red, y

[1] http://www.nsnam.org

pueden ser utilizados en la simulación de protocolos de enrutamiento, en la investigación acerca de redes ad hoc, en el desarrollo de nuevos esquemas de seguridad, entre otras cosas.

El simulador ns comenzó a desarrollarse en 1989 como una variante del simulador de red REAL. En 1995, ns había ganado el apoyo de DARPA, el proyecto Vint de LBL, Xerox PARC, UCB y USC/ISI. ns ya no está en desarrollo pero aún es mantenido por voluntarios conformados por una serie de investigadores e instituciones, incluida la SAMAN (con el apoyo de DARPA), CONSER (a través de la NSF (National Science Fundation)), e ICIR (antes ACIRI). También han llegado contribuciones de largo alcance por parte de Sun Microsystems, la UCB Daedalus y Carnegie Mellon.

El simulador ns-2 está desarrollado en el lenguaje C++ y provee una interfaz de simulación a través de OTcl, una variante orientada a objetos del lenguaje Tcl (Tool command language). Es decir, los usuarios utilizan el lenguaje OTcl para describir la red de interconexión y el tráfico de datos asociado. ns-2 funciona en la mayoría de los sistemas operativos incluyendo Unix, MacOS y Windows a través del uso de Cygwin. Se distribuye bajo la licencia GNU GPLv2 y su última versión (v2.35) fue liberada en noviembre de 2011.

El simulador ns-3 surge en febrero de 2005 impulsado por Tom Henderson, PI (Principal Investigator) de ns-2, el cual inicialmente propuso discutir acerca de cómo ns-2 podría ser rediseñado como parte de un esfuerzo de desarrollo futuro, además de mantener el código de ns-2 existente. En el proceso de discusión acerca de los cambios necesarios, se encontró que en general no valía la pena mantener la compatibilidad hacia atrás con ns-2 debido a que la mayoría de los modelos más útiles estaban implementados en bifurcaciones o partes separadas que eran generalmente incompatibles unas con las otras [18]. Por este motivo, se decidió que el nuevo simulador iba a ser escrito desde cero, usando el lenguaje de programación C++.

En 2004, Mathieu Lacage comenzó a desarrollar YANS (Yet Another Network Simulator), el cual se utilizó después como base de ns-3. El desarrollo de ns-3 inicialmente fue patrocinado por la NSF, la Universidad de Washington, INRIA (Institut National de Recherche en Informatique et en Automatique) y Georgia Tech, y se proyectó para un periodo de tiempo de cuatro años, comenzando en julio de 2006. La primera versión, ns-3.1, se hizo pública en junio de 2008, y después el proyecto siguió haciendo lanzamientos trimestrales de versiones del software. En la actualidad, ns-3 hizo su décimo quinto lanzamiento con la versión ns-3.16 en diciembre del 2012.

4.3 Descripción General del Proyecto

ns-3 permite a los investigadores estudiar los protocolos de Internet y sistemas a gran escala en un ambiente controlado. La meta del proyecto ns-3 es desarrollar un entorno de simulación abierto, el cual se convierta en el medio preferido para realizar la investigación y estudio de las redes [18].

El proyecto ns-3 está comprometido a construir una base sólida de simulación que esté bien documentada, que sea fácil de usar y depurar, además, que responda a la necesidad de cumplir con el flujo de trabajo de la simulación completa partiendo desde la configuración de la simulación a una colección de registros obtenidos y su análisis.

La infraestructura de ns-3 permite el estudio y desarrollo de modelos de simulación de alto desempeño tanto en redes basadas en IP (Internet Protocol), como en las no IP. Sin embargo, la gran mayoría de los usuarios han tendido a centrase en simulaciones de escenarios IP inalámbricos los cuales involucran modelos como WiFi, WiMAX, o LTE (Long Term Evolution) para capas 1 y 2 del modelo OSI y una variedad de protocolos de enrutamiento dinámicos como OLSR [6] (Optimized Link State Routing) y AODV [32] (Ad hoc On-Demand Distance Vector Routing).

4.4 Soporte para C++ y Python

Muchos simuladores usan un lenguaje de modelado de dominio específico para describir modelos y el flujo del programa. ns-3 utiliza C++ o Python, permitiendo a los usuarios tomar ventaja del soporte completo para cada lenguaje. Los usuarios normalmente interactúan con la biblioteca ns-3 mediante la codificación de aplicaciones C++ o Python. Estas aplicaciones crean instancias de los modelos de simulación para establecer el escenario de simulación de interés, donde dichas instancias interactúan entre sí durante la simulación.

El conjunto de modelos de simulación de redes que provee ns-3[2] son implementados como objetos de C++. Sin embargo, existe la posibilidad de acceder a estos modelos desde Python gracias a los *Python Wrappers* con los cuales se da soporte a Python en ns-3 como se muestra en la Figura 4.1. Para ello, se utiliza la biblioteca *pybindgen* la cual delega el análisis de las cabeceras de C++ de ns-3 a *gccxml* y *pygccxml* que generan automáticamente el correspondiente enlace con C++. De esta manera, se generan archivos C++ que son finalmente compilados en el módulo Python de ns-3. En otras palabras, se logra encapsular Python dentro de ns-3 permitiendo la interacción con el núcleo y los modelos escritos en C++ a través de scripts Python.

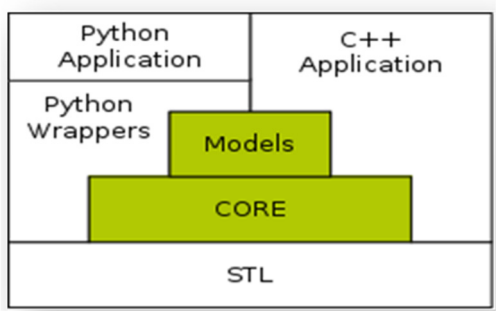


Figura 4.1: Claves de la Tecnología

4.5 Enfoque Orientado a Objetos

ns-3 es una herramienta de simulación diseñada para ser fácilmente extendida. Para alcanzar este objetivo [19], se llevó a cabo el diseño del núcleo de ns-3 en C++ bajo el enfoque orientado a objetos con una gran cantidad de clases base, que describen la funcionalidad básica y subclases que implementan dicha funcionalidad [18]. Por ejemplo, ns-3 cuenta con una clase base Queue que describe los métodos necesarios para todas las implementaciones específicas de colas. Entonces, las subclases de Queue implementan varios tipos de métodos de encolamiento, como Droptail y RED (Random Early Detection). Similarmente, una clase TCP base describe la implementación de muchas de las funcionalidades del protocolo TCP, pero las subclases proveen funcionalidades específicas de las variantes de TCP como lo son Reno, new Reno y SACK.

Algunas funcionalidades pueden ser logradas mediante el uso de *Callbacks* en varios puntos de la simulación. Por ejemplo, un *Callback* entre las capas 2 y 3 pueden implementar funcionalidades de un firewall u otro dispositivo de filtrado de red sin requerir modificación alguna de la implementación del modelo en la capa 2 o 3.

[2] http://www.nsnam.org/overview/key-technologies

4.6 Realismo

El diseño básico de una simulación está relacionado de manera muy cercana con el diseño de una red real y con sus elementos. Cuando surgen preguntas acerca del diseño de un objeto, función, o interfaz, las repuestas deberían ser encontradas pensando en cómo serían implementadas en la vida real, a menos de que existan fuertes consideraciones de desempeño y eficiencia. Estos principios de diseño deberían influir positivamente en la portabilidad del código y en su uso de manera educativa.

ns-3 mantiene una clara distinción entre las capas de protocolos en el diseño del software, con los bien definidos puntos de intercambio entre las capas como se muestra en la Figura 4.2 [25], facilitando la decisión de multiplexación y demultiplexación en el flujo de paquetes hacia arriba y abajo en la pila. Objetos de la simulación en representación de los paquetes consisten en uno o más PDUs (Protocol Data Unit) con objetos subclases que representan los distintos tipos de protocolos. También se tienen objetos que representan aplicaciones las cuales interactúan con protocolos de la capa 4 de manera muy familiar para los desarrolladores de aplicaciones de Internet, incluyendo llamadas para el establecimiento de conexión, envío y recepción de datos, cierre de conexiones, etc.

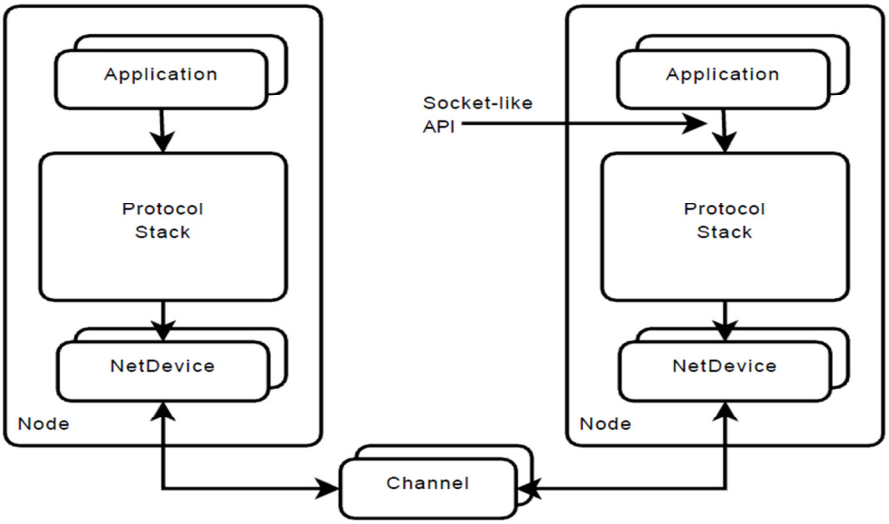


Figura 4.2: Modelo Básico de Comunicación de ns-3

Al utilizar una simulación hay que entender bien qué se está modelando y qué no se encuentra permitido. Un modelo es por definición una abstracción de la realidad[3]. Es en última instancia la responsabilidad por parte del autor del script determinar cuál es el rango de precisión y el dominio de aplicabilidad de su simulación.

4.6.1 Énfasis en Emulación

La infraestructura del software ns-3 estimula el desarrollo de modelos de simulación que son lo suficientemente realistas para permitir que ns-3 sea utilizado como un emulador de una red en tiempo real. Gracias al planificador de ns-3 se simplifica el manejo de la simulación en estos casos donde se interactúa con sistemas reales.

ns-3[4] tiene la facultad de integrarse en bancos de pruebas y ambientes con máquinas virtuales (Figura 4.3). Para ello, se proporcionan dos tipos de interfaz de red. El primer tipo denominado

[3] http://www.nsnam.org/docs/tutorial/singlehtml/index.html
[4] http://www.nsnam.org/docs/release/3.12/models/singlehtml/index.html#emulation-overview

EmuNetDevice permite enviar data en una red real. El segundo tipo denominado *TapNetDevice* le permite a un host real participar en una simulación ns-3 como si fuera uno de los nodos simulados.

Para intercambiar paquetes entre el simulador y una red verdadera, se crean objetos ns-3 de tipo *packet* que son almacenados internamente en un *buffer* de bytes (similar a paquetes en un sistema operativo real), listo para ser serializado y enviado a una interfaz de red real.

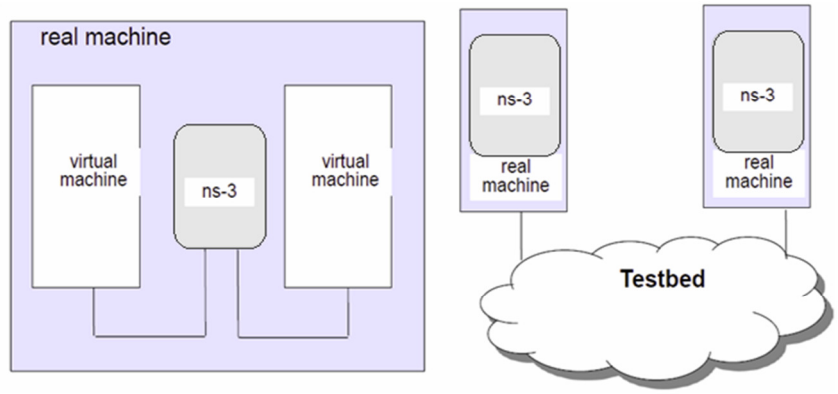


Figura 4.3: Emulación entre Máquinas Virtuales y entre Máquinas Reales

4.6.2 Correspondencia con las Interfaces del Mundo Real

Los nodos en ns-3 siguen el modelo de la arquitectura de red de Linux, donde interfaces claves y objetos (sockets, dispositivos de red) corresponden con los de un computador con Linux. Esto hace destacar el realismo del modelo y hace el flujo de control del simulador más fácil de comparar con sistemas reales. Además, los nodos están diseñados para hacer una representación más fiel de los equipos reales, donde se pueden tener múltiples interfaces de red por nodos, múltiples direcciones IP para una sola interfaz, etc.

4.6.3 Software de Integración

Otro énfasis del simulador está en la reutilización de aplicaciones de red existentes. ns-3 [16] posee una arquitectura para soportar la incorporación de software para sistemas de redes de código libre tales como la pila de protocolos del kernel, demonios de enrutamiento y analizadores de paquetes. Gracias a que ns-3 cuenta con capas de abstracción e interfaces para portar implementaciones de código, se reduce la necesidad de reescribir modelos y herramientas requeridas para la simulación.

Existen frameworks para ejecutar aplicaciones sin modificar su código fuente o para reutilizar toda la pila de red del kernel Linux dentro de ns-3. Estos aspectos están siendo probados y evaluados (Figura 4.4) [16] a través del proyecto DCE (Direct Code Execution).

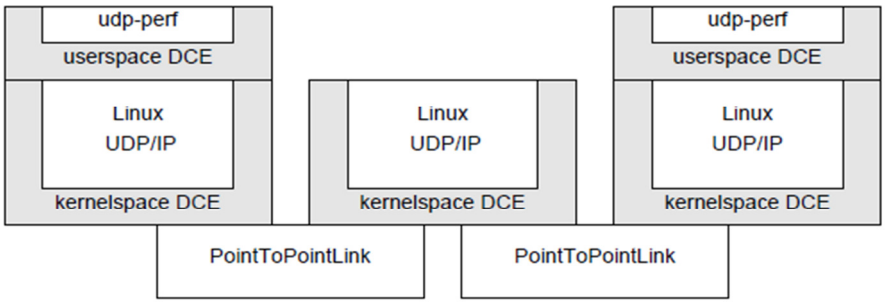


Figura 4.4: Escenario de Prueba con DCE

4.7 Núcleo Flexible con Nivel de Ayuda

ns-3 cuenta con un potente API (Application Programming Interface) a bajo nivel que da a los usuarios la flexibilidad suficiente para configurar características y objetos de diferentes maneras. En capas superiores se encuentran un conjunto de APIs que conforman la capa de *helpers* que proveen funciones de uso muy sencillo para realizar tareas diversas. Los usuarios de ns-3 pueden mezclar y combinar entre el API simple y los *helpers*.

4.8 Falta de un IDE

El proyecto no mantiene un entorno de desarrollo integrado o IDE (Integrated Development Enviroment) para configurar, depurar, ejecutar y visualizar simulaciones en un mismo ambiente, como se tiene en otros simuladores. Dicho de otra manera, ns-3 no cuenta con un entorno de programación empaquetado, conformado por un editor de código, un compilador, un depurador y una interfaz gráfica. En cambio, el flujo de trabajo típico es trabajar a través de la línea de comandos y utilizar herramientas independientes de apoyo al desarrollo como gdb, NetAnim, etc. Sin embargo, algunos desarrolladores han utilizado Eclipse como IDE de desarrollo. Eclipse puede ser integrado con ns-3[5] de forma muy fácil y rápida mediante algunas configuraciones que permiten usar este IDE para desarrollar, compilar, depurar y ejecutar scripts de ns-3.

4.9 Comunidad ns-3

Como proyecto de código abierto, ns-3 confía en la contribución de la extensa comunidad de usuarios e investigadores. Es posible retribuirle algo al proyecto mediante:

- La lista de discusión[6]: si se conoce la respuesta a una pregunta o simplemente se siente que se puede contribuir de manera constructiva en una discusión.
- El reporte de posibles bugs o errores: pueden ser reportados y se puede hacer un seguimiento de su resolución.
- Tutoriales o contenido en el wiki[7]: si no se tiene una solución significativa u obvia a un problema, se puede escribir algo de documentación.
- Código: pueden ser escritos nuevos modelos o modificar alguno de ellos, agregar nuevas características, o solucionar algún bug.

4.10 Documentación

A pesar de ser un proyecto de código abierto, n-3 cuenta con una muy buena documentación. En la página principal de ns-3[8] se encuentra un tutorial, un manual, los documentos de los modelos y del API, y muchos otros documentos.

- Tutorial: tiene como propósito introducir a los nuevos usuarios de ns-3 al sistema de manera estructurada. En ocasiones se les hace difícil a los nuevos usuarios captar la información esencial del manual de ns-3 y convertir esta información en simulaciones funcionales. En este tutorial se realizan muchos ejemplos de simulaciones, de manera tal que se introduzcan y se expliquen los conceptos claves y especificaciones para comenzar a utilizar la herramienta.
- Manual: es un documento que cubre en mayor profundidad la arquitectura y el núcleo de ns-3. Además, este manual presenta los modelos y las capacidades que soporta el simulador.

[5] http://www.nsnam.org/wiki/index.php/HOWTO_configure_eclipse_with_ns-3
[6] http://groups.google.com/group/ns-3-users
[7] http://www.nsnam.org/wiki/index.php/Main_Page
[8] http://www.nsnam.org

- Model Library: este manual está enfocado hacia la documentación de los modelos ns-3 y el software de soporte. Hasta la fecha este documento está muy incompleto.
- API de ns-3: se encuentra extensamente documentado por medio de Doxygen, un generador de documentación para C++, C y Java que permite generar documentación para código fuente.

4.11 Módulos de ns-3

En la Figura 4.5 [17] se puede apreciar algunos de los módulos de ns-3. Cabe destacar que varios de estos módulos son módulos padres que contienen sub-módulos. Es el caso de los módulos Core, Network, Applications e Internet que son módulos generales que contienen varios sub-módulos. Estos módulos serán descritos a continuación para conocer su alcance y las tecnologías que soporta ns-3.

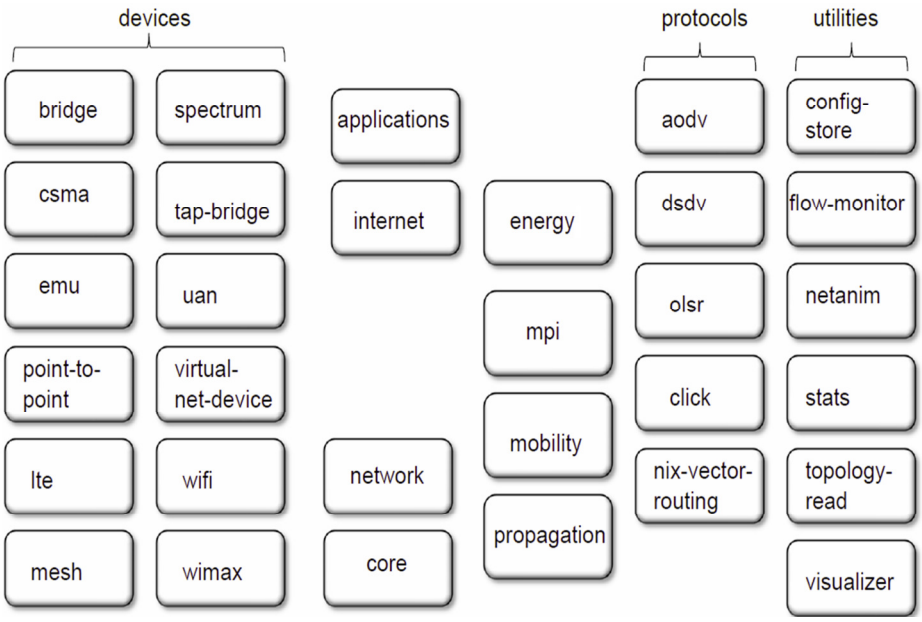


Figura 4.5: Módulos de ns-3 Disponibles a la Fecha de Mayo de 2011

4.11.1 Módulo Core

El módulo Core representa el módulo principal de ns-3 ya que como su nombre lo indica, contiene al núcleo del simulador. Como se puede ver en la Figura 4.6, tiene varios sub-módulos como Time, Schedule, Object, etc. El módulo Time contiene una clase manejadora de tiempo que permite almacenar valores de tipo tiempo y hacer conversiones de éste entre diversas unidades de tiempo. El módulo Schedule contiene una clase base para planificadores y se utiliza para implementar nuevos planificadores de eventos de simulación. El módulo Object provee manejo de memoria y agregación de objetos. Además, hay módulos encargados de manejar el sistema de atributos (Attribute Helper), la creación de funciones *Callbacks* (MakeCallback), el sistema de seguimiento (Tracing), etc.

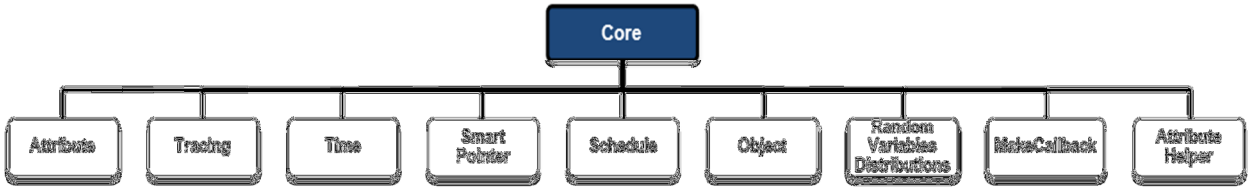


Figura 4.6: Módulo Core

4.11.2 Módulo Applications

El módulo Applications (Figura 4.7) contiene una clase base para las aplicaciones de ns-3 y un conjunto de módulos que describen las diversas aplicaciones soportadas. El módulo OnOffApplication permite crear aplicaciones On-Off (aplicaciones que generan tráfico hacia un destino dado alternando periodos con una demanda constante con periodos sin demanda). El módulo PacketSink permite implementar aplicaciones que reciben paquetes antes de descartarlos. Además, existen módulos para modelar aplicaciones UDP cliente/servidor sin echo (UdpClient/UdpServer), UDP cliente/servidor con echo (UdpEchoClient/UdpEchoServer), demonios de autoconfiguración para IPv6 (Radvd), etc.

Figura 4.7: Módulo Applications

4.11.3 Módulo Internet

El módulo Internet (Figura 4.8) contiene los módulos que describen o modelan la pila TCP/IP. Entre ellos se encuentran modelos para los protocolos IPv4 (Internet Protocol version 4), IPv6 (Internet Protocol version 6), ARP (Address Resolution Protocol), UDP (User Datagram Protocol) y TCP (Transmission Control Protocol*)*, así como también distintas versiones mejoradas de TCP (Reno, New Reno y Tahoe).

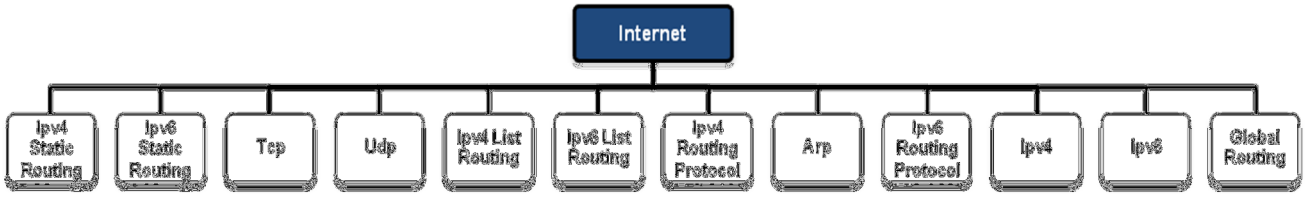


Figura 4.8: Módulo Internet

Además, dentro del módulo Internet se tienen los módulos Ipv4 Rounting Protocol (Figura 4.9) e Ipv6 Rounting Protocol (Figura 4.10) donde se encuentran los módulos que representan protocolos de enrutamiento tanto para IPv4 como para IPv6.

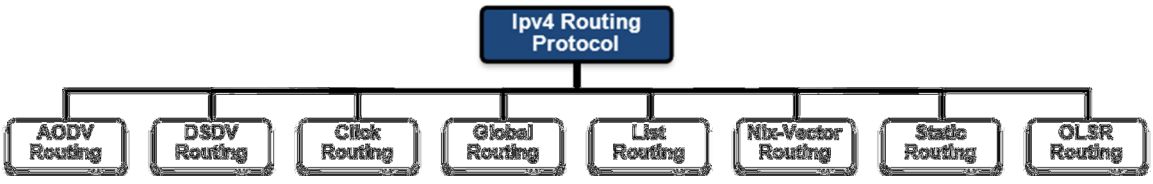


Figura 4.9: Módulo Ipv4 Routing Protocol

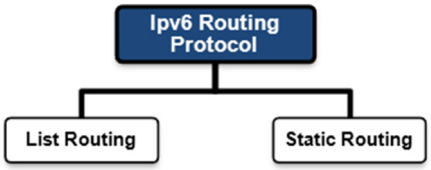


Figura 4.10: Módulo Ipv6 Routing Protocol

4.11.4 Módulo Network

El módulo Network (Figura 4.11) contiene los módulos que se encargan de modelar algunos elementos implicados en las redes de comunicación. El módulo Socket permite modelar sockets a bajo nivel basado en el API de socket BSD (Berkeley Software Distribution). El módulo Packet se utiliza para modelar paquetes de red donde cada paquete contiene un buffer de bytes que almacena el contenido serializado de las cabeceras y colas añadidas al paquete. Además, también se tienen modelos para las direcciones IP (Address), tanto de IPv4 como de IPv6, modelos para corromper paquetes (Error Model), modelos de los canales lógicos por donde fluye la información (Channel), etc.

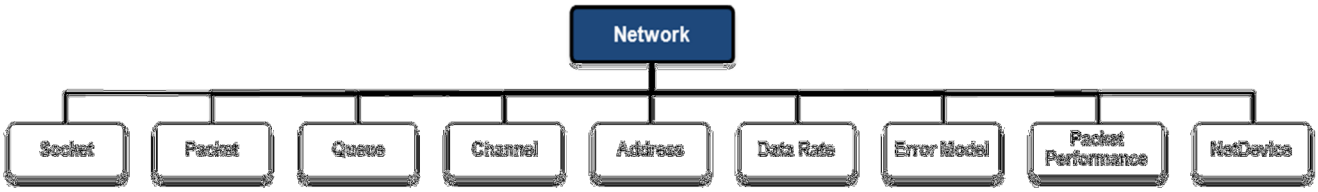


Figura 4.11: Módulo Network

Los dispositivos de red son modelados en el módulo NetDevice indicado en la Figura 4.12. Como se puede observar, ns-3 tiene soporte para una gran variedad de tecnologías de red incluyendo CSMA, Point-to-Point, WiFi, WiMAX, LTE, etc.

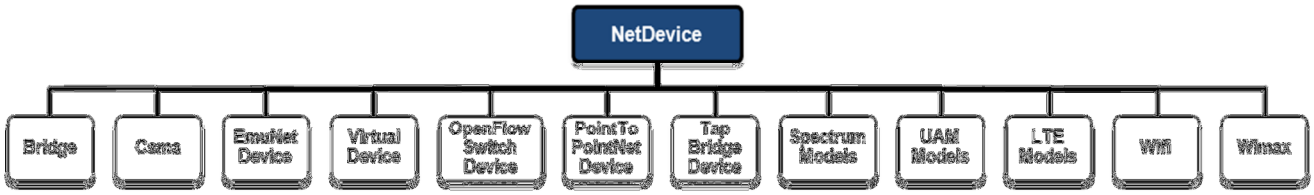


Figura 4.12: Módulo NetDevice

4.11.5 Otros Módulos

Existen también un grupo de módulos ilustrados en la Figura 4.13 que ofrecen un conjunto de herramientas que son útiles al momento de realizar una simulación. El módulo Config-Store brinda la posibilidad de almacenar y cargar los atributos de configuración de la simulación. El módulo NetAnim permite mostrar de forma gráfica la simulación, creando una animación como se muestra en la Figura 4.14. El módulo Tools permite calcular promedios, el retardo de los paquetes, la varianza del retardo (*jitter*), entre otras cosas. Además, existen módulos con modelos de propagación (Propagation Models), modelos de energía (Energy Models), modelos de movilidad (Mobility Models), una herramienta para correr simulaciones de forma distribuida (MPI Distributed Simulation), etc.

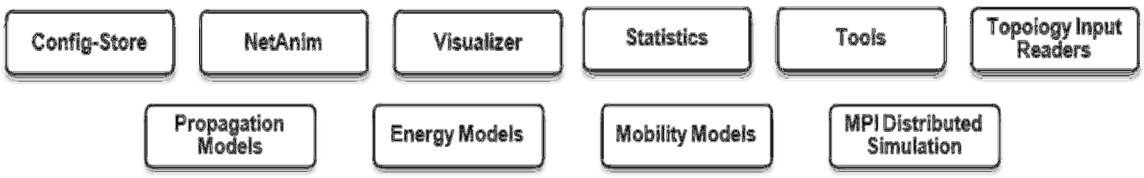


Figura 4.13: Otros Módulos

Con NetAnim (Network Animator), los usuarios puede cambiar el zoom de la red simulada (zoom in, zoom out), dar un paso hacia adelante o hacia atrás en la simulación (step forward, step backward), observar el recorrido de los paquetes desde el emisor hasta el destinatario, etc.

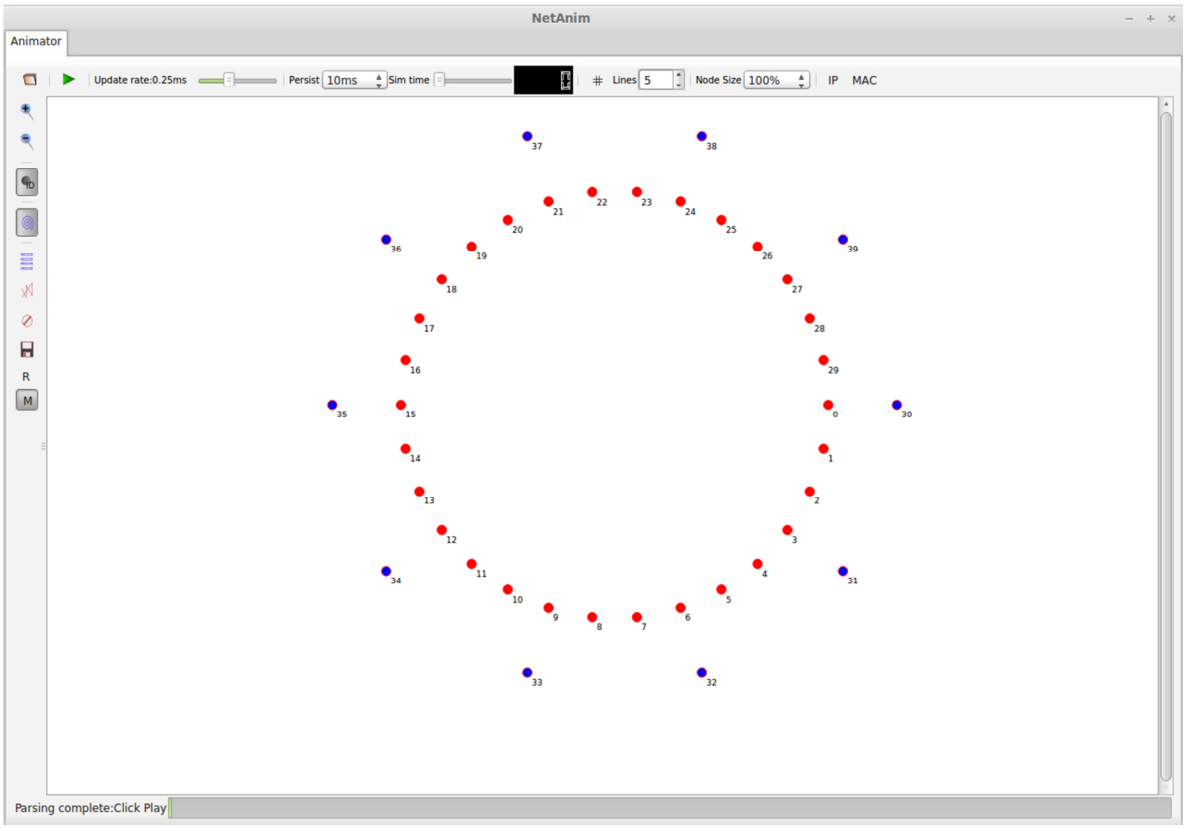


Figura 4.14: NetAnim

5. OMNeT++

En este capítulo se presenta un estudio acerca del simulador de red OMNeT++ resaltando sus principales características, ventajas y desventajas. Adicionalmente, se analizan todas las tecnologías que soporta y las herramientas que pueden ser utilizadas conjuntamente con el simulador para facilitar y potenciar sus características.

5.1 Introducción

OMNeT++ [37] es un entorno de simulación basado en eventos discretos distribuido bajo Licencia Publica Académica[9]. La simulación basada en eventos discretos [1] complementa a las herramientas analíticas (matemáticas) y a los métodos experimentales para la estimación y análisis del desempeño de las redes de comunicaciones. Por esta razón, OMNeT++ es utilizado principalmente en el área de la simulación de redes de comunicaciones. Además, permite diseñar buenos modelos de simulación que pueden proveer información importante acerca de las debilidades existentes dentro de la estructura de una red de computadores o de protocolos de comunicación.

En lugar de ser un simulador especializado en algún área de aplicación, OMNeT++ fue diseñado para ser lo más general posible. Por esta razón, posee una arquitectura genérica que puede ser (e incluso ha sido) usada en problemas de diferentes dominios. Generalmente ha sido utilizado para modelar tráfico en sistemas de redes de comunicaciones, protocolos de red, sistemas paralelos y distribuidos, validación de arquitectura de hardware, evaluación de desempeño de sistemas software, y en general para modelar cualquier sistema que pueda simularse con eventos discretos.

OMNeT++ [37] fue diseñado desde un principio para soportar simulaciones de redes a gran escala con periodos de procesamiento rápidos y sin limitar la ejecución de distintos eventos que se puedan simular en la red. Para cumplir estos objetivos, los modelos de simulación se construyen jerárquicamente y a partir de componentes reutilizables, tanto como sea posible. Además, OMNeT++ facilita la visualización y depuración de los modelos de simulación de forma tal que pueda reducirse el tiempo invertido en depurar, lo cual tradicionalmente representa un gran porcentaje en el desarrollo del proyecto de simulación. Estas últimas características son también un punto a favor en cuanto al uso del simulador con propósitos educativos.

La principal motivación para el desarrollo de OMNeT++ [37] fue crear y desarrollar una poderosa herramienta de simulación que pudiese ser utilizada con propósitos de educación y de investigación acerca de las redes de computadores y sistemas distribuidos. OMNeT ++ intenta llenar la brecha que existe entre sistemas de código libre como ns-3 (descrito en el Capítulo 3) y alternativas comerciales costosas como OPNET, un simulador de redes desarrollado por OPNET Technology[10]. En la actualidad, OPNET se perfila como una de las soluciones propietarias más completa para la simulación de redes existente en el mercado.

OMNeT++ [37] se describe como un framework donde, en lugar de proveer directamente los componentes para simular redes de computadores o de otros dominios, proporciona el entorno y las herramientas básicas para escribir tales simulaciones. Diferentes áreas de aplicación son soportadas por frameworks. Estos frameworks son desarrollados de manera completamente independiente de OMNeT++. Estas características hacen de OMNeT++ un simulador bastante potente con la capacidad de adaptarse a las condiciones del problema para hacer que la simulación sea lo más real posible.

[9] http://omnetpp.org/home/license
[10] http://www.opnet.com

5.2 Historia

Las siglas OMNeT++ (Objective Modular Network Testbed in C++) identifican a un entorno de simulación creado en el año 1992. Surge a partir de una asignación de programación de la Universidad Técnica de Budapest (Hungría). András Vargas lo diseñó tomando como base al simulador OMNeT escrito en lenguaje de programación Pascal por su profesor, el Dr. Gyrgy Pongor. Durante años, muchas personas han contribuido con OMNeT++, en particular estudiantes provenientes de la Universidad Técnica de Delft (Holanda) y Budapest. Actualmente, András Vargas todavía se encarga del mantenimiento de OMNeT++.

OMNeT++ fue liberado al público en el año 1997 y para el año 1998 se le agregaron funcionalidades de animación que hicieron que el simulador fuese mucho más utilizable para la educación. Para el año 2000, miembros de la Universidad de Karlsruhe (Alemania) crearon el modelo TCP para OMNeT++. La última versión de OMNeT++ (versión 4.2) fue liberada en noviembre de 2011.

5.3 Descripción General del Proyecto

OMNeT++ [37] es una biblioteca y framework de simulación extensible, modular, con componentes basados en C++ y que además incluye un entorno gráfico integrado para el diseño, ejecución, visualización y depuración de la simulación. Como se mencionó anteriormente, OMNeT++ en sí mismo no es un simulador de algo en concreto sino que proporciona la infraestructura y las herramientas para la escritura de la simulación. En OMNeT++ los modelos son ensamblados partiendo de componentes reutilizables denominados *módulos.* A través de la infraestructura de OMNeT++ es posible construir simulaciones basadas en estos componentes y su configuración (lenguaje NED, archivos INI). El simulador también provee funcionalidades para la ejecución de simulaciones por lotes, en tiempo real, emulación, conectividad con bases de datos, etc.

El kernel y las bibliotecas de OMNeT++ están escritos en C++. Las bibliotecas de simulación proveen mecanismos para manejar mensajes, soportar la configuración y ensamblado de módulos, generar números aleatorios con varias distribuciones, crear colas, recolectar estadísticas, enrutar, entre otros.

Las simulaciones en OMNeT++ [37] pueden ser ejecutadas bajo varias interfaces de usuario. La interfaz gráfica de usuario para animaciones es altamente útil para realizar demostraciones y para realizar depuración mientras que la interfaz de línea de comando es la mejor opción para ejecuciones en lote. El simulador, así como sus interfaces de usuario y herramientas, son muy portables y han sido adoptados para los sistemas operativos más comunes (Unix, Windows, MacOS).

OMNeT++ [37] también soporta simulaciones distribuidas y paralelas. Puede usar varios mecanismos para la comunicación entre las partes de una simulación distribuida y paralela como MPI (Message Passing Interface) o los denominados *pipes*. Los algoritmos de simulación en paralelo pueden ser fácilmente extendidos o incluso nuevos algoritmos pueden ser añadidos fácilmente. Los modelos no necesitan ninguna instrumentación especial para ejecutarse en paralelo, sólo es cuestión de configuración. OMNeT++ puede ser utilizado para hacer presentaciones de algoritmos en paralelo en salones de clase ya que la simulación puede ser ejecutada en paralelo incluso bajo el GUI (Graphic User Interface) obteniendo un *feedback* detallado acerca de lo que está ocurriendo.

OMNEST[11] es la versión comercial de OMNeT++ que requiere una licencia de Simulcraft Inc para poder ser utilizada.

[11] http://www.omnest.com

5.4 Estructura del Modelo

Muchos simuladores de red tienen más o menos definida la manera con la cual representan a los elementos de la red dentro del modelo. En contraste, OMNeT++ [37] provee una arquitectura de componentes genérica y le corresponde al diseñador del modelo mapear los conceptos de dispositivos de redes, protocolos o canales inalámbricos dentro de los componentes del modelo. Si se logra hacer un buen diseño de estos componentes o módulos, pueden ser usados en diferentes entornos y estarían en la capacidad de ser combinados de varias maneras como bloques de LEGO. Estos módulos principalmente se comunican mediante el pase de mensajes. Los mensajes pueden representar eventos, paquetes, comandos, trabajos u otras entidades dependiendo del dominio del modelo.

Un modelo en OMNeT++ [36] está compuesto por módulos que se comunican mediante el intercambio de mensajes. Los módulos activos son denominados módulos simples (simple modules) y son escritos en C++. Los módulos simples pueden ser agrupados dentro de módulos compuestos (compound modules) donde el número de niveles jerárquicos es ilimitado, es decir la profundidad de anidación de los módulos no se encuentra limitada. Los mensajes pueden enviarse por medio de las conexiones existentes entre los módulos o directamente hacia el destinatario. En la Figura 5.1 se puede apreciar los módulos simples, compuestos y las puertas (gates) de comunicación. Al modelo completo en OMNeT++ se le llama *network* y es un módulo compuesto sin puertas hacia el mundo exterior.

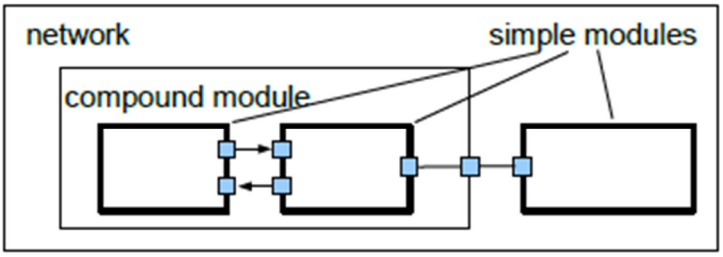


Figura 5.1: Módulos Simples y Compuestos

En la simulación de redes, los módulos simples pueden representar agentes de usuarios, generadores de tráfico y *sinks*, entidades de protocolos tales como TCP, dispositivos de red como interfaces IEEE 802.11, estructuras de datos como tablas de enrutamiento, etc. Las funciones correspondientes a la simulación tales como la asignación automática de direcciones IP o el control del movimiento de nodos móviles también son representadas a menudo a través de los módulos simples. Los nodos de la red tales como hosts y routers son representados típicamente mediante módulos compuestos que son ensamblados a partir de módulos simples.

Los mensajes de comunicación entre los módulos contienen datos arbitrarios además de los atributos usuales como la marca de tiempo (*timestamp)*. Los módulos simples envían mensajes a través de las puertas, pero también es posible enviarlos directamente al módulo destino. Las puertas son las interfaces de entrada y salida del módulo, los mensajes se envían por medio de las puertas de salida y son recibidos a través de las puertas de entrada. Una puerta de entrada y una de salida pueden estar enlazadas por una conexión. Las conexiones son creadas dentro de un solo nivel de la jerarquía de módulos. Dentro de un módulo compuesto, se pueden conectar las puertas correspondientes a dos sub-módulos o una puerta de un sub-módulo con una del módulo compuesto.

Debido a la estructura jerárquica del modelo, los mensajes generalmente viajan a través de una cadena de conexiones, partiendo y llegando en módulos simples. Los módulos compuestos actúan

como “cajas de cartón” en el modelo, de forma transparente transmiten mensajes entre su mundo interior y el exterior. Parámetros como retardo de propagación, tasa de datos, y tasa de errores pueden ser asignados a las conexiones a través de canales (channel).

Como lo muestra la Figura 5.2 los módulos y canales son una especie de componentes. Los módulos son representados por la clase abstracta cModule y los canales con la clase abstracta cChannel, ambas subclases de cComponent. Los módulos simples se representan por con la clase cSimpleModule y los compuestos con la clase cCompoundModule.

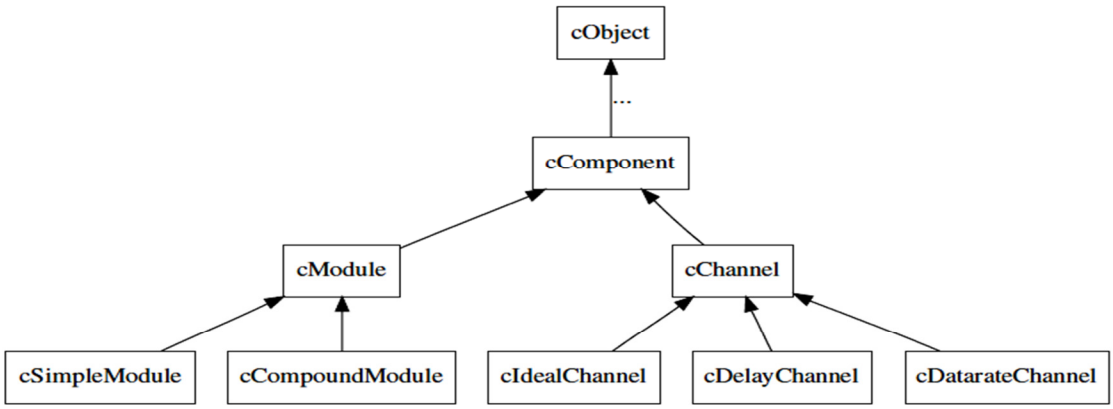


Figura 5.2: Jerarquía entre las clases cComponent, cModule, y cChannel.

Los módulos pueden tener parámetros los cuales se utilizan para pasar datos de configuración a los módulos simples y también para ayudar a definir la topología del modelo. Los parámetros pueden tomar valores numéricos (enteros o reales), string o booleanos. Los parámetros se representan como objetos en el programa, gracias a esto pueden actuar como fuentes de números aleatorios definiendo en la configuración del modelo alguna distribución. También pueden obtener valores por parte de los usuarios de forma interactiva y contener expresiones que referencian a otros parámetros.

5.5 Lenguaje NED

La estructura del modelo de simulación se describe en el lenguaje NED (Network Description). Los ingredientes típicos de una descripción NED son las declaraciones de módulos simples, módulos compuestos y definiciones de *network*. Un módulo simple puede tener puertas y parámetros, mientras que los módulos compuestos consisten de declaraciones de las interfaces de los módulos externos (puertas y parámetros) además de la definición de sub-módulos y sus conexiones. En pocas palabras, este lenguaje les permite a los usuarios declarar módulos simples y además conectar y ensamblar éstos en módulos compuestos para construir el modelo de la simulación.

El lenguaje NED [38] ha sido diseñado para ser escalable y tener una buena adaptabilidad en proyectos grandes. Las principales características del lenguaje NED son:

- Jerárquico: la forma tradicional de manejar la complejidad es introduciendo jerarquías. En OMNeT++ cualquier módulo que pudiese ser complejo puede desensamblarse en sub-módulos y utilizarse como un módulo compuesto.
- Basado en componentes: módulos simples y compuestos son reusables por naturaleza con lo cual no solo se reduce la copia de código, sino que más importante aún, permite la existencia de bibliotecas de componentes (INET Framework[12], MiXiM[13], Castalia[14], etc).

[12] http://inet.omnetpp.org

- Interfaces: las interfaces de módulo y canal se pueden utilizar como un contenedor donde normalmente un módulo o un canal se utilizaría y el módulo en concreto o el tipo de canal es determinado en tiempo de configuración de la red mediante un parámetro. Por ejemplo, dado un tipo de módulo compuesto llamado MobileHost que contiene un sub-módulo *mobility* del tipo IMobility (donde IMobility es una interfaz), el tipo actual de *mobility* puede ser escogido desde los tipos de módulos implementados por IMobility (RandomWalkMobility, TurtleMobility, ConstSpeedMobility, etc).
- Herencia: los módulos y canales pueden tener subclases. A los módulos derivados se les puede añadir nuevos parámetros, puertas, sub-módulos y conexiones (en el caso de módulos compuestos). Se pueden ajustar los parámetros existentes a un valor específico. Esto hace que sea posible por ejemplo, tomar un módulo GenericTCPClientApp y derivar de este un FTPClientApp por medio de la configuración de ciertos parámetros a valores fijos. También es posible obtener un módulo compuesto WebClientHost partiendo de un módulo compuesto BaseHost, agregando sub-módulos WebClientApp y conectándolo con sub-módulos TCP heredados.
- Paquetes: el lenguaje NED tiene como característica una estructura en paquetes similar a la de Java con la finalidad de reducir el riesgo de conflictos de nombres entre los diferentes modelos.
- Inner types: los tipos de módulos y de canales usados a nivel local por un módulo compuesto pueden definirse dentro de éste, de forma tal de reducir la polución del espacio de nombres.
- Propiedades: es posible comentar tipos de módulo o canal, parámetros, puertas y sub-módulos por medio de la agregación de propiedades. Las propiedades no son utilizadas directamente por el núcleo de la simulación, pero éstas pueden acarrear información extra para varias herramientas, para el ambiente en tiempo de ejecución o incluso para otros módulos en el modelo. Por ejemplo, la representación gráfica de un módulo (el icono, etc.) o la cadena indicadora se especifican como propiedades.

5.6 Programación de Módulos Simples

Los módulos simples [37] son elementos activos en el modelo y también son elementos atómicos dentro de la jerarquía del módulo ya que no pueden seguir siendo divididos. Como se mencionó anteriormente, el comportamiento de los módulos simples es definido por los usuarios mediante código C++. Los usuarios implementan la funcionalidad de un módulo simple extendiendo la clase cSimpleModule y es añadida a través de uno de los dos enfoques alternativos de programación:

- La programación basada en *co-rutina*: donde el código del módulo se ejecuta en su propio hilo que es despertado por el núcleo de simulación cada vez que el módulo recibe un mensaje. La función que contiene el código con la co-rutina típicamente no retorna, usualmente contiene un ciclo infinito para enviar y recibir mensajes.
- La programación basada en *procesamiento de eventos*: donde el núcleo de simulación simplemente llama a una función dada del módulo con el mensaje como argumento. La función tiene que retornar inmediatamente luego de procesar el mensaje.

Es posible escribir código que sea ejecutado en la inicialización y finalización del módulo. El código de finalización es llevado a cabo cuando la simulación termina con éxito y se utiliza sobre todo para guardar los resultados en un archivo. OMNeT++ también soporta múltiples fases de inicialización (situaciones donde la inicialización del modelo necesita llevarse a cabo en varias fases). Múltiples

[13] http://mixim.sourceforge.net
[14] http://castalia.npc.nicta.com.au/index.php

fases de inicialización no están soportadas en la mayoría de los paquetes de simulación y por lo general son emulados con eventos en broadcast planificados en el tiempo cero de la simulación, lo que representa una solución menos limpia.

El envío y recepción de mensajes son las tareas más frecuentes en los módulos simples. Los mensajes pueden ser definidos por medio de la especificación de su contenido en un archivo MSG. A partir de estos archivos, OMNeT++ se encarga de crear las clases de C++ necesarias.
Es posible modificar dinámicamente la topología de la red. Se pueden crean y eliminar módulos y reordenar conexiones mientras la simulación se está ejecutando. Incluso los módulos compuestos con topología interna parametrizada pueden ser creados sobre la marcha.

5.7 IDE de la Simulación

OMNeT++ [38] incluye en su paquete a un IDE basado netamente en Eclipse[15] (Figura 5.3) que incorpora un conjunto de herramientas muy importantes para el simulador.

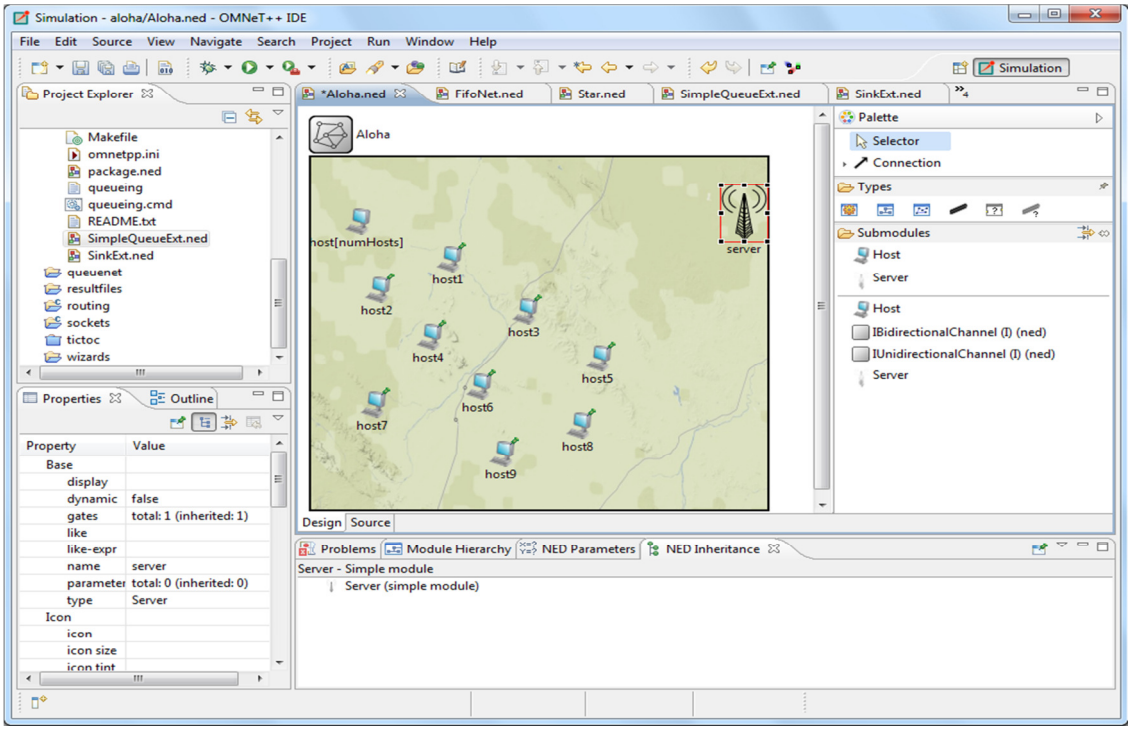


Figura 5.3: IDE de OMNeT++

El IDE de simulación provee un editor para la configuración de la simulación, soporte para escribir código en C++, representación gráfica de resultados y herramientas de análisis, analizadores de trazas que visualizan la ejecución de la simulación en diagramas de secuencia, etc.

La intención del equipo de OMNeT++ es procurar que el IDE de la simulación se convierta en la plataforma huésped y de integración para varias utilidades y herramientas de simulación creadas por terceros. En la actualidad existen varias interfaces de usuario y herramientas de línea de comando para la generación de topologías, generación de escenarios de red y para la ejecución de simulaciones en lote los cuales usan una variedad de lenguajes y herramientas.

[15] http://www.eclipse.org

5.7.1 Soporte del IDE para Generar Archivos NED

Los paquetes de OMNeT++ incluyen un editor gráfico para generar archivos NED (Figura 5.4) como un editor de texto para código NED (Figura 5.5). El editor es una herramienta totalmente bidireccional, los usuarios puede editar la topología bien sea gráficamente o con vista al código NED e incluso cambiar a cualquiera de las dos vistas cuando quiera. Esto es posible gracias a las características del lenguaje NED. NED es un lenguaje declarativo, por lo tanto no utiliza un lenguaje de programación imperativo para la definición de las estructuras internas de un módulo compuesto.

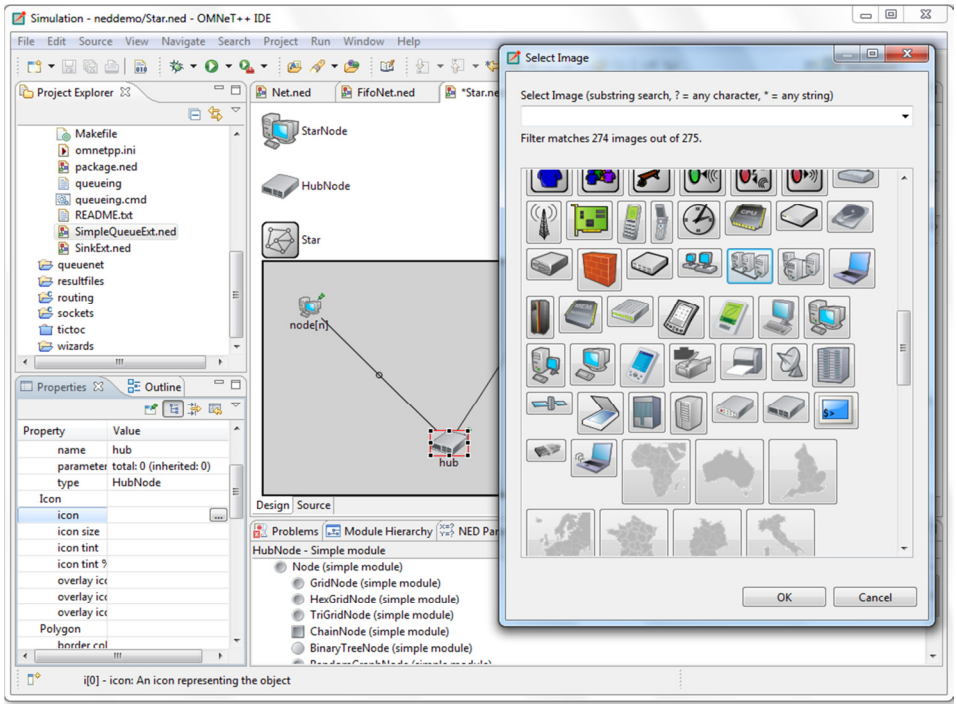


Figura 5.4: Editor Gráfico de Archivos NED

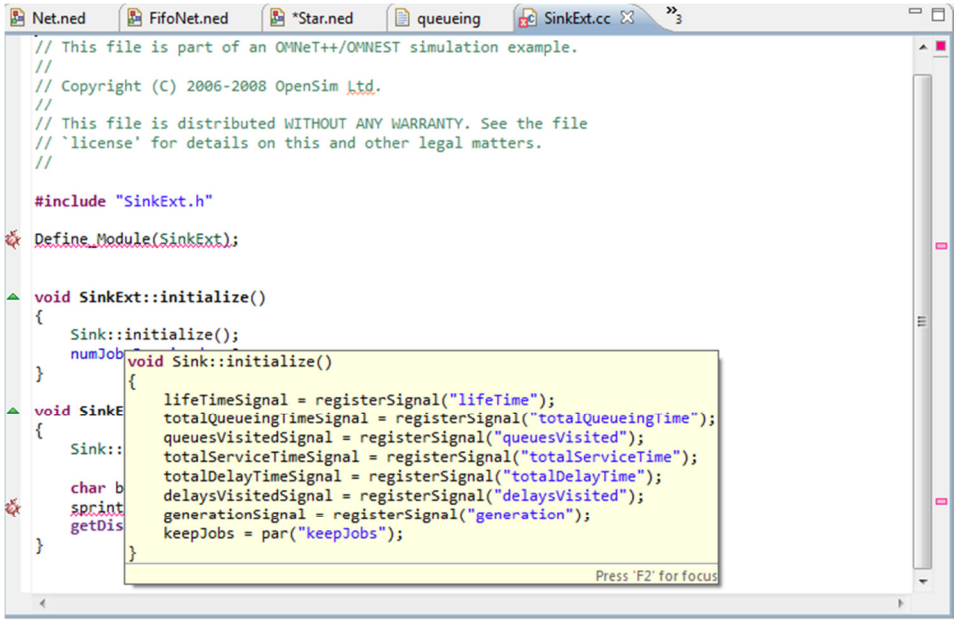


Figura 5.5: Editor de Código de Archivos NED

Los atributos gráficos son almacenados en *display strings* los cuales son parte del lenguaje NED conocidos como propiedades. El editor de código provee asistencia de contenido, validación sobre la marcha, navegación hacia las declaraciones y otras funciones básicas además de la edición y el resaltado de la sintaxis.

La mayoría de los editores gráficos solo permiten la creación de topologías fijas. Sin embargo, NED contiene constructores declarativos (similares a los ciclos y condicionales en los lenguajes imperativos) los cuales permiten parametrizar topologías. Es posible crear topologías comunes como anillo, malla, estrella, árbol o interconexiones aleatorias cuyas características son pasadas como parámetros con valores numéricos. Con topologías parametrizadas, NED tiene una ventaja en muchos escenarios de simulación sobre otros simuladores de red donde solo modelos de topologías fijas pueden ser diseñados.

5.7.2 Animación

La animación de la red es provista por la interfaz gráfica *Tkenv* (Figura 5.6) y es automática, esto quiere decir que el código de la simulación no tiene que ser modificado para la animación. Tkenv permite a los usuarios abrir un inspector gráfico (una ventana para la visualización de la simulación) para cualquier módulo compuesto. El inspector muestra los sub-módulos (nodos de red, protocolos, etc.) y sus interconexiones.

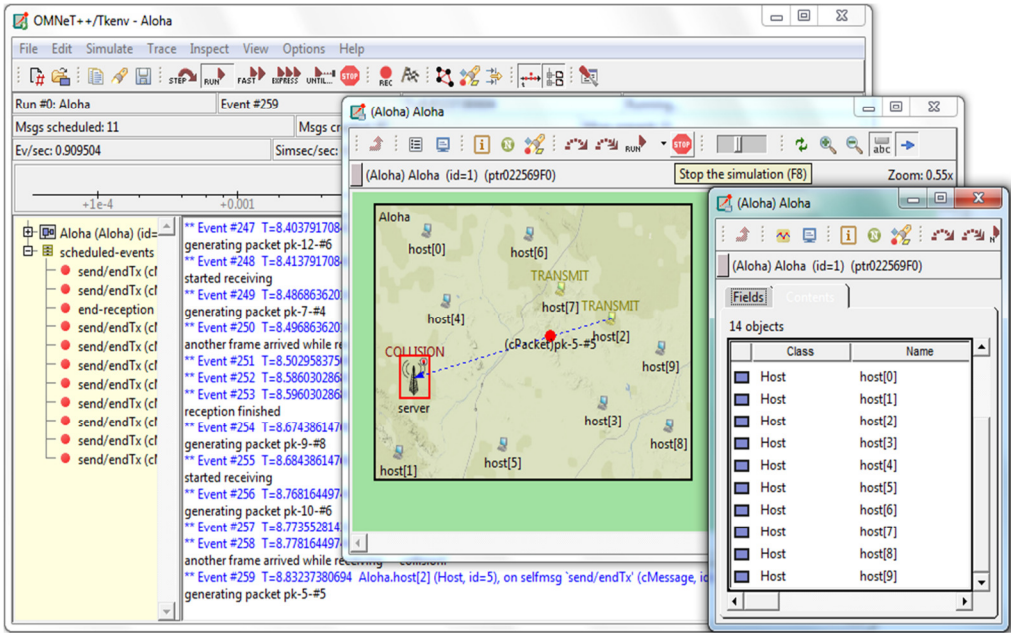


Figura 5.6: Interfaz Gráfica Tkenv

Durante la simulación, Tkenv muestra como los mensajes o paquetes viajan entre los módulos y también muestra la llamada a los métodos entre los módulos. El usuario puede influir sobre la animación a través de la manipulación de los *display strings* cambiando los valores de los atributos como por ejemplo actualizando las coordenadas de un nodo móvil o cambiando el color del icono que representa a un protocolo dependiendo de su estado durante la simulación.

Tkenv provee animación en vivo a diferencia de la reproducción proporcionada en ns-3 con NetAnim por ejemplo. En comparación con la reproducción, la animación en vivo tiene sus ventajas ya que permite que todos los objetos puedan ser examinados en detalle en cualquier momento y también

puede combinarse con el depurador de C++. Entre las desventajas se tiene que no es posible volver pasos hacia atrás en la simulación ni reproducir partes de la historia de la misma.

Además de la animación, Tkenv también muestra la salida de un log de depuración de módulos y permite a los usuarios inspeccionar el modelo a nivel de objetos y campos. Por ejemplo, es posible examinar el contenido de las colas o mirar dentro de los paquetes de red. El contenido de la lista de eventos futuros también se visualiza en una línea de tiempo a gran escala. Es posible conocer en tiempo de ejecución información acerca de todos los objetos en el modelo de simulación y además es posible realizar búsquedas acerca de, por ejemplo, todos los datagramas IP en la red.

5.7.3 Logs de Eventos y Diagramas de Secuencia

Las simulaciones de OMNeT++ pueden crear opcionalmente un archivo log para los eventos (Figura 5.7). Con este archivo se registran los eventos de la simulación tales como la creación y eliminación de mensajes, planificación y cancelaciones de eventos, mensajes enviados y transmisiones de paquetes, cambios en la topología del modelo, mensajes de depuración de los módulos simples, entre otros. Los campos de los mensajes y paquetes pueden también ser capturados por el archivo log de eventos con un nivel de detalle configurable.

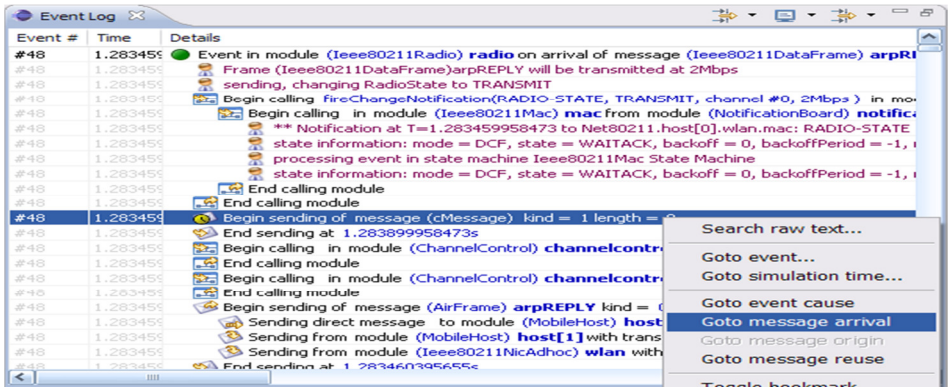


Figura 5.7: Archivo Log de Eventos

El IDE de la simulación puede visualizar el log usando un diagrama de secuencia interactivo (Figura 5.8) que facilita mucho la verificación de protocolos del modelo. El diagrama se puede visualizar de forma panorámica y se le puede hacer zoom, además existen muchas maneras (lineal, no lineal, por pasos, etc.) de mapear el tiempo y eventos de simulación en el eje de las coordenadas X. En el diagrama se puede hacer filtrado por módulos y por otros criterios distintos. Las herramientas de información muestran las propiedades de los eventos y mensajes, la información detallada de los paquetes y también es posible navegar por el log detallando todas las acciones de los eventos de la simulación.

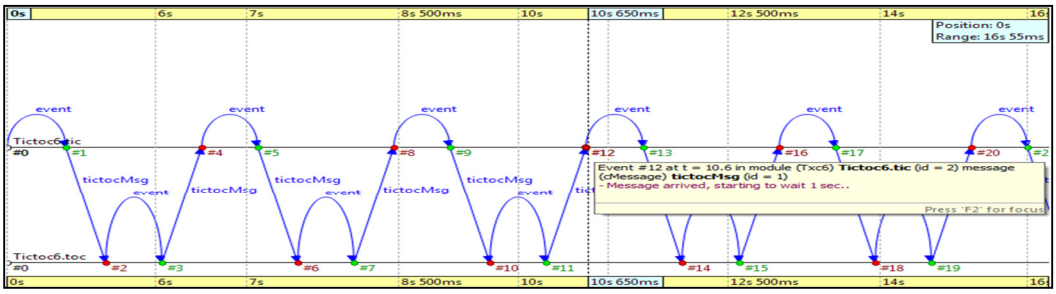


Figura 5.8: Diagrama de Secuencia de los Eventos

5.8 Separación del Modelo y del Experimento

Siempre es una buena práctica tratar de separar los diferentes aspectos de la simulación tanto como sea posible. El comportamiento de los modelos es encapsulado dentro de los archivos C++ como código, mientras que la topología y los parámetros son definidos por los archivos NED. Este enfoque permite que los usuarios mantengan diferentes aspectos en diferentes lugares que hacen un diseño más limpio del modelo y más compatible con otras herramientas de soporte.

En un escenario de simulación genérico, usualmente se quiere conocer cómo se comportará la simulación con diferentes entradas. Estas variables no pertenecen ni al comportamiento (código), ni a la topología (archivos NED) ya que estos pueden cambiar de ejecución en ejecución. Los archivos INI son los que se utilizan para almacenar estos valores. Los archivos INI proveen una mejor opción para especificar cómo estos parámetros cambian y permiten ejecutar las simulaciones para cada combinación de parámetros que sea de interés. La simulación generada puede ser fácilmente capturada y procesada por las herramientas de análisis.

5.9 Recolección y Almacenamiento de Resultados

OMNeT++ distingue tres tipos de resultados:

- Escalares: es un solo número.
- Vectoriales: son series de vectores marcados en el tiempo.
- Estadísticos: son registros compuestos de propiedades estadísticas, es decir, medidas, varianzas, mínimo, máximo, etc. Posiblemente también data de histogramas.

La manera tradicional de almacenar escalares y estadísticas en OMNeT++ es recoger los valores en variables dentro de las clases de los módulos para entonces almacenarlos durante la fase de finalización. Los vectores son almacenados generalmente en los objetos *output vector.* El almacenamiento de los escalares, vectores y estadísticas puede ser habilitado o deshabilitado mediante la configuración de los archivos INI en los cuales también se puede configurar intervalos para el almacenamiento de los vectores. Los resultados de la simulación son guardados en archivos de texto.

5.9.1 Visualización de los Resultados

El IDE de simulación de OMNeT++ provee de forma integrada una herramienta para el análisis de resultados. La herramienta intenta combinar el uso sencillo de la interfaz gráfica con el poder del *scripting*. Una de las metas de diseño de la herramienta fue la eliminación del trabajo repetitivo. Los usuarios no desean rehacer todas las tablas luego de volver a ejecutar las simulaciones debido a algún cambio en el código o en la configuración.

La herramienta les permite a los usuarios especificar un conjunto de archivos de resultados para trabajar con ellos y permite también navegar por los datos que contienen los mismos. Para la navegación, los datos pueden ser desplegados en tablas, organizados como árboles por las etiquetas de medición del experimento o de otras maneras y también se pueden realizar filtrados.

La herramienta permite a los usuarios crear varios gráficos de los resultados de la simulación (Figura 5.9) como gráficos de líneas, de barra, diagramas de dispersión de valores escalares, histogramas, etc.

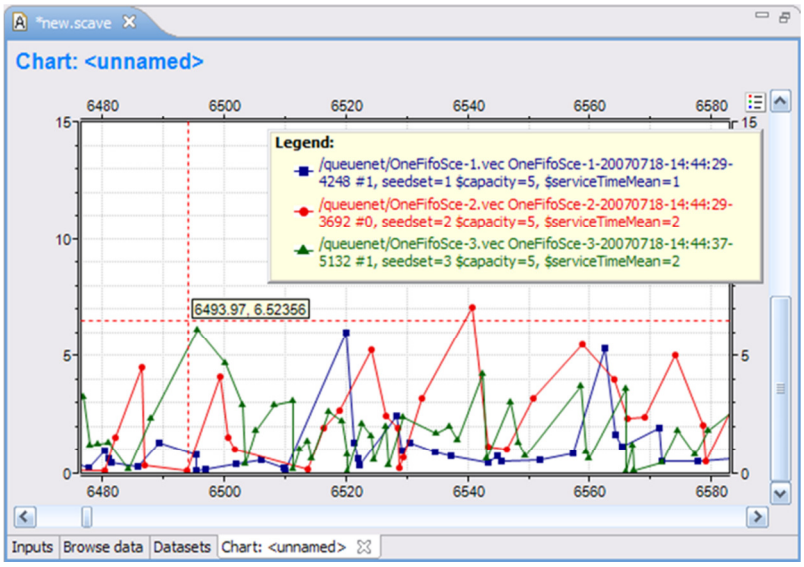


Figura 5.9: Gráficas de Resultados

5.10 Soporte para Simulaciones Distribuidas y Paralelas

OMNeT++ tiene soporte para la ejecución de simulaciones en paralelo. Simulaciones de gran tamaño pueden beneficiarse de las características de la simulación distribuida ya sea para conseguir mejores tiempos de ejecución o para distribuir los requerimientos de memoria. Si la simulación requiere de una gran cantidad de memoria, la distribución sobre un *cluster* puede ser la única manera de ejecutarla. La capa de comunicación es MPI[16] pero en realidad es configurable, por lo que si los usuarios no tienen MPI es posible utilizar los llamados *pipes*. La Figura 5.10 explica la arquitectura lógica del kernel para la simulación en paralelo.

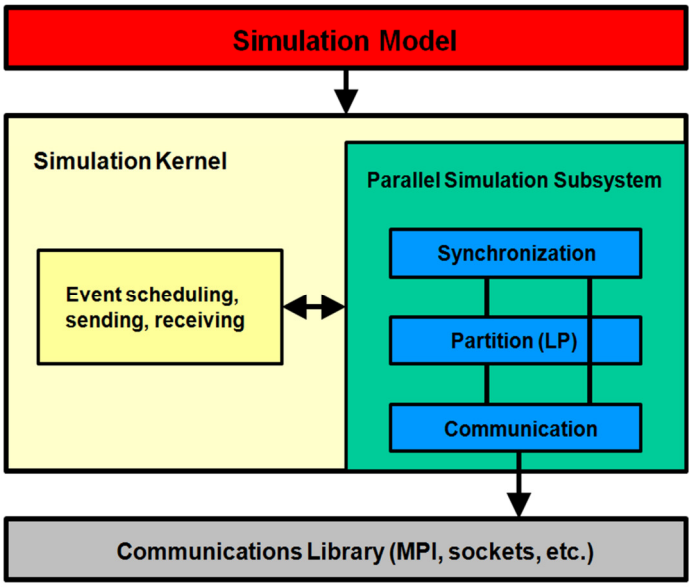


Figura 5.10: Arquitectura Lógica del Kernel para Simulaciones en Paralelo

[16] http://www.mcs.anl.gov/research/projects/mpi

5.11 Contenido de la Biblioteca de Simulación

En esta sección se proporciona una breve introducción al catálogo de los principales módulos de OMNeT++. Estos han sido diseñados para cubrir la mayoría de las tareas más comunes de la simulación.

5.11.1 Módulo del Núcleo de Simulación

Contiene las clases que definen al núcleo de la simulación. Entre estas clases se tienen:

- cObject y cOwnerObject: son las clases base para la mayoría de las demás clases en OMNeT++.
- cModule, cCompoundModule y cSimpleModule: representan a los módulos en la simulación.
- cMessage: representa los eventos y también los mensajes enviados entre los módulos.
- cGate: representa las puertas del módulo.
- cPar: representa los parámetros de los módulos y canales.
- cSimulation: almacena todos los módulos de la red y las estructuras de datos para la planificación de eventos (el conjunto de eventos futuros). La mayoría de estos métodos son de uso interno pero son útiles para los desarrolladores de modelos.

5.11.2 Módulo de las Clases Contenedoras

Define un conjunto de clases para estructuras de datos. Entre estas clases se encuentran:

- cQueue: una clase genérica para representar colas.
- cPacketQueue: una cola especializada para objetos mensaje (cMessage).
- cArray: arreglos dinámicos.

Además, se puede hacer uso de las clases contenedoras estándar de C++ como std::vector o std::map.

5.11.3 Módulo para la Generación de Números Aleatorios

A través de este módulo, OMNeT++ tiene la habilidad de generar números aleatorios. Están soportadas las distribuciones comunes y es posible agregar nuevas distribuciones. Este módulo contiene los siguientes tres sub-módulos:

- Módulo de distribuciones continuas: contiene las distribuciones uniforme, exponencial, normal, gamma, beta, weibull, etc.
- Módulo de distribuciones discretas: contiene las distribuciones binomial, geométrica, poisson, binomial negativa, etc.
- Módulo para la generación de números aleatorios: contiene clases para la generación de números aleatorios entre un rango dado.

5.11.4 Módulo para las Estadísticas y Colección de Datos

Este módulo le provee a OMNeT++ una variedad de clases para realizar estadísticas y recolectar resultados. Existen clases con las cuales se calculan estadísticas básicas como la media y la desviación estándar mientras que algunas clases tratan con estimación de densidad y otras con la detección automática de estadísticas recolectadas. Entre esas principales clases se tienen:

- cStatistic y cDensityEstBase: las principales clases base abstractas de las cuales la mayoría de las demás clases hacen polimorfismo para obtener funcionalidades a través de funciones virtuales.
- cTransientDetection y cAccuracyDetection: las clases base abstractas de donde se derivan las clases para la detección y cálculo de precisión de los resultados.
- cOutVector: se utiliza para registrar los resultados de la simulación en vectores.
- cStdDev: calcula la media, desviación estándar, valor mínimo y máximo, etc.
- cWeightedStdDev: es similar a cStdDev pero acepta muestras ponderadas. Se puede utilizar por ejemplo para calcular el promedio de tiempo. Es la única clase de estadísticas ponderadas.
- cLongHistogram y cDoubleHistogram: son clases derivadas de cStdDev y mantienen una aproximación de la distribución de las muestras usando histogramas con *bins* del mismo ancho.

5.11.5 Módulo con las Clases de Utilidades

Este módulo contiene a las clases que hacen más fácil escribir modelos de simulación:

- cTopology: soporta el enrutamiento en redes de telecomunicaciones o multiprocesador.
- cStringTokenizer: divide un string en palabras (tokens).
- cFSM: se utiliza para construir máquinas de estados finitos.
- cVisitor: para recorrer el árbol de objetos de la simulación.

5.11.6 Otros Módulos

Existen muchos otros módulos para el manejo de las interfaces de usuario, para los enumerados, tipos y definición de funciones, para el uso de macros, para la definición de funciones, para el soporte de la simulación en paralelo, etc. Todos ellos se encuentran documentados en el API[17] de OMNeT++.

5.12 Frameworks para la Simulación de Redes

Los siguientes frameworks para simulación de redes han sido desarrollados para OMNeT++ y son los de mayor uso común:

- INET Framework: contiene modelos para IP, TCP, UDP, PPP, Ethernet, MPLS con LDP, y señalización RSVP-TE, entre otros protocolos. Adicionalmente, tiene soporte para simulaciones móviles e inalámbricas y por lo general es utilizado como base para otros frameworks.
- INETMANET: es una extensión del INET Framework. Contiene una gran cantidad de protocolos de enrutamiento para redes MANETs. Entre estos protocolos se encuentran: AODV, DSR (Dynamic Source Routing), BATMAN (Better Approach To Mobile Ad hoc Networking), etc.
- MiXiM: utilizado para el modelado de redes inalámbricas móviles y fijas (redes inalámbricas de sensores, redes ad hoc, redes vehiculares, etc.). MiXiM se concentra en las capas inferiores de la pila de protocolos y ofrece modelos detallados de la propagación de las ondas de radio, la estimación de interferencias, el consumo de energía de un radio transmisor-receptor y protocolos inalámbricos de la capa MAC.

[17] http://www.omnetpp.org/doc/omnetpp/api/index.html

- Castalia: es un simulador para WSN (Wireless Sensor Network), BAN (Body Area Networks) y en general las redes de dispositivos embebidos de baja potencia. Es utilizado por los investigadores y desarrolladores para poner a prueba sus algoritmos distribuidos y/o sus protocolos en canales inalámbricos y modelos de radio realistas.
- xMIPv6: es un preciso y extensible modelo de simulación móvil IPv6 para el INET Framework.
- ReaSE: es una extensión del INET Framework capaz de crear un entorno de simulación realista con respecto a topologías de red jerárquicas y ataques al tráfico basados en herramientas de ataques reales.
- OverSim: es un framework para el soporte de protocolos para redes superpuestas (red virtual de nodos conectados virtualmente y que se construye en la parte superior de otra red) y simulación de redes peer-to-peer. Contienen muchos modelos para estos sistemas y también está basado en el INET Framework.

6. JiST/SWANS

En este capítulo se presenta un estudio acerca de la plataforma de simulación de red JiST/SWANS resaltando sus principales características, ventajas y desventajas. Además, se analizan todas las tecnologías y herramientas adicionales que esta plataforma soporta.

6.1 Introducción

Los simuladores de eventos discretos son programas que manejan eventos marcados en el tiempo procesándolos en el orden que indique su marca de tiempo así como también pueden modificar el estado de la simulación y generar eventos futuros. Estos programas son herramientas científicas de gran importancia en la que muchas áreas de investigación en computación pretenden lograr obtener un diseño y ejecución eficiente.

La investigación en redes inalámbricas depende fundamentalmente de la simulación. Cuantificar analíticamente el desempeño y el comportamiento incluso de un simple protocolo es a menudo impreciso. Por otro lado, realizar experimentos resulta bastante costoso, implica adquirir cientos de dispositivos, gestionar y configurar su software, darle movimiento a los dispositivos, tener el espacio físico para realizar las pruebas, aislar interferencias y en general garantizar igualdad de circunstancias. Son por estas razones que la simulación se convierte en una solución de gran importancia y a través de ella se busca simular redes con miles de nodos. Solo unos pocos simuladores de eventos discretos tienen una alta escalabilidad y muchos no tienen los modelos adecuados para redes inalámbricas, estos son algunos aspectos que han motivado a la creación del proyecto JiST/SWANS.

JiST (Java in Simulation Time) [40] es un motor de alto desempeño para simulaciones de eventos discretos el cual se ejecuta sobre una máquina virtual de Java estándar. Permite a los programadores simular diferentes tipos de escenarios generales de manera eficiente y transparente. Es una plataforma muy eficiente y altamente optimizada en cuanto al consumo de memoria y tiempo de ejecución de la simulación.

SWANS (Scalable Wireless Adhoc Network Simulator) [41] basado en la plataforma JiST, fue creado debido a que las herramientas para la simulación de redes de la época no cubrían las necesidades de los investigadores. SWANS está organizado como un componente de software independiente que permite formar redes inalámbricas o armar redes de sensores permitiendo simular redes con un gran número de nodos. Busca aprovecharse del diseño de JiST para lograr simulaciones de alto rendimiento, ahorrar memoria y ejecutar aplicaciones de red estándar de Java sobre las redes simuladas.

6.2 El Proyecto JiST

La motivación clave de JiST es crear un sistema de simulación que puede ejecutar simulaciones de eventos discretos de manera eficiente y transparente usando solo un lenguaje estándar y tiempo de ejecución donde:

- Por eficiencia: se refiere a obtener tiempos de ejecución favorables con respecto a sistemas de simulación existentes y altamente optimizados.
- Transparencia: implica que las simulaciones sean transformadas automáticamente para ser ejecutadas con semánticas de tiempo de simulación.

- Estándar: denota la escritura de simulaciones en un lenguaje de programación convencional, lo opuesto a tener un lenguaje de dominio específico diseñado únicamente para la simulación.

Estos atributos (particularmente el último) resaltan una diferencia importante entre los sistemas de simulación previos y JiST ya que permite que el código de simulación que se ejecuta en JiST no este escrito en un lenguaje de dominio específico evitando el uso de callbacks y llamadas especiales al sistema para dar funcionalidades en tiempo de ejecución de la simulación. JiST introduce la semántica de ejecución en tiempo de simulación a programas o simulaciones escritas en Java plano y son ejecutadas sobre una máquina virtual de Java sin modificaciones.

6.2.1 Arquitectura

La arquitectura del sistema JiST (Figura 6.1) está compuesta por un compilador, un módulo para sobrescribir instrucciones bytecode, el kernel de simulación y una máquina virtual. Una vez que la simulación es escrita en texto plano, será compilada en instrucciones bytecode utilizando un compilador del lenguaje Java. A continuación, estas clases compiladas son entonces modificadas a través del generador a nivel de bytecode para ejecutarse sobre un núcleo de simulación y soportar las semánticas de tiempo de simulación que serán descritas más adelante. Todo este proceso ocurre dentro de un estándar y sin modificar la máquina virtual de Java.

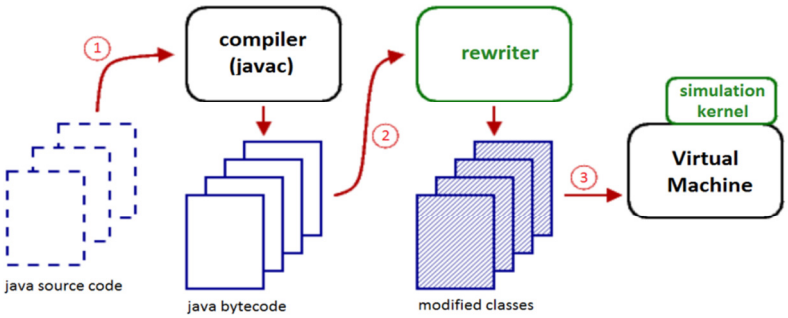


Figura 6.1: Arquitectura de JiST

Son muchos los beneficios que se tienen al simular sobre sistemas tradicionales. Integrando la semántica de simulación dentro del lenguaje Java permite reusar una gran cantidad de código incluyendo el lenguaje Java, sus librerías estándar y los compiladores existentes. Otros beneficios de JiST inherentes al lenguaje Java son el manejo de memoria, concurrencia, reflexión, entre otros. Estos beneficios disminuyen la curva de aprendizaje y facilitan el reúso de código para la construcción de simulaciones. El uso de una máquina virtual estándar provee una eficiente y portable plataforma de ejecución que permite un importante nivel de optimización entre el núcleo de la simulación y la simulación en ejecución. Este diseño resulta en un sistema que es conveniente para usar, robusto y eficiente, permitiendo que el núcleo y la simulación se ejecuten dentro del mismo espacio reduciendo así la serialización y sobrecargas en el cambio de contexto.

6.2.2 Tiempo de simulación

En el modelo estándar de ejecución, el paso del tiempo no es dependiente del progreso de la aplicación, es decir, el reloj del sistema avanza sin importar cuántas instrucciones bytecode son procesadas. Además, el programa puede avanzar a una tasa variable, ya que no solo depende de la velocidad del procesador sino también de otros factores impredecibles, tales como interrupciones y operaciones de entrada/salida. Para solventar este tipo de problemas, mucha investigación ha sido realizada para ejecutar aplicaciones en tiempo real o con modelos de rendimiento más predecibles, donde el tiempo de ejecución puede garantizar qué instrucciones o conjuntos de instrucciones

cumplirán con los plazos dados. Así la tasa del progreso de una aplicación es hecha dependiente del paso del tiempo.

En JiST, sucede lo contrario, es decir, el progreso del tiempo depende del progreso de la aplicación. El tiempo de la aplicación, que representa el tiempo de simulación, no avanza al próximo evento discreto hasta que todo el procesamiento para el actual evento discreto haya sido completado.

En la ejecución en tiempo de simulación, las instrucciones bytecode son procesadas secuencialmente siguiendo el control de flujo estándar de Java, pero el tiempo de aplicación permanece invariable. El tiempo de ejecución JiST procesa una aplicación en su orden de simulación temporal hasta que todos sus eventos son agotados o hasta que un predeterminado tiempo de finalización es alcanzado.

6.2.3 Ventajas de JiST

Además de rendimiento y escalabilidad, JiST posee una serie de características, muchas de ellas inherentes al lenguaje Java que le da grandes ventajas comparado con otros simuladores. Las ventajas se mencionan a continuación:

- Seguridad de tipos: el origen y objetivo de los eventos de la simulación son estáticamente chequeados por tipo por el compilador, eliminando una larga clase de errores.
- Tipos de eventos: numerosas constantes y el casting asociado de tipo en código no es requerido dado que los eventos son implícitamente tipados.
- Depuración: los estados de cada evento están disponibles durante el procesamiento de los eventos, esto permite generar trazas para determinar el origen de una falla que llegase a ocurrir.
- Reflexión: permite configuración basada en script, depuración y seguimiento en una forma que es transparente para la implementación de la simulación.
- Seguridad: permite para un objeto aislamiento de estado entre entidades, asegura que todas las llamadas en tiempo de simulación pasen a través del kernel, provee flexibilidad en la agregación de estado de entidad.
- Recolección de residuos: la memoria para objetos con largo y variable tiempo de vida, como paquetes de red, es automáticamente manejada, evitando pérdidas de memoria y la necesidad de complejos protocolos para el manejo de memoria.
- Comunicación inter-proceso: Ya que las entidades comparten el mismo espacio de proceso, no hay necesidad de cambio de contexto.
- Núcleo basado en Java: permite optimización entre el kernel y la simulación en ejecución para despacho de eventos y llamadas al sistema más rápidos.
- Robustez: la verificación estricta de Java asegura que las simulaciones no terminarán de forma abrupta; la recolección de residuos protege contra pérdidas de memoria en el tiempo.
- Concurrencia: el modelo de objeto de simulación y las semánticas de ejecución soportan ejecución paralela y óptima transparente con respecto a la aplicación.
- Portabilidad: Permite portar fácilmente las simulaciones entre un sistema y otro.

6.3 Proyecto SWANS

El software SWANS es organizado como componentes de software independientes que se pueden combinar para formar simulaciones completas de redes inalámbricas o redes de sensores. Sus capacidades son similares a ns-2 y GloMoSim, dos populares simuladores de red. Hay componentes que implementan diferentes tipos de protocolos de aplicación, red, enrutamiento y control de acceso

al medio; modelos de transmisión de radio, recepción y ruido; modelos de propagación y pérdida de señal, y modelos de movilidad de nodos. Las instancias de cada tipo de componente son mostradas en la Figura 6.2.

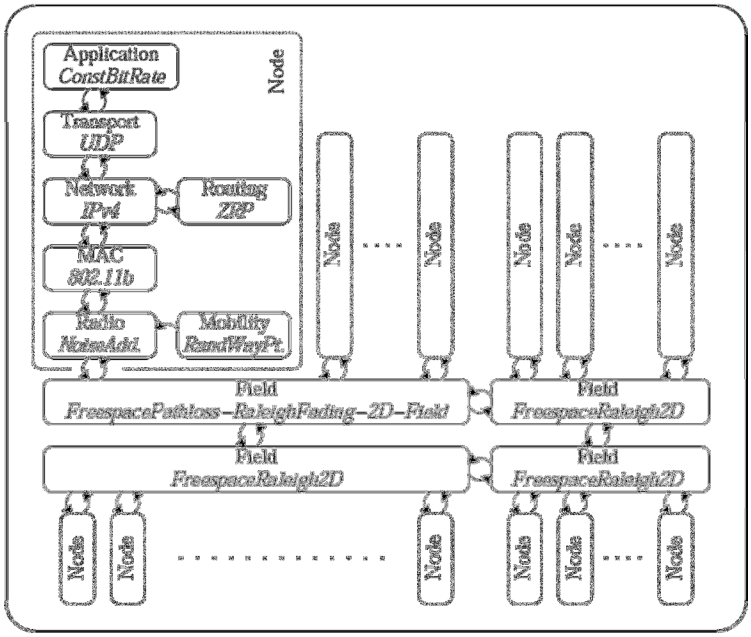


Figura 6.2: Componentes de SWANS

Cada componente en SWANS es encapsulado como una entidad JiST, es decir, es almacenado en su propio estado local e interactúa con otros componentes a través de interfaces basadas en eventos. SWANS contiene componentes para la construcción de una pila de protocolos, como también componentes para una gran variedad de modelos de movilidad y configuración de parámetros y campos. Este patrón simplifica el desarrollo de la simulación dado que reduce el problema de la creación de componentes relativamente pequeños dirigidos por eventos. También particiona explícitamente el estado de la simulación y el grado de interdependencia entre componentes, a diferencia del diseño de ns-2 y GloMoSim. Además permite que los componentes estén listos para ser intercambiados con implementaciones alternas de las interfaces comunes y por cada nodo simulado permite que sean independientemente configurados. Finalmente también confina el patrón de la comunicación en la simulación, es decir, componentes de distintos nodos no se pueden comunicar directamente, solo pueden hacerlo a través del paso de mensajes mediante su propia pila de protocolos.

Es importante destacar que la comunicación entre entidades es muy eficiente en JiST. El diseño incurre en suprimir el costo de la serialización, copia, o cambio de contexto entre entidades desde que los objetos Java contenidos dentro de los eventos son pasados por referencia a través del motor de simulación. Este diseño conserva memoria que permitiría simulaciones de red a gran escala.

SWANS tiene una única e importante ventaja sobre los simuladores de red existentes. Puede funcionar de forma normal sin modificar las aplicaciones Java de red sobre la red simulada, permitiendo así la inclusión de software existente basado en Java tales como servidores web, aplicaciones peer-to-peer y protocolos multicast a nivel de aplicación. Estas aplicaciones simplemente no envían paquetes al simulador desde otros procesos, ellos operan en tiempo de simulación dentro del mismo espacio de proceso JiST permitiendo gran escalabilidad.

6.3.1 Componentes

SWANS está conformado por una serie de componentes que contienen todas las características necesarias para la simulación de redes inalámbricas, brindando el soporte requerido para las aplicaciones y entidades ya existentes en JiST. Los componentes pueden ser mejorados por la comunidad o se pueden agregar otros nuevos para aumentar la funcionalidad del proyecto. A continuación se listan los componentes existentes en SWANS:

- Physical: En este componente se define la capa física y todo lo que ella implica. Se encuentra definida la interfaz dedicada para la movilidad de nodos y la propagación, desvanecimiento y pérdida de la señal. También define modelos de localización de nodos y movilidad genéricos como Random Waypoint. Los modelos de propagación y pérdida de la señal también se ubican en este componente.
- Link: Recibe llamadas desde la entidad radio definida en el componente Physical y las pasa a la entidad de red y viceversa. Este componente es el responsable de la implementación del control de acceso al medio y del encapsulamiento del paquete de red en una trama. Incluye IEEE 802.11b y una implementación llamada MacDumb que solo transmite si el transmisor está inactivo. La implementación 802.11b incluye la funcionalidad completa de DCF y todas las características inherentes a ella.
- Network: Contiene las características y funcionalidades necesarias para la capa de red. Envía paquetes a la entidad de enrutamiento para recibir la información del próximo salto, permite a la entidad de enrutamiento analizar a todos los paquetes que se reciban, y también encapsula los mensajes con la cabecera IP apropiada. Este componente usa una implementación IPv4, el loopback y broadcast están totalmente implementados.
- Routing: En esta entidad se definen los protocolos de enrutamiento que se pueden usar en una simulación. Los protocolos implementados son ZRP (Zone Routing Protocol), DSR (Dynamic Source Routing) y AODV (Ad hoc On-demand Distance Vector). También se encarga de enviar la información del próximo salto según el protocolo de enrutamiento usado en la entidad de red.
- Transport: Define los protocolos de transporte para dar soporte a la entidad de aplicación. Los protocolos de transporte implementados son UDP y TCP (este último parcialmente) que encapsulan los datos de la aplicación utilizada con la cabecera correspondiente del protocolo utilizado por la aplicación.
- Application: Se encuentra las entidades necesarias para proveer servicios de aplicación. La clase de aplicación más genérica y útil es la integración para aplicaciones regulares de Java. Se puede ejecutar aplicaciones de red de Java estándares sin modificación sobre SWANS. Estas aplicaciones operan dentro de un contexto que incluye la fundamental implementación de transporte para un nodo en particular. Así estas aplicaciones pueden establecer comunicaciones comunes vía sockets, para poder transmitir paquetes desde el nodo simulado por medio de la red simulada.
- Common: Hay varias interfaces que son comunes entre las capas de SWANS. La más importante interfaz común es Message que representa a un paquete transferido a lo largo de la pila de red y que debe ser inmutable. Muchos componentes en varias de las capas definen su propia estructura de mensaje. Otros elementos comunes son la localización de nodos, ciertos controles globales de la capa MAC, entre otros.

7. Redes Vehiculares

7.1 Introducción

El mundo de las redes de comunicación ha crecido de forma exponencial desde sus inicios abarcando prácticamente todas las áreas en donde el ser humano se desarrolla. Como los vehículos son parte de la vida cotidiana, las redes de comunicación también están presentes allí y existen tecnologías que permiten la comunicación entre los vehículos, formando lo que actualmente se conoce como las redes vehiculares. Como el movimiento de los vehículos en las carreteras es altamente dinámico e incluso caótico por la variabilidad de la velocidad, resulta difícil plantear cómo establecer redes entre ellos. Por lo tanto, el tema ha suscitado muchos esfuerzos de investigación para así dar soluciones y en un futuro se pueda implementar a cabalidad las redes vehiculares.

En los últimos años, los sistemas de control de los vehículos han pasado de analógico a digital, facilitando de este modo las redes vehiculares. En particular, sistemas *drive-by-wire* están apareciendo y permiten cambiar procesos mecánicos en los vehículos por sistemas electrónicos, como la dirección y el frenado. Los ECUs [12] (Electronic Control Units) se están desplegando a gran escala en los vehículos para realizar diversas funcionalidades, tales como el despliegue del air-bag, la gestión del motor e incluso los sistemas de frenado inteligente. Por ejemplo, al menos 70 ECUs se emplean en un vehículo modelo Mercedes S-Class[18] [12] y así en muchos modelos actuales de vehículos. En el área de redes vehiculares se establecieron nuevas formas de comunicación [12][33], como lo son las redes In-Vehicle (InV), Vehicle-to-Vehicle (V2V) y Vehicle-to-Infrastructure (V2I). Dichas redes permitirán una variedad de aplicaciones para la seguridad, el control del tráfico, la asistencia al conductor, así como información, los servicios basados en localización, y el entretenimiento.

Una red vehicular organiza y conecta los vehículos entre sí, y con los dispositivos de comunicación ubicados en lugares fijos en las carreteras conocidos como RSUs (Road-Side Unit). Muchas arquitecturas telemáticas [20] ya están implementadas hoy en día. Éstas incluyen los servicios de navegación, de información de tráfico, de servicios basados en localización, de servicios de entretenimiento, de emergencia y de seguridad. En estas arquitecturas, la información de tráfico y servicios de navegación son generalmente proporcionados por centrales TSP[19] (Telematics Service Providers) [20]. Los servicios de emergencia y seguridad son suministrados por una plataforma a bordo, la cual es por lo general instalada por los fabricantes de vehículos.

En contraste con las arquitecturas convencionales adoptadas, las arquitecturas telemáticas rara vez se aplican en los establecimientos públicos con conexiones locales, tales como los estacionamientos públicos, hoteles, restaurantes, aeropuertos y centros comerciales. En la arquitectura *hot-spot* local, un vehículo se considera como una plataforma de cómputo móvil alternativo (lógicamente equivalente a un computador portátil, un PDA o un teléfono móvil) con corto alcance WLAN (Wireless LAN), tales como Bluetooth y WiFi [20]. Esta arquitectura *hot-spot* permite al conductor interactuar con muchos de los servicios locales. Las arquitecturas telemáticas serán de utilidad para los servicios de información solo en los vehículos que necesiten servicios tradicionales. Dichos servicios pueden ser la información de tráfico, la navegación provista por una central TSP y los servicios locales dados por los proveedores de servicios.

Los sistemas telemáticos actuales dependen de infraestructuras móviles para ofrecer servicios a los usuarios. Por lo tanto, la implementación de un servicio telemático entre las redes móviles es una

[18] http://en.wikipedia.org/wiki/Mercedes-Benz_S-Class
[19] http://www.wirelesscar.com

tarea muy costosa. Los desarrolladores de sistemas deben tener un sólido conocimiento acerca de la red móvil subyacente. La Figura 7.1 muestra una visión general de la arquitectura de red en el vehículo (In-Vehicle) y la arquitectura de red fuera del vehículo (Out-Vehicle).

La red In-Vehicle puede adoptar tres protocolos vehiculares de bus que son CAN (Controller Area Network), LIN (Local Interconnect Network), y FlexRay. Sin embargo, estos protocolos no pueden intercomunicarse entre sí y necesitan gateways.

En la red Out-Vehicle el OBU (On-Board Unit) en el vehículo puede comunicarse con Internet a través de la infraestructura donde se ubican los RSUs. Los proveedores de servicios ofrecen servicios específicos a un usuario del vehículo. Las redes vehiculares se están convirtiendo rápidamente en algo común ya que muchos datos críticos de interés deben ser intercambiados en las carreteras.

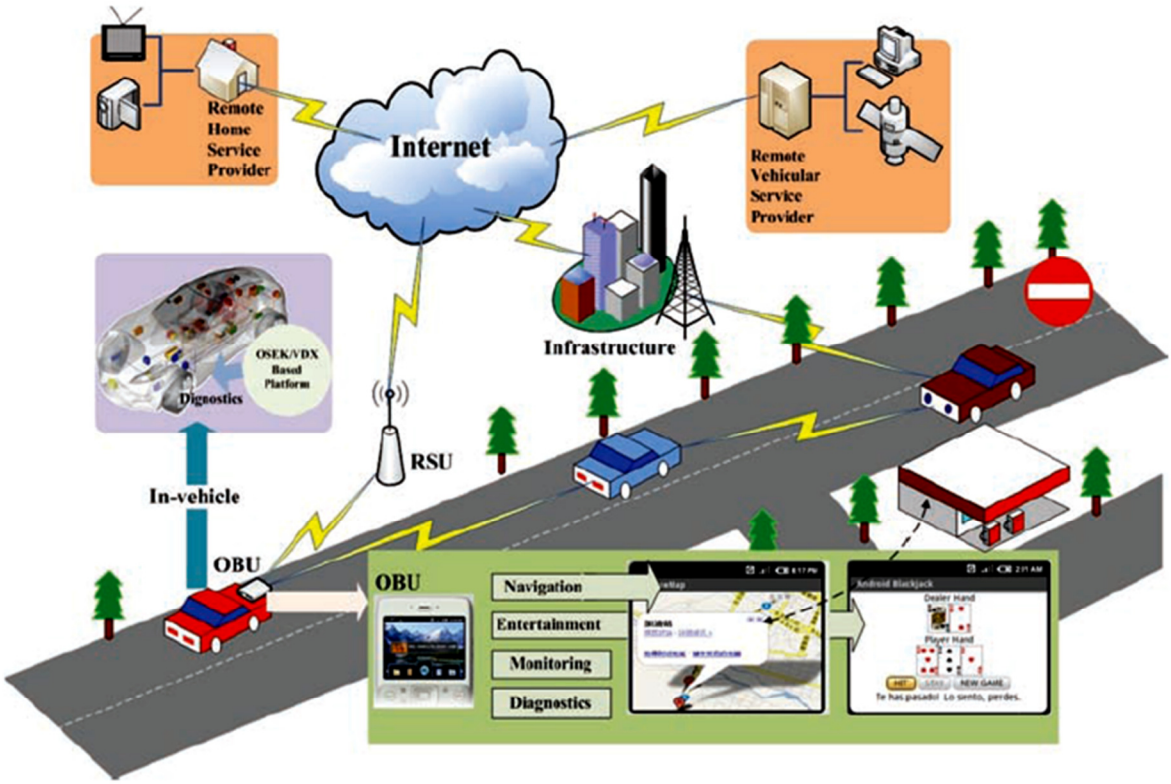


Figura 7.1: Arquitectura In-Vehicle y Out-Vehicle en las Redes Vehiculares

7.2 Motivación

En el comienzo de la industria automotriz [33], las calles y los diferentes tipos de vehículos eran considerados como sistemas autónomos. Más tarde, se introdujeron regulaciones y ciertas normas en las calles, pero una vez más los vehículos seguían siendo considerados como sistemas autónomos. Actualmente, debido al gran crecimiento de la industria automotriz y el establecimiento del vehículo como medio de transporte terrestre por defecto, el número de accidentes y averías ha ido en aumento, ya que el sistema actual de regulación no es suficiente. Entonces, la próxima meta es crear una nueva forma de controlar el sistema tradicional [33], es decir, desarrollar nuevas aplicaciones o tecnologías que permitan solventar en gran escala los problemas actuales en las carreteras. La idea es poder automatizar maniobras específicas que al conductor del vehículo le es difícil controlar por sí mismo. Esas maniobras sin automatización están dando lugar a un creciente número de accidentes y problemas en las vías [12][33], por lo tanto, la seguridad debe ser tomada en cuenta para las nuevas

tecnologías que se desarrollen para el tránsito terrestre. Ejemplos de nuevas aplicaciones y tecnologías que se pueden desarrollar son la adaptación automática de la velocidad, la entrada automática en una carretera o parking, entre otras. Para lograr implementar ese tipo de aplicaciones, lo primordial es tener en cuenta la percepción del entorno que rodea cada vehículo y así poder capturar y analizar los eventos que sucedan. Por lo tanto, el uso de tecnologías de comunicación juega un rol importante, donde la meta es la cooperación entre vehículos en las carreteras.

La investigación sobre ITSs (Intelligent Transport Systems) se remonta a finales del año 1980 y principios de 1990 como necesidad emergente de mejorar los problemas relacionados con el tránsito de vehículos. Desde sus inicios, la investigación pasó de sistemas de autopistas automatizadas o AHS[20] (Automated Highway Systems) a la iniciativa de tener vehículos inteligentes o dispositivos en los vehículos que permitieran funcionalidades específicas [33]. Todo esto con el fin de mejorar la seguridad, tanto para los conductores como para los peatones, el uso de los recursos (carreteras, combustible, etc), ofrecer información y la asistencia al conductor, así como también entretenimiento. La Figura 7.2 muestra las aplicaciones a ser implementadas en las redes vehiculares. Desde el inicio de la investigación, han habido tres áreas que posiblemente se pueden mejorar: (1) en los vehículos, por ejemplo, control adaptativo de la velocidad, sistemas de prevención de colisiones, entre otros; (2) en las carreteras, por ejemplo, gestión avanzada del tráfico, notificación de eventos como colisiones o averías, control adaptativo de la velocidad a mayor escala, entre otros; y (3) en los conductores, por ejemplo, difusión de información anticipada del tráfico en tiempo real, notificación de eventos acontecidos en las carreteras, entre otros.

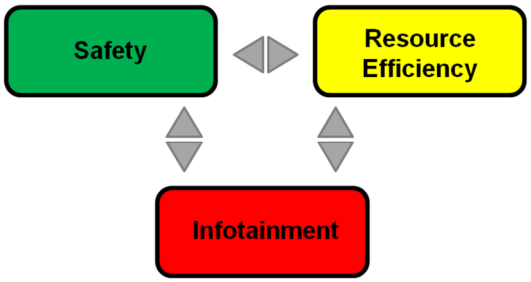


Figura 7.2: Aplicaciones a ser Consideradas en las Redes Vehiculares

Por supuesto, la mayoría de los proyectos de investigación siguieron un enfoque y desarrollo mixto, tratando de mejorar las tres áreas mencionadas anteriormente. Por otra parte, las empresas automotrices de hoy en día han implementado en sus modelos de vehículos modernos tecnologías que mejoran en parte las áreas de interés en el transporte terrestre. Muchos vehículos actuales incluyen sistemas de control adaptativo de la velocidad, detección de la cercanía de otros vehículos para mantener una distancia adecuada, así como el soporte para paradas, por mencionar algunas. Sin embargo, el impacto de estas tecnologías a largo plazo no se ha verificado a su cabalidad. Además, cada fabricante implementa su propia tecnología lo cual puede afectar la meta común de solventar los problemas de viabilidad de forma global y lograr un estándar común. Es por ello que hoy en día se está realizando un gran esfuerzo para la estandarización de las comunicaciones que se pueden dar entre los vehículos.

7.3 Arquitecturas de Comunicación en Redes Vehiculares

Muchas nuevas tecnologías están surgiendo y siendo desarrolladas para aportar en el ámbito de las redes vehiculares y así proveer métodos eficaces y eficientes para las comunicaciones complejas que

[20] http://faculty.washington.edu/jbs/itrans/ahssummary.htm

se dan entre los vehículos de hoy en día. Las redes vehiculares son una meta de los ITSs [12][33], al permitir que los ECUs de un mismo vehículo se comuniquen entre sí por medio de la comunicación InV, mientras que la comunicación entre vehículos se da a través de la comunicación V2V, y la comunicación de los vehículos con los RSUs a través de la comunicación V2I como se muestra en la Figura 7.3. Con la implementación de estas redes se contribuye a tener vías más seguras y eficientes, proporcionando información oportuna a los conductores y autoridades interesadas.

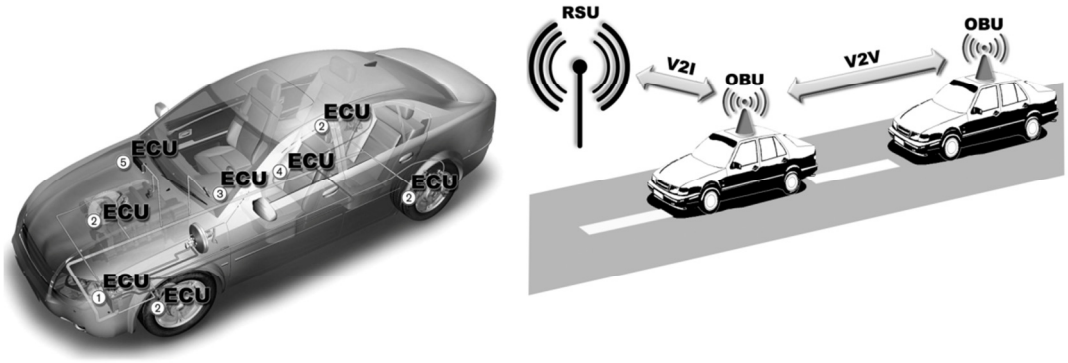


Figura 7.3: Componentes Básicos de la Comunicación en una Red Vehicular

Actualmente existe una serie de grupos de estudio y de trabajo sobre comunicaciones entre vehículos, quienes definen los estándares para las diversas aplicaciones a ser implementadas bajo este ámbito de comunicación [12]. Como ya se mencionó, en las comunicaciones InV se utilizan tecnologías como CAN, LIN y FlexRay. En las comunicaciones V2V se utilizan tecnologías como IEEE 802.11p [22], DSRC (Dedicated Short Range Communications) [26], y la familia IEEE 1609 WAVE (Wireless Access in Vehicular Environments) [21]. Estas tecnologías proveen los mecanismos necesarios para que cada arquitectura de red vehicular coopere de forma correcta con las demás arquitecturas relacionadas con la comunicación vehicular.

La comunicación Vehicle-to-Backoffice se refiere a establecer comunicaciones vehiculares mediante la infraestructura ya existente como GSM (Global System for Mobile Communications) o UMTS (Universal Mobile Telecommunications System). La comunicación Roadside-to-Backoffice también se refiere a establecer comunicación entre los RSUs mediante el uso de tecnología existente. La comunicación de corto alcance entre vehículos es primordial, además, las comunicaciones InV, V2V, y V2I deben estar operando de forma controlada y síncrona para lograr implementar las aplicaciones que se desean para las redes vehiculares.

7.4 Arquitectura de Redes In-Vehicle

En esta sección se presenta la arquitectura de red In-Vehicle o InV que combina diferentes protocolos de bus vehicular, es decir, CAN, LIN, y FlexRay[21].

7.4.1 Controller Area Network

El objetivo era establecer un estándar para la comunicación confiable y eficiente mediante la integración de dispositivos, sensores y unidades en un sistema para aplicaciones de control en tiempo real. El protocolo CAN desarrollado por Bosch[22] combina en una red los ECUs, lo que reduce tanto el cableado como la complejidad. CAN ha ganado un amplio uso en el mundo automotriz, sin

[21] http://www.flexray.com
[22] http://www.bosch.com/worldsite_startpage/en/default.aspx

embargo, también tiene aplicaciones de automatización industrial. La especificación de CAN consta de dos partes [5]. La parte A describe el formato de los mensajes CAN tal como se define en la especificación CAN 1.2, y la parte B describe los formatos de mensajes estándares y extendidos. Para lograr la transparencia de diseño y flexibilidad de aplicación, CAN se divide en tres capas: (1) el objeto, (2) la transferencia, y (3) las capas físicas.

CAN posee las siguientes propiedades [5]:

- Priorización de los mensajes mediante un identificador (ID) del mensaje que representa la prioridad.
- Codificación de mensajes a transmitir por el bus mediante NRZ (Non-Return-To-Zero) y monitorización de éstos por todos los nodos.
- Flexibilidad en la configuración.
- Garantía en los tiempos de respuesta.
- Recepción multicast con sincronización en tiempo.
- Sistema de consistencia de datos.
- Multi-maestro.
- Detección de errores y señalización.
- Retransmisión automática de mensajes erróneos tan pronto como el bus esté libre.
- Distinción entre errores temporales y fallas permanentes de nodos, como también la desconexión autónoma de nodos defectuosos.

Cada nodo CAN requiere de:

- Procesador host: procesa los mensajes recibidos y determina qué mensajes enviar.
- Controlador CAN: posee un reloj síncrono. Recibe bit por bit lo que llega de forma serial a través del bus hasta recibir el mensaje completo el cual puede ser entregado al procesador host. También tiene la tarea de recibir el mensaje a enviar por el procesador host y serializarlo para transmitirlo a través del bus.
- Transceptor: adapta los niveles de la señal del bus a niveles que el controlador CAN puede procesar, además tiene una circuitería que protege a dicho controlador. También tiene la función de convertir los bits a transmitir en señales.

Dado que los procesos de aplicación son asíncronos, el bus dispone de un mecanismo CSMA/CD (Carrier Sense Multiple Access/Collision Detection) para resolver los conflictos. La Figura 7.4 ilustra un bus CAN con tres nodos. CAN tiene cuatro tipos de tramas que son las tramas de (1) datos, (2) remotas, (3) error, y (4) sobrecarga. La trama de datos es la única trama que se adapta para la transmisión de datos.

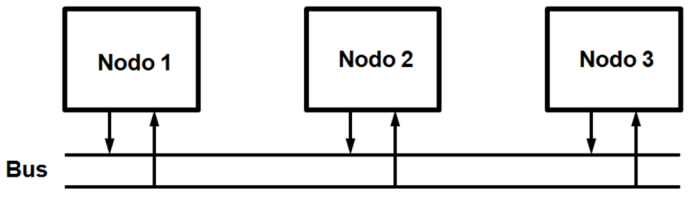


Figura 7.4: Bus CAN

7.4.2 Local Interconnect Network

El bus LIN (Local Interconnect Network Bus) es un estándar de bus de vehículo o de redes de computadores usado en las actuales arquitecturas de red vehicular. La especificación LIN es desarrollada por el LIN-Consortium, con la primera versión liberada en 1999 [33]. La especificación ha evolucionado a la versión 2.1 para satisfacer las necesidades actuales de redes. Debido a que CAN cubre la necesidad de alto ancho de banda y redes con manejo avanzado de errores, los costos de hardware y software para la implementación de CAN se han vuelto prohibitivos para dispositivos de menor rendimiento como controladores de ventanas y asientos. El bus LIN es un pequeño y lento sistema de red que se utiliza como una alternativa menos costosa a la que ofrece un bus CAN para integrar los ECUs en los vehículos modernos. LIN proporciona comunicación rentable en aplicaciones donde el ancho de banda y la versatilidad de CAN no son requeridos.

Las redes automotrices modernas usan una combinación de LIN para aplicaciones de bajo costo principalmente en electrónicos, CAN para la comunicación del tren de potencia y carrocería, y FlexRay para comunicaciones de datos sincronizados de alta velocidad en sistemas avanzados como mecanismos asociados a la seguridad.

Las principales características de LIN incluyen [33]:

- Arquitectura maestro/esclavo, con un máximo de 15 esclavos.
- Bus cableado.
- Velocidades de transmisión de 1 - 20 kbps, donde 2.4 kbps, 9.6 kbps y 19.2 kbps se utilizan generalmente en aplicaciones vehiculares.
- Mensajes multicast y broadcast.
- Auto-sincronización de los esclavos (solo el maestro tiene un reloj de precisión).
- Mensajes con 2, 4, u 8 bytes de datos, y 3 bytes de control.
- Detección de errores por 8 bits de checksum y 2 bits de paridad en el identificador.
- Capa física: ISO9141.
- Capacidad *sleep/wake-up*.

La especificación de LIN describe tres de las siete capas del modelo de referencia OSI, en particular, la capa física, la capa de enlace de datos y la capa de aplicación. El bus LIN en una aplicación automotriz conecta generalmente los sensores y las unidades inteligentes con un ECU, que a menudo es una puerta de entrada con un bus CAN. En la Figura 7.5 se muestra como sería la interconexión entre el maestro y los esclavos en una implementación LIN.

Figura 7.5: Interconexión de un Maestro LIN con varios Esclavos LIN

Entre las ventajas de LIN se encuentra que es mucho más fácil de usar e implementar, además que es mucho más económico su uso como ya se ha mencionado. Como su principal desventaja se encuentra que está limitado a solo usar un máximo de 15 esclavos, lo que reduce la escalabilidad de esta solución. Por eso, se recomienda el uso de la tecnología LIN en sistemas que no requieran mucho rendimiento. Como consecuencia de esta desventaja, LIN no es un reemplazo total de CAN,

pero es una buena alternativa para aquellos sistemas donde el costo es algo crítico, y los requerimientos de velocidad y ancho de banda no son elevados.

7.4.3 FlexRay

FlexRay es un nuevo estándar que proporciona una comunicación serial de alta velocidad, planificación *time triggered* del bus y tolerancia a fallas de comunicación entre los dispositivos electrónicos para el futuro de las aplicaciones InV. La planificación *time triggered* permite calcular la carga del bus en función del número de mensajes que se transmiten y así ajustar las tasas de transmisión, soportando velocidades de hasta 20 Mbps. FlexRay fue desarrollado para la próxima generación de vehículos y aplicaciones [33], incluida la x-by-wire del FlexRay Consortium (Consortium fundado en el año 2000 que incluye a BMW, Bosch, DaimierChrysler y Philips).

FlexRay está diseñado específicamente para la creación de redes InV, y por lo tanto, no reemplaza a las redes existentes sino que trabaja en conjunto con sistemas ya bien establecidos como CAN y LIN. En la Figura 7.6 se muestra una red InV con FlexRay que sirve como el *backbone* que proporciona determinismo para la gestión del motor y la tolerancia a fallos de los sistemas *steer-by-wire*, *brake-by-wire* y otras aplicaciones de seguridad avanzadas. Los sistemas *steer-by-wire* permiten sustituir los sistemas mecánicos tradicionales de dirección del vehículo por sistemas electrónicos e implementaciones de seguridad, al igual que los sistemas *brake-by-wire* que permiten sustituir los sistemas tradicionales por sistemas electrónicos que puedan gestionar el frenado del vehículo.

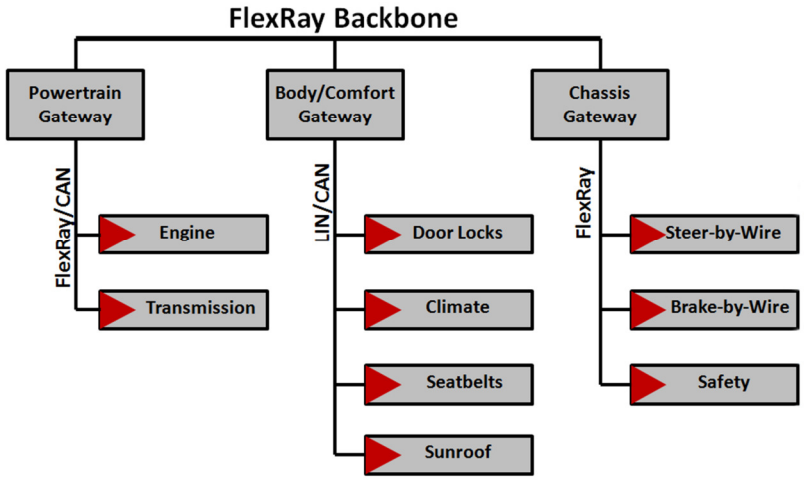


Figura 7.6: Red In-Vehicle con un Sistema FlexRay como Backbone para los Otros Sistemas

Al poder trabajar con las tecnologías ya existentes y formar redes de comunicación, Flexray permite realizar una categorización por tecnología y asignar funciones específicas para las redes formadas por cada tecnología. Por ejemplo, como la tecnología LIN es mucho más lenta y económica, se puede asignar las funciones de comunicación de sistemas como el aire acondicionado y el seguro de puertas a las redes gestionadas por esta tecnología, y unir estas redes con las demás existentes mediante un *backbone* FlexRay. Las funciones de mayor importancia como los sistemas *x-by-wire* y la seguridad deberían ser administradas completamente por la tecnología FlexRay, debido a las características de esta tecnología. La Tabla 7.1 muestra una comparación entre las tecnologías LIN, CAN y FlexRay.

Bus	Velocidad	Costo	Aplicaciones Típicas
LIN	20 kbps	Económico	Sistemas de confort al conductor (aire acondicionado, ventanas, etc).
CAN	1 Mbps	Accesible	Sistema de encendido y apagado.
FlexRay	20 Mbps	Costoso	Seguridad, sistemas *x-by-wire*, sistema de encendido y apagado que requiere mayor rendimiento.

Tabla 7.1: Comparación entre las Tecnologías LIN, CAN y FlexRay

7.5 Características de las Redes Vehiculares

Las redes vehiculares cuentan con características únicas y específicas, las cuales se nombran a continuación [28]:

- Sin problemas de batería: los problemas de alimentación del dispositivo de comunicación no suele ser un obstáculo en este tipo de redes como ocurre en el caso de redes ad hoc convencionales o redes de sensores, ya que el nodo móvil (el vehículo) puede proporcionar energía continua a los dispositivos de comunicación y de cómputo.
- Mayor capacidad de cálculo: los vehículos pueden proporcionar cómputo más significativo, comunicación y capacidades de detección mayores.
- Movilidad previsible: a diferencia de las redes ad hoc tradicionales donde es difícil predecir la movilidad de los nodos, los vehículos tienden a tener movimientos muy predecibles que son en general limitados por la infraestructura vial y los vehículos cercanos. La información de ubicación está disponible gracias a tecnologías como GPS[23] (Global Positioning System), entonces, teniendo en cuenta la velocidad media, la velocidad actual y la trayectoria de la carretera, la posición futura de un vehículo se puede predecir.

Sin embargo, las redes vehiculares tienen que hacer frente a algunas características desafiantes [28], las cuales son:

- Escalabilidad: las redes vehiculares pueden tener tamaños muy grandes en particular en horas pico, cuando aumenta el tráfico vial.
- Gran movilidad: el entorno en que operan las redes vehiculares es extremadamente dinámico, donde hay que considerar situaciones drásticas como velocidades de hasta 200 km/h en las autopistas.
- Topología dinámica: por el movimiento de los vehículos, la topología de la red cambia con frecuencia, por lo que se presenta mucho la conexión y desconexión de enlaces de comunicación. De hecho, puede ser que la duración de las conexiones sea muy corta debido al movimiento de los vehículos.

7.6 Arquitectura de Redes Vehiculares

Las redes vehiculares pueden ser desplegadas por los operadores de red y proveedores de servicios en conjunto con entes gubernamentales. Los recientes avances en las tecnologías inalámbricas permiten establecer una serie de arquitecturas de despliegue de las redes vehiculares, teniendo en cuenta todas las áreas posibles, como las carreteras, áreas rurales, y entornos de la ciudad [28]. Estas arquitecturas permiten la comunicación V2V y la comunicación V2I, donde se plantean dos alternativas que incluyen:

[23] http://www.gps.gov

- Una red inalámbrica totalmente ad hoc que permita la comunicación entre vehículos de forma independiente, es decir, sin apoyo de la infraestructura.
- Una arquitectura híbrida que no dependa de una infraestructura fija de una forma constante, pero pueda hacer uso de la misma para mejorar el rendimiento y proveer acceso a servicios adicionales cuando estén disponibles.

En el último punto, los vehículos pueden comunicarse con la infraestructura, ya sea de modo one-hop o modo multi-hop de acuerdo a las posiciones de los vehículos en relación con los RSUs. La arquitectura básica de una red vehicular debe incluir los siguientes tres ámbitos: (1) el vehículo, (2) la red ad hoc, y (3) la infraestructura. La Figura 7.7 muestra las capas de una arquitectura básica de red vehicular, donde se deben definir claramente las tecnologías en la capa física y MAC que juegan un rol muy importante para la comunicación, así como también considerar los planos de gestión y seguridad, tener en cuenta el enrutamiento y regímenes de comunicación para poder establecer los servicios y aplicaciones.

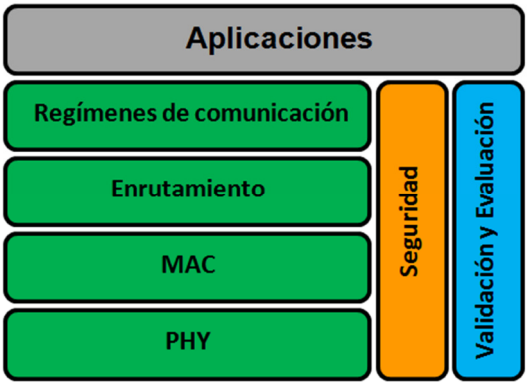


Figura 7.7: Capas de una Arquitectura Básica de Red Vehicular

Para la comunicación In-Vehicle cada vehículo tiene establecido un conjunto de ECUs que se comunican entre sí para lograr establecer ciertas funcionalidades claves, mientras que para la comunicación Out-Vehicle el vehículo tiene integrado dos tipos de unidades más [20][28]: (1) el OBU y (2) uno o más OBCs (On-Board Computer). Un OBU es un dispositivo integrado en el vehículo que tiene la capacidad de comunicación hacia el exterior mediante la tecnología que se implemente en las capas física y MAC. Mientras que un OBC es un dispositivo que tiene la capacidad de ejecutar una o un conjunto de aplicaciones, haciendo uso de las capacidades que ofrece el OBU.

La red vehicular está compuesta por los vehículos equipados con OBUs y los RSUs que se ubican en la infraestructura vial de forma fija. Los OBUs de diferentes vehículos forman entre sí redes ad hoc o VANETs (Vehicular Ad Hoc Networks), donde cada OBU está equipado con elementos de comunicación, incluyendo al menos un dispositivo de corto y mediano alcance dedicado de comunicación inalámbrica. Un RSU puede estar conectado a una red con infraestructura y a su vez estar conectado a Internet [28]. Los RSUs también pueden comunicarse entre sí directamente o a través de múltiples saltos y su función principal es la mejora de la seguridad vial, mediante la ejecución de aplicaciones, y el envío y la recepción de datos. La Figura 7.8 muestra la comunicación que se puede dar entre los OBUs y los RSUs para una red vehicular. Existen dos tipos de dominio de infraestructura [28]: (1) RSU y (2) *hot-spot*. Los RSUs permiten a los OBUs acceder a la infraestructura, tener acceso a Internet y a los servicios que ofrece la infraestructura. Los vehículos también pueden tener acceso a Internet y servicios a través de *hot-spots* públicos, comerciales o privados (WiFi *hot-spots*).

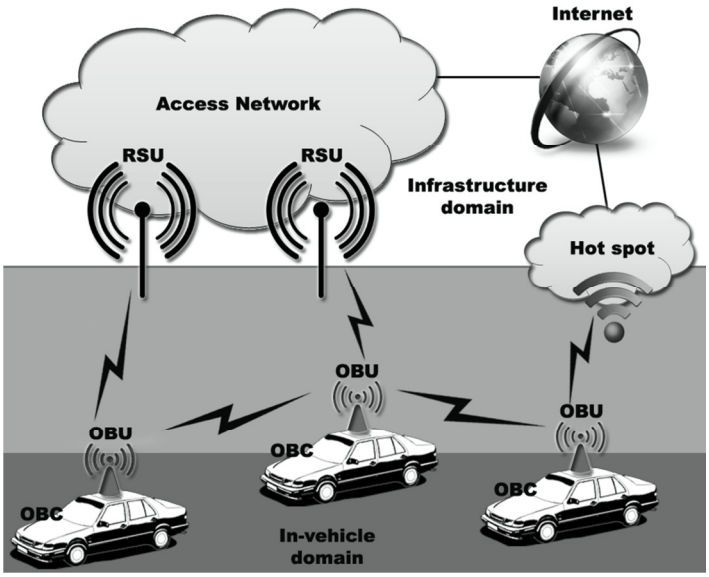


Figura 7.8: Componentes y Tipos de Comunicación que Conforman una Red Vehicular

7.7 Vehicular Ad Hoc Network

Con el fin de evitar los costos de comunicación y garantizar poco retardo para intercambiar datos relacionados con la seguridad entre vehículos, los sistemas de comunicación V2V, basado en redes inalámbricas ad hoc, representan una solución prometedora para el futuro de los escenarios de comunicación vehicular. V2V permite a los vehículos que se organicen localmente en redes ad hoc sin ningún tipo de infraestructura pre-instalada [20]. La comunicación en futuros sistemas V2V no se limitará a los vehículos vecinos que viajan dentro de un rango de transmisión de radio específico. Al igual que en los escenarios inalámbricos típicos, el sistema ad hoc V2V proporcionará capacidades multi-hop de comunicación mediante el uso de los vehículos intermedios que viajan entre el emisor y el receptor. La Figura 7.9 ilustra esta idea básica. En este ejemplo en particular, el vehículo origen es todavía capaz de comunicarse con el vehículo destino, aunque el vehículo destino no está en el rango de comunicación inmediato del vehículo origen. Los vehículos intermedios retransmiten los datos hacia el receptor. Como resultado, la capacidad multi-hop del sistema V2V aumenta significativamente el alcance de la comunicación, permitiendo la comunicación con los vehículos más distantes.

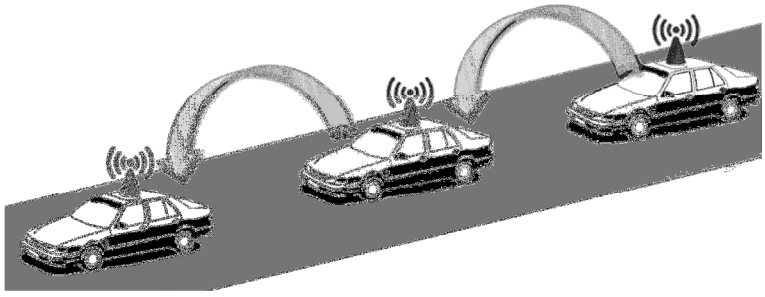


Figura 7.9: Modo Multi-Hop entre Vehículos

La optimización del flujo de tráfico vehicular es uno de los principales retos en el estudio de la congestión y la seguridad en las calles con el fin de maximizar el confort [12]. Propuestas destinadas a disminuir la congestión de tráfico, causado en parte por el flujo de tráfico vehicular ineficiente, han incluido a menudo costosos proyectos de construcción. Estos proyectos, sin embargo, han tenido un éxito limitado.

Aunque los sistemas de seguridad pasivos como cinturones de seguridad y *air-bag* se han utilizado para reducir significativamente el número total de lesiones graves y muertes por accidentes de vehículos, no hacen nada para mejorar realmente el flujo de tráfico vehicular o disminuir el número real de accidentes automovilísticos. Con el fin de reducir los accidentes, los expertos en informática y redes proponen sistemas de seguridad activa, incluyendo ITSs que se basan en la comunicación V2V y V2I [12][33]. En la actualidad, las tecnologías relacionadas con las redes vehiculares pueden de manera más eficiente llevar a cabo la administración del flujo de tráfico vehicular, y a su vez pueden tener importancia en la seguridad, ecología, y economía.

Los sistemas vehiculares activos emplean redes inalámbricas ad hoc y GPS para determinar y mantener las distancias entre los vehículos, para que se establezcan las comunicaciones one-hop y multi-hop necesarias para mantener el espaciado entre vehículos. Algoritmos de enrutamiento basados en la localización geográfica pueden ayudar en el desarrollo de redes vehiculares debido a su flexibilidad y eficiencia. Aunque varios algoritmos basados en la localización ya existen, incluyendo GLS (Grid Location Service), LAR (Location Aided Routing), GPSR (Greedy Perimeter Stateless Routing), y DREAM (Distance Routing Effect Algorithm for Mobility) [12], estos no cumplen a cabalidad con la meta que se quiere tener en las redes que pueden conformar los vehículos en las autopistas, debido a la alta velocidad de los vehículos y a los cambios topológicos frecuentes.

7.7.1 Dedicated Short Range Communications

DSRC es un conjunto de canales de comunicación inalámbrica de medio rango específicamente diseñado para uso automotriz [33], además especifica cómo se usará el rango de frecuencia asignado para los canales. Es compatible tanto con la seguridad pública y las operaciones privadas de V2V como con entornos de comunicación V2I.

DSRC es un complemento a las comunicaciones móviles, proporcionando velocidades de transferencia muy elevadas en casos donde la mínima latencia en el enlace de comunicación y el aislamiento de zonas relativamente pequeñas de comunicación son importantes. En los Estados Unidos, el FCC (Federal Communications Commission) ha designado el uso de las frecuencias 5,850 GHz – 5,925 GHz para los canales.

El espectro de DSRC [33] se estructura en siete canales, cada uno con un ancho de banda de 10 MHz. El CCH (Control Channel) se limita a las comunicaciones de seguridad. Los dos canales en los extremos de la banda del espectro están reservados para aplicaciones especiales. Los SCHs (Services Channels) están disponibles tanto para la seguridad como para el uso de otras aplicaciones.

La capa física especificada por el DSRC se estructura en súper-tramas de duración de 100 ms. Cada intervalo asociado a una súper-trama se divide en dos períodos y se dedica a un aspecto particular de la comunicación [26]. El primero de ellos es el CCH, cuya duración por defecto es de 50 ms y que se encarga del envío de información importante, normalmente relacionada con la seguridad en la conducción. El segundo es el SCH, compuesto por la multiplexación temporal cada 100 ms de canales que operan a distintas frecuencias dentro de la banda asignada. El SCH permite la transmisión de información relacionada con seguridad, entretenimiento, y administración remota a través de paquetes IP.

La diferencia fundamental entre CCH y SCH radica en el hecho de que el primero no puede usar IP para la transmisión de paquetes. Para ello recurre a un protocolo de propósito específico que opera al mismo nivel que IP conocido como WSMP [26][28] (WAVE Short Message Protocol). WSMP toma en consideración las características especiales que definen a los entornos de tránsito vehicular y reduce

sustancialmente la carga de los paquetes para mejorar las transmisiones. Los OBUs que se ubican en los vehículos soportan la comunicación a través de DSRC y así pueden establecer conexión con otros vehículos a su alrededor. En la Figura 7.10 se muestra el espectro DSRC asignado por el FCC para el uso de las redes vehiculares donde se especifican cada uno de los siete canales.

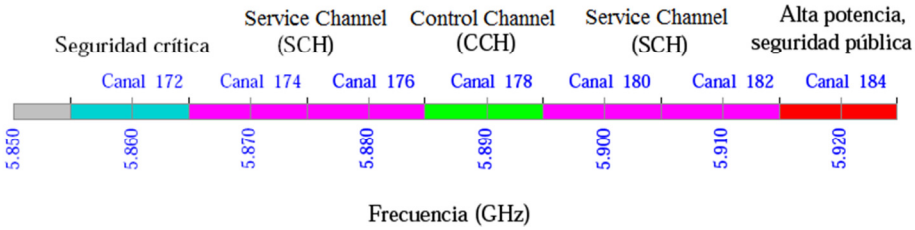


Figura 7.10: Espectro de Frecuencias de DSRC

7.7.2 IEEE 1609/802.11p

Un entorno vehicular requiere de un conjunto de nuevas exigencias en los sistemas modernos de comunicación inalámbrica. Aplicaciones de comunicación para seguridad vehicular no pueden tolerar largos retrasos para el establecimiento de la conexión antes de comunicarse con otros vehículos encontrados en el camino [28]. El estándar IEEE 802.11p, también conocido como WAVE (Wireless Access in Vehicular Environments), está diseñado para resolver estos problemas. El protocolo WAVE ofrece mejoras en la capa física (PHY) y en la capa MAC de los actuales estándares inalámbricos 802.11.

El Departamento de Transporte de los Estados Unidos promueve y soporta el ITS sobre la base de comunicaciones DSRC. El ITS se centra principalmente en permitir que las aplicaciones de seguridad pública puedan salvar vidas y mejorar el flujo de tráfico.

Para comunicaciones V2X, los estándares IEEE 1609 son desarrollados para suministrar los servicios necesarios en una capa superior dentro de la carga útil de tramas IEEE 802.11p [21][26]. La Figura 7.11 muestra la arquitectura de protocolos de WAVE.

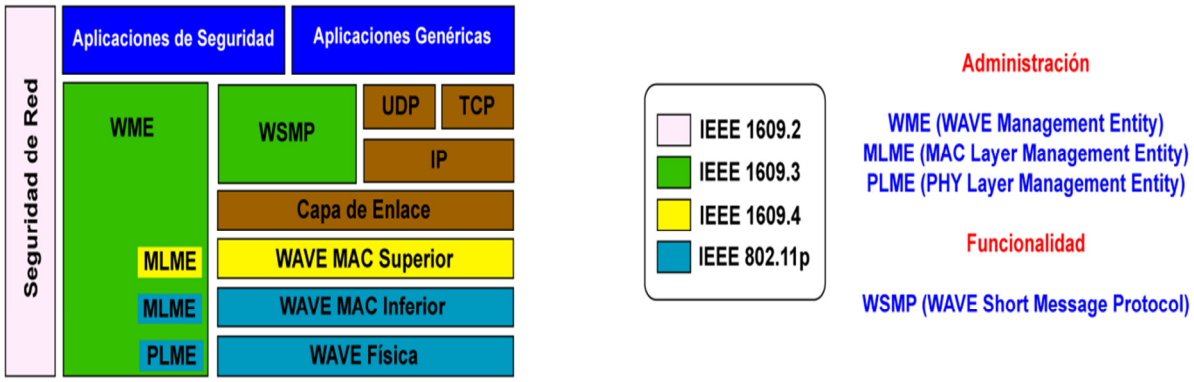


Figura 7.11: Arquitectura de Protocolos de WAVE

Cada uno de estos estándares tiene una función específica en la arquitectura WAVE [21][26]:

- IEEE 1609.1: define un administrador de recursos que permite la interoperabilidad de las aplicaciones que se ejecuten en las redes vehiculares. Forma parte de la capa de aplicación.
- IEEE 1609.2: establece los servicios de seguridad para las comunicaciones V2X, como la autenticación de las estaciones y el cifrado de mensajes.

- IEEE 1609.3: especifica los servicios de redes para las comunicaciones V2X, con un nuevo protocolo llamado WSMP que permite el intercambio de mensajes WSM (WAVE Short Message).
- IEEE 1609.4: permite la operación de múltiples canales. Tiene una fuerte relación con los mecanismos EDCA (Enhanced Distributed Channel Access) descritos en la Sección 7.7.3.
- IEEE 802.11p: define operaciones de acceso al medio a nivel inferior de la MAC y establece las adaptaciones del medio físico de comunicación.

7.7.3 Capa de Acceso al Medio

Los procesos de autenticación y asociación de IEEE 802.11 antes de un intercambio de la primera trama podrían tardar mucho tiempo. En consecuencia, la autenticación y la asociación no son proporcionados por las sub-capas PHY/MAC IEEE 802.11p, pero tienen que ser soportados por la SME (Station Management Entity) o por un protocolo de capa superior [28][33]. Todas las tramas innecesarias se eliminan y sólo queda un pequeño número de tramas necesarias, los datos se transmiten mediante uso de tramas con QoS (Quality of Service) para permitir la priorización de las tramas en un paquete de nivel básico de acuerdo a los mecanismos EDCA.

EDCA se basa en el servicio de paquetes de datos según la prioridad asignada a la aplicación que los genera. Para ello, cada estación mantiene configuraciones independientes para cada una de las cuatro interfaces definidas lo cual permite establecer distintas tipos de QoS según el tipo de aplicación al que quiera darse servicio. Cada interfaz opera con mecanismos de acceso al medio según la escucha de la portadora, es decir, cada estación virtual monitoriza el medio durante un intervalo determinado de tiempo AIFS (Arbitration Inter-Frame Space) a partir del cual ejecutará un proceso de *backoff* decremental en el que se escogerá un número de *slots* aleatorio entre el tamaño mínimo y máximo de la ventana de *backoff*. En caso de detectarse una colisión entre paquetes en el momento de transmitir, se reproducirá una nueva fase de *backoff* en la que los parámetros del tamaño mínimo y máximo de la ventana se modificarán para reducir la probabilidad de colisión en intentos sucesivos de acceso. Si se alcanza el número máximo de intentos posibles, se descarta el paquete y se reiniciarán los valores de la ventana a los iniciales para el nuevo paquete que se desee enviar al medio. Las tramas unicast son asentidas (ACK) y podrán ser precedidas por un intercambio opcional de tramas RTS/CTS.

Una trama especial de administración es introducida (la trama de tiempo e información). Se recomienda que a los RSUs se les permitan hacer publicidad de la información sobre los servicios prestados de una manera rápida. Esta información puede incluir fecha y hora y la información de sincronización de tiempo, con el apoyo de transmisión de datos o información sobre EDCA, y la posibilidad de anunciar los servicios de las capas superiores como, por ejemplo, se especifica en el IEEE 1609.

Un aspecto importante para las comunicaciones relativas a la seguridad vehicular será la priorización de mensajes importantes. El estándar IEEE 802.11p por lo tanto, se adapta específicamente a mejorar el acceso distribuido del canal [26][28] que fue originalmente propuesto en la enmienda IEEE 802.11e, introduciendo la calidad de servicio. Las reglas de acceso al medio definido por el DCF (Distributed Coordination Function) se sustituyen por las de EDCA, donde cuatro ACs (Access Category) se definen. A cada trama se le asigna una de los cuatro ACs por la aplicación que crea el mensaje, dependiendo de la importancia y urgencia de su contenido. Cada AC se identifica por su ACI (Access Category Index), mantiene su propia cola de tramas, y tiene un conjunto de parámetros individuales de coordinación del acceso al medio.

El arbitraje entre el número de espacio por trama o AIFSN (Arbitration Inter-Frame Space Number) reemplaza el tiempo DIFS (DCF Inter-Frame Space) fijos definidos en el DCF. El tiempo por el cual el medio tiene que estar inactivo antes de que se pueda acceder tiene que superar el tiempo de un SIFS (Short Inter-Frame Space) por los tiempos de *slot* AIFSN. Además, la ventana de contención mínima y máxima es individual para cada país [26]. En resumen, tramas con un ACI de 0 tienen un acceso regular, un ACI de 1 está previsto para tráfico sin prioridad, mientras que los ACIs de 2 y 3 están reservados para los mensajes de prioridad, por ejemplo, advertencias de seguridad críticas. Sin embargo, no hay priorización estricta: la discordia entre las categorías de acceso se resuelve internamente, solo la estación que tenga el menor tiempo de espera en realidad contiende con otras estaciones en el medio. Hay que tener en cuenta que colisiones internas son posibles, en este caso la trama con un mayor ACI es elegida. Como ya se mencionó, todas las funciones que se omiten para permitir un intercambio de mensajes instantáneos tienen que ser tratadas en un nivel superior de abstracción.

7.7.4 Capa Física

La capa física de IEEE 802.11p es similar a IEEE 802.11a, con algunas adaptaciones para el dominio de aplicación específico [28]. La operación se lleva a cabo en una banda independiente y reservada. Como ya se mencionó, el FCC ha destinado un ancho de banda de 75 MHz, cuyo espectro se ubica entre 5,850 y 5,925 GHz. En Europa, un ancho de banda de 30 MHz fue asignado por el ECC (Electronic Communications Committee), con una posible ampliación a 50 MHz. El estándar IEEE 802.11p especifica la operación de canales de 5, 10 y 20 MHz, mientras que las redes inalámbricas clásicas suelen utilizar canales de 20 MHz. Los canales de 10 MHz son previstos para las redes V2X debido a problemas de robustez y la posibilidad de reutilizar los chipsets inalámbricos ya existentes. Varias mediciones realizadas por una serie de investigadores en la anterior década mostraron una propagación Doppler (causada por los nodos de movimiento rápido) de hasta 2 kHz y la propagación del retardo RMS (Root Mean Square) causada por la propagación multipath de hasta 0,8 ms [28][33]. En un canal de 20 MHz de IEEE 802.11a, el intervalo de guarda entre símbolos tiene una longitud de 0,8 ms y por lo tanto es fundamental, siendo demasiado corto para mitigar las interferencias entre símbolos. Un intervalo de guarda de 1,6 ms se obtiene cuando se utiliza la mitad del ancho de banda, como se hace en el estándar IEEE 802.11p. La duración de un símbolo de datos también se duplica a 6,4 ms. Por lo tanto, la dispersión del retardo medido es menor que el intervalo de guarda, mitigando las interferencias entre símbolos.

Las interferencias inter-carrier o ICIs (Inter-Carrier Interferences) se ven mitigadas porque la propagación Doppler es mucho menor que la mitad de la distancia de separación de la sub-portadora de 156,25 kHz. Utilizando sólo la mitad del ancho de banda de la capacidad del canal también se reduce a la mitad, es decir, sólo 3 Mbps en lugar de 6 Mbps en el modo más básico [28]. El IEEE 802.11p especifica las características que debe soportar el hardware de los dispositivos de comunicación, con respecto a las temperaturas que estos alcanzan, las tolerancias permitidas de las frecuencias y los relojes de temporización. Las bajas tasas de error de bit apoya a la comunicación de alta fiabilidad, y por lo tanto, IEEE 802.11p opcionalmente especifica las regulaciones más estrictas en relación con el rechazo de canal adyacente y no adyacente, y transmitir las máscaras de espectro. Esto debería reducir la influencia de los canales vecinos el uno del otro.

7.8 Detalles de Difusión y Enrutamiento

Debido al dinamismo presente en las redes vehiculares, el enrutamiento debe ser eficiente y adaptarse a sus características y las aplicaciones que se ofrecen en ellas [28], permitiendo la transmisión de las tramas con diferente prioridad de acuerdo con el tipo de aplicación (relacionadas con la seguridad o no). Hasta ahora, la mayor parte de la investigación de las redes vehiculares se ha centrado en el análisis de los algoritmos de enrutamiento para manejar el problema de la gran cantidad de envíos en una topología de red de gran cantidad de nodos. En la actualidad, la penetración de la tecnología de redes vehiculares es un tanto débil, y por lo tanto, estas redes deben contar con el apoyo de la infraestructura existente para el despliegue a gran escala. Sin embargo, en el futuro, se espera observar un mayor uso de estas redes y con un soporte de infraestructura menor, es por ello que se debe considerar el problema de las desconexiones de enlaces en cada momento por la gran movilidad, que es un reto para la investigación fundamental para el desarrollo de un protocolo de enrutamiento confiable y eficiente. En cuanto a los envíos por broadcast de mensajes, los algoritmos de enrutamiento dependerán del tamaño de la red, así como del tipo de aplicación.

Los protocolos de enrutamiento ad hoc tienen como objetivos de diseño la optimización de la red, la simplicidad, el bajo costo operativo, la robustez, la estabilidad, la convergencia rápida y la flexibilidad. Sin embargo, ya que los nodos móviles sufren de problemas de suministro de energía, la velocidad de procesamiento y memoria, y el bajo costo operativo se vuelve aún más importante que en las redes fijas convencionales, aunque este problema no se presenta en las redes vehiculares [33]. La alta movilidad presente en la comunicación V2V también otorga gran importancia en la rápida convergencia. Por lo tanto, es imperativo que los protocolos de enrutamiento ad hoc compensen efectivamente los retrasos inherentes a la tecnología subyacente, se adapten a los diferentes grados de movilidad y sean lo suficientemente sólidos para hacer frente a la pérdida potencial de transmisión debido a la deserción. Además, estos protocolos deben enrutar los paquetes con mayor eficacia que los algoritmos tradicionales de red con el fin de compensar los recursos de ancho de banda limitados [28]. Varios algoritmos de enrutamiento para redes ad hoc han surgido para hacer frente a las dificultades relacionadas con el enrutamiento unicast. Estos algoritmos pueden clasificarse ya sean proactivos o reactivos, en función de su mecanismo de descubrimiento de ruta. Es vital el estudio de estos protocolos de enrutamiento y establecer un esfuerzo de desarrollo para incorporar las redes inalámbricas ad hoc en la industria automotriz.

En los algoritmos proactivos, cada nodo actualiza continuamente las rutas a todos los demás nodos de la red. En consecuencia, la ruta está disponible de inmediato cuando un nodo necesita enviar un paquete a otro nodo en la red. La principal ventaja de los algoritmos proactivos [28] es que tienen un retardo más corto. Como ejemplos de algoritmos proactivos se pueden mencionar OLSR y TBRPF (Topology Dissemination Based on Reverse-Path Forwarding). La desventaja de los protocolos OLSR y TBRPF es su estrategia de difusión del estado de enlace de enrutamiento. Los cambios de enlace reconocidos causan que los nodos inunden paquetes de control a través de toda la red, los cuales comprometen los recursos de la misma.

Por el contrario, los algoritmos reactivos de enrutamiento descubren las rutas por demanda. Por lo tanto, una ruta se descubre cuando un nodo origen necesita comunicarse con un nodo destino para el cual la ruta no ha sido aún establecida. El descubrimiento se basa en las inundaciones [28], donde los nodos emisores difunden un mensaje de solicitud de ruta para todos los vecinos inmediatos y éstos a su vez retransmiten la solicitud de ruta para sus vecinos. Cuando la solicitud llega al destino o a un nodo que tiene una ruta válida para el destino, un mensaje de respuesta de ruta se genera y se transmite al origen. Por lo tanto, tan pronto como el origen reciba la respuesta de ruta, se crea una ruta desde el origen al destino y viceversa. La ventaja de los algoritmos reactivos es que no hay

mensajes de control de las rutas no activas. El mayor inconveniente es la latencia en establecer las rutas de transmisión. Ejemplos de algoritmos reactivos incluyen AODV y DSR.

A pesar que los algoritmos mencionados fueron desarrollados para redes ad hoc, no se adaptan correctamente a las redes vehiculares, debido al alto dinamismo de los vehículos al transitar por las carreteras. Por esta razón, el enrutamiento es un problema abierto para las redes vehiculares y algunos investigadores han propuesto algoritmos como AGF (Advanced Greedy Forwarding) [4][28] y PGB (Prefered Group Broadcasting) [28].

8. Herramientas de Generación de Trazas para Redes Vehiculares

Una de las características de los simuladores ns-3 y OMNeT++ es la posibilidad de trabajar en conjunto a otra herramienta de simulación. Tanto ns-3 como OMNeT++ no cuentan con un generador de trazas que simule el flujo de vehículos en una red de carreteras. A pesar de que en estos simuladores existe un conjunto de modelos de movilidad, no son lo suficientemente completos como para simular el comportamiento de una red vehicular. Por fortuna, ns-3 y OMNeT++ pueden utilizarse junto a la herramienta SUMO (Simulation of Urban Mobility) o VanetMobiSim, especializadas en la generación de trazas para tráfico vehicular.

8.1 SUMO

SUMO es una plataforma de simulación de tráfico de nivel microscópico, multimodal, y código abierto que emula el flujo del tráfico de forma continua en el espacio y discreta en el tiempo[24]. Los modelos microscópicos son aquellos que simulan el movimiento de cada vehículo sobre las vías, donde su desplazamiento es determinado tanto por las capacidades físicas del vehículo para moverse como por el comportamiento del conductor para controlarlo[25]. Gracias a esto es posible simular una demanda de tráfico que consista de un conjunto de vehículos individuales, con sus propias rutas y que se mueven a través de una determinada red de carreteras. Es posible que cada conductor intente utilizar el camino más corto a través de la red pero cuando esto sucede, alguna de las carreteras (sobre todo las avenidas principales) se van a congestionar lo cual reduce el beneficio de su uso. La solución a este problema en ingeniería de tráfico se conoce como *user assignment*. SUMO [10] usa DUA (Dynamic User Assignment) para tratar con escenarios de este estilo.

El desarrollo de SUMO comenzó en el año 2000 por parte del laboratorio alemán DLR (Deutsches Zentrum für Luft- und Raumfahrt) bajo licencia GPL. La razón principal para el desarrollo de esta plataforma fue dar soporte a la comunidad investigadora con una herramienta para desarrollar y evaluar algoritmos de simulación de tráfico. Además, el DLR buscaba poner a disposición de la comunidad de investigadores una arquitectura en común para poder comparar entre sí los algoritmos.

Existen dos principales metas de diseño alcanzadas. El software debe ser rápido y portable. La primera versión fue desarrollada para ejecutarse únicamente desde la línea de comandos y con la inserción de todos los parámetros de forma manual. Esto debería aumentar la velocidad de ejecución ya que no se invierte tiempo en la visualización. Alcanzar estas metas también hizo que el software se dividiera en varias aplicaciones, cada una de estas con un cierto propósito y que serían ejecutadas de forma individual. Esta característica hace a SUMO diferente a otros paquetes de simulación ya que permite una fácil extensión de cada una de sus aplicaciones porque son más pequeñas y menos complejas. Además, permite el uso de estructuras de datos rápidas, cada una ajustada a su propósito actual en lugar de usar estructuras complicadas y muy sobrecargadas. Aun así, todo esto hace del uso de SUMO un poco incómodo para los usuarios en comparación con otros paquetes de simulación. A pesar de que no se piensa hacer ningún rediseño en la actualidad, se han integrado algunas características para mejorar su usabilidad como un comando con interfaz gráfica y la posibilidad de crear archivos con todos los parámetros en lugar de colocarlos todos en la línea de comando.

[24] http://sumo.sourceforge.net
[25] http://arxiv.org/abs/cond-mat/0007053

8.1.1 Especificaciones

Las versiones actuales de SUMO[26] cuentan con el siguiente conjunto de especificaciones:

- Movimiento de los vehículos de forma continua en el espacio y discreta en el tiempo.
- Diferentes tipos de vehículos.
- Calles de varios canales con cambio de canal.
- Diferentes reglas para ceder el paso en las intersecciones.
- Interfaz gráfica de usuario (Figura 8.1) basada en OpenGL (Open Graphics Library).
- Manejo de redes de carreteras con una gran cantidad de calles (10.000 edges).
- Velocidad de ejecución rápida (hasta 100.000 actualizaciones por segundo de vehículos en una máquina de 1GHz).
- Interoperabilidad con otras aplicaciones en tiempo real.
- Salidas en toda la red, basado en bordes, vehículos, y en detectores de salidas.
- Enrutamiento con diferentes algoritmos de asignación de usuarios dinámicos.
- Alta portabilidad, ya que SUMO está desarrollado en C++ y utiliza bibliotecas comunes.
- Instalación en los principales sistemas operativos (Unix, Windows y Mac OS).
- Uso de archivos XML (eXtensible Markup Language) para los datos generados, que simplifica la integración con otras herramientas.

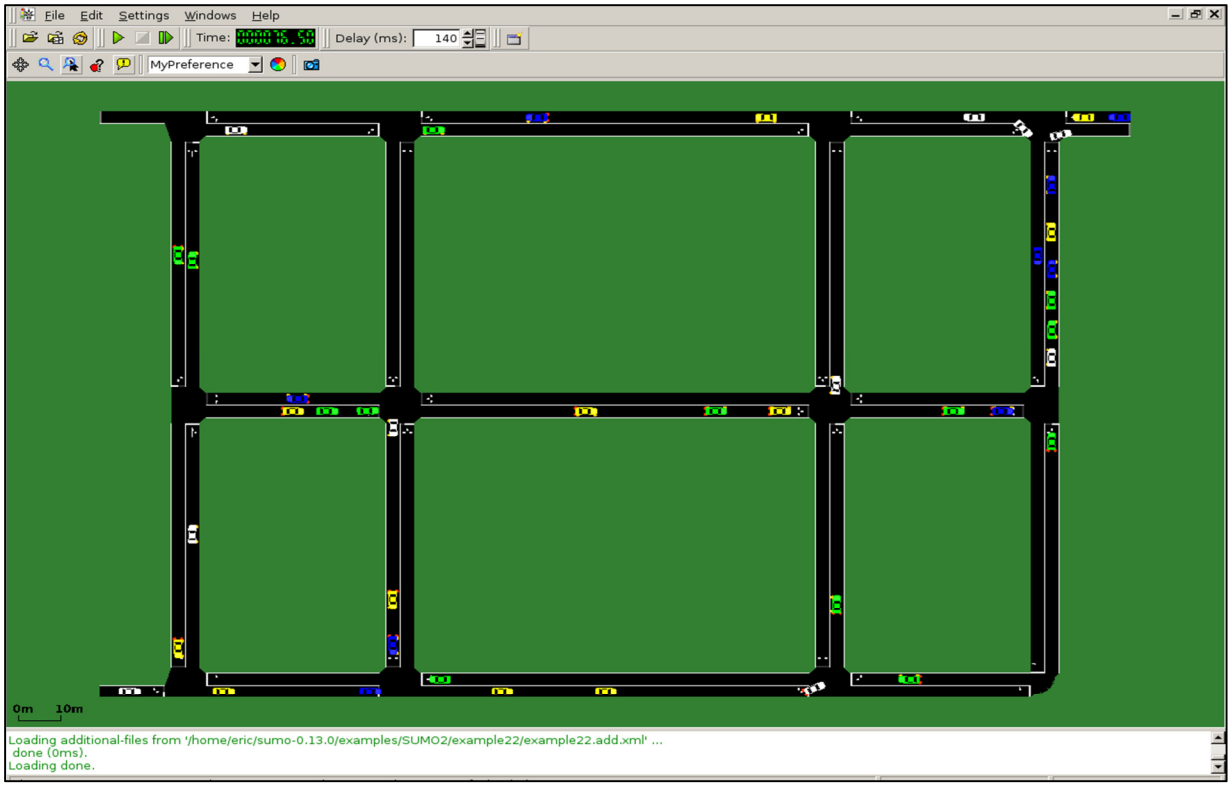


Figura 8.1: Interfaz Gráfica de SUMO

[26] http://sumo.sourceforge.net/doc/current/docs/Sumo_at_a_Glance.html

8.1.2 Aplicaciones

SUMO contiene un conjunto de aplicaciones adicionales utilizadas principalmente para importar o preparar redes de carreteras y los datos necesarios que serán utilizados para la simulación. Estas aplicaciones se muestran en la Tabla 8.1.

Nombre de la Aplicación	Descripción Corta
SUMO	La simulación microscópica sin visualización, aplicación de línea de comando.
SUMO-GUI	La simulación microscópica con interfaz gráfica de usuario.
NETCONVERT	Importador y generador de redes. Lee redes de carreteras desde distintos formatos y convierte estos al formato SUMO.
NETGEN	Genera redes abstractas para la simulación en SUMO.
DUAROUTER	Importa diferentes definiciones de demanda y con ellas construye rutas rápidas a través de la red. También se encarga de calcular las rutas durante un *user assignment*.
JTRROUTER	Calcula las rutas utilizando porcentajes para girar en las intersecciones.
DFROUTER	Calcula las rutas partiendo de un punto de observación dado como medida. Estos puntos se denominan *induction loop*.
OD2TRIPS	Importa matrices O/D y divide estas en los viajes de cada vehículo.
POLYCONVERT	Importa puntos de interés y polígonos desde distintos formatos y traduce estos en una descripción que puede ser visualizada por SUMO-GUI.
ACTIVITYGEN	Genera una demanda basada en los deseos de movilidad de una población modelo.
Additional Tools	Existen algunas tareas para las cuales escribir una aplicación de gran escala no es necesario. Muchas soluciones para problemas diferentes pueden cubrirse mediante estas herramientas adicionales[27].

Tabla 8.1: Aplicaciones de SUMO

8.1.3 OpenStreetMap

SUMO también tiene la capacidad de importar formatos de archivos que definen redes de carreteras. Esto brinda la posibilidad de construir una red de carreteras partiendo de archivos y de mapas que ya están creados y se encuentran disponibles. De esta manera, es posible construir simulaciones más reales donde los vehículos se muevan sobre carreteras existentes las cuales no tienen que ser diseñadas desde cero dentro de SUMO. Uno de los formatos que soporta SUMO para importar estas redes de carreteras es el formato de mapas OSM (OpenStreetMap).

OpenStreetMap es un proyecto colaborativo para crear mapas libres y editables. Mediante OSM se crean y proveen datos geográficos tales como los mapas de las carreteras para cualquier persona que los necesite. El proyecto comenzó como consecuencia de que la mayoría de los mapas actualmente tienen limitaciones bien sean legales o técnicas para poder utilizarlos. Los mapas son creados utilizando información geográfica capturada con dispositivos GPS, representaciones

[27] http://sumo.sourceforge.net/doc/current/docs/Tools/Main.html

fotográficas de las zonas u otras fuentes. Tanto las imágenes creadas como los datos vectoriales almacenados en sus bases de datos se distribuyen bajo licencia Creative Commons Attribution-ShareaLike 2.0. Actualmente la comunidad de OSM está considerando cambiar a otra licencia abierta ODbL (Open Database License) la cual ofrece una mejor protección y mayor claridad en cuanto a la definición de los usos de los datos geográficos.

Para obtener un mapa donde los vehículos puedan transitar es posible utilizar directamente la página web de OpenStreetMap[28] donde existe una función para descargar el mapa o mediante el uso de la herramienta JOSM[29] un editor de OSM escrito en Java. De ambas maneras se puede almacenar una selección de objetos (por lo general mediante un rango rectangular) dentro de un archivo OSM. Simplemente con utilizar este archivo se cuenta con un mapa para la simulación de tráfico en SUMO. Por lo general los datos de los mapas OSM no están completos y se tienen que mejorar para hacer una buena simulación.

Una vez que se cuenta con un buen mapa OSM para la simulación es necesario hacer una conversión a este archivo ya que SUMO cuenta con su propio formato denominado SUMO network. Entonces, primero se tiene que convertir el formato OSM al formato SUMO network por medio de la herramienta NETCONVERT que se encarga de extraer la información relacionada con la simulación almacenada en el archivo OSM y la coloca en un archivo SUMO network. Una vez obtenido este archivo ya es posible generar tráfico vehicular sobre la red de carreteras del área seleccionada en el mapa OSM.

8.2 VanetMobiSim

Vehicular Ad Hoc Networks Mobility Simulator (VanetMobiSim) [14][15] es un conjunto de extensiones de CanuMobiSim[30], un framework usado para la simulación genérica de movilidad por el CANU[31] (Communication in Ad Hoc Networks for Ubiquitous Computing) en la Universidad de Stuttgart, Alemania. CanuMobiSim es una plataforma independiente basada en Java, provee una arquitectura de movilidad eficiente y fácilmente extensible. El framework incluye modelos de movilidad, convertidor de datos geográficos en varios formatos y un entorno gráfico. Genera trazas de movilidad para los siguientes simuladores de red: ns-2, GloMoSim[32] (Global Mobile Information System Simulation Library), y QualNet[33].

Al ser de propósito general, CanuMobiSim no cuenta con un alto nivel de detalle que permita representar escenarios específicos como la simulación de ambientes vehiculares [15]. Es por ello que VanetMobiSim extiende a CanuMobiSim para soportar la movilidad vehicular con un alto grado de realismo. Estas extensiones consisten principalmente de un modelo de topología vial usando estructuras de datos compatibles con GDF (Geographic Data Files) [13][15] y un conjunto de modelos de movilidad orientados al ambiente vehicular. El modelo topológico está compuesto de elementos espaciales, sus atributos y las relaciones que unen esos elementos espaciales para describir las áreas vehiculares, es decir, las calles y carreteras.

[28] http://www.openstreetmap.org
[29] http://josm.openstreetmap.de
[30] http://vanet.eurecom.fr
[31] http://xurl.es/i630f
[32] http://pcl.cs.ucla.edu/projects/glomosim
[33] http://www.scalable-networks.com

8.2.1 Características de Macro-Movilidad

En la macro-movilidad se toma en cuenta una serie de especificaciones de las carreteras y calles como la topología, la estructura (si es unidireccional o bidireccional, simple o multicanal), las características (límites de velocidad, restricciones de clases de vehículos), y la presencia de señales de tránsito [13][15]. Además, el concepto de macro-movilidad también incluye los efectos de la presencia de puntos de interés que influencian en los patrones de movilidad de los vehículos en la topología vial.

La elección de la topología vial es un factor clave importante para obtener resultados más reales cuando se está simulando ambientes vehiculares. De hecho, la longitud de las calles, la frecuencia de intersecciones, o la densidad de edificios pueden afectar las métricas de movilidad como la velocidad mínima, máxima y promedio de los vehículos, o la densidad de vehículos sobre el mapa simulado. VanetMobiSim permite la definición de la topología vial [13][15] de las siguientes maneras:

- Grafo definido por el usuario: la topología es especificada por el usuario indicando los vértices del grafo y sus conexiones.
- Mapa GDF: la topología es importada de un GDF. Desafortunadamente, muchas librerías de archivos GDF no están libremente accesibles.
- Mapas TIGER/Line: la topología es extraída de un mapa obtenido de la base de datos TIGER proporcionada por el US Census Bureau. El nivel de detalle de los mapas TIGER/Line no es tan alto como el que provee el estándar GDF, pero esta base de datos es abierta y contiene descripciones digitales de todas las áreas urbanas y rurales de todos los distritos de los Estados Unidos. De hecho, las descripciones de la topología en base a los datos TIGER se están convirtiendo común en las simulaciones con VANETs.
- Grafo Voronoi: la topología es aleatoriamente generada por la creación de *tessellations* Voronoi en un conjunto de puntos distribuidos de forma no uniforme.

La Figura 8.2 muestra cómo se verían gráficamente la definición de la topología vial que se escoja para realizar una simulación en VanetMobiSim.

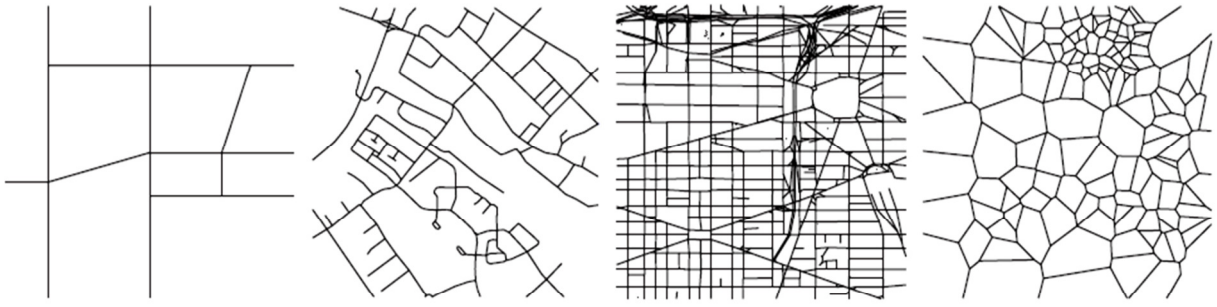


Figura 8.2: Ejemplos de la Topología Vial: (a) definido por el usuario, (b) definido por un mapa GDF, (c) definido por un mapa TIGER, (d) definido mediante un grafo Voronoi

En todos los casos, la topología vial es implementada como un grafo donde el movimiento de los vehículos está restringido por las conexiones del grafo. El concepto de macro-movilidad vehicular no se limita a las restricciones de movimiento obtenidas por un grafo dado por la topología, sino que también se incluyen todos los aspectos relacionados a la caracterización de la estructura de las calles y carreteras [15]:

- Separación física de los flujos de tráfico contrarios en cada calle.
- Introducción de calles con múltiples canales en cada dirección.
- Restricciones de velocidad en cada segmento de calle.
- Implementación de señales de tránsito en cada intersección.

8.2.2 Características de Micro-Movilidad

El concepto de micro-movilidad vehicular incluye todos los aspectos relacionados a la simulación de la velocidad y aceleración de un vehículo individual [15]. VanetMobiSim agrega dos modelos de movilidad microscópicos para incluir la gestión de las intersecciones reguladas por las señales de tránsito y las carreteras de múltiples canales. Estos modelos son [13]:

- IDM_IM (Intelligent Driving Model with Intersection Management): agrega capacidades de gestionar intersecciones, donde se pueden dar dos posibles escenarios: (1) un cruce de calles regulado por señales de parada y (2) una unión de calles regulada por semáforos.
- IDM_LC (Intelligent Driver Model with Lane Changes): extiende el modelo IDM_IM con la posibilidad de que los vehículos puedan cambiarse de canal para adelantar a otros.

8.2.3 Archivos TIGER/Line

TIGER (Topollogically Integrated Geographic Encoding and Referencing) es un formato usado por el US Census Bureau para describir los atributos topológicos y geográficos de su territorio [35], incluyendo Puerto Rico y las Islas Vírgenes. El diseño de la base de datos TIGER adapta las teorías topológicas de grafos y asocia los campos de las matemáticas para proveer la descripción de la estructura geográfica. La estructura topológica define la localización y relación de calles, ríos, sistemas de ferrocarriles y otras características y entidades geográficas para que el US Census Bureau cumpla con sus funciones de forma más sencilla [35]. Además de su uso para mecanismos de censo, este formato de mapa es muy útil en el ámbito de la simulación de redes vehiculares, ya que VanetMobiSim posee la capacidad de definir la topología vial mediante un mapa TIGER/Line.

La construcción de la base de datos TIGER incluyó una variedad de técnicas de codificación [35] tales como escaneo automático de mapas, digitalización manual de mapas, datos claves estándar y mecanismos sofisticados de cómputo. La meta era proveer acceso automatizado a información geográfica relevante acerca del territorio de los Estados Unidos. Los mapas TIGER están disponibles sin costo alguno, pueden usarse sin ningún problema con VanetMobiSim.

Los archivos TIGER/Line contienen datos que describen tres grandes tipos de características:

- Características de línea.
  - Calles.
  - Rieles.
  - Hidrografía.
  - Características variadas de transporte, líneas eléctricas, y tuberías seleccionadas.
  - Límites.
- Características de áreas.
  - Áreas puntuales como escuelas e iglesias.
  - Áreas extensas como parques y cementerios.
- Características de forma.
  - Códigos de entidades geográficas para áreas usadas por censos.

- Localización de áreas.

La información de las características de línea y de forma comprende la mayor parte de los datos en un archivo TIGER/Line. Algunos de los datos que describen las líneas incluyen coordenadas, identificadores o nombres, códigos de clasificación, rangos de dirección, y códigos geográficos [35]. También los archivos TIGER/Line contienen puntos y etiquetas de áreas que describen características relevantes que proveen referencias locales para quienes usan los mapas.

9. Trabajos Relacionados

En este capítulo se abordará una serie de trabajos que se relacionan con el análisis de desempeño para distintas herramientas y software de simulación. Estos trabajos abarcan desde el área de simulación de redes clásicas (cableadas o MANETs) hasta el análisis de la escalabilidad de algoritmos de enrutamientos para las redes vehiculares a través de la simulación.

9.1 Análisis de Desempeño y Escalabilidad de Algoritmos de Enrutamiento para Redes Móviles

Muchos trabajos relacionados con el análisis del desempeño a través de la simulación se enfocan en los algoritmos de enrutamiento para redes MANETs. En el trabajo titulado "Performance Analysis of AODV, DSR and OLSR in MANET" [3], se estudia el rendimiento que tienen los protocolos AODV, DSR y OSLR en el ámbito de las MANETs, utilizando OPNET como herramienta de simulación. Establecen una serie de escenarios de simulación distintos variando como único parámetro el número de nodos móviles, es decir, escenarios de 20, 40 y 80 nodos. En cada escenario se calcula el retardo, throughput y la carga de red, realizando este estudio para cada uno de los algoritmos de enrutamiento ya mencionados. La simulación está basada en la utilización de tráfico FTP en una red de área de 1000 m^2 donde se ubican los nodos móviles y un nodo WLAN fijo que actúa como servidor. En el análisis de los resultados obtenidos concluyen que el algoritmo OLSR se adapta mejor en ambientes MANETs que los otros dos algoritmos, ofreciendo para cada escenario de simulación bajo retardo y mayor throughput. Sin embargo, no necesariamente el rendimiento OLSR siempre es el mejor [3], varía según la escalabilidad de la red.

Otro trabajo orientado a la evaluación de desempeño de algoritmos de enrutamiento en redes MANETs es "Scalability Study of Ad Hoc Wireless Mobile Network Routing Protocol in Sparse and Dense Networks" [24]. Dicho trabajo se centra en el análisis de la escalabilidad y performance de algoritmos de enrutamientos para redes MANETs en escenarios de redes con gran densidad de nodos y de redes con nodos dispersos. Para ello se hace uso de ns-2 como herramienta de simulación. Los algoritmos de enrutamiento evaluados fueron AODV y DSDV (Destination-Sequenced Distance-Vector Routing). Cada ambiente de simulación está compuesto de una cierta cantidad de nodos y un área para su ubicación, donde según sea el caso, el área se adecua para lograr que sea un ambiente disperso o denso. La meta es evaluar la sobrecarga de los algoritmos de enrutamiento cuando se aplican en estos ambientes densos y dispersos. Según los autores, la escalabilidad para los dos algoritmos es pobre en redes dispersas en comparación a las redes densas. Sin embargo, es de notar que la sobrecarga en las redes dispersas es mínima. El algoritmo que posee una mayor sobrecarga es AODV. También realizan un análisis comparativo combinando los dos escenarios (escenario híbrido) demostrando que la porción densa de la red tiene siempre un mayor impacto en el rendimiento de la red en general, sin importar si el algoritmo de enrutamiento es AODV o DSDV.

9.2 Análisis de Desempeño en Redes Vehiculares y VANETs

Debido al auge de las redes vehiculares, muchos trabajos presentan diversos aspectos de las herramientas especializadas en la simulación de este tipo de sistema de comunicación. Por ejemplo, el artículo "A Survey and Comparative Study of Simulators for Vehicular Ad Hoc Networks (VANETs)" [27] se enfoca en el estudio y comparación de una serie de herramientas y software open source que se pueden utilizar para el estudio y simulación de VANETs. Dividen las herramientas en tres categorías: (1) los generadores de trazas y movilidad, (2) los simuladores de red, y (3) los simuladores de VANETs. En cada categoría hacen un estudio de una serie de herramientas para determinar cuáles son las más robustas y completas para el estudio de VANETs.

En la categoría de generadores de trazas y movilidad se encuentran las herramientas VanetMobiSim, SUMO, MOVE, STRAW, FreeSim, y CityMob. Se concluye que VanetMobiSim, SUMO, MOVE y STRAW poseen mayor robustez y mejores características para el ámbito de VANETs [27], destacándose VanetMobiSim como el único que provee un excelente soporte de trazas. FreeSim y CityMob proveen buenos resultados pero se encuentran limitados en ciertas características críticas.

Como simuladores de red estudiados se encuentran ns-2, GloMoSim, JiST/SWANS y SNS. Se evalúan parámetros como la escalabilidad, la portabilidad, facilidad de usar, entre otros; pero la principal característica que se evalúa es el soporte para VANET, es decir el soporte para 802.11p y modelos de tráfico vehicular. De los simuladores estudiados, solo ns-2 en su versión 2.33 soporta el estándar 802.11p [27]. Sin embargo, los simuladores estudiados poseen los requerimientos necesarios para la simulación de VANETs utilizando otra tecnología de comunicación como WiFi.

En el caso de simuladores de VANETs se estudiaron las herramientas TraNS, MobiREAL, NCTUns, y GrooveNet. Se comparan parámetros como el tipo de topologías que soportan, generación de trazas, facilidad de usar, entre otros. Se concluye que NCTUns y GrooveNet son los mejores simuladores de VANETs [27] ya que tienen un alto soporte en el ámbito vehicular y proveen de un gran realismo en las simulaciones, además que muchos otros trabajos hacen énfasis en el uso de estas herramientas.

Otro artículo titulado “Measuring the Performance of IEEE 802.11p Using ns-2 Simulator for Vehicular Networks” [29], se enfoca en el análisis del desempeño de IEEE 802.11p usando ns-2 como herramienta de simulación. Para ello hacen uso de la combinación de ns-2, C++, TCL, y AWK donde simulan y analizan una carretera simple de un solo canal con condiciones variantes. Estas condiciones fueron el número de vehículos, la velocidad y la distancia promedio entre los vehículos. La cantidad de vehículos fue establecida entre un mínimo de 2 vehículos hasta un máximo de 200, moviéndose a una velocidad de 0 a 120 km/h y con distancia entre vehículos de 5 a 26 metros. Implementaron un script TCL que contiene una configuración parcial necesaria para simular y soportar los mecanismos de 802.11p. Para la generación de tráfico implementaron una aplicación en C++. Para el análisis de los resultados implementaron un script en AWK que genera archivos de salida independientes de los resultados de la simulación para luego ser graficados y estudiados. Como análisis de desempeño, se tomaron en cuenta las variables de pérdida de paquetes, throughput, y retardo, donde se variaba la cantidad de vehículos. Gracias a este trabajo se encontraron dos bugs en ns-2 que limitaban en gran medida la simulación de estos escenarios [29]. Además, se concluye que el factor más importante que determinará el rendimiento de las redes vehiculares en un futuro será la cantidad de vehículos. Cabe acotar que para este trabajo no fueron incluidas las características de multi-canal y otras configuraciones necesarias.

9.3 Comparación entre Distintos Simuladores

La comparación entre simuladores es vital ya que se determinan las fortalezas y debilidades que cada uno posee para así poder decidir que simulador usar para cierto escenario en específico que se quiera simular. En el trabajo “Simulation of Ad Hoc Networks: ns-2 compared to JiST/SWANS” [34], se compara el rendimiento de ns-2 y de JiST/SWANS en base de los algoritmos de enrutamiento AODV y CGGC. Se establecen varios escenarios de simulación, donde se varía el número de nodos, la distancia entre los nodos y el área que abarca la red. Todas las simulaciones fueron ejecutadas en una máquina con sistema operativo Gentoo v1.12.6 con kernel de Linux v2.6.15, CPU Pentium IV a 3.0 GHz, y 882 MB de RAM. Como resultados a evaluar tomaron en cuenta retardo, número de saltos, entregas exitosas de los mensajes, el tiempo de procesamiento, y la cantidad de memoria consumida para la simulación. En los resultados de performance se observa que los dos simuladores difieren bastante, a pesar que ambos arrojan resultados buenos en simulación. Sin embargo, a nivel de tiempo de procesamiento y de cantidad de memoria consumida para la simulación se observa que

JiST/SWANS tiene mejores resultados en este aspecto. Por lo tanto, JiST/SWANS [34] está mejor capacitado que ns-2 para usarse en simulaciones de redes ad hoc e inclusive se recomienda en trabajos futuros expandir las características de JiST/SWANS para dar un mayor soporte en este tipo de redes.

En el artículo "A Performance Comparison of Recent Network Simulator" [39] se compara el rendimiento de simuladores de red recientes a través de escenarios de simulación de redes cableadas. Para realizar la comparación de los simuladores se establecieron escenarios donde varían el número de nodos, llegando a un máximo de 3025 nodos en la red. Todas las simulaciones fueron ejecutadas en un *workstation* AMD Athlon 64 3800+ con 2 GB de RAM ejecutando Ubuntu Linux 8.04 LTS [39]. Los simuladores a comparar fueron ns-2 v2.33, ns-3 v3.1, OMNeT++ v3.4b2, SimPy v1.9.1 y JiST/SWANS v1.06. Los resultados a evaluar fueron el tiempo de procesamiento y la cantidad de memoria consumida por la simulación. En todas las simulaciones se observa que JiST/SWANS [39] tiene mejores tiempos de cómputo pero es la herramienta que más memoria consume al momento de ejecutar la simulación. A nivel de tiempo de procesamiento le sigue ns-3, que consume menos memoria en comparación con las demás herramientas, siendo más idónea para simulaciones a gran escala. Sin embargo, se concluye que ns-3, JiST/SWANS, y OMNeT++ son las herramientas con mayor capacidad para la simulación de redes a gran escala de forma eficiente, destacando JiST/SWANS por su alta eficiencia en tiempo pero sin dejar a un lado su problema con el uso de memoria. Además los autores concluyen que un punto fuerte que posee OMNeT++ [39] es su capacidad de proveer una interfaz de usuario amigable en la cual el usuario puede interactuar con la simulación, pero dicha interfaz acarrea cierta lentitud.

Otro trabajo que compara distintos simuladores es "Comparison of Network Simulators Revisited" [31]. Se enfoca en una comparación de las herramientas de simulación JavaSim, ns-2, y SSFNet. Como escenario de simulación se hace uso de sesiones TCP entre hosts, lográndose establecer hasta 10000 sesiones TCP en el ambiente de simulación más pesado. Se usa la cantidad de tiempo de la simulación y la cantidad de memoria consumida como variables para el análisis de desempeño. Se determina que JavaSim es una herramienta muy lenta en comparación con ns-2 y SSFNet, dando tiempos de ejecución muy largos. Sin embargo a nivel de memoria consumida tiene un buen comportamiento, al igual que en la cantidad de eventos generados por simulación.

9.4 Evaluación de Desempeño en Redes de Gran Tamaño

Las herramientas de simulación proveen los modelos necesarios para hacer evaluación de la escalabilidad y eficiencia de una red que posee una gran cantidad de nodos y conexiones sin necesidad de tener la red real. El trabajo "Large-Scale Network Simulation: How Big? How Fast?" [8], se enfoca en la evaluación de desempeño de la simulación de grandes redes bajo el enfoque del paralelismo usando ns-2. Proponen una medida para especificar la velocidad del simulador llamada PTS (Packet Transmissions that can be simulated per Second of wallclock time). Esta métrica es muy útil ya que se puede estimar la cantidad de tiempo que será requerido para completar la ejecución de la simulación, siempre y cuando se conozca la cantidad de tráfico que debe ser simulado y la cantidad promedio de saltos requeridos para transmitir un paquete. Proponen una serie de modelos matemáticos para llevar a cabo el análisis de desempeño. Como simuladores que soportan paralelismo se enfocaron en PDNS (Parallel/Distributed Network Simulator) basado en ns-2, y GTNetS (Georgia Tech Network Simulation) cuyos modelos están escritos en C++ y usa muchas de las técnicas que implementa PDNS para permitir la ejecución en paralelo. Como ambiente de simulación se propusieron modelar una red de universidad formada por varios campus. Cada CN (Campus Network) consiste de 4 servidores, 30 routers y 504 clientes haciendo un total de 538 nodos. La meta principal es conectar múltiples CNs en una topología anillo. Las conexiones entre los nodos variaban según el contexto. Para el estudio del rendimiento se enfocaron en tráfico TCP puro

entre los clientes y los servidores. Para ejecutar el ambiente de simulación usaron una plataforma clúster que consistía en 17 máquinas, un total de 136 CPUs. Cada máquina era un SMP (Symmetric Multi-Processor) con 8 procesadores Pentium III Xeon de 550 MHz. Los CPUs en cada máquina compartían 4 GB de RAM. Las 17 máquinas SMP fueron conectadas vía un switch Dual Gigabit Ethernet con agregación de EtherChannel. Todas las máquinas poseían como sistema operativo la distribución Red Hat Linux 7.3. Como resultado de la simulación concluyeron que bajo el esquema PDNS se obtenían mejor rendimiento que GTNetS, donde se tomaron como variables de análisis de desempeño el tiempo consumido por la simulación, el PTS, y la cantidad de mensajes enviados.

Otro trabajo que se enfocó en realizar análisis de desempeño de la simulación de grandes redes ad hoc usando como herramienta de simulación a ns-2 es abordado en el artículo “Simulation of Large Ad Hoc Networks” [30]. Debido a las limitaciones de ns-2 para este enfoque, proponen mejorar éste agregando una serie de optimizaciones y modelos matemáticos para realizar la simulación. Realizan variaciones en el número de nodos y el área donde se ubicarán los mismos, la máxima cantidad de nodos es de 1.000 y el área máxima es de 25 km^2. Como principales medidas de análisis de rendimiento se tomó en cuenta el tiempo de ejecución y la cantidad de memoria consumida, haciendo una comparación entre el uso de ns-2 sin ninguna modificación o mejora y ns-2 con la mejora propuesta en el trabajo. Como conclusión, el ns-2 modificado ofrece mejor desempeño en tiempo que el ns-2 original, sin embargo, la cantidad de memoria consumida bajo los dos esquemas hace a ns-2 una herramienta limitada para la simulación de redes de gran cantidad de nodos.

10. Marco Metodológico

Para lograr cumplir con los objetivos planteados en el Capítulo 2, es necesario definir un esquema o metodología de trabajo que permita desarrollar la aplicación de manera estructurada y planificada. A continuación se presenta la especificación de la metodología utilizada y otros detalles importantes que fueron tomados en cuenta para el desarrollo e implementación del conjunto de benchmarks.

10.1 Adaptación de la Metodología de Desarrollo

La programación extrema o XP [1] (eXtreme Programming) es una metodología de desarrollo ágil [2] basada en una serie de valores y buenas prácticas que persigue el objetivo de aumentar la productividad a la hora de desarrollar programas. Este modelo de programación busca dar prioridad a los trabajos que dan un resultado directo y dejar en segundo plano aquellas actividades más burocráticas que existen alrededor de la programación.

Al igual que otras metodologías ágiles, la programación extrema se diferencia de las metodologías tradicionales principalmente en que hace mayor énfasis en la adaptabilidad y menos en la previsibilidad. El enfoque de XP se basa en que los cambios que ocurren en los requisitos son un aspecto a menudo natural e inevitable en los proyectos de desarrollo de software. Ser capaz de adaptarse a las cambiantes necesidades en cualquier momento durante la vida del proyecto es un enfoque más realista y mejor que intentar definir todos requisitos al comienzo de un proyecto y luego gastar esfuerzo para controlar cambios en los requisitos.

La programación extrema[34] define cuatro actividades básicas que se realizan dentro del proceso de desarrollo de software: Planificación, Diseño, Codificación y Pruebas.

10.1.1 Planificación

El primer paso de cualquier proyecto que siga la metodología XP es la de generar los requerimientos, establecer los tiempos de implementación ideales de cada uno de estos y la prioridad con la que serán implementados.

Una vez definidos los requerimientos, el tiempo de implementación y la prioridad de los mismos, el proyecto se ha de dividir en pequeñas iteraciones de aproximadamente 2 o 3 semanas de duración con las cuales se irán cubriendo pequeñas características o funcionalidades requeridas.

10.1.2 Diseño

El diseño crea una estructura que organiza la lógica del sistema, un buen diseño permite que el sistema crezca con cambios en un solo lugar. Los diseños deben de ser sencillos, si alguna parte del sistema es de desarrollo complejo, se divide en varias partes. Si hay fallos en el diseño o malos diseños, estos deben de ser corregidos cuanto antes.

Es importante codificar porque sin código no hay programas pero también hacer buen diseño evitará una gran cantidad de dependencias dentro de un sistema, lo que significa que el cambio en una parte del sistema no afectará a otras partes del mismo.

[34] http://www.extremeprogramming.org/rules.html

10.1.3 Codificación

El único producto verdaderamente importante del proceso de desarrollo del sistema es el código. Sin código no existe un producto de trabajo. La codificación también se puede utilizar para determinar cuál es la solución más adecuada y también puede ayudar a comunicar pensamientos acerca de problemas de programación. Es la única actividad de la que no se puede prescindir.

10.1.4 Pruebas

Las características del software que no pueden ser demostradas mediante pruebas simplemente no existen. Las pruebas dan la oportunidad de saber si lo que fue implementado es lo que en realidad se quería implementar, indican si el trabajo realizado funciona cuando no es posible pensar en ninguna prueba que pudiese originar un fallo en el sistema.

En la programación extrema se considera el hecho de que si con pocas pruebas se puede eliminar algunos defectos, con muchas pruebas se pueden eliminar una gran cantidad de defectos más. Es necesario pensar en todas las posibles pruebas para el código, estas deben de ser sensatas y valientes. La idea es diseñar pruebas que examinen el sistema a fondo, ir programando y probando ya que es más rápido que solo programar. Así se invierte menos tiempo en la depuración y serán menos los errores.

10.2 Tecnologías a Utilizar

Será utilizado (1) el simulador de red OMNeT++ el cual está escrito en el lenguaje C++ y donde las simulaciones también son escritas en dicho lenguaje. Adicionalmente, es necesario integrar en OMNeT++ el framework (2) INETMANET que contiene los modelos para redes de comunicación de datos. Otro de los simuladores a utilizar es (3) JiST un simulador que está escrito en Java y donde las simulaciones son construidas en dicho lenguaje también. Adicionalmente a JiST, es necesaria también la integración de (4) SWANS, conjunto de paquetes que contiene todo lo necesario para la simulación de redes. Por último, también será utilizado el simulador de red (5) ns-3 y los simuladores de tráfico (6) SUMO y (7) VanetMobiSim.

A continuación se describen brevemente cada una de las tecnologías mencionadas:

- C++: es un lenguaje de programación imperativo, orientado a objeto derivado del lenguaje C.
- Java: lenguaje de programación creado por Sun Microsystems, Inc. Permite crear programas que funcionan en cualquier tipo de computador y sistema operativo ya que está basado en una máquina virtual.
- OMNeT++: es un entorno de simulación basado en eventos discretos distribuido bajo Licencia Publica Académica. Fue diseñado desde un principio para soportar simulaciones de redes a gran escala con periodos de procesamiento rápidos y sin limitar la ejecución de distintos eventos que se puedan simular en la red.
- INETMANET: es una extensión del INET framework. Generalmente provee la misma funcionalidad que el framework INET pero contiene protocolos adicionales y componentes que son especialmente útiles para modelar comunicaciones inalámbricas.
- JiST: es un motor de alto desempeño para simulaciones de eventos discretos que se ejecuta sobre una máquina virtual de Java estándar. Permite a los programadores simular diferentes tipos de escenarios de computación de manera eficiente y transparente.
- SWANS: es un componente de software que se instala sobre JiST que permite formar redes inalámbricas o armar redes de sensores, permitiendo hacer simulaciones con un gran número de nodos.

- ns-3: simulador de red basado en eventos discretos que cuenta con unas herramientas poderosas y completas para simular sistemas de redes cableadas e inalámbricas.
- SUMO: es una plataforma de simulación de tráfico de nivel microscópico, multimodal, y código abierto que emula el flujo del tráfico de forma continua en el espacio y discreta en el tiempo.
- VanetMobiSim: es un conjunto de extensiones de CanuMobiSim35, un framework usado para la simulación genérica de movilidad.
- JFreeChart: es un paquete de Java que permite la creación de tablas, gráficas, histogramas, entre otros, de forma simple y organizada. Tiene gran variedad de gráficas y opciones que permiten personalizar muchas características de las gráficas.

10.3 Prototipo General de la Interfaz

Para el desarrollo del conjunto de benchmarks no existe un prototipo de interfaz estándar a desarrollar debido a que la interfaz con la cual el usuario interactuará viene dada por el simulador en cuestión. Por lo tanto no fue desarrollado ningún prototipo de interfaz pero si fue definida la interacción o configuración de las simulaciones mediante el uso de archivos de configuración.

Para el desarrollo de la herramienta cuya función es analizar las trazas de movilidad que se generen, se toma en cuenta la capacidad que posee el lenguaje de programación Java para el desarrollo de interfaces. La herramienta se llamará ns-2 Trace Toolkit y permitirá al usuario una gran cantidad de opciones que permita validar la correctitud de una traza generada.

A continuación se presentan las características de las interfaces provista por los simuladores para la interacción con el usuario y el prototipo de interfaz para la herramienta ns-2 Trace Toolkit:

10.3.1 Interfaz de OMNeT++

Para visualizar la simulación, OMNeT++ provee una interfaz gráfica de usuario como también una interfaz de línea de comando. Para la configuración de la simulación, OMNeT++ provee los archivos *omnetpp.ini* y los archivos *.ned* los cuales son utilizados para asignar valores a los parámetros de la simulación y de esta manera personalizar el comportamiento de los módulos simples y la topología.

Para el conjunto de benchmarks la idea es definir los parámetros con valores por defecto en los archivos *.ned* y definir los parámetros con valores a personalizar en los archivos *omnetpp.ini*. De esta manera se busca que el usuario configure la simulación mediante el uso de solamente los archivos *omnetpp.ini*.

10.3.2 Interfaz de JiST/SWANS

Para ejecutar una simulación, JiST/SWANS solo provee interfaz de línea de comando, no posee interfaz gráfica. JiST/SWANS tampoco provee de un sistema de archivos de configuración para personalizar la simulación, por esta razón fue definido el archivo *Parameters.java* el cual tendrá la misma funcionalidad que los archivos *omnetpp.ini* y *.ned* de OMNeT++.

10.3.3 Interfaz de ns-3

Para la ejecución y visualización de la simulación en ns-3 se cuenta con la interfaz de línea de comando y con un módulo (NetAnim) que permite visualizar de manera gráfica la reproducción de la simulación. Para la configuración de los parámetros de los benchmarks se usa un archivo *.h* (*CircleParameters.h* para el benchmark circular y *CityParameters.h* para el benchmark de la ciudad)

[35] http://vanet.eurecom.fr

que contiene todos aquellos parámetros que el usuario puede personalizar para la simulación, siguiendo la lógica de lo que se realizó en JiST/SWANS.

10.3.4 Interfaz de ns-2 Trace Toolkit

Es primordial establecer un estándar en la interfaz gráfica entre los diferentes módulos que poseerá la herramienta para asegurar la usabilidad, eficiencia y robustez de la aplicación, manteniendo así la consistencia. Por eso, se diseñó un prototipo de interfaz general que permita cumplir con los requerimientos necesarios de la aplicación.

El lenguaje de programación Java provee una facilidad para crear interfaces gráficas mediante el uso de la clase JFrame o de la clase JPanel. También ofrece una alta portabilidad, asegurando que la aplicación funcione en cualquier sistema operativo. Con el uso de IDEs como JCreator o NetBeans se aumenta la facilidad de la generación de interfaces a través de Java. Se estableció para todas las ventanas desarrolladas que el lenguaje nativo de la aplicación sea inglés y que el *look & feel* sea el propio del sistema operativo donde la herramienta se está ejecutando.

En la Figura 10.1 se muestra la ventana principal de la aplicación (llamada Home) donde se especificarán las opciones iniciales para la carga de la traza. La ventana será desplegada a través de un JFrame con dimensiones de 300 píxeles de alto por 400 píxeles de ancho. Esta ventana desplegará un explorador de archivos a través de la clase JFileChooser para buscar y cargar la traza a analizar.

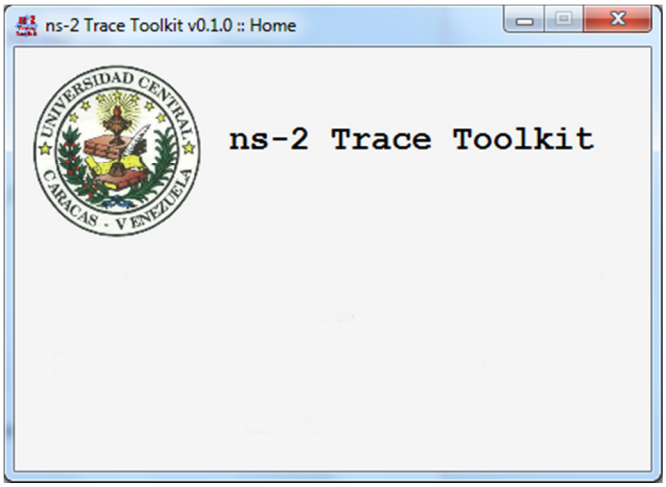


Figura 10.1: Prototipo de Interfaz de la Ventana Principal de la Herramienta ns-2 Trace Toolkit

Una vez cargada la traza, la ventana principal debe ofrecer una opción que permita ir a una ventana donde se pueda analizar la traza cargada. La Figura 10.2 muestra el prototipo de la ventana de análisis para la traza cargada en la ventana principal. En esta ventana el usuario podrá ver la información asociada a la traza (número de nodos, límites, velocidades reportadas, etc), además de poder hacer un análisis más profundo mediante gráficas usando JFreeChart. Adicionalmente, se tendrán opciones para la generación de subconjuntos de la traza general según las opciones que se especifiquen por la interfaz. Se podrá regresar a la ventana principal mediante un botón de retorno.

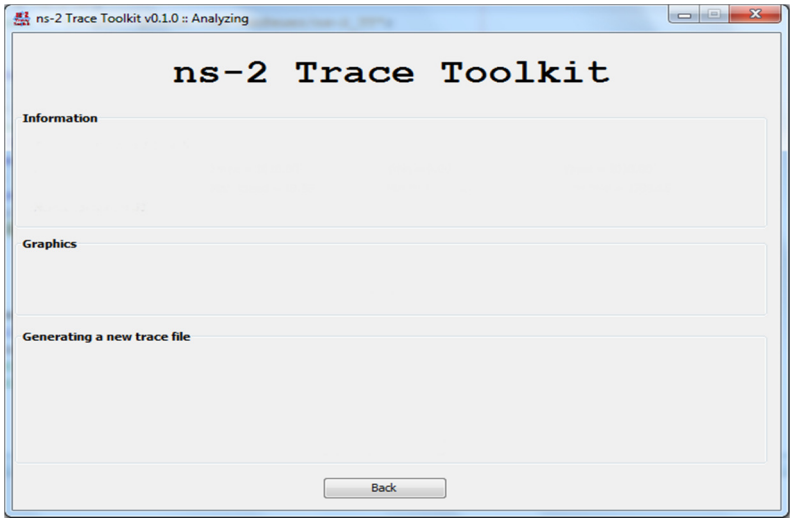


Figura 10.2: Prototipo de Interfaz de la Ventana de Análisis de la Herramienta ns-2 Trace Toolkit

Por último, se debe poder observar una simulación de la traza analizada y así poder observar los movimientos de los nodos. Para eso se diseñó un prototipo para esta ventana como se puede apreciar en la Figura 10.3, donde en la parte superior se ubicarán todas las opciones posibles que se tendrán para ejecutar la simulación (botón de inicio, botón de parada, entre otros). En el Panel más grande es donde se desplegará la simulación, donde cada nodo será dibujado como un punto con su respectivo identificador asociado. Por último, en el panel menor ubicado a la derecha se colocará una tabla con la información actual de cada uno de los nodos para que el usuario tenga un *feedback* de lo que está sucediendo en la simulación por cada nodo.

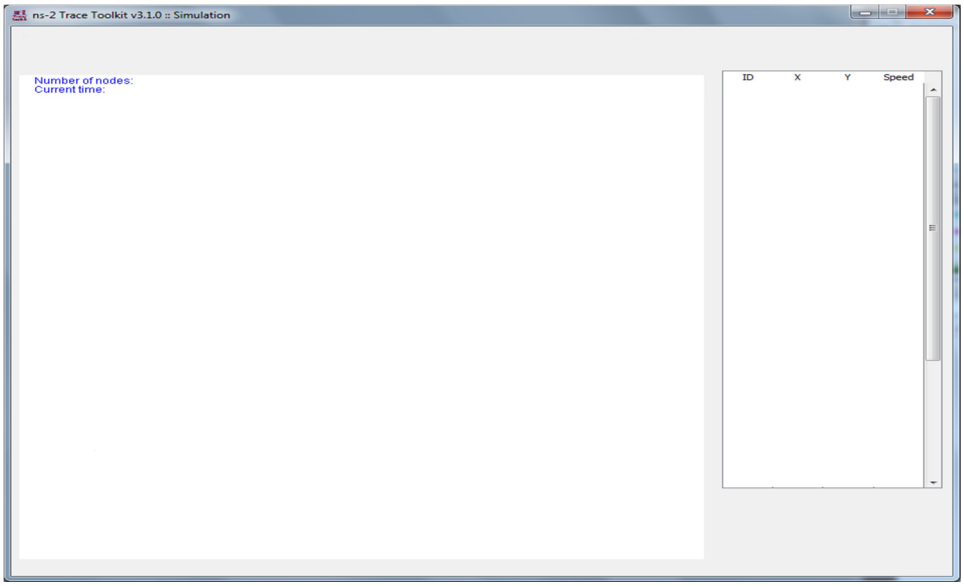


Figura 10.3: Prototipo de Interfaz de la Ventana de Simulación de la Herramienta ns-2 Trace Toolkit

11. Marco Aplicativo

En este capítulo se explica el diseño de la solución a través de las iteraciones definidas en base a los lineamientos de la metodología XP, descrita en el Capítulo 10. Cada iteración se define en base al desarrollo de las fases de Diseño, Codificación y Pruebas.

11.1 Análisis General

Antes de iniciar el desarrollo del conjunto de benchmarks, se llevó a cabo una fase de análisis general donde se determinó de manera global los principales requerimientos para lograr la construcción del conjunto de benchmarks. Estos requerimientos deben cubrir los objetivos estipulados en el Capítulo 2 para lograr construir la solución a la problemática planteada.

A continuación, se muestra cómo se estructuró la lista de requerimientos del conjunto de benchmarks:

- Diseñar un benchmark en el cual se logre simular una red vehicular donde el movimiento de los vehículos esté confinado a una carretera circular.
- Diseñar un benchmark en el cual la topología sea más realista y se fundamente en mapas de carreteras reales donde el movimiento de los nodos corresponda con la red de calles definida.
- Reportar estadísticas para cada benchmark acerca del comportamiento de la red (número de paquetes enviados y recibidos, delay, etc.).
- Implementar los benchmarks planteados en OMNeT++.
- Implementar los benchmarks planteados en ns-3.
- Implementar los benchmarks planteados en JiST/SWANS.
- Reportar estadísticas acerca del consumo de CPU, de la memoria y del tiempo de ejecución relacionados por la simulación.
- Diseñar y crear scripts para lograr la ejecución de la simulación en modo batch.
- Generar reportes o archivos de salida donde se almacene toda la información y los resultados obtenidos de la simulación.
- Generar trazas ns-2 adecuadas con las herramientas SUMO y VanetMobiSim para los escenarios planteados en el benchmark realista.
- Diseñar e implementar una herramienta adecuada para el manejo y depuración de trazas ns-2.
- Obtener y adecuar mapas de ciudades (nacionales o internacionales) de manera que puedan ser utilizados como escenarios para la simulación de redes vehiculares.

11.1.1 Desarrollo de la Aplicación

El desarrollo de la aplicación consta de tres partes. La primera parte se basa en el diseño y desarrollo de los benchmarks, descrito en las iteraciones 1 y 2. La segunda parte se fundamenta en el manejo de los simuladores de red y la implementación de los benchmarks en cada uno de ellos, descrita en las iteraciones 3, 4 y 5. Por último, la tercera parte trata la herramienta ns-2 Trace Toolkit desarrollada para visualizar y depurar trazas de movilidad ns-2, se describe en la iteración 6.

A continuación se especifican cada una de las iteraciones que fueron necesarias y las fases implementadas en ellas según la metodología de desarrollo ágil XP.

11.1.2 Iteración 1: Diseño y Planificación del Circle Benchmark

- **Fase de Diseño**: Muchos simuladores de red tienen modelos de movilidad ya implementados pero la mayoría de ellos no ofrecen lo que se desea para los benchmarks. La idea es tener una movilidad sencilla que se pueda programar y que no dependa de una traza de movilidad. La movilidad escogida debe permitir a los nodos moverse de una forma coherente apegada a la realidad y reproducir simulaciones idénticas cuando se requiera para así evaluar la correctitud del benchmark según la aplicación y el número de nodos que se estén simulando. Algunos tipos de movilidad ofrecen esas características pero la movilidad circular (tipo redoma) fue la que se decidió tomar para realizar un benchmark sin el uso de trazas de movilidad ya que es sencilla de implementar y no tiene problemas para aumentar la velocidad.

  La movilidad circular se encuentra implementada en la mayoría de los simuladores actuales, y de no ser así, no es muy complicado implementarla. Hay una serie de parámetros que pueden ser modificados antes de la ejecución de la simulación y así obtener casos de pruebas distintos. Esos parámetros son el número total de vehículos, el radio del círculo y la separación entre los vehículos en segundos. Estos parámetros son tomados en cuenta según el modo de operación escogido, existen 3 modos de operación: (1) para el primer modo se tiene que especificar el número total de vehículos que se desean en la simulación y la separación entre ellos. Entonces el programa calcula el radio del circulo necesario para soportar el número de vehículos dado con la separación entre ellos especificada. (2) Para el segundo modo se debe indicar la distancia entre los vehículos y el radio del círculo para que el programa calcule el número de vehículos totales bajo esos parámetros. Y por último, (3) para el tercer modo el usuario debe especificar el número total de vehículos y el radio del círculo, siendo no significativa la separación entre los vehículos.

  En resumen se tiene que la topología circular puede contener más de un canal con un ancho dado llamado *laneWidth* y cada canal soporta una velocidad indicada por parámetro. Los canales pueden tener sentidos distintos, esto se logra a través de las velocidades, si la velocidad es positiva los vehículos circulan en sentido horario y si es negativa irán en sentido anti-horario. De los simuladores usados, sólo OMNeT++ dispone de un modelo de movilidad circular. Para ns-3 y JiST/SWANS se diseñó el modelo de movilidad circular siguiendo la lógica de OMNeT++. En la Figura 11.1, se observa la topología circular para los benchmarks.

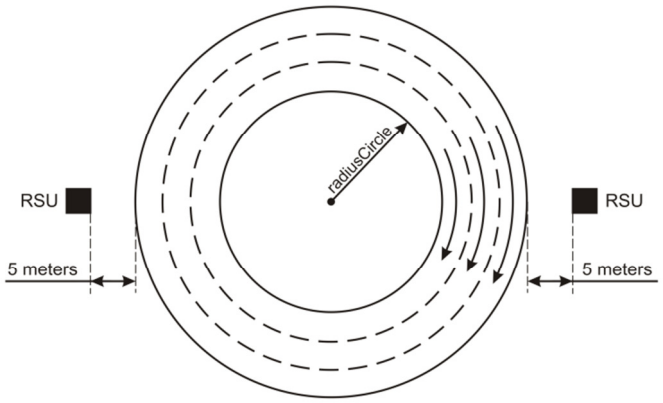


Figura 11.1: Topología Circular

- **Fase de Codificación**: En esta fase se implementó el diseño propuesto en cada uno de los simuladores obteniendo lo siguiente:

  - OMNeT++: El modelo de movilidad utilizado se denomina *CircleMobility*. Es un modelo ya definido en el simulador listo para usarse. Este modelo solo fue instalado para los vehículos ya que son los únicos nodos que se mueven en la simulación. Para los RSUs se utilizó el modelo *StationaryMobility* que permite mantener a los nodos fijos en la posición inicial dada. Los RSUs se ubican fuera del último canal a una distancia especificada como parámetro. Los vehículos se irán ubicando según sea el modo utilizado en el benchmark y la cantidad de canales especificada. En la Figura 11.2, se muestra la definición de la movilidad para los vehículos.

```
module MyWirelessCar extends Vehicle
{
parameters:
    IPForward       = default(true);
    mobilityType    = "CircleMobility";
    mobility.speed = default(0.0 mps);
    mobility.cx     = default(0.0);     // Center X of the circle.
    mobility.cy     = default(0.0);     // Center Y of the circle.
    mobility.r      = default(0.0);     // Radius of the circle.
}
```

Figura 11.2: Definición de la Movilidad de los Vehículos para OMNeT++

  - ns-3: Para este simulador se tuvo que definir la movilidad de los vehículos y RSUs. A pesar que los RSUs no se mueven, ns-3 no define un *Allocator* que ubique los nodos en forma circular. Para la definición del modelo de movilidad se tiene que calcular la coordenada polar de cada nodo, conservando la separación dada por parámetros entre los nodos. En la clase *CircleBenchmark* el método *CreateAllNodes* crea los nodos y los ubica en su posición inicial. Cuando la aplicación se instala en *InstallApplications*, a cada vehículo se le coloca un evento futuro que le permitirá avanzar un paso siguiendo el patrón circular. Dicho evento es ejecutado mediante el método *AdvancePosition* que calculará la nueva posición para el nodo y colocará como evento futuro su próximo movimiento. En la Figura 11.3 se puede apreciar parte de la movilidad implementada en ns-3.

  - JiST/SWANS: Al igual que ns-3, JiST/SWANS no dispone de un modelo de movilidad circular para los nodos. Es por ello que en este simulador se creó la clase *CircleMobility* que hereda de *Mobility* donde se define toda la lógica para la movilidad circular. Para el desarrollo de este modelo se siguió el mismo esquema que se tiene en OMNeT++. La clase tiene como atributos el centro del círculo y la velocidad con que se moverán los vehículos. Mediante el método *next* se hace un nuevo movimiento siguiendo el patrón circular. De esta manera, cuando los vehículos se están configurando solo se debe instalar está movilidad. En la Figura 11.4, se puede observar una parte del código desarrollado para el modelo de movilidad circular.

```
double cx, cy, rf, angle;
cx = cy = border + numlanes*laneWidth + radiusCIrcle

// Calculates all initial positions for cars
for(int lane = 0; lane < numLanes; lane++)
{
    rf = radiusCircle + laneWidth*(0.5 + double(lane));

    for(int i = 0; i < numCarsLane[lane]; i++)
    {
        angle = (2*PI/numCarsLane[lane])*i;
        positionAllocCar->Add(Vector(cx + rf*cos(angle),
                                     cy + rf*sen(angle), 0.0))
    }
}

// Installs the mobility model for cars
mobilityCar.SetPositionAllocator(positionAllocCar);
mobilityCar.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobilityCar.Install(cars);
```

Figura 11.3: Parte del Código donde se Asigna el Modelo de Movilidad en ns-3

```
public void next(FieldInterface f, Integer id, Location loc, Mobility info)
{
    CircleMobilityInfo nodeInfo = (CircleMobilityInfo) info;
    double radiusCircle = nodeInfo.radiusCircle;

    nodeInfo.angle += nodeInfo.deltaAngle;
    if(nodeInfo.angle >= 360.0)
        nodeInfo.angle -= 360.0;

    double posX = cx + radiusCircle*Math.cos(nodeInfo.angle/360.0*2.0*Math.PI);
    double posY = cy + radiusCircle*Math.sin(nodeInfo.angle/360.0*2.0*Math.PI);

    JistAPI.sleep((long)(updateInterval*Constants.SECOND));
    f.moveRadio(id, new Location2D((float) posX, (float) posY));
}
```

Figura 11.4: Sección de Código del Modelo de Movilidad en JiST/SWANS

- **Fase de Pruebas**: Para probar la movilidad circular en cada simulador se tomó en cuenta algunas ventajas inherentes que tiene cada uno. En OMNeT++ no es tan complicado probar y depurar los modelos de movilidad gracias a la potente interfaz gráfica que ofrece. ns-3 no ofrece una interfaz gráfica como OMNeT++ pero mediante la integración de NetAnim se pudo verificar la correctitud del modelo de movilidad circular implementado en este simulador. También se procedió a hacer capturas de trazas de la movilidad de los nodos y verificando las posiciones en un archivo plano de texto. Para JiST/SWANS, no quedo más que hacer un análisis de las posiciones vía la consola de comandos o la impresión de las posiciones en un archivo de texto plano. Para todos los simuladores se hicieron diversas pruebas, variando el número de canales, el radio o el modo.

11.1.3 Iteración 2: Diseño y Planificación del City Benchmark

- **Fase de Diseño**: Uno de los tópicos más importantes en la simulación de redes vehiculares es la movilidad de los nodos. Es importante la utilización de un modelo de movilidad realista

para que los resultados de la simulación reflejen el comportamiento real de una red vehicular. Por ejemplo, la movilidad de un vehículo está típicamente limitada a las calles que están separadas por edificios, árboles u otros objetos. Un modelo de movilidad realista con suficiente nivel de detalle es crítico para obtener resultados precisos. Son estas las razones que motivaron al diseño y planificación del City Benchmark en donde la movilidad de los nodos simulados se realiza sobre topologías de redes de carreteras reales.

La solución fue diseñada pensando en utilizar mecanismos sencillos que permitan obtener y reproducir la movilidad de los nodos sobre la red de carreteras deseada sin que sea necesario hacer modificaciones significativas en la aplicación entre un simulador y otro. Por esta razón, es necesario hacer uso de los modelos de movilidad que los simuladores proveen y se utilizan para modelar escenarios de este estilo. Estos modelos de movilidad tienen en común el uso de archivos de trazas de movilidad ns-2 los cuales describen el movimiento de los nodos en espacio y tiempo haciendo referencia también a la velocidad con la cual se desplazan. Utilizar estos archivos de trazas de movilidad garantiza la posibilidad de crear simulaciones de redes vehiculares obteniendo una movilidad bastante realista y brindando la capacidad de reproducir el mismo experimento varias veces y bajo las mismas condiciones independientemente del simulador que se utilice. Todas estas características hacen de este diseño de la solución adecuado ya que se logra cubrir importantes requisitos de la aplicación.

De manera más esquematizada, la solución consta de dos partes fundamentales. La primera parte trata la utilización de los simuladores de tráfico vehicular SUMO y VanetMobiSim, necesarios para la creación de los archivos de trazas de movilidad ns-2. La importancia que tienen estos simuladores de tráfico es que son capaces de importar mapas digitales (OSM para SUMO y TIGER/Line para VanetMobiSim) con lo cual se pueden crear simulaciones de tráfico vehicular sobre escenarios reales. Cada uno de estos simuladores de tráfico tienen sus características y limitaciones, ambos tienen la capacidad de generar trazas de movilidad adecuadas a lo que se requiera, por esta razón para generar trazas de movilidad se utilizaron ambas herramientas aprovechando así las ventajas que cada una ofrece. La segunda parte de la solución consiste en la utilización de los modelos de movilidad propios de cada simulador de red para importar archivos de trazas de movilidad ns-2. Una vez que se obtiene una traza adecuada de movilidad, es importante que ésta sea leída de manera correcta por el simulador y que el movimiento de los nodos en la simulación corresponda con lo que sugiere el archivo de traza de movilidad ns-2. En la fase de codificación se explicará de manera detallada la forma que estos archivos son procesados por los distintos simuladores de red en cuestión.

- **Fase de Codificación**: En esta fase es donde se codifica el diseño de la solución planteado para simular redes vehiculares sobre escenarios realistas mediante el uso de archivos de trazas de movilidad ns-2. Para lograr esto, es importante conocer la manera en la cual cada simulador interactúa con los archivos de trazas. De esta manera se tiene lo siguiente:

    - OMNeT++: El modelo de movilidad utilizado en este simulador para simular escenarios realistas se denomina *Ns2MotionMobility*. Para hacer uso de este modelo es necesario instalarlo como un componente de los vehículos. Los vehículos se declaran como un módulo compuesto denominado *MyWirelessCar* en el archivo MyNetwork.ned donde uno de los parámetros a especificar es el modelo de movilidad. El siguiente ejemplo (
    - Figura 11.1) muestra como se le indica a los vehículos que usen el modelo de movilidad *Ns2MotionMobility* y también muestra la especificación de la ruta donde se encuentra el archivo de trazas de movilidad ns-2.

```
module MyWirelessCar extends Vehicle
{
parameters:
    mobilityType            = "Ns2MotionMobility";
    mobility.updateInterval = 1s;
    mobility.traceFile      = default("traces/mobility.tcl");
    mobility.nodeId         = default(1);
}
```

Figura 11.1: Movilidad para el City Benchmark en OMNeT++

- ns-3: Este simulador cuenta con una clase denominada *Ns2MobilityHelper* específica para leer archivos de trazas de movilidad ns-2 y configurar la movilidad de los nodos. La utilización de esta clase es sumamente sencilla, como se muestra en la Figura 11.2, solamente es necesario declararla e instanciarla pasando como parámetro un String con la ruta donde se encuentra almacenado el archivo de trazas de movilidad ns-2. Una vez instanciada la clase *helper* y que los nodos ya fueron creados y configurados, se procede a instalar la movilidad mediante el método *Install* provisto por *Ns2MobilityHelper*.

```
// Set the ns-2 mobility model
Ns2MobilityHelper ns2 = Ns2MobilityHelper(tracefile);

/* ... Configure all nodes for simulation ... */

// Install the mobility into the nodes
ns2.Install();
```

Figura 11.2: Movilidad para el City Benchmark en ns-3

- JiST/SWANS: Cuenta con el modelo de movilidad *MobilityReplay* específico para interactuar con archivos de trazas de movilidad ns-2. Como se muestra en la Figura 11.3, es necesario declarar una variable de tipo *Mobility* la cual será instanciada con el modelo de movilidad *MobilityReplay*. El primer parámetro del constructor especifica los límites del área en la cual los nodos se van a desplazar, si se le asigna *null*, los límites serán obtenidos según lo que indique el archivo de trazas de movilidad ns-2. El segundo parámetro funciona para indicar la precisión requerida (en metros) con la cual el modelo de movilidad irá realizando un paso cada tantos metros se hayan especificado. El tercer parámetro indica la ruta donde se encuentra almacenado el archivo de trazas de movilidad ns-2. Por último, se especifica el nombre de la clase deseada encargada de leer el archivo de trazas de movilidad ns-2, en este caso la clase *MobilityReaderNs2*.

```
Mobility mobility = new MobilityReplay(null, 10, traceFile,
                                       MobilityReaderNs2.class.getName());
```

Figura 11.3: Movilidad para el City Benchmark en JiST/SWANS

- **Fase de Pruebas**: En esta iteración las pruebas se dividen en dos partes:
  - La primera parte consiste en la generación adecuada de trazas de movilidad ns-2, mediante un proceso de ensayo y error en el cual se tendrá que validar la correcta lectura y representación de los mapas en los simuladores de tráfico vehicular. Se tiene

que garantizar que las trazas generadas contengan la mayor cantidad de nodos en movimiento y evitar en lo posible tener nodos detenidos gran parte del tiempo de la simulación. Por último, se comprueba que los archivos de trazas de movilidad ns-2 se hayan generado de forma correcta y que puedan ser utilizados por los modelos de movilidad de los simuladores de red.

- La segunda parte de las pruebas tiene que ver con la comparación entre la movilidad descrita por los archivos de trazas y la movilidad reproducida en los nodos simulados. Es importante verificar que existe una correspondencia entre el movimiento que realizan los nodos simulados y el movimiento que indica el archivo de trazas de movilidad ns-2. Para lograr esto, se puede evaluar que la posición de los nodos simulados en el tiempo corresponda con lo que indica el archivo de trazas. No es necesario visualizar la simulación, solo con ir visualizando mensajes con la posición actual de los nodos se puede realizar una comparación con la información contenida en el archivo de traza.

11.1.4 Iteración 3: Implementación en OMNeT++

- **Fase de Diseño**: Esta aplicación está formada por un conjunto de benchmarks (descritos en las iteraciones 1 y 2) para evaluar la escalabilidad del simulador OMNeT++ y su framework INETMANET 2.0 simulando redes vehiculares. Para el diseño de la solución en OMNeT++, es necesario crearla en base a la metodología o flujo de trabajo propio de este simulador. Por esta razón, el diseño de la solución es basado en las siguientes actividades:
  - Un modelo en OMNeT++ es construido partiendo de componentes (módulos) las cuales se comunican mediante el intercambio de mensajes. Para simular redes vehiculares es necesaria la construcción de un modelo formado por módulos que contengan la infraestructura y los protocolos adecuados para crear vehículos y RSUs que sean capaces de crear una VANET. En este tópico se identifican las componentes requeridas por el modelo.
  - Para definir la estructura del modelo de simulación se utiliza el lenguaje NED con el cual se declaran módulos simples y se ensamblan y conectan éstos en módulos compuestos. OMNeT++ permite crear estos archivos mediante un editor de texto o por medio del editor gráfico basado en Eclipse del IDE de OMNeT++. En la
  - Figura 11.4, se puede apreciar la definición de la estructura de un módulo compuesto con el editor gráfico.

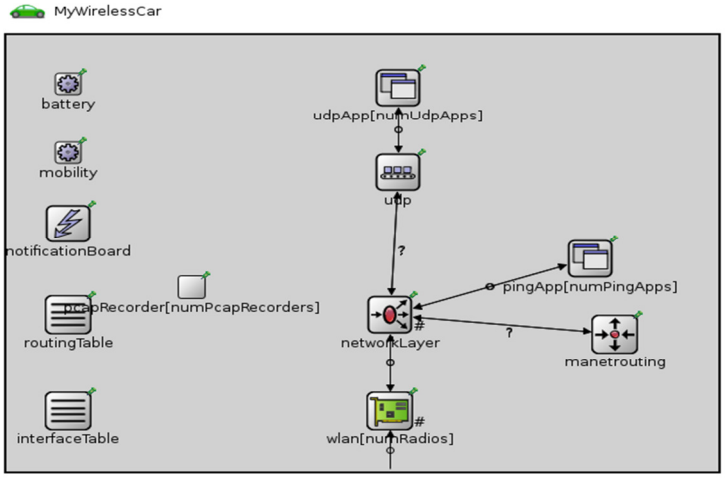


Figura 11.4: Definición de la Estructura de un Vehículo en el Lenguaje NED

- La programación en C++ es otra de las fases importantes ya que a través de ella se pueden construir simulaciones más dinámicas y adaptadas a la problemática en cuestión. También representan la forma con la cual se crean los componentes activos de los modelos (módulos simples) haciendo uso del kernel de simulación y bibliotecas de clases.
- Otro de los aspectos relevantes de este simulador es la utilización de los archivos con extensión .ini con los cuales se configura y parametriza la simulación. Adicionalmente, los archivos .ini son el mecanismo que permite también definir ejecuciones por lote. Mediante el uso de los archivos .ini, se establecen cuáles son los valores de los parámetros de la simulación.
- Por último, este simulador brinda la posibilidad de visualizar la simulación mediante una interfaz gráfica o por interfaz de línea de comando. Ambas interfaces pueden ser utilizadas para visualizar la ejecución de los benchmarks. Los resultados obtenidos son reportados por pantalla y almacenados en un archivo de salida.

- **Fase de Codificación**: La codificación de los benchmarks en OMNeT++ se llevó a cabo en 3 partes.

  La primera parte consiste en la definición de la estructura del modelo, es decir la creación del archivo MyNetwork.ned donde son definidas las componentes a utilizar. Entre los módulos definidos se tiene el modelo completo denominado MyNetwork el cual contiene a toda la simulación de la VANET, el módulo MyWirelessCar que modela a los vehículos y el módulo MyWirelessRSU que modela a los dispositivos RSU. En la Figura 11.5 se puede observar un fragmento del archivo MyNetwork.ned.

```
package inet.benchmarks.circleBenchmark;

import inet.world.radio.ChannelControl;
import inet.networklayer.IManetRouting;
import inet.benchmarks.manet.Vehicle;
import inet.benchmarks.manet.RSU;
import inet.benchmarks.circleBenchmark.CircleController;

module MyWirelessCar extends Vehicle
{
parameters:
   IPForward = default(true);
   mobilityType="CircleMobility";
   mobility.speed = default(0.0 mps);  // Will be overwritten
   mobility.cx=default(0.0); // Center X of the circle. Will be overwritten
   mobility.cy=default(0.0); // Center Y of the circle. Will be overwritten
   mobility.r=default(0.0);  // Radius of the circle. Will be overwritten
   numUdpApps= default(1);
   udpApp[*].typename = "inet.benchmarks.manet.UDPForCarApp";
   udpApp[*].localPort = 3000;
   udpApp[*].destPort = 3000;
   udpApp[*].messageLength = 500B;   // UDP payload lenght. Will be overwritten
   udpApp[*].sendInterval = default(0.2s);
   // Frequence of message (seconds) for vehicles. Will be overwritten
   udpApp[*].timeToLive = 64;
   udpApp[*].startTime = 1s;
```

Figura 11.5: Segmento del Archivo MyNetwork.ned

La segunda parte consiste en la creación del archivo omnetpp.ini en el cual se especifican los valores de los parámetros de entrada con los cuales se configura la simulación. Entre estos parámetros están el tiempo de simulación, número de vehículos, protocolo de enrutamiento a utilizar, la frecuencia de envío de mensajes, etc. En la Figura 11.6 se muestra un fragmento del archivo omnetpp.ini.

```
sim-time-limit=120.0 s          # Duration of simulation (simulation time)

*.totalRSUs = 1                 # Number of Road Side Units (RSUs)
*.streetRSUDistance = 5.0 m     # Distance between the streets and the RSUs
*.laneWidth = 2.6 m             # Lane width (meters)
*.numLanes = 3                  # Number of lanes
*.border = 30.0 m               # Space around the rectangular road in GUI
*.speed = "30.0 30.0 30.0"      # Speed of each lane (km/h). Set negatives
*.carDistance = 2.0 s           # Distance between cars (seconds).
*.routingProtocol= "AODVUU"     # Routing protocol "AODVUU","DYMOUM", "DYMO",
                                # "DSRUU","OLSR","OLSR_ETX","DSDV_2","Batman"
```

Figura 11.6: Segmento del Archivo omnetpp.ini

La tercera parte consiste en la implementación de los benchmarks en C++. Entre los principales métodos utilizados están:

- initialize(): El flujo de la simulación inicia mediante la invocación de este método que es llamado luego de que OMNeT++ ha configurado la red (creado los módulos y conectados de acuerdo a lo que indica el archivo MyNetwork.ned). Este método se utiliza principalmente para configurar la simulación obteniendo y validando los parámetros de entrada establecidos en el archivo omnetpp.ini.
- activity(): Una vez ejecutado el método initialize(), el control de la simulación se realiza a través del método activity() que inicia como una corutina al comienzo de la simulación y continua ejecutándose hasta el final de la misma. En este método es donde se crean los vehículos y dispositivos RSUs simulados, se les configura el modelo de movilidad y son inicializadas y configuradas las capas física, enlace, red, transporte y aplicación con los parámetros obtenidos con el método initialize().
- finish(): Este método es llamado cuando la simulación ha terminado de manera exitosa. Aquí es donde se ejecuta el código para reportar y guardar los resultados finales mostrándolos por pantalla y almacenándolos en un archivo de salida. En este método son reportados los parámetros con los cuales se configuró la simulación, estadísticas del comportamiento de la red y el consumo de CPU, memoria y tiempo de ejecución de la simulación.

Adicionalmente, fue necesaria la creación de un modelo propio para simular una aplicación UDP en la cual se envían datagramas UDP a destinos de forma aleatoria. Se crearon las aplicaciones *UDPforCarApp* para los vehículos y *UDPforRSUApp* para los RSUs. En *UDPforCarApp* los vehículos envían datagramas UDP con una cierta frecuencia, el destino se escoge al azar primero entre la probabilidad de enviarlo a otro vehículo o a un RSU y luego se escoge al vehículo o RSU en cuestión. Para *UPDforRSUApp* no existe la posibilidad de envío entre RSUs, por lo tanto cada vez que un RSU desea enviar un datagrama UDP se escoge un vehículo destino de forma aleatoria.

También fue necesaria la creación del modelo MyIPv4 para contabilizar el número de mensajes enviados por el protocolo de enrutamiento ya que el modelo IPv4 original de OMNeT++ no realiza dicha función.

- **Fase de Pruebas**: Las pruebas unitarias para evaluar el correcto funcionamiento de los benchmarks en OMNeT++ consistieron en la ejecución de simulaciones con pocas cantidades de nodos. Gracias a la poderosa e interactiva interfaz gráfica de OMNeT++ fue posible verificar que tanto los vehículos como los RSUs tenían todos sus modelos inicializados y configurados con los parámetros establecidos. Visualizar la simulación también hace posible verificar y validar la correcta movilidad de los nodos. Reportar los resultados finales da indicios de lo que ha sucedido durante la simulación y ayuda a la depuración de errores.

11.1.5 Iteración 4: Implementación en ns-3

- **Fase de Diseño**: Esta aplicación está conformada por un conjunto de benchmarks (descritos en las iteraciones 1 y 2) para evaluar la escalabilidad del simulador ns-3 en el área de redes vehiculares. Para el diseño de la solución se siguió un esquema orientado a objetos para crear el flujo de la red y sus componentes. Se realizaron las siguientes actividades para cumplir la meta planteada:
    - Para definir la estructura de la red y sus componentes es necesario modularizar varias partes de la red para tener control de la simulación. Para ello se define la estructura que deben poseer tanto los vehículos como los RSUs para así poder establecer comunicación entre ellos y establecer la VANET.
    - Un aspecto importante es la parametrización de la simulación, para eso se estableció un estándar de qué variables se pueden modificar en cada simulación y así obtener casos de prueba diferentes. Tanto en ns-3 como en OMNeT++ y JiST/SWANS se sigue este esquema.
    - El simulador no ofrece una interfaz gráfica integrada y potente a diferencia de OMNeT++, aun así se puede utilizar NetAnim para la verificación del diseño planteado.
    - Para definir una aplicación idónea para las redes vehiculares, se esquematizó una aplicación UDP. Debe ser una aplicación que no congestione la red y se asegure de que los mensajes puedan saltar entre nodos intermedios hasta llegar a su destino.

- **Fase de Codificación**: La codificación de los benchmarks en ns-3 se llevó a cabo en cuatro partes.

  La primera parte consiste en la definición de la red y cómo debe estar estructurada, es decir, los protocolos a usar y estructuras de datos necesarias para el funcionamiento de la red. Para ello en cada benchmark se definieron una serie de primitivas que configuran la red según los parámetros dados. Un parámetro vital es el alcance de propagación de los nodos, tanto de los vehículos como de los RSUs. Para ello se escogió el Free Space Propagation Loss Model y en cada simulación se configura dicho modelo en función de lo especificado por el usuario.

  La segunda parte consiste en la definición del archivo que contiene todos los parámetros de la red. Este archivo sigue un estándar establecido para el desarrollo de estos benchmarks. En OMNeT++ y JiST/SWANS se tiene estructuras de archivos de entradas muy parecidos a los de ns-3. Entre los parámetros se tienen el tiempo de la simulación, el número de vehículos, el tipo de estándar WiFi a utilizar, el protocolo de enrutamiento para la simulación, entre otros. En la
  Figura 11.5 se muestra una porción del archivo de configuración para ns-3.

```
// mode = 3 You must give totalCars and radiusCircle.
// Value of carDistance is not significative. The
// distance between cars (in meters) will be more or
// less the same.
static int mode = 3;

// Duration of simulation (simulation time) (seconds)
double simTimeLimit = 120.0;

// Interval between of node position update (seconds)
double updateInterval = 0.25;

// Lane width (meters)
double laneWidth = 2.6;
```

Figura 11.5: Parte del Archivo de Configuración para ns-3

La tercera parte consiste en la implementación total de los benchmarks en ns-3 bajo el lenguaje de programación C++. Como todo programa en C++, el flujo de la simulación comienza en la función principal o main(). Los archivos CircleBenchmark.cc y CityBenchmark.cc son quienes poseen el método main. Se crearon dos clases que definen el city benchmark y el circle benchmark respectivamente. En estas clases se ubican una serie de métodos comunes que determinan la lógica del benchmark:

- CreateAllNodes(): En este método se definen y se crean todos los nodos que van a participar en la simulación, instalando la movilidad y la ubicación inicial de los nodos.
- CreateAllDevices(): Una vez que los nodos son creados, este método se encarga de la instalación de la capa física y MAC en todos los nodos. En este método se especifican el modelo de propagación y el estándar WiFi.
- InstallInternetStack(): En este método se instala el stack TCP/IP en los nodos. También se define en este paso el algoritmo de enrutamiento a usar.
- InstallApplications(): Una vez se haya instalado el stack de Internet en los nodos, este método se encarga de instalar la aplicación, estableciendo el momento que comenzará y finalizará su ejecución.
- ReportResults(): Este método es invocado al final de cada simulación. Imprime los resultados obtenidos de la simulación, almacenándolos en un archivo de texto de salida y mostrándolos también por la línea de comandos.

La última parte consistió en desarrollar una aplicación común entre los nodos. Dicha implementación fue hecha en los archivos UdpForCar.h, UdpForCar.cc, UdpForRsu.h y UdpForRsu.cc donde se definen las clases con sus métodos y atributos necesarios para el funcionamiento de la aplicación. La aplicación no es más que una conexión vía Socket UDP provisto ya por ns-3 y siguiendo la misma lógica y funcionamiento de la aplicación implementada para OMNeT++.

- **Fase de Pruebas**: Las pruebas consistieron en un conjunto de casos sencillos con poca cantidad de nodos y verificando la movilidad mediante NetAnim. También se habilitaron la impresión de trazas ya que ns-3 soporta esta funcionalidad y mediante Wireshark o tcpdump se pudo analizar el correcto funcionamiento de la aplicación.

11.1.6 Iteración 5: Implementación en JiST/SWANS

- **Fase de Diseño**: Esta aplicación está formada por un conjunto de benchmarks (descritos en las iteraciones 1 y 2) para evaluar la escalabilidad del simulador JiST y su framework SWANS simulando redes vehiculares. El diseño de la solución en JiST/SWANS resulta ser bastante simple ya que la mayoría de la codificación de los benchmarks se realiza en el lenguaje de programación Java puro. Java como lenguaje orientado a objetos hace que el diseño de la solución esté compuesta por un gran número de clases que colectivamente implementan la lógica del modelo de simulación. Durante la ejecución de la simulación, el estado del programa está contenido enteramente dentro de objetos individuales. Estos objetos se comunican mediante el paso de mensajes representados como llamadas a métodos de estos objetos.

  Para facilitar el diseño de las simulaciones, JiST extiende el modelo tradicional de programación con el concepto de las entidades. En el código del programa, las entidades se definen como instancias de las clases que implementan la interfaz JistAPI.Entity. Aunque las entidades son objetos regulares de la máquina virtual de Java, también permiten encapsular lógicamente objetos de aplicación demarcando componentes de simulación independientes (Figura 4.1).

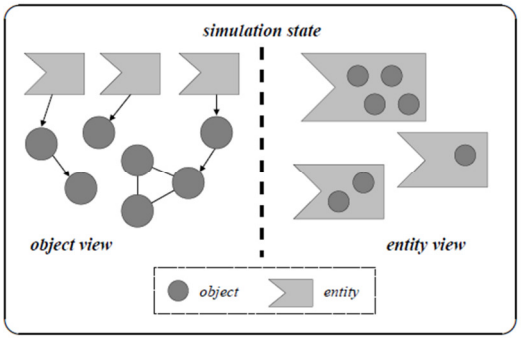


Figura 11.7: División en Entidades Alrededor de los Límites del Objeto

  Debido a estas características del simulador, la solución es desarrollada mediante la codificación del modelo de simulación en Java plano particionando el estado de la simulación no solo en objetos sino también en un conjunto de entidades independientes a lo largo de los límites de la aplicación. De esta manera, las entidades no comparten ningún estado de la aplicación y pueden progresar en forma independiente entre interacciones en el tiempo.

- **Fase de Codificación**: Para lograr la construcción de los benchmarks en JiST/SWANS, el proceso de codificación fue realizado de la siguiente manera:
  - Como cualquier programa escrito en el lenguaje de Java, el flujo de la simulación comienza mediante la ejecución de la función principal o main(). En esta función principal, se realiza la lectura y validación de los parámetros de entrada, se crea e inicializa el modelo de movilidad y la información que es compartida entre los nodos, se instancian los vehículos y RSUs y se muestra los resultados de la simulación una vez que ésta ha terminado.
  - Los parámetros de entradas son especificados en el archivo Parameters.java en el cual se definió una interfaz Java de tipo pública denominada *Parameters* la cual contiene la definición de todos los parámetros de entrada de la simulación. Los valores

especificados en esta interfaz son obtenidos y validados en la función principal o main().

- El código para la creación de los vehículos está contenido en el archivo Cars.java en donde se define a la clase Car implementada mediante la interfaz *JistAPI.Entity*. Esta entidad es la que modela a los vehículos de la red. En esta clase son configuradas las capas física, enlace, red, transporte y aplicación de cada vehículo. El funcionamiento de la aplicación UDP desarrollada en JiST/SWANS para los vehículos es exactamente el mismo que se describió en la implementación de los benchmarks en ns-3 y OMNeT++.
- El código para la creación de los RSUs está contenido en el archivo Rsu.java en donde se define a la clase Rsu implementada mediante la interfaz *JistAPI.Entity*. Esta entidad es la que modela a los RSUs de la red. En esta clase son configuradas las capas física, enlace, red, transporte y aplicación de cada RSU. El funcionamiento de la aplicación UDP desarrollada en JiST/SWANS para los RSUs es exactamente el mismo que se describió en la implementación de los benchmarks en ns-3 y OMNeT++.
- Para el manejo de los resultados fue creada la clase *ShowResult* en el archivo ShowResult.java que contiene toda la codificación para el cálculo, impresión y almacenamiento de los resultados. Esta clase es invocada al finalizar la simulación para que muestre los resultados obtenidos por pantalla y para que almacene todos los parámetros de configuración de la simulación y los resultados obtenidos.

En la Figura 11.8 se muestra un segmento de código del benchmark circular en JiST/SWANS.

```
int indexCar = 0;
for(int lane = 0; lane < numLanes; lane++)
    for(int i = 0; i < numCarsLane[lane]; i++)
    {
        carSpeed[indexCar] = speed[lane];

        Car car = new Car();
        double r = radiusCircle + laneWidth/2.0 + laneWidth*lane;
        double angle = 360.0/numCarsLane[lane]*i;

        car.initCar(indexCar, cx, cy, r, angle, speed[lane],
            moveUpdateInt, messageLengthCar, sendIntervalCar);

        messageSentByCar[indexCar] = 0;
        messageReceivedByCar[indexCar] = 0;
        indexCar++;
    }
messageSentByRSU = new long[totalRSUs];
messageReceivedByRSU = new long[totalRSUs];
```

Figura 11.8: Segmento de Código del Benchmark Circular en JiST/SWANS

- **Fase de Pruebas**: Las pruebas unitarias para evaluar el correcto funcionamiento de los benchmarks en JiST/SWANS consistieron en la ejecución de simulaciones con pocas cantidades de nodos de manera que fuese más fácil controlar y depurar su correcto funcionamiento. JiST/SWANS no posee interfaz gráfica de usuario por lo cual a simple vista no se puede observar lo que sucede en la simulación. Por esta razón se llevó a cabo el control y chequeo de la simulación mediante la impresión de mensajes para la depuración, estos

mensajes pueden especificar, por ejemplo la posición de un nodo en un momento determinado o la recepción o envío de un mensaje por parte de la aplicación UDP.

11.1.7 Iteración 6: ns-2 Trace Toolkit

- **Fase de Diseño**: Existen una gran cantidad de herramientas que brindan la posibilidad de generar archivos de trazas ns-2 a partir de un escenario dado o una parte de un mapa del mundo. Muchos investigadores generan trazas y las aportan a la comunidad, pero no todos dan fiabilidad de que las trazas son correctas o que siguen con algún patrón deseado. Hace falta una herramienta que permita analizar una traza ns-2 generada y verificar la correctitud del archivo de trazas. Son muy escasas las herramientas que permiten hacer un análisis de las trazas en ns-2 y las que existen no brindan la posibilidad de hacer subconjuntos de las trazas o mover las posiciones de los nodos a un nuevo rango. Es por ello que se diseñó e implementó una herramienta (ns-2 Trace Tool Kit) que tuviera todas esas características y que además permitiera simular la traza analizada. Para la herramienta ns-2 Trace Toolkit, se definió una serie de funcionalidades, tanto de interfaz gráfica como de análisis de trazas, que permitieran brindar al usuario la mayor cantidad de opciones y que fuesen fáciles de entender al momento de usar la herramienta. En la Figura 11.9 se puede apreciar la ventana principal de la herramienta.

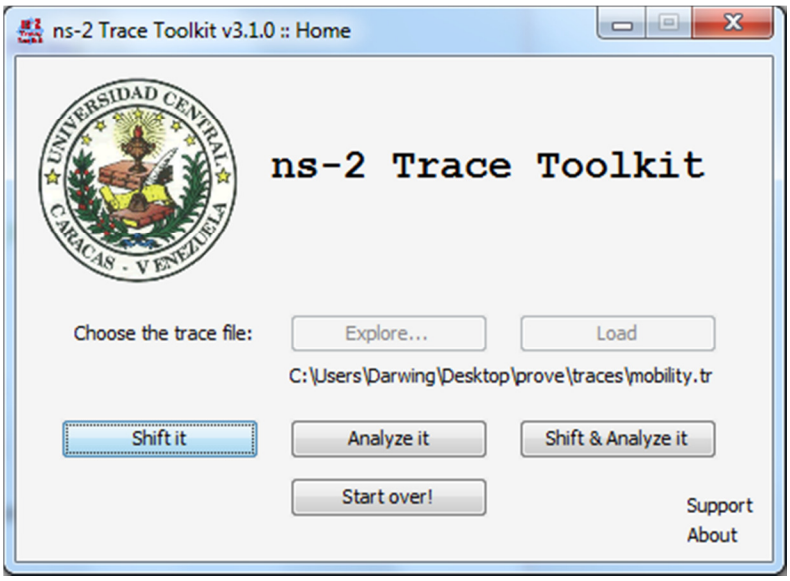


Figura 11.9: Ventana Principal de ns-2 Trace Toolkit

- **Fase de Codificación**: Para implementar la herramienta se utilizó Java como lenguaje de programación, debido a la gran cantidad de ventajas que este lenguaje posee, como la portabilidad y la facilidad de crear interfaces sin usar *plugins* foráneos. Para la implementación en código de la herramienta se dividió el proceso en dos partes.

  La primera parte consiste en la definición, estructuración e implementación de la interfaz de usuario. Se crearon tres interfaces: *Home*, *AnalyzeWindow* y *Simulation*. En Home, se encuentra las funcionalidades iniciales que puede realizar el usuario como cargar un archivo de trazas, mover sus posiciones a un rango establecido o analizar la traza. En *AnalyzeWindow* se tiene toda la información relacionada con la traza cargada, como la velocidad mínima y máxima, posiciones iniciales de los nodos, entre otros. En esa ventana el usuario también puede generar subconjuntos de la traza original. Por último la ventana

Simulation permite al usuario ejecutar una simulación de la traza cargada. Además el usuario puede observar en tiempo real como los nodos van actualizando sus posiciones y los valores de las mismas.

La segunda parte consiste en implementar toda la lógica por detrás de las interfaces. Un punto crítico para esta herramienta es la cantidad de información que puede contener un archivo de trazas. Es por ello que se debe utilizar una estructura de datos eficiente para almacenar tanto los nodos como sus movimientos. Para ello se creó una clase *Node* que contiene todo la información referente a un nodo. También se creó la clase *Movement* que tiene los atributos necesarios para cada movimiento de un nodo. Cada nodo posee un *ArrayList* de *Movement* donde estarán almacenados todos sus movimientos. Al final se tiene un *HashTable* de *Node* para manipular los nodos en la ejecución de la herramienta.

Muchas gráficas como las de las posiciones iniciales de los nodos, como los histogramas son realizadas mediante el uso del *plugin* JFreeChart que ofrece una solución potente para este tipo de gráficas. En la Figura 11.10 se muestra una de las gráficas generadas gracias a este paquete. Para lograr la simulación de la traza se tiene la clase *Viewport* que define los métodos necesarios para mover los nodos a través del tiempo. *Viewport* realiza los cálculos correspondientes para cada movimiento de un nodo con el uso de interpolación lineal.

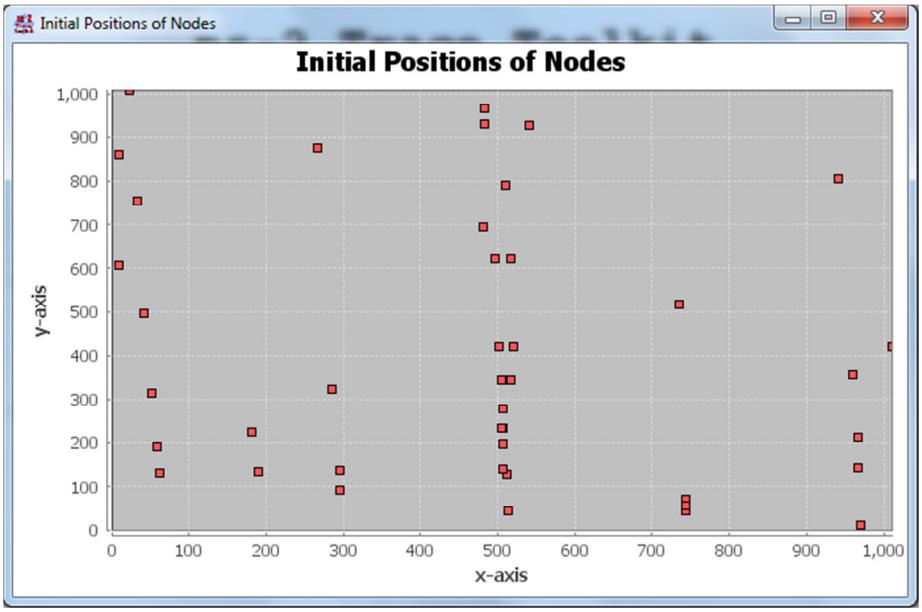


Figura 11.10: Gráfica de la Posición Inicial de los Nodos Usando JFreeChart

- **Fase de Pruebas**: Para probar la herramienta, se usó una serie de trazas pequeñas en las cuales los movimientos de los nodos están claramente definidos y son fáciles de calcular, de esta manera es posible depurar con esta herramienta si los nodos seguían o no la trayectoria especifica de la traza ns-2. Después que se validó el correcto funcionamiento del movimiento de los nodos y la información mostrada, se hicieron pruebas con trazas más grandes. En la Figura 11.11 se observa la ventana de simulación con una traza de 50 nodos.

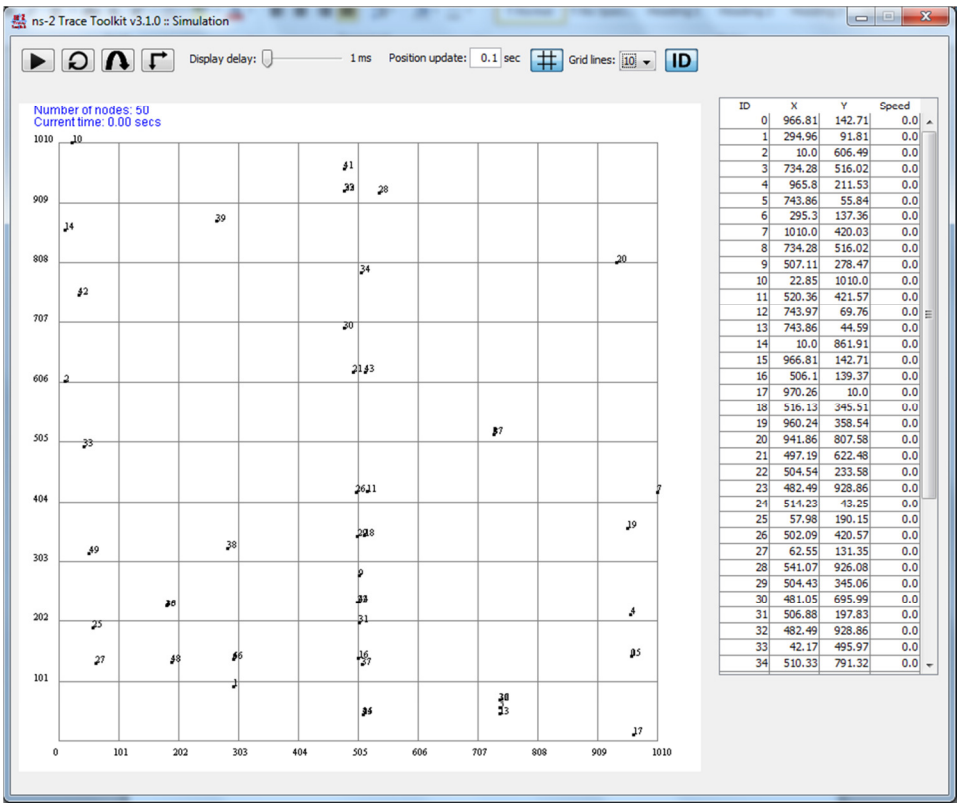


Figura 11.11: Ventana de Simulación de ns-2 Trace Toolkit con 50 Nodos

12. Pruebas y Análisis de los Resultados

En este capítulo se presentan las pruebas realizadas y los resultados obtenidos con los benchmarks propuestos para evaluar la escalabilidad de los simuladores de red ns-3, OMNeT++ y JiST/SWANS. Para todas las pruebas realizadas se tomaron en cuenta las siguientes métricas para evaluar el desempeño de los simuladores estudiados:

- Packet Delivery Ratio (PDR): es la relación entre el número de paquetes de datos recibidos y el número de paquetes de datos enviados.
- End-to-End Delay Promedio: promedio de la diferencia entre el tiempo de salida y el tiempo de llegada de todos los paquetes de datos que fueron recibidos.
- Número de Saltos Promedio: promedio del número de saltos que hicieron los paquetes de datos recibidos.
- Normalized Routing Load (NRL): es la relación entre el número de paquetes transmitidos por el protocolo de enrutamiento y el número de paquetes de datos recibidos.
- Tiempo Real de Simulación: es el tiempo real de duración de la simulación en segundos.
- Consumo de Memoria: el consumo de memoria de la simulación en kB.

12.1 Resultados de las Pruebas Realizadas

En líneas generales, se realizaron dos tipos de pruebas por cada benchmark considerando o no la presencia de RSUs. De esta manera se obtuvieron los siguientes escenarios:

12.1.1 Circle Benchmark sin RSUs

El objetivo de esta prueba es evaluar el comportamiento de los simuladores de red en un escenario con una carretera circular en la cual se forma una red vehicular sin la presencia de RSUs. Los parámetros más relevantes con valores constantes establecidos para este benchmark son:

- Tiempo de duración de la simulación: 120 segundos.
- Número de vehículos: 400.
- Distancia entre los vehículos: regla de 2 segundos.
- Número de RSUs: 0.
- Número de canales: 3.
- Velocidad de los vehículos en cada canal: 60, 80 y 100 km/h, desde el más interno hasta el más externo.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Intervalo de envío de mensajes de los vehículos: 3 segundos.
- Protocolo de enrutamiento: AODV.

El parámetro a variar en esta prueba es el bitrate, la idea es observar cómo influye la variación de este parámetro en el comportamiento de la red simulada. Una vez obtenido los resultados de esta prueba en cada simulador, se comparan entre sí. Cada simulación se realizó utilizando los valores permitidos por el estándar 802.11a, es decir: 6, 9, 12, 18, 24, 36, 48 y 54 Mbps. El protocolo de enrutamiento escogido fue AODV dado que es el único protocolo común entre los tres simuladores.

En la Figura 12.1 se observa cómo afecta la variación del bitrate sobre el PDR. Como se esperaba, el PDR disminuye cuando el bitrate aumenta. JiST/SWANS fue el simulador que arrojó un mejor PDR para todos los bitrates. OMNeT++ tiene un PDR bastante elevado hasta valores del bitrate de 36 Mbps. Para 48 y 54 Mbps, OMNeT++ sufre una fuerte disminución del PDR. ns-3 reportó los peores resultados para el PDR, al investigar más, se notó que ns-3 envía un alto número de paquetes de control en AODV. Aparentemente, es un bug de la implementación de AODV en ns-3 que ha sido reportado[36]. La gran cantidad de mensajes de control congestiona la red perdiéndose muchos mensajes de datos afectando el PDR y muchas otras métricas tomadas en cuenta para estas pruebas.

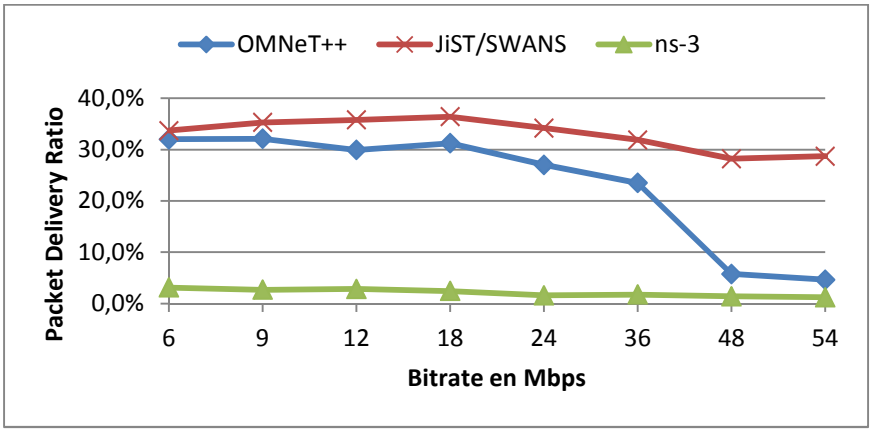


Figura 12.1: Packet Delivery Ratio para la Prueba Circle Benchmark sin RSUs

En la Figura 12.2 se observa cómo el bitrate afecta el promedio del delay para los tres simuladores. Como AODV es un protocolo de enrutamiento reactivo (establece la ruta hacia un destino solo bajo demanda) debería presentar un delay promedio elevado. Se puede observar que tanto ns-3 como OMNeT++ presentan un delay promedio de 1 segundo para la entrega de un mensaje a un destino en específico para todos los bitrates (a excepción de OMNeT++ para bitrates mayores a 36 Mbps). En cambio para JiST/SWANS se obtuvo un comportamiento diferente donde el delay empezó a aumentar después de los 18 Mbps. La justificación de este comportamiento se puede observar en la Figura 12.3, donde el número promedio de saltos va en aumento en este simulador.

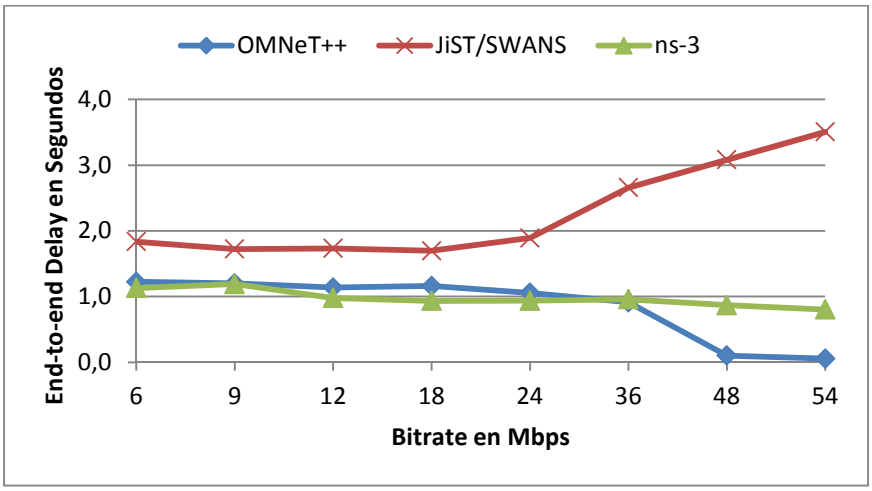


Figura 12.2: End-to-End Delay Promedio para la Prueba Circle Benchmark sin RSUs

[36] http://tinyurl.com/ns3-AODV-performance-MANET

Tanto OMNeT++ como JiST/SWANS alcanzan a destinatarios que están a 4 o más saltos como se puede apreciar en la Figura 12.3, lo que demuestra que funcionan bien en este escenario. En cambio, ns-3 no funciona correctamente ya que solamente alcanza a vecinos que están a uno o dos saltos, ocasionando un PDR bajo.

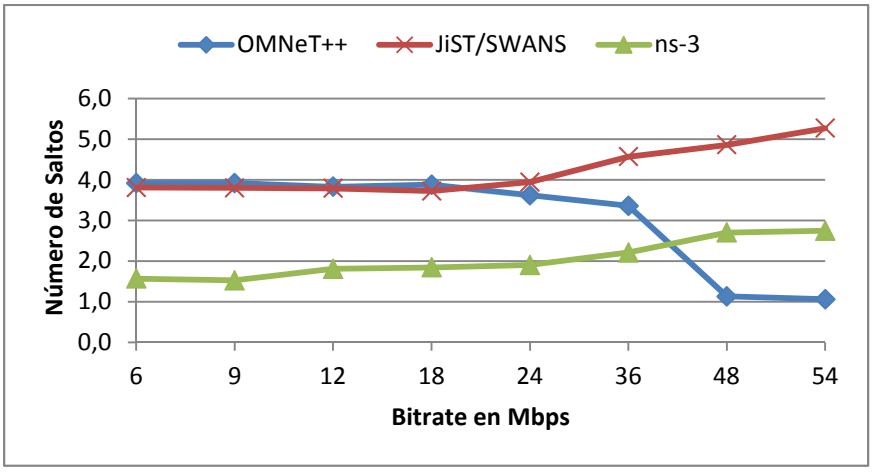


Figura 12.3: Número Promedio de Saltos para la Prueba Circle Benchmark sin RSUs

En la Figura 12.4 se observa el NRL obtenido en los tres simuladores para cada uno de los bitrates permitidos por el estándar 802.11a. Como ya se mencionó, dado al bug que presenta ns-3 para AODV, hay un número alto de mensajes de control. OMNeT++ presenta una ligera ventaja sobre JiST/SWANS hasta 36 Mbps, para tasas mayores JiST/SWANS se mantiene casi invariante mientras que OMNeT++ aumenta drásticamente el NRL.

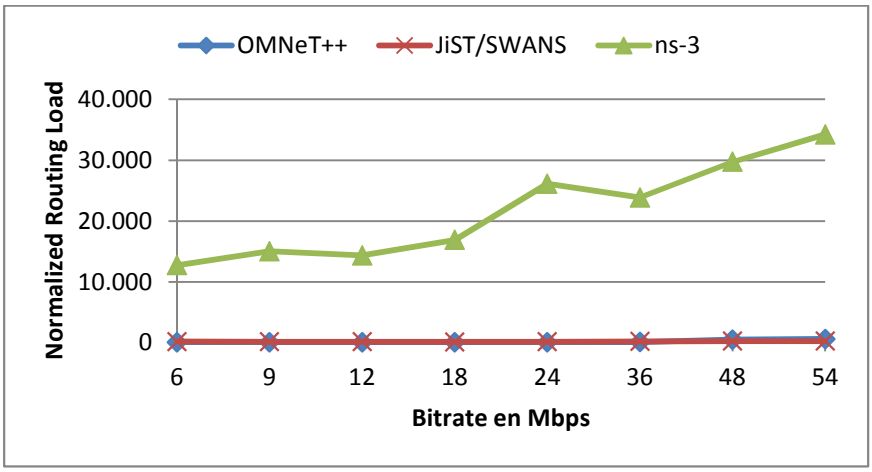


Figura 12.4: Normalized Routing Load para la Prueba Circle Benchmark sin RSUs

En la Figura 12.5 se puede observar cómo afectó la variación del bitrate en el tiempo real para la ejecución de la simulación. JiST/SWANS fue el que obtuvo mejores tiempos sin ni siquiera llegar a 30 minutos por simulación. En cambio ns-3 fue el que tuvo los peores tiempos dado por la sobrecarga que genera AODV por la gran cantidad de mensajes de control enviados por el protocolo de enrutamiento.

En la Figura 12.6 se observa el consumo de memoria para cada simulador según la variación del bitrate. Tanto JiST/SWANS como OMNeT++ presentan un consumo de memoria entre los 150 MB y 200 MB, teniendo JiST/SWANS una ligera ventaja para tasas mayores a 12 Mbps. ns-3 presentó el mayor consumo de memoria (en promedio de 350 MB), debido al excesivo envío de mensajes de control.

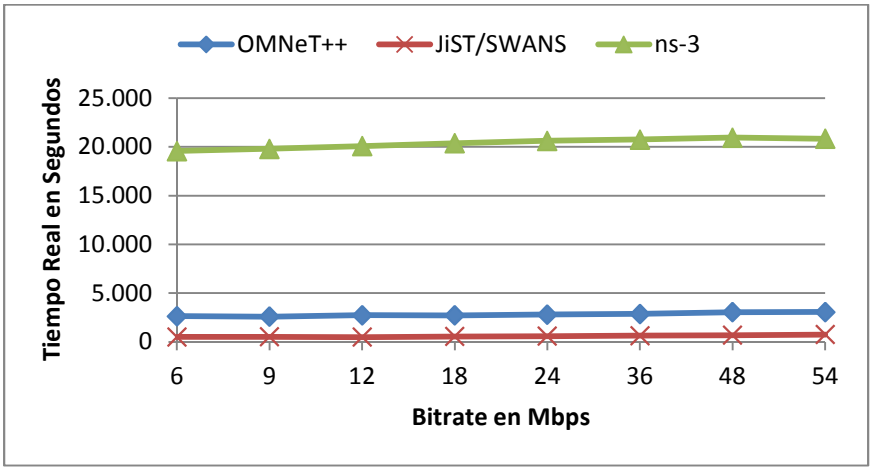


Figura 12.5: Tiempo Real de la Simulación para la Prueba Circle Benchmark sin RSUs

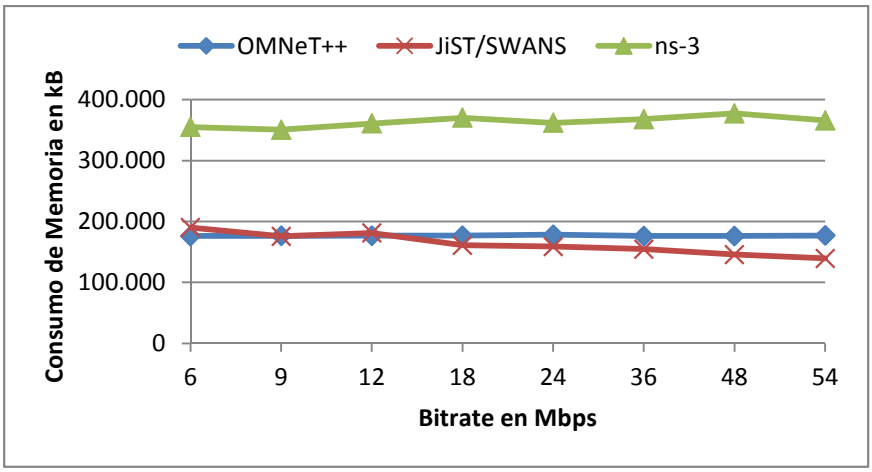


Figura 12.6: Consumo de Memoria para la Prueba Circle Benchmark sin RSUs

12.1.2 City Benchmark sin RSUs

El objetivo de esta prueba es evaluar el comportamiento de los simuladores de red en un escenario realista con carreteras derivadas de mapas reales en el cual se forma una red vehicular sin la presencia de RSUs. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 120 segundos.
- Número de vehículos: 400.
- Número de RSUs: 0.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.

- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Protocolo de enrutamiento: AODV.

El parámetro a variar en esta prueba es el intervalo para el envío de mensajes de los vehículos, dicho de otra manera, lo que se varía en cada simulación es la cantidad de mensajes que envían los vehículos por segundo. La idea es observar cómo influye la variación de este parámetro en el comportamiento de la red simulada. Cada simulación se realizó con los siguientes intervalos para el envío de mensajes consecutivos: 1, 0.2, 0.1, 0.05, 0.04, 0.02 segundos o lo que es el equivalente a enviar 1, 5, 10, 20, 25 y 50 datagramas UDP por segundo. Para estas pruebas se utilizaron trazas de ns-2 hechas sobre el mapa de Caracas con la intención de obtener resultados que se acerquen más a la realidad.

En la Figura 12.7 se observa cómo afectó la variación del intervalo para el envío de mensajes sobre el PDR reportado en cada simulador. OMNeT++ y JiST/SWANS tienen curvas muy parecidas. ns-3 presenta de nuevo el PDR más pobre de los tres simuladores. Todos los simuladores estudiados reportan para esta prueba un comportamiento similar en el PDR, donde éste disminuye a medida que la cantidad de paquetes por segundo aumenta. Eso se debe a que hay una mayor cantidad de mensajes en la red y muchos de ellos se pierden por el alto dinamismo de los nodos y por el aumento de colisiones.

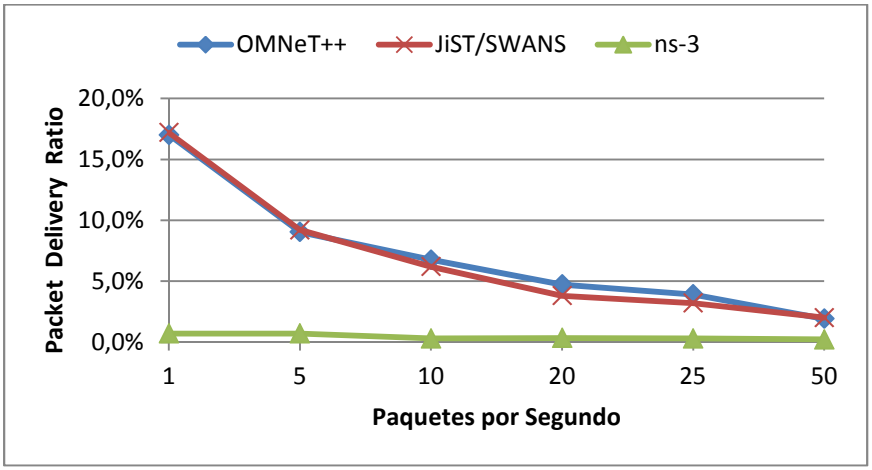


Figura 12.7: Packet Delivery Ratio para la Prueba City Benchmark sin RSUs

En la Figura 12.8 se puede observar el end-to-end delay promedio reportado para los tres simuladores. JiST/SWANS y OMNeT++ tienen curvas similares. ns-3 presenta un mayor delay para la entrega de un mensaje de datos por causa del envío de una alta cantidad de mensajes de control de su implementación de AODV. En la Figura 12.9 se observa la variación del número de saltos promedio dados por un mensaje antes de ser entregado a un destinatario. OMNeT++ y JiST/SWANS presentan curvas similares teniendo una pequeña ventaja para JiST/SWANS. Para estos dos simuladores se observa que el número de saltos va decrementando cuando el número de paquetes por segundo aumenta. Eso se debe a que para mayor cantidad de mensajes por segundo se envían, la red se congestiona más y ocasiona que solo los nodos más cercanos son los que reciben mensajes de datos. En cambio ns-3 presenta en promedio un número de saltos de 1, por lo que se concluye que para esta prueba, AODV en ns-3 no funciona bien dado que solo asegura que los nodos que están a un salto reciban los paquetes de datos.

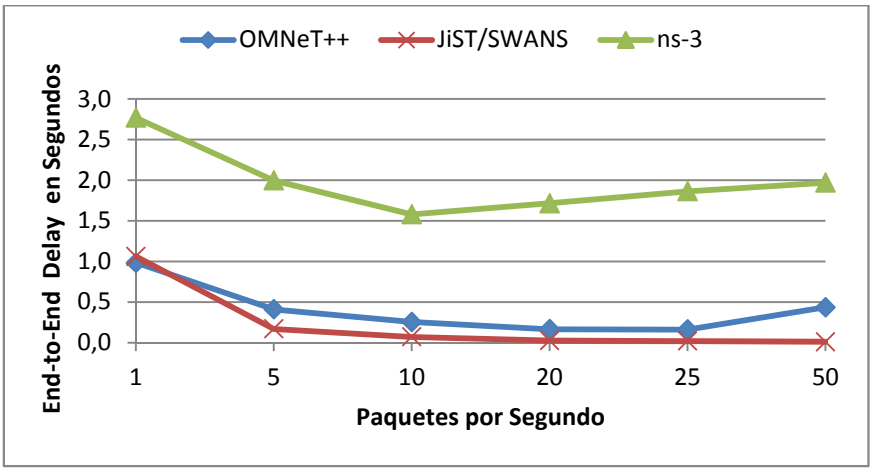


Figura 12.8: End-to-End Delay Promedio para la Prueba City Benchmark sin RSUs

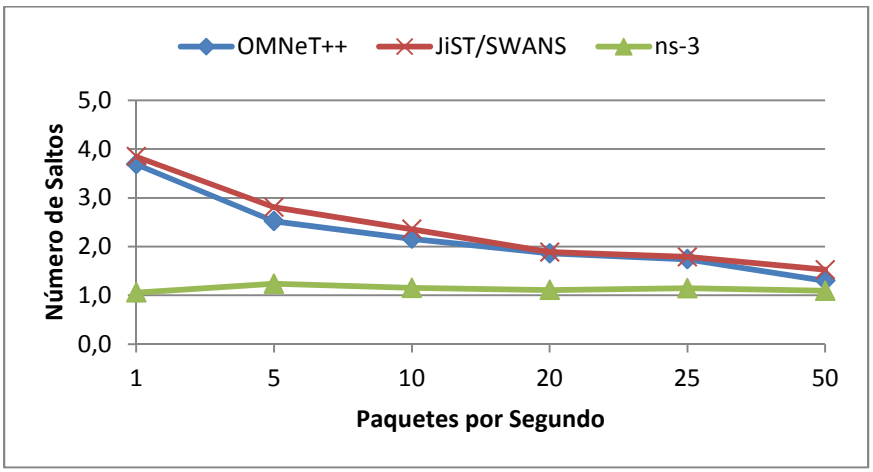


Figura 12.9: Número de Saltos Promedio para la Prueba City Benchmark sin RSUs

En la Figura 12.10 se observa cómo el NRL se ve afectado por la variación del número de paquetes por segundo. En este caso también, ns-3 tiene un NRL superior al reportado por los otros dos simuladores, pero cuando se va aumentando la cantidad de paquetes por segundo, el NRL va disminuyendo que es lo esperado ya que se están enviando más mensajes de datos por unidad de tiempo aumentando la probabilidad que una ruta válida hacia un cierto nodo ya exista cuando se le tiene que enviar un paquete. OMNeT++ y JiST/SWANS vuelven a presentar curvas parecidas, teniendo OMNeT++ una fuerte ventaja sobre JiST/SWANS, ya que en promedio envía 22 mensajes de control contra los 340 mensajes que envía JiST/SWANS por cada mensaje de datos entregado.

En la Figura 12.11 se puede observar el tiempo real de la simulación para los tres simuladores. ns-3 reportó tiempos similares para cada simulación, manteniéndose casi invariante. En cambio JiST/SWANS y OMNeT++ presentaron curvas parecidas donde va aumentando el tiempo según va aumentando la cantidad de paquetes por segundo, presentando OMNeT++ una ligera ventaja respecto a JiST/SWANS para una tasa mayor a 10 paquetes por segundo.

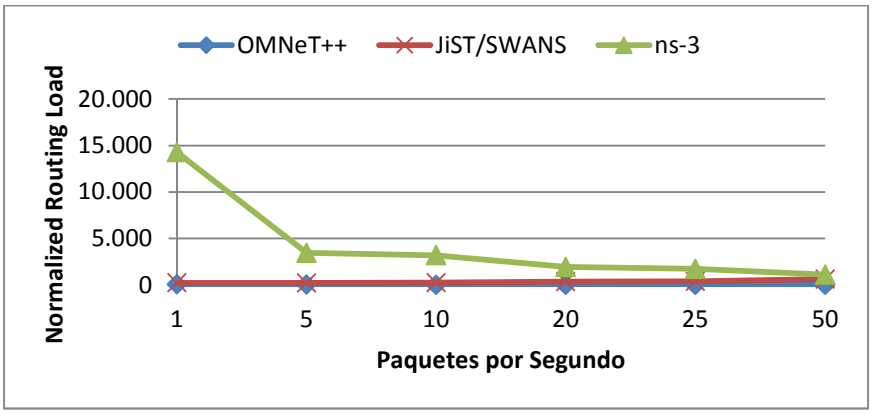


Figura 12.10: Normalized Routing Load para la Prueba City Benchmark sin RSUs

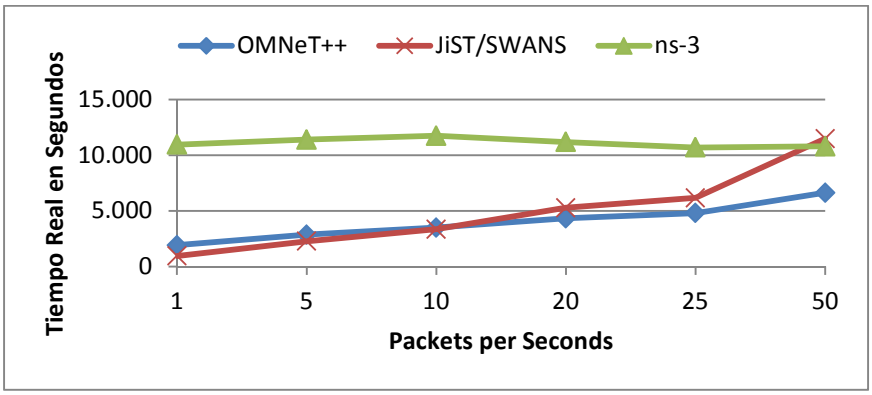


Figura 12.11: Tiempo Real de la Simulación para la Prueba City Benchmark sin RSUs

Por último, en la Figura 12.12 se observa cómo afecta la variación de la cantidad de mensajes por unidad de tiempo en el consumo de memoria. Nuevamente ns-3 presenta un mayor consumo de memoria siendo nada más superado por JiST/SWANS para el caso de 50 paquetes por segundo. Tanto ns-3 como JiST/SWANS tienen un comportamiento parecido donde la cantidad de memoria consumida aumenta en función del aumento del número de paquetes por segundo. En cambio OMNeT++ presenta un comportamiento inverso, es decir, la cantidad de memoria consumida decrementa cuando el número de paquetes por segundo aumenta, aunque el decremento es lento.

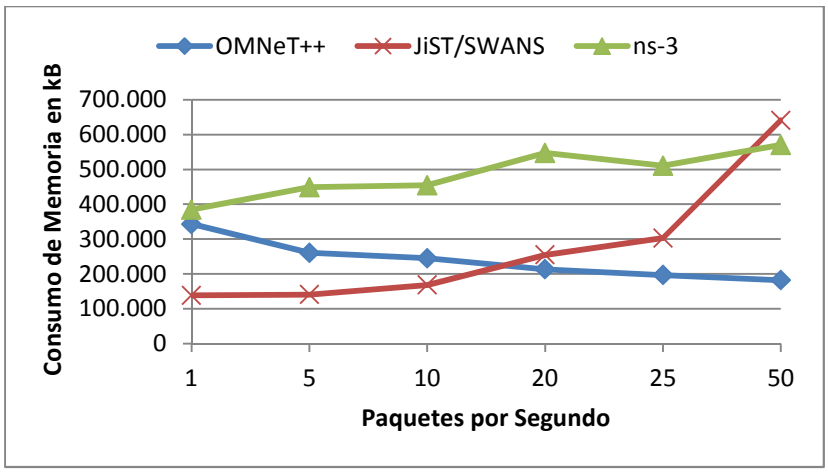


Figura 12.12: Consumo de Memoria para la Prueba City Benchmark sin RSUs

12.1.3 Circle Benchmark con RSUs

El objetivo de esta prueba es evaluar el comportamiento de los simuladores de red en un escenario con una carretera circular en la cual se forma una red vehicular con la presencia de RSUs. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 180 segundos.
- Número de vehículos: 400.
- Distancia entre los vehículos: regla de 2 segundos.
- Número de RSUs: 5.
- Número de canales: 3.
- Velocidad de los vehículos en cada canal: 60, 80 y 100 km/h, desde el más interno hasta el más externo.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Intervalo de envío de mensajes para los vehículos: 0.1 segundos.
- Intervalo de envío de mensajes para los RSUs: 0.01 segundos.
- Protocolo de enrutamiento: AODV.

Esta prueba fue diseñada para observar cómo influye la presencia de los RSUs en el comportamiento de la red vehicular simulada. Para ello, los RSUs fueron ubicados alrededor de la carretera circular de forma tal que no se solapen y generen interferencias entre sí. Se calculó que 5 RSUs con un rango de propagación de 300 metros es suficiente para cubrir la circunferencia de la topología circular modelada de aproximadamente 900 metros. El parámetro a variar en esta prueba es la probabilidad de enviar un mensaje de un vehículo a otro (car-to-car messages). Mientras la probabilidad sea baja hay más chance de que el vehículo envíe el mensaje a un RSU, en caso contrario hay más chance que el destinatario sea otro vehículo. Los RSUs envían mensajes solo a vehículos. De esta manera se realizaron simulaciones con las siguientes probabilidades: 0.1, 0.25, 0.4, 0.5, 0.75, 0.9.

En la Figura 12.13 se observa cómo afecta la variación del parámetro en el PDR en los tres simuladores. El comportamiento fue similar en los simuladores estudiados, donde el PDR aumenta a medida que la probabilidad de mandar mensajes a los vehículos aumenta, siendo el resultado esperado ya que debido a la poca cantidad de RSUs, hay una alta demanda para enviarles mensajes, resultando en mucha pérdida de estos envíos. JiST/SWANS presenta el mayor PDR de los tres simuladores y como era de esperarse por los resultados anteriores, ns-3 ofrece un PDR muy bajo.

En la Figura 12.14 se puede apreciar como varía el end-to-end delay promedio cuando se varía la probabilidad de enviar mensaje a los vehículos. Para este caso ns-3 mantuvo un delay casi invariable mientras que OMNeT++ va disminuyendo el delay en función que la probabilidad de enviar mensajes a vehículos aumenta. JiST/SWANS presenta el delay más bajo y casi invariante.

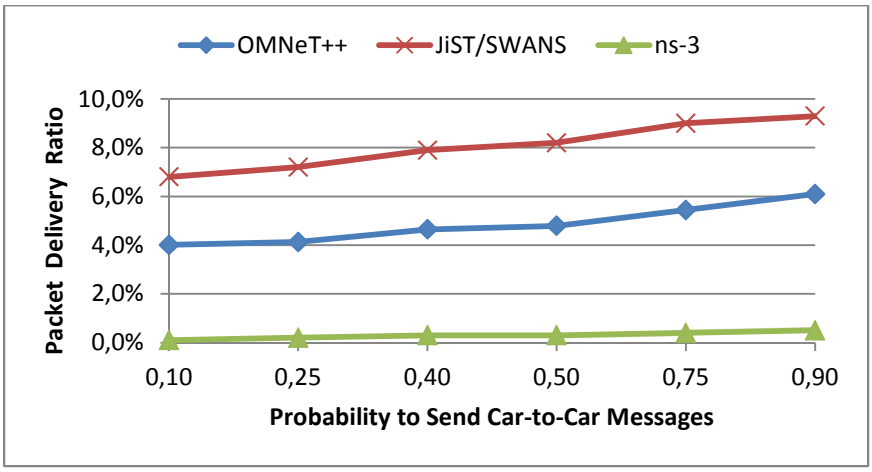


Figura 12.13: Packet Delivery Ratio para la Prueba Circle Benchmark con RSUs

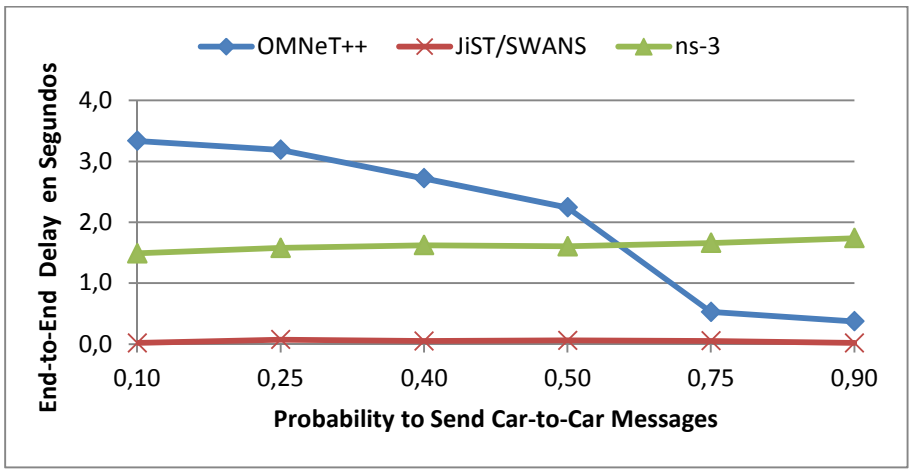


Figura 12.14: End-to-End Delay Promedio para la Prueba Circle Benchmark con RSUs

En la Figura 12.15 se observa cómo afecta el parámetro variado en el número de saltos promedio. Para los tres simuladores se observa que en promedio el número de saltos es casi el mismo quedando entre 1 y 2 saltos, lo que quiere decir que bajo este escenario se pudo hacer pocas entregas a más de dos saltos que no es ideal para las VANETs que se implementarán en un futuro.

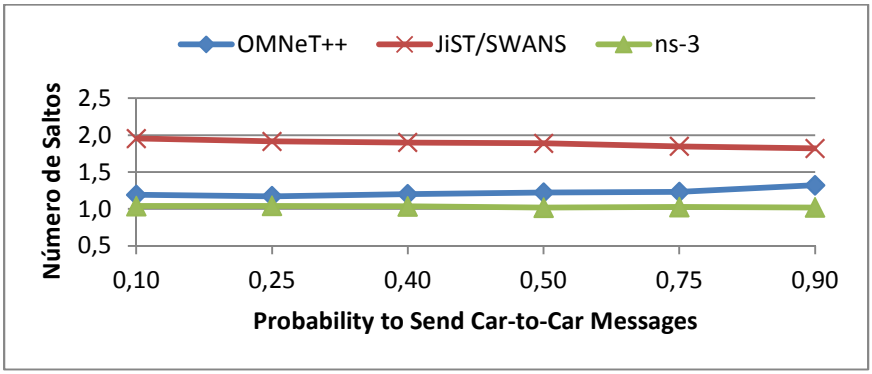


Figura 12.15: Número de Saltos Promedio para la Prueba Circle Benchmark con RSUs

En la Figura 12.16 se puede apreciar el NRL reportado para esta prueba. Como se esperaba ns-3 obtuvo un NRL muy alto en comparación a los otros dos simuladores y mientras va aumentando la probabilidad de enviar mensajes a un vehículo el NRL disminuye para ns-3. Lo mismo ocurre en OMNeT++. De los tres simuladores, OMNeT++ posee el menor NRL, con un promedio de 30 mensajes de control por cada mensaje de datos entregado. En cambio JiST/SWANS reportó 95 mensajes de control en promedio por cada mensaje entregado y su comportamiento difiere de los otros dos simuladores ya que el NRL aumenta a medida que la probabilidad aumenta.

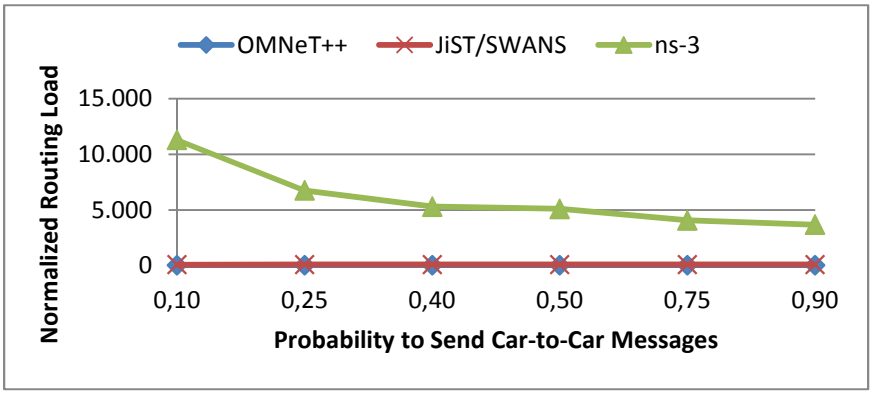


Figura 12.16: Normalized Routing Load para la Prueba Circle Benchmark con RSUs

Con respecto al tiempo real de la simulación y la memoria consumida, se pueden observar los resultados en la Figura 12.17 y la Figura 12.18 respectivamente. Para el tiempo real de la simulación, los tres simuladores se mantienen casi invariantes, siendo ns-3 el que tiene un mayor tiempo y JiST/SWANS el menor. Prácticamente el comportamiento es el mismo para la memoria consumida, donde ns-3 es quien tiene la mayor cantidad de memoria consumida y JiST/SWANS es el que consume menos memoria.

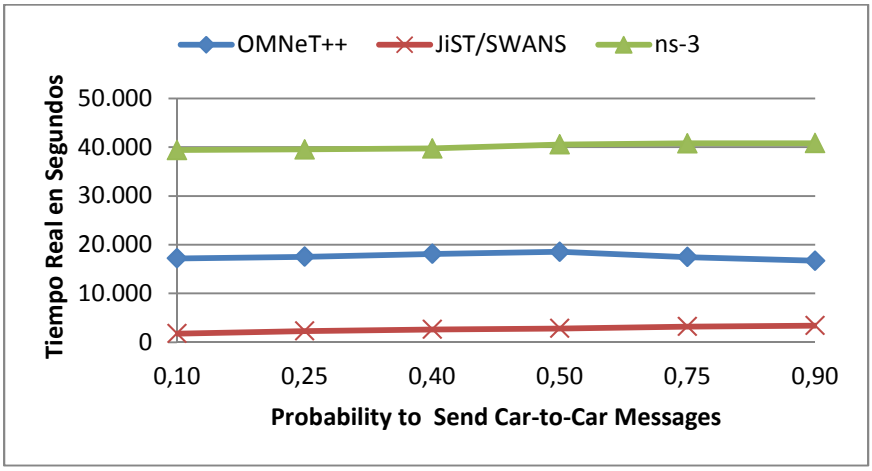


Figura 12.17: Tiempo Real de la Simulación para la Prueba Circle Benchmark con RSUs

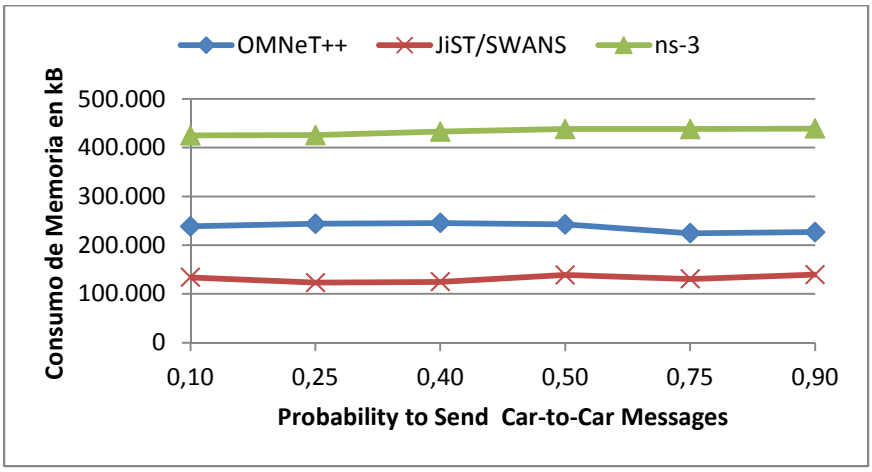


Figura 12.18: Consumo de Memoria para la Prueba Circle Benchmark con RSUs

12.1.4 City Benchmark con RSUs

Al igual que en la prueba Circle Benchmark con RSUs, esta prueba consiste en evaluar el efecto que causa tener RSUs en la red, solo que esta vez se utiliza una red de carreteras real en donde el movimiento de los nodos simulados lo especifica la traza ns-2. Los RSUs son ubicados en unas posiciones fijas dentro del área que abarca la red de carreteras de forma tal que se logre cubrir gran parte del área sin que estos se solapen y generen interferencias entre sí. Se determinó que con 10 RSUs con un rango de propagación de 300 metros era suficiente para cubrir el área por la cual los vehículos circulan. Se usa la misma traza usada en la prueba City Benchmark sin RSUs. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 180 segundos.
- Número de vehículos: 400.
- Número de RSUs: 10.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Intervalo de envío de mensajes para los vehículos: 0.1 segundos.
- Intervalo de envío de mensajes para los RSUs: 0.01 segundos.
- Protocolo de enrutamiento: AODV.

El parámetro a variar en esta prueba es la probabilidad de enviar un mensaje de un vehículo a otro (car-to-car messages). De esta manera se realizaron corridas con las siguientes probabilidades: 0.1, 0.25, 0.4, 0.5, 0.75, 0.9.

En la Figura 12.19 se puede apreciar el PDR reportado para los tres simuladores. Los tres simuladores reportan un aumento del PDR a medida que la probabilidad aumenta que es el resultado esperado. Sin embargo, se tienen PDRs muy bajos para esta topología que se aproxima más a la realidad, de lo que se puede concluir que AODV no se adaptaría para las VANETs que se implementarán en un futuro. JiST/SWANS es el simulador que presenta el mejor PDR y ns-3 el peor dado los bugs ya mencionados del modelo de AODV en este simulador.

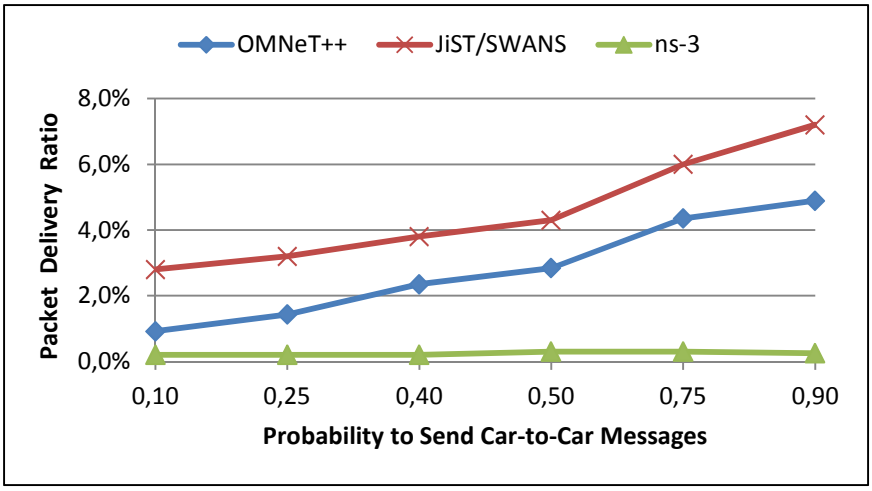


Figura 12.19: Packet Delivery Ratio para la Prueba City Benchmark con RSUs

En la Figura 12.20 se observa cómo varía el end-to-end delay promedio con la variación de la probabilidad. JiST/SWANS se muestra casi invariante mientras que OMNeT++ va disminuyendo su delay a medida que la probabilidad aumenta, lo cual indica que los modelos de AODV en estos dos simuladores no son muy cercanos a la realidad. ns-3 reporta un delay que incrementa si la probabilidad aumenta, que es el resultado esperado.

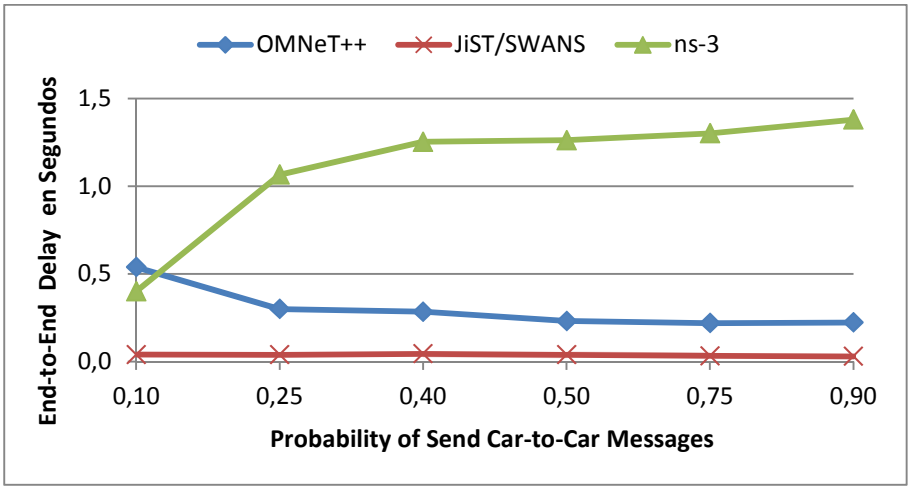


Figura 12.20: End-to-End Delay Promedio para la Prueba City Benchmark con RSUs

En la Figura 12.21 se observa la variación del número de saltos al cambiar la probabilidad. Para los tres simuladores se mantuvo casi invariante la cantidad de saltos, presentando un mayor número de satos el simulador JiST/SWANS. Para ns-3 se presentó un ligero decremento pero luego se mantuvo casi invariable. Fue el simulador que reporto el promedio de saltos más bajo lo que indica que solo los vecinos cercanos a un origen recibieron paquetes de datos.

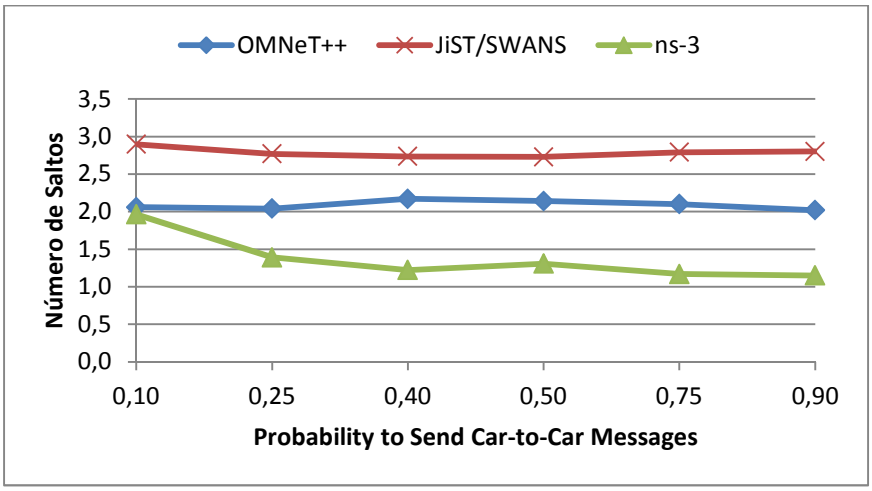


Figura 12.21: Número de Saltos Promedio para la Prueba City Benchmark con RSUs

La variación del NRL para los tres simuladores se puede observar en la Figura 12.22. ns-3 presenta un incremento del NRL a medida que la probabilidad aumenta, teniendo el mayor NRL con respecto a los otros dos simuladores. JiST/SWANS y OMNeT++ en cambio reportan un decremento del NRL a medida que la probabilidad aumenta, siendo OMNeT++ el mejor de todos reportando un NRL promedio de 48.

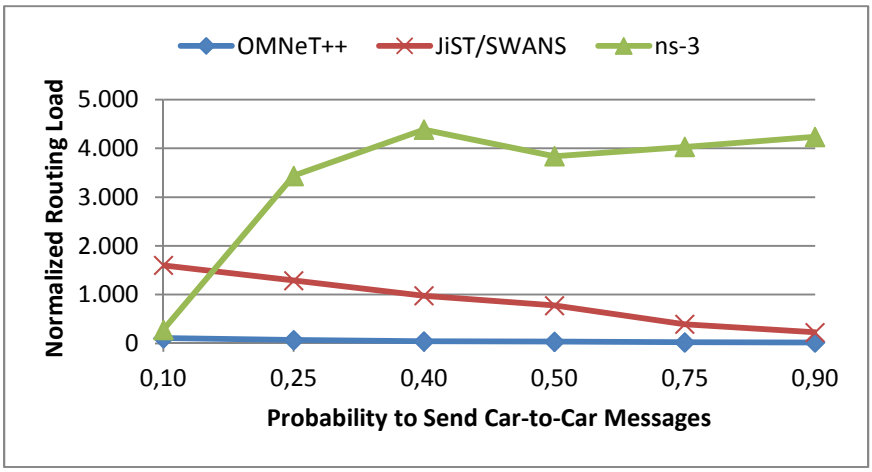


Figura 12.22: Normalized Routing Load para la Prueba City Benchmark con RSUs

Por último, en la Figura 12.23 y la Figura 12.24 se reportan el tiempo real de la simulación y la memoria consumida respectivamente para esta prueba. Con respecto al tiempo real, se observa que tanto JiST/SWANS como OMNeT++ tienen un decremento en el tiempo a medida que la probabilidad aumenta. En cambio ns-3 reporta un incremento a medida que la probabilidad aumenta, pero para probabilidades mayores a 0.25 el tiempo se mantiene casi constante. Con respecto a la cantidad de memoria consumida los tres simuladores presentaron un consumo de memoria casi constante, siendo ns-3 él que consumió más memoria por simulación y OMNeT++ fue él que consumió menos.

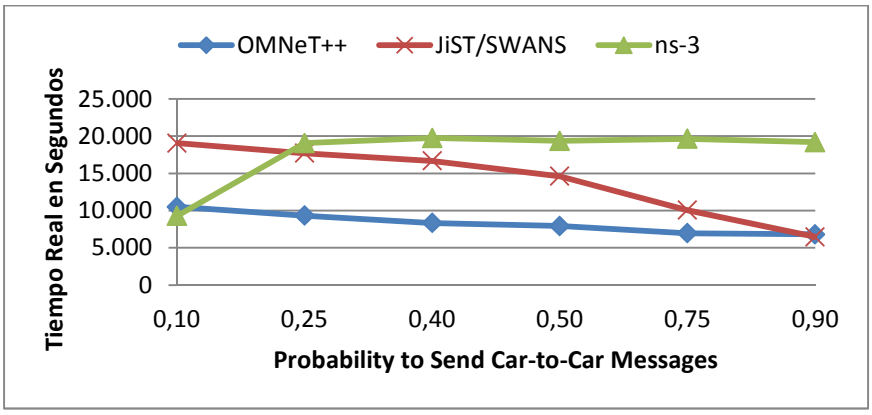


Figura 12.23: Tiempo Real de la Simulación para la Prueba City Benchmark con RSUs

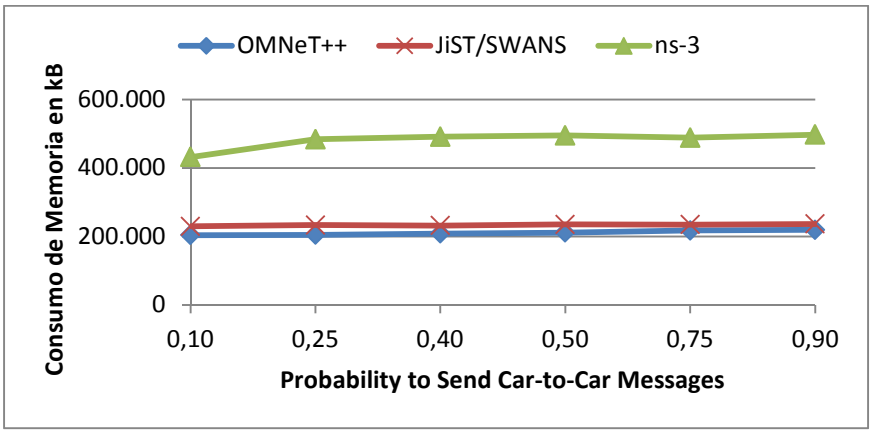


Figura 12.24: Consumo de Memoria para la Prueba City Benchmark con RSUs

12.1.5 Desempeño de Protocolos de Enrutamiento en OMNeT++ sin RSUs

El objetivo de esta prueba es evaluar el comportamiento de los protocolos de enrutamiento MANET en el simulador de red OMNeT++ en una red cercana a la realidad con carreteras derivadas de mapas reales en la cual se forma una red vehicular sin la presencia de RSUs. Es el mismo mapa utilizado en la pruebas de City Benchmark. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 120 segundos.
- Número de vehículos: 400.
- Número de RSUs: 0.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Protocolos de enrutamiento: AODV, DYMO y BATMAN.

El parámetro a variar en esta prueba es el intervalo para el de envío de mensajes de los vehículos, dicho de otra manera, lo que se varía en cada simulación es la cantidad de mensajes que envían los vehículos por segundo. La idea es observar cómo influye la variación de este parámetro en el

comportamiento de la red simulada por cada protocolo de enrutamiento. Cada simulación se realizó con los siguientes intervalos para el envío de mensajes: 1, 0.2, 0.1, 0.05, 0.04, 0.02 segundos o lo que es el equivalente a enviar 1, 5, 10, 20, 25 y 50 datagramas UDP por segundo.

En la Figura 12.25, se observa el PDR obtenido para cada uno de los protocolos de enrutamiento MANET en OMNeT++. BATMAN tiene un PDR muy bajo, esto puede ser ocasionado por la cantidad de mensajes de control del protocolo que se envían por cada mensaje de datos entregado, como se puede observar en la Figura 12.28. Cuando la cantidad de paquetes por segundo aumenta, los tres protocolos sufren de una fuerte disminución del PDR ocasionado por la congestión del medio.

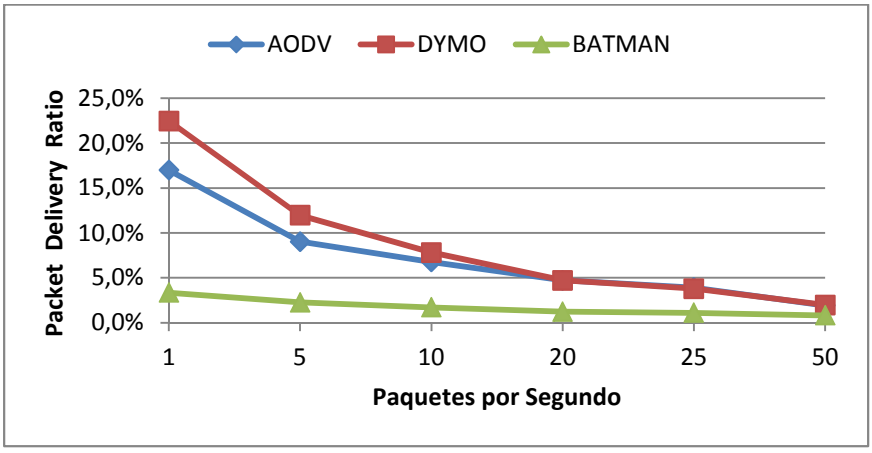


Figura 12.25: Packet Delivery Ratio en OMNeT++ sin RSUs

En la Figura 12.26 se observa el end-to-end delay promedio. BATMAN presenta un incremento en el delay ocasionado por el sobre-envío de mensajes de control del protocolo. En cambio, DYMO y AODV reportan una disminución en el delay a medida que la cantidad de paquetes por segundo aumenta.

En la Figura 12.27 se observa el promedio del número de saltos de los paquetes que fueron recibidos para los tres protocolos. DYMO y AODV funcionan bien dado que se hacen entregas de mensajes a más de 3 saltos. En cambio, BATMAN presenta un número de saltos promedio muy bajo, cercano a 1 y ocasionado por la gran cantidad de mensajes de control que este envía.

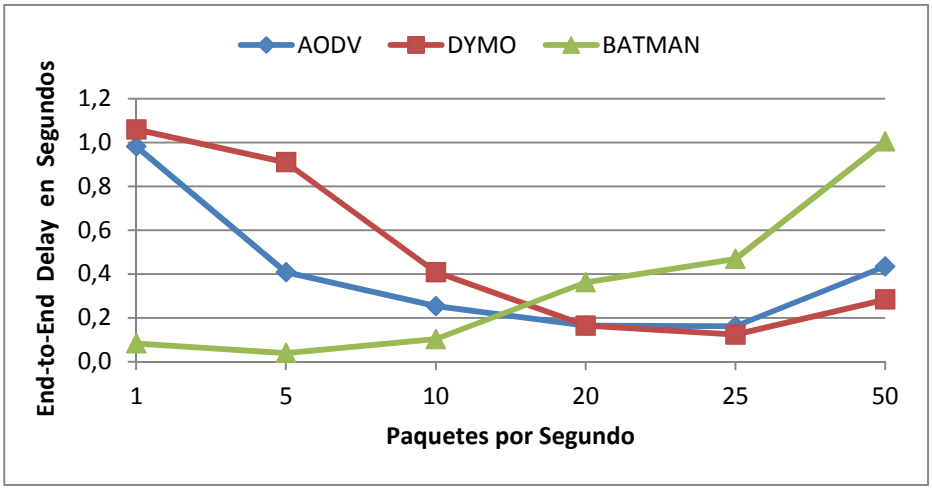


Figura 12.26: End-to-end Delay Promedio en OMNeT++ sin RSUs

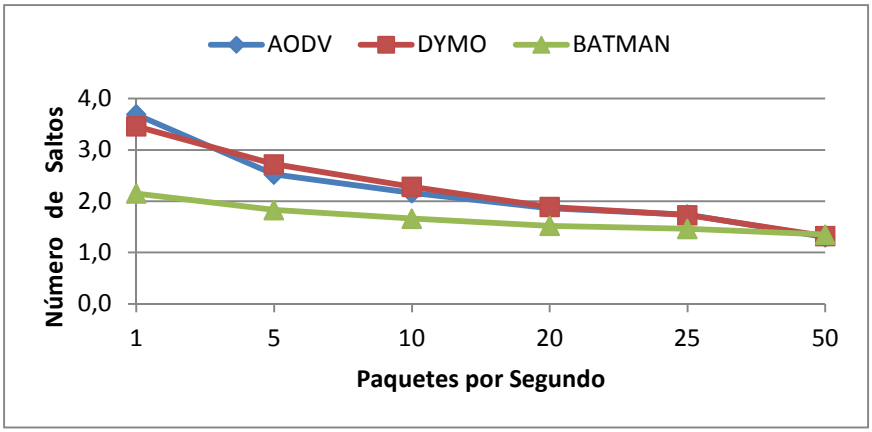


Figura 12.27: Número de Saltos Promedio en OMNeT++ sin RSUs

Como ya se ha mencionado, BATMAN presenta una gran cantidad de mensajes de control del protocolo por cada mensaje de datos entregado. Esto provoca un alto NRL sobre los otros dos protocolos como se puede observar en la Figura 12.28. De acuerdo a nuestros resultados, BATMAN es uno de los protocolos de enrutamiento para MANETs que menos se adapta para los escenarios de las VANETs. En la Figura 12.29 se observa el tiempo real de ejecución para los tres protocolos, presentando BATMAN un mayor tiempo por cada simulación ejecutada.

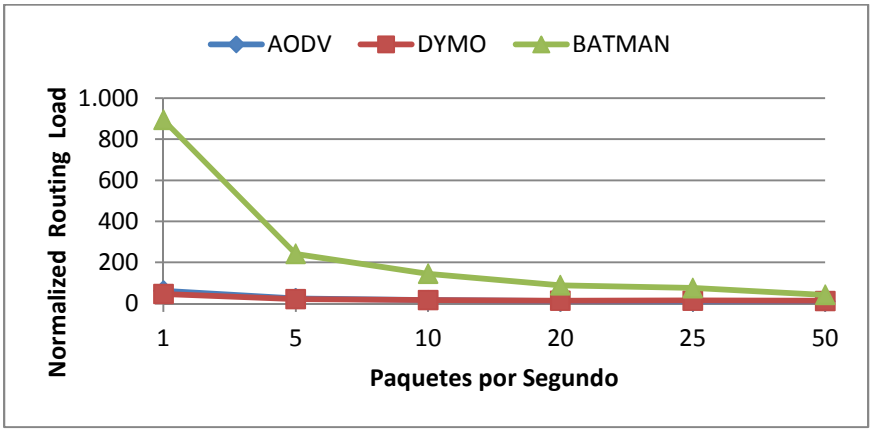


Figura 12.28: Normalized Routing Load en OMNeT++ sin RSUs

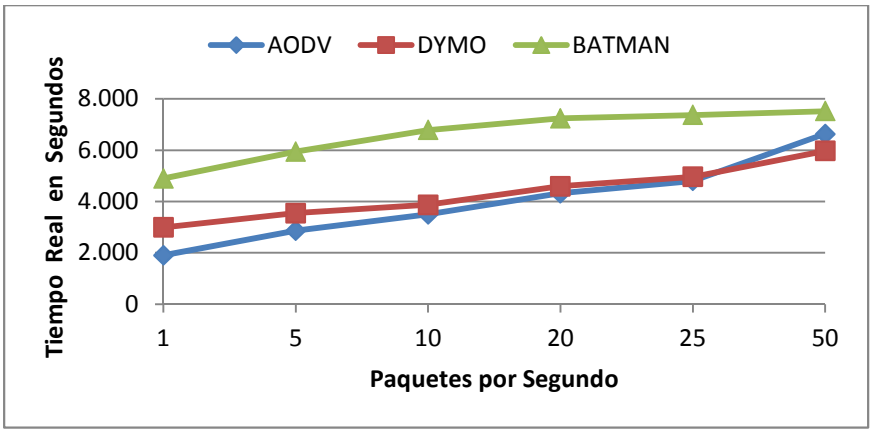


Figura 12.29: Tiempo Real de Simulación en OMNeT++ sin RSUs

Por último, la Figura 12.30 muestra la memoria consumida para los tres protocolos estudiados en OMNeT++. BATMAN presenta un incremento notable en la cantidad de memoria consumida ocasionado por la cantidad de mensajes de control del protocolo mientras que AODV y DYMO mantienen un consumo casi invariante.

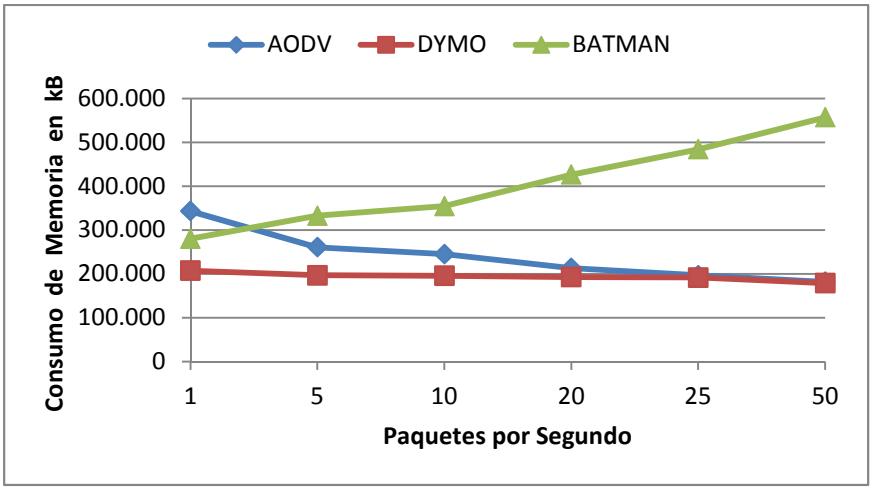


Figura 12.30: Consumo de Memoria en OMNeT++ sin RSUs

12.1.6 Desempeño de Protocolos de Enrutamiento en OMNeT++ con RSUs

Al igual que en la Prueba para el Desempeño de Protocolos de Enrutamiento en OMNeT++ sin RSUs, esta prueba consiste en evaluar el desempeño de los protocolos de enrutamiento MANETs en este simulador de red pero ahora con la presencia de RSUs. Se ubican 10 RSUs siguiendo los mismos lineamientos de la prueba City Benchmark sin RSUs, utilizando la misma traza. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 180 segundos.
- Número de vehículos: 400.
- Número de RSUs: 10.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Intervalo de envío de mensajes para los vehículos: 0.1 segundos.
- Intervalo de envío de mensajes para los RSUs: 0.01 segundos.
- Protocolos de enrutamiento: AODV, DYMO y BATMAN.

El parámetro a variar en esta prueba es la probabilidad de enviar un mensaje de un vehículo a otro (car-to-car messages). De esta manera se realizaron corridas con las siguientes probabilidades: 0.1, 0.25, 0.4, 0.5, 0.75, 0.9.

A medida que la probabilidad de enviar mensajes a un vehículo aumenta, el PDR aumenta como se observa en la Figura 12.31. Esto es debido a que los RSUs son concentradores de comunicación, y a medida que se va liberando su trabajo, aumenta el PDR ya que hay menos colisiones. BATMAN

presenta, al igual que en la prueba anterior, un PDR muy bajo mientras que DYMO y AODV presentan un mejor comportamiento bajo este escenario.

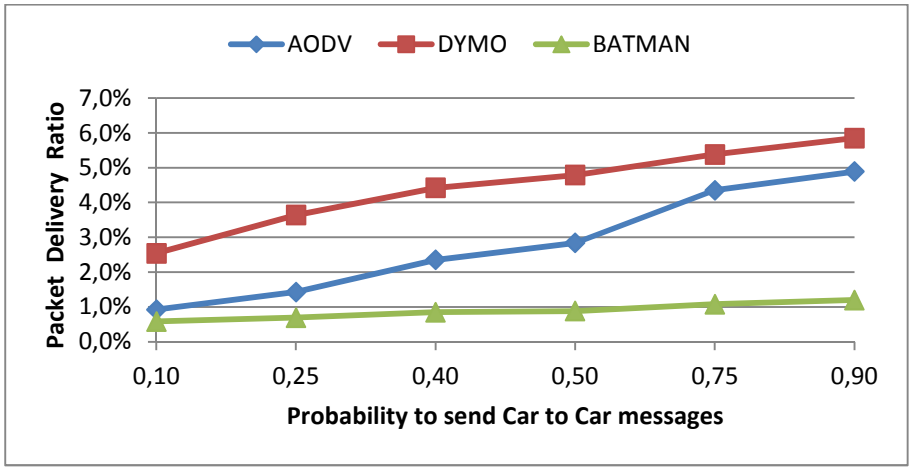


Figura 12.31: Packet Delivery Ratio en OMNeT++ con RSUs

En la Figura 12.32 se observa el end-to-end delay promedio para los tres protocolos. Este parámetro debería disminuir a medida que la probabilidad aumenta como ocurre con AODV. Pero para BATMAN reporta un delay muy bajo que aumenta a medida que la probabilidad aumenta. El delay bajo reportado para BATMAN es debido a que sólo los vecinos más cercanos reciben paquetes de datos (como se puede observar en la Figura 12.33) y su aumento es ocasionado por la gran cantidad de mensajes de control enviados.

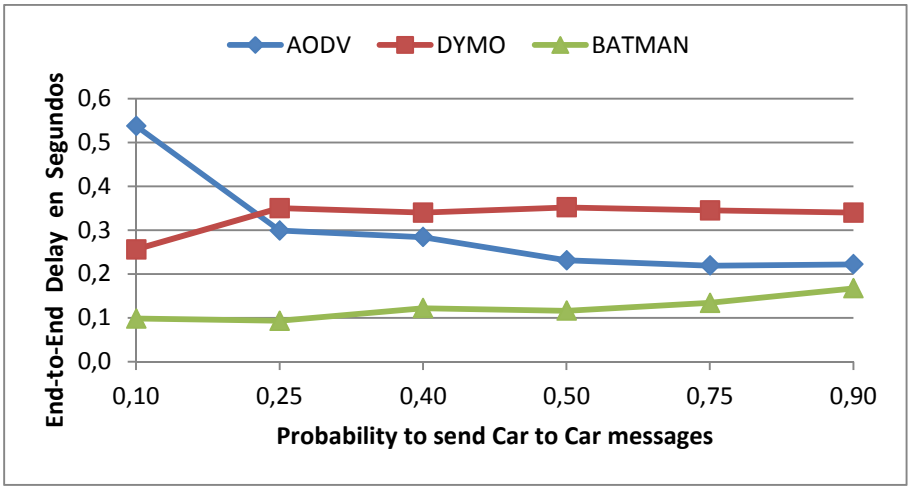


Figura 12.32: End-to-end Delay Promedio en OMNeT++ con RSUs

En la Figura 12.33 se observa el número de saltos promedio para los tres protocolos. Para AODV y DYMO se tiene entregas de mensajes de datos a destinos que están a más de dos saltos, presentando un pequeño decremento con respecto a la prueba anterior BATMAN sigue presentando un número de saltos muy bajo asegurando que sólo los destinos que estén a un salto reciban paquetes de datos.

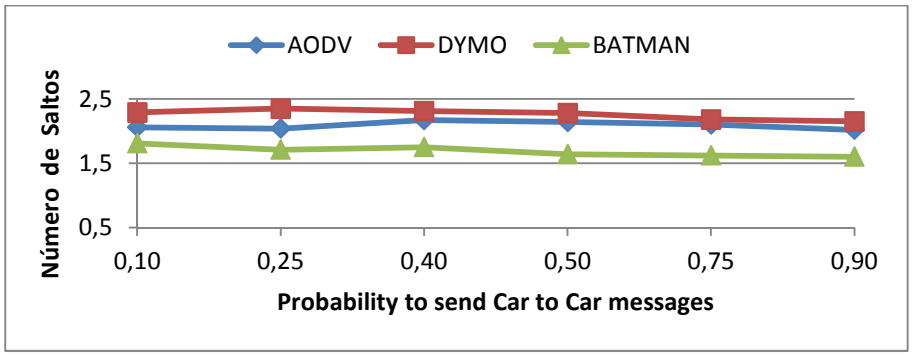


Figura 12.33: Número de Saltos Promedio en OMNeT++ con RSUs

En la Figura 12.34 se muestra el NRL para los tres protocolos. BATMAN reporta de nuevo un NRL muy alto debido a su gran cantidad de mensajes de control enviados por cada mensaje de datos entregado. Los 3 protocolos presentan una disminución en el NRL a medida que la probabilidad aumenta, esto se debe a que los vehículos tienen más chance de que estén rodeados de otros vehículos que de estar cerca de un RSU, lo que provoca que a mayor probabilidad, los vehículos cercanos recibirán más mensajes de datos disminuyendo así el NRL.

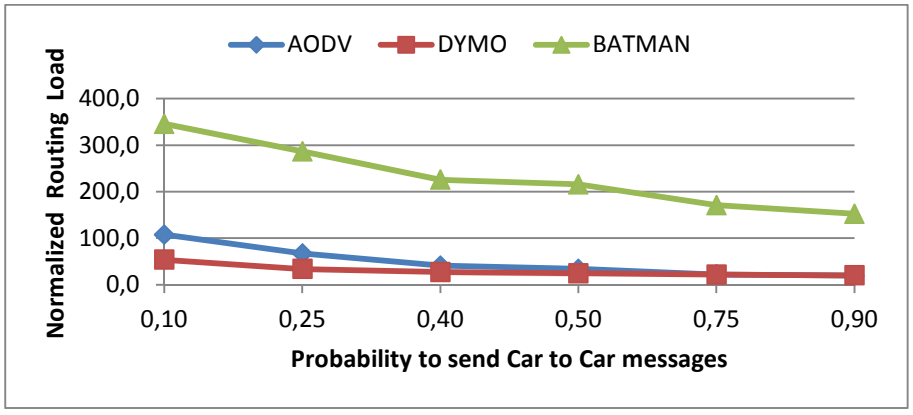


Figura 12.34: Normalized Routing Load en OMNeT++ con RSUs

En la Figura 12.35 se puede apreciar el tiempo de ejecución para las simulaciones realizadas en con cada uno de los protocolos de enrutamiento. De nuevo BATMAN reporta el mayor tiempo y para los tres protocolos el tiempo se mantiene casi invariante. En la Figura 12.36 se muestra el consumo de memoria para esta prueba. Se observa que BATMAN de nuevo posee un alto consumo de memoria ocasionado por la gran cantidad de mensajes de control.

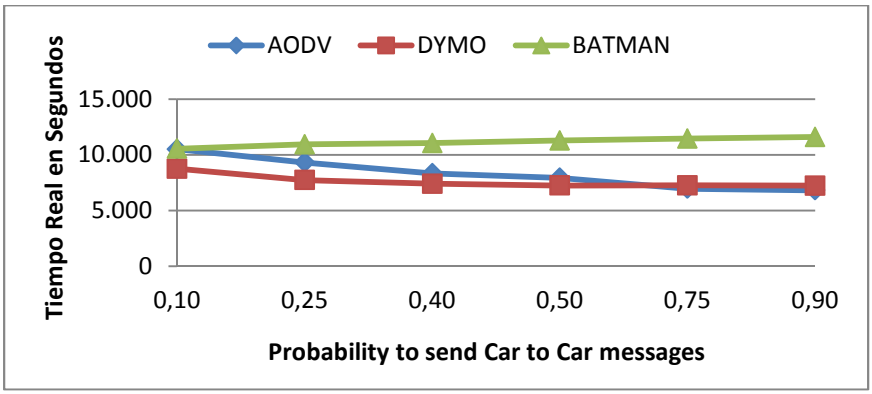


Figura 12.35: Tiempo Real de Simulación en OMNeT++ con RSUs

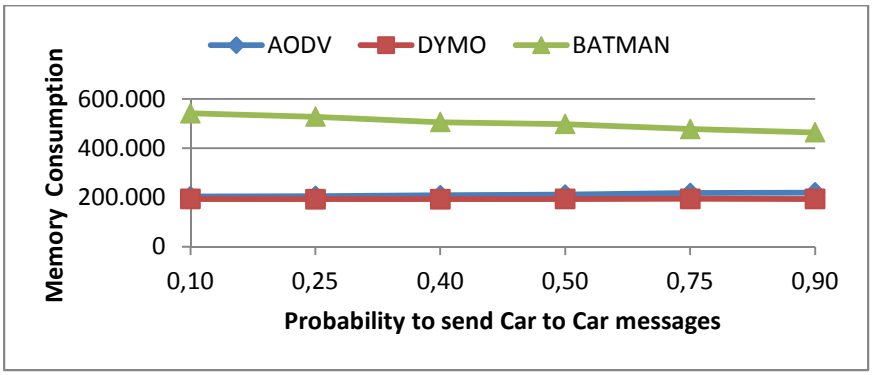


Figura 12.36: Consumo de Memoria en OMNeT++ con RSUs

12.1.7 Desempeño de Protocolos de Enrutamiento en JiST/SWANS sin RSUs

El objetivo de esta prueba es evaluar el comportamiento de los protocolos de enrutamiento MANET en el simulador de red JiST/SWANS en una red realista con carreteras derivadas de mapas reales en el cual se forma una red vehicular sin la presencia de RSUs. Es el mismo mapa utilizado en la pruebas de City Benchmark. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 120 segundos.
- Número de vehículos: 400.
- Número de RSUs: 0.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Protocolo de enrutamiento: AODV y DSR.

El parámetro a variar en esta prueba es el intervalo para el de envío de mensajes de los vehículos, dicho de otra manera, lo que se varía en cada simulación es la cantidad de mensajes que envían los vehículos por segundo. La idea es observar cómo influye la variación de este parámetro en el comportamiento de la red simulada por cada protocolo de enrutamiento. Cada simulación se realizó con los siguientes intervalos para el envío de mensajes: 1, 0.2, 0.1, 0.05, 0.04, 0.02 segundos o lo que es el equivalente a enviar 1, 5, 10, 20, 25 y 50 datagramas UDP por segundo.

En la Figura 12.37 se muestra el PDR para los dos protocolos. El PDR va disminuyendo a medida que la cantidad de mensajes aumenta, presentando DSR una ligera ventaja con AODV. Esto se debe a que DSR posee una cantidad de mensajes de control menor que AODV (como se puede apreciar en la Figura 12.40) y esto causa una menor congestión en la red, asegurando que una mayor cantidad de mensajes de datos lleguen a sus destinos.

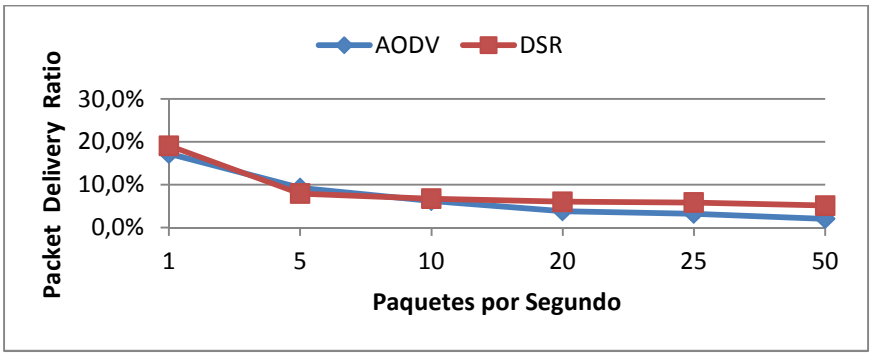


Figura 12.37: Packet Delivery Ratio en JiST/SWANS sin RSUs

DSR presenta un end-to-end delay bastante alto comparado con los resultados que se han obtenido en otros protocolos en pruebas anteriores, como se puede observar en la Figura 12.38. En cambio, AODV muestra un delay promedio demasiado bajo lo que hace dudar del realismo de este modelo.

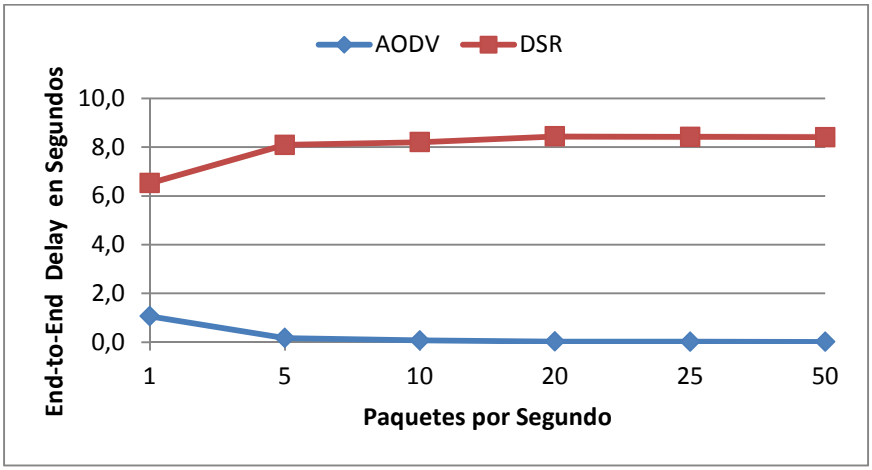


Figura 12.38: End-to-End Delay Promedio en JiST/SWANS sin RSUs

AODV y DSR permiten alcanzar destinos que están a más de 3 saltos como se muestra en la Figura 12.39. Este número va disminuyendo a medida que la congestión aumenta por el aumento de los paquetes por segundo. DSR presenta una ligera ventaja debido a que la red presenta menor congestión porque la cantidad de mensajes de control en DSR es menor a AODV (Figura 12.40).

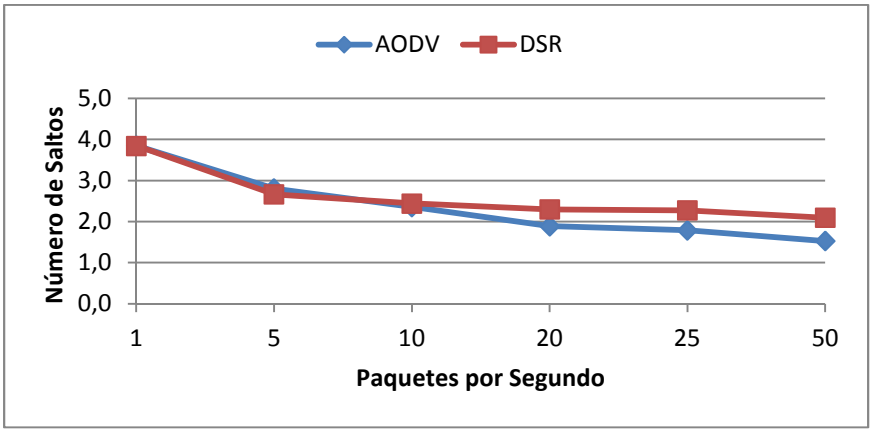


Figura 12.39: Número de Saltos Promedio en JiST/SWANS sin RSUs

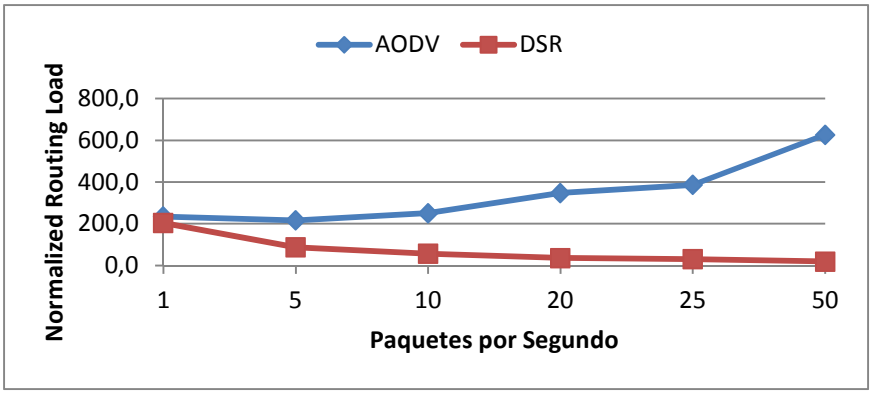


Figura 12.40: Normalized Routing Load en JiST/SWANS sin RSUs

El tiempo de ejecución de la simulación se ve afectado por el NRL de AODV, como se puede observar en la Figura 12.41. En cambio, DSR permanece casi invariante a medida que el número de paquetes por segundo aumenta.

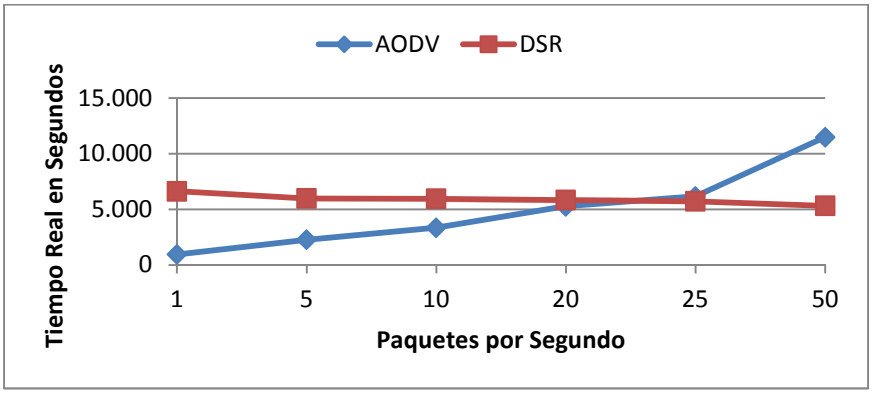


Figura 12.41: Tiempo Real de Simulación en JiST/SWANS sin RSUs

En la Figura 12.42 se puede observar el consumo de memoria para los dos protocolos. DSR presenta un consumo mayor respecto a AODV a pesar que éste último tiene un NRL mucho mayor al que reporta DSR.

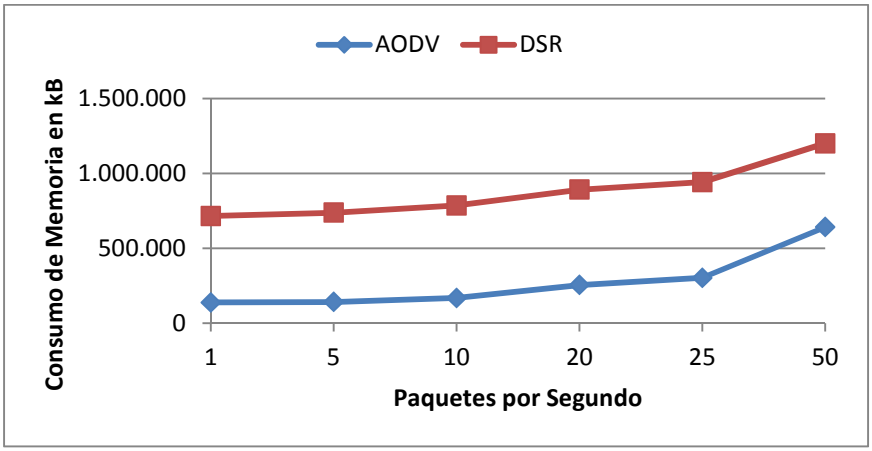


Figura 12.42: Consumo de Memoria en JiST/SWANS sin RSUs

12.1.8 Desempeño de Protocolos de Enrutamiento en JiST/SWANS con RSUs

Al igual que en la Prueba para el Desempeño de Protocolos de Enrutamiento en JiST/SWANS sin RSUs, esta prueba consiste en evaluar el desempeño de los protocolos de enrutamiento MANETs en este simulador de red pero ahora con la presencia de RSUs. Se ubican 10 RSUs siguiendo lo realizado en la prueba City Benchmark sin RSUs, utilizando la misma traza. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 180 segundos.
- Número de vehículos: 400.
- Número de RSUs: 10.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Intervalo de envío de mensajes para los vehículos: 0.1 segundos.
- Intervalo de envío de mensajes para los RSUs: 0.01 segundos.
- Protocolo de enrutamiento: AODV y DSR.

El parámetro a variar en esta prueba es la probabilidad de enviar un mensaje de un vehículo a otro (car-to-car messages). De esta manera se realizaron corridas con las siguientes probabilidades: 0.1, 0.25, 0.4, 0.5, 0.75, 0.9.

En la Figura 12.43 se aprecia el PDR para los dos protocolos. AODV presenta un aumento en el PDR a medida que la probabilidad aumenta. En DSR sucede lo contrario debido a que presenta un delay alto para la entrega de cada mensaje de datos a medida que la probabilidad aumenta, como se puede observar en la Figura 12.44. AODV presenta un delay muy bajo lo que se puede inferir que hay problemas en el modelo o no es muy realista.

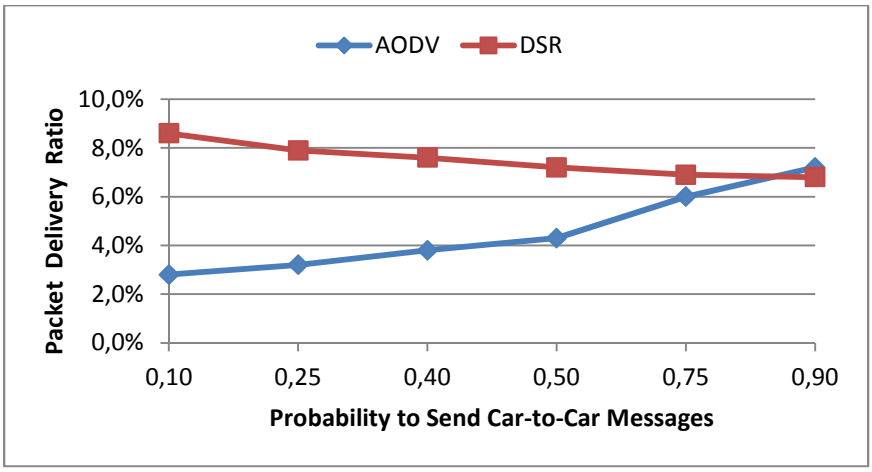


Figura 12.43: Packet Delivery Ratio en JiST/SWANS con RSUs

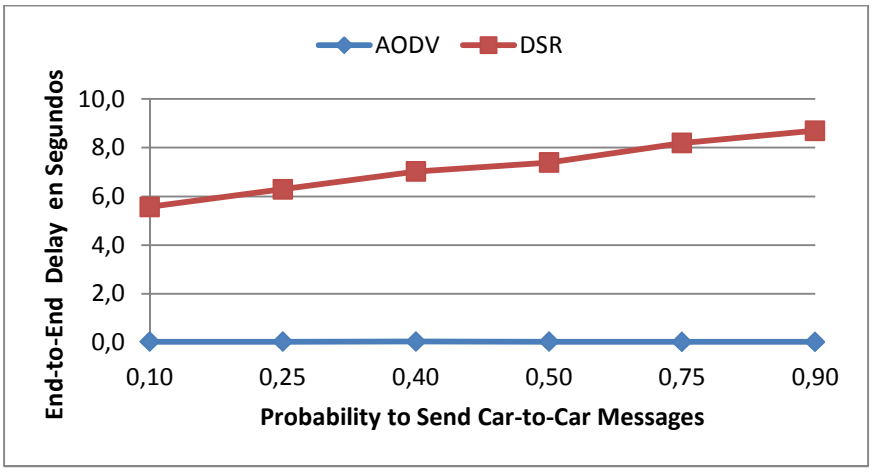


Figura 12.44: End-to-End Delay Promedio en JiST/SWANS con RSUs

En la Figura 12.45 se observa el número de saltos promedio para la entrega de un mensaje de datos para los dos protocolos. AODV y DSR funcionan bien bajo este escenario ya que aseguran la entrega de mensajes a destinos que estén a más de 3 saltos.

En la Figura 12.46 se observa el NRL reportado para los dos protocolos. Como en la prueba anterior, AODV presenta un NRL superior que va disminuyendo a medida que la probabilidad aumenta. Esto es debido a que los vehículos tienen un mayor chance de estar rodeados de otros vehículos que de estar cerca de un RSU. DSR se mantiene casi invariante y con un NRL muy bajo, lo que se puede inferir que el modelo es muy simple.

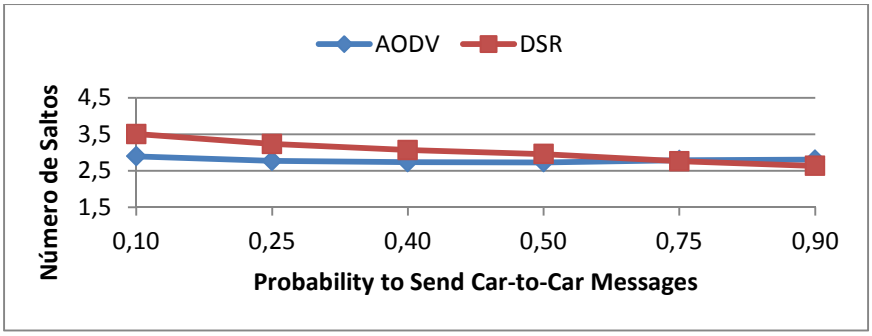


Figura 12.45: Número de Saltos Promedio en JiST/SWANS con RSUs

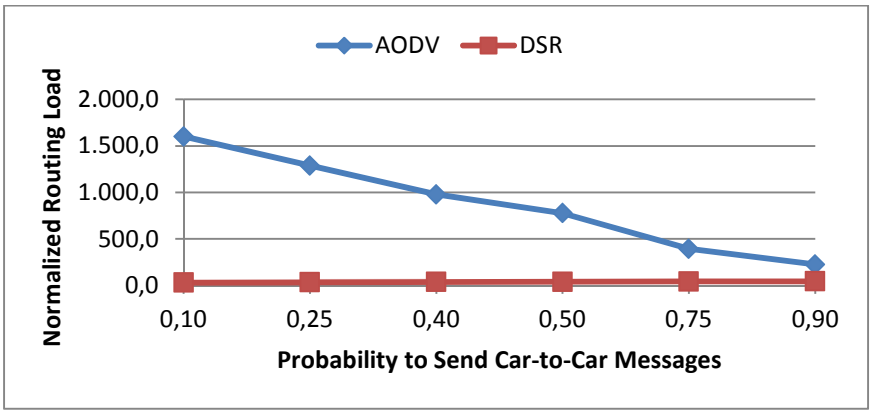


Figura 12.46: Normalized Routing Load en JiST/SWANS con RSUs

En la Figura 12.47 se observa el tiempo real de ejecución para la simulación en los dos protocolos. Nuevamente DSR permanece casi invariante en el resultado para esta métrica mientras que AODV muestra un tiempo mayor pero que va disminuyendo dado que el NRL también va disminuyendo. Con respecto al consumo de memoria, DSR vuelve a ser el protocolo que consume más memoria, como se puede observar en la Figura 12.48.

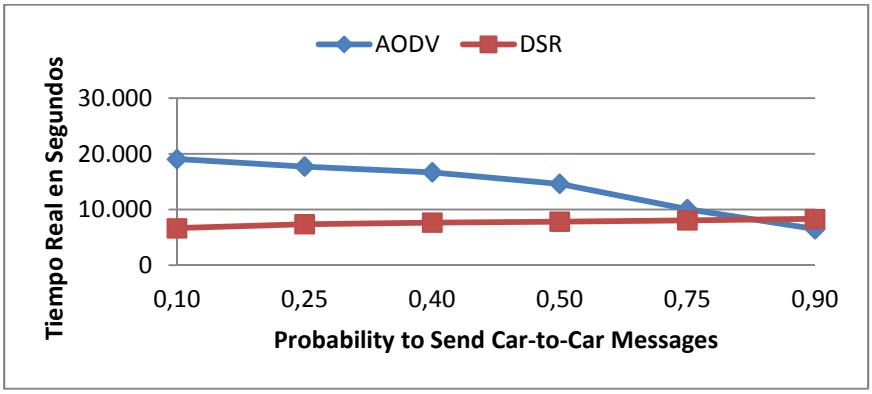


Figura 12.47: Tiempo Real de Simulación en JiST/SWANS con RSUs

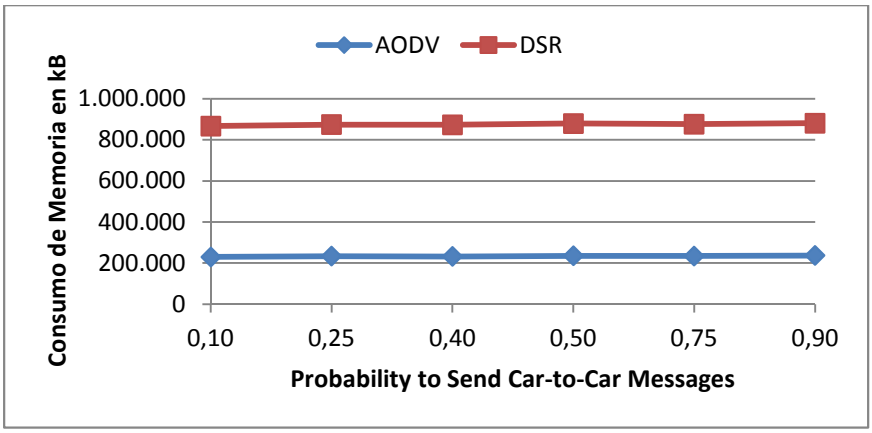


Figura 12.48: Consumo de Memoria en JiST/SWANS con RSUs

12.1.9 Desempeño de Protocolos de Enrutamiento en ns-3 sin RSUs

El objetivo de esta prueba es evaluar el comportamiento de los protocolos de enrutamiento MANET en el simulador de red ns-3 en una red realista con carreteras derivadas de mapas reales en el cual se forma una red vehicular sin la presencia de RSUs. Es el mismo mapa utilizado en la pruebas de City Benchmark. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 120 segundos.
- Número de vehículos: 400.
- Número de RSUs: 0.
- Estándar WiFi: 802.11a.
- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.

- Protocolo de enrutamiento: AODV, DSDV y OLSR.

El parámetro a variar en esta prueba es el intervalo para el de envío de mensajes de los vehículos, dicho de otra manera, lo que se varía en cada simulación es la cantidad de mensajes que envían los vehículos por segundo. La idea es observar cómo influye la variación de este parámetro en el comportamiento de la red simulada por cada protocolo de enrutamiento. Cada simulación se realizó con los siguientes intervalos para el envío de mensajes: 1, 0.2, 0.1, 0.05, 0.04, 0.02 segundos o lo que es el equivalente a enviar 1, 5, 10, 20, 25 y 50 datagramas UDP por segundo.

OLSR presenta un PDR alto respecto a los otros dos protocolos, como se puede observar en la Figura 12.49. Esto es debido al bajo NRL que reporta este protocolo como se puede observar en la Figura 12.52. AODV y DSDV se mantienen por debajo del 2% en PDR debido a que solo aseguran que destinos que estén a solo un salto reciban mensajes de datos. Además, estos dos protocolos poseen un NRL alto comparado con el que reporta OLSR.

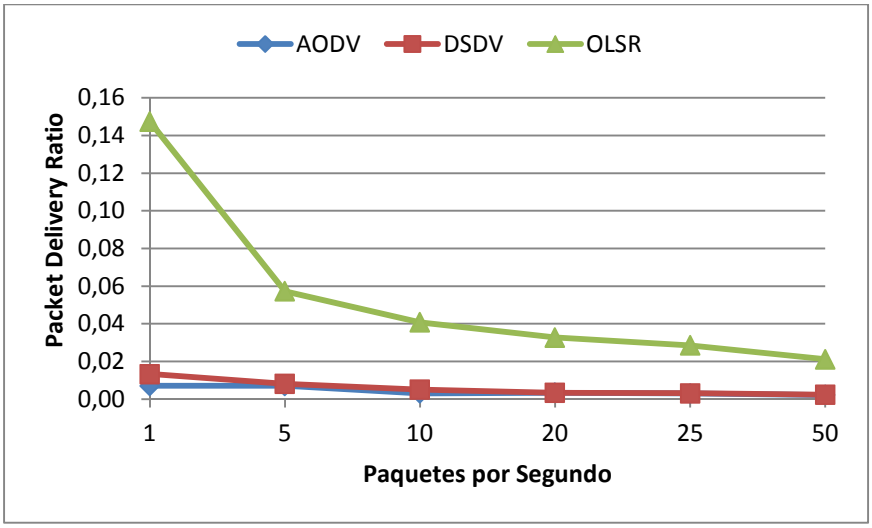


Figura 12.49: Packet Delivery Ratio en ns-3 sin RSUs

En la Figura 12.50 se puede apreciar el end-to-end delay para los tres protocolos estudiados en este simulador. DSDV presenta un mayor delay, lo que puede estar afectando al PDR reportado para este protocolo. AODV presenta el menor delay pero dado a su alto NRL ocasiona que sólo destinos a un salto reciban mensajes de datos y por eso el bajo delay reportado.

OLSR reporta un número de saltos promedio mayor que los otros protocolos, asegurando que funciona bien bajo este escenario ya que destinos que están a más de 3 saltos pueden recibir mensajes de datos, como se puede observar en la Figura 12.51. En cambio, AODV y DSDV solo pueden hacer entregas en promedio a destinos que estén entre uno y dos saltos, lo que ocasiona un bajo PDR y un alto NRL para estos dos protocolos.

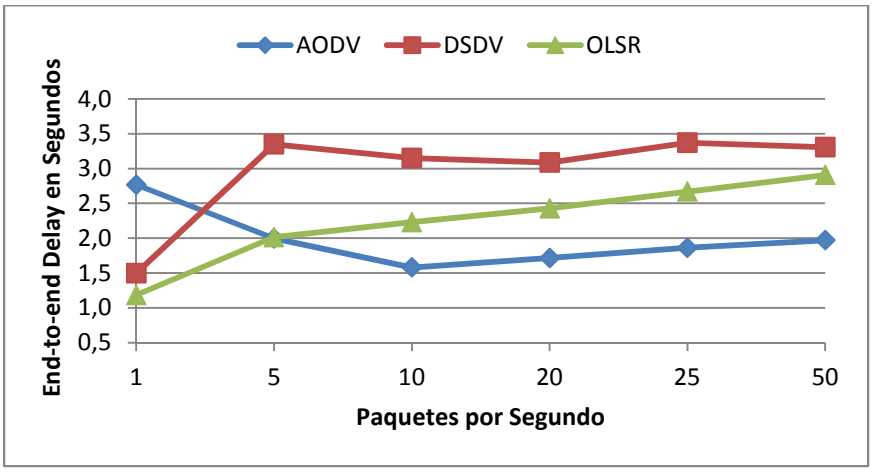


Figura 12.50: End-to-End Delay Promedio en ns-3 sin RSUs

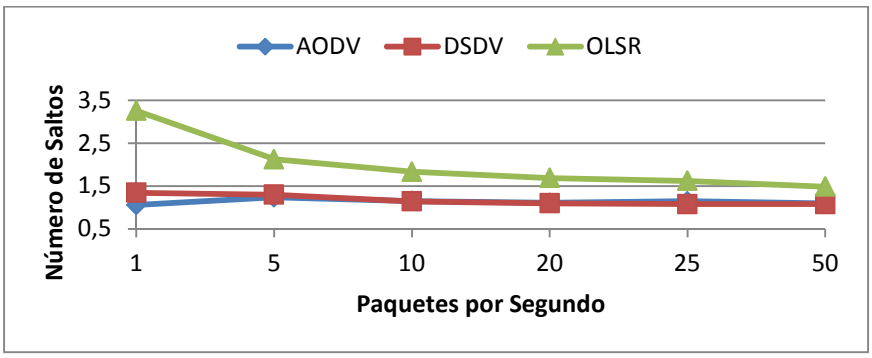


Figura 12.51: Número de Saltos Promedio en ns-3 sin RSUs

OLSR reporta un NRL demasiado bajo, en promedio de 4 mensajes de control por cada mensaje de datos entregado, como se puede apreciar en la Figura 12.52. Este comportamiento muestra que OLSR no es un modelo que se apega mucho a la realidad comparado con los otros dos protocolos que reportan un NRL acorde a lo que sucede en el escenario de la simulación. AODV posee un alto NRL pero una posible causa de esto es el bug que presenta este modelo ya mencionado en pruebas anteriores.

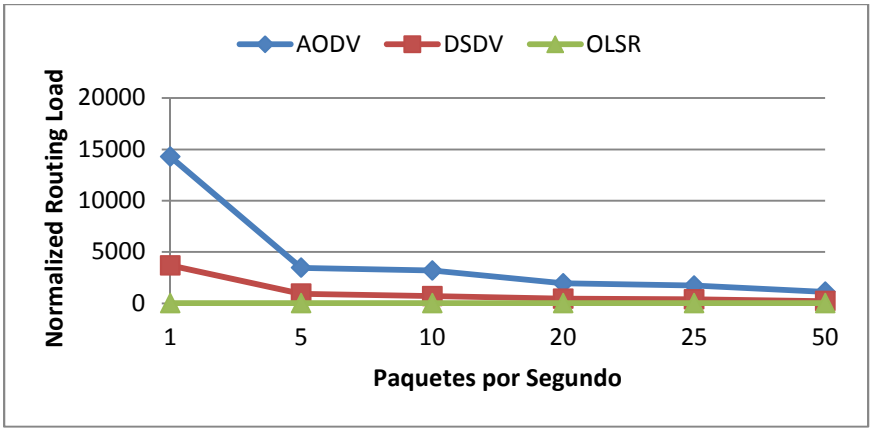


Figura 12.52: Normalized Routing Load en ns-3 sin RSUs

En la Figura 12.53 se puede observar el tiempo real de ejecución para cada uno de los protocolos estudiados. AODV reporta un tiempo mayor dado al bug que posee este modelo. OLSR es el protocolo que menos tiempo consume.

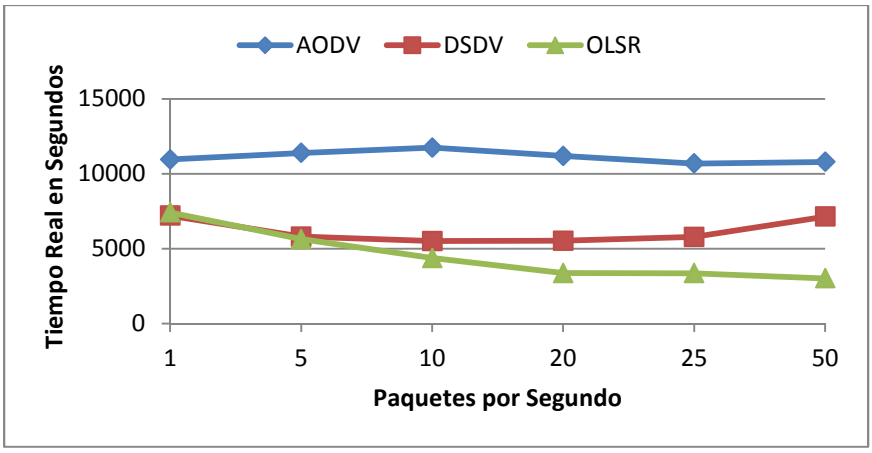


Figura 12.53: Tiempo Real de Simulación en ns-3 sin RSUs

En la Figura 12.54 se observa el consumo de memoria reportado por los protocolos estudiados. AODV en este aspecto consume menos memoria, mientras que DSDV va en aumento a medida que la cantidad de paquetes por segundo aumenta.

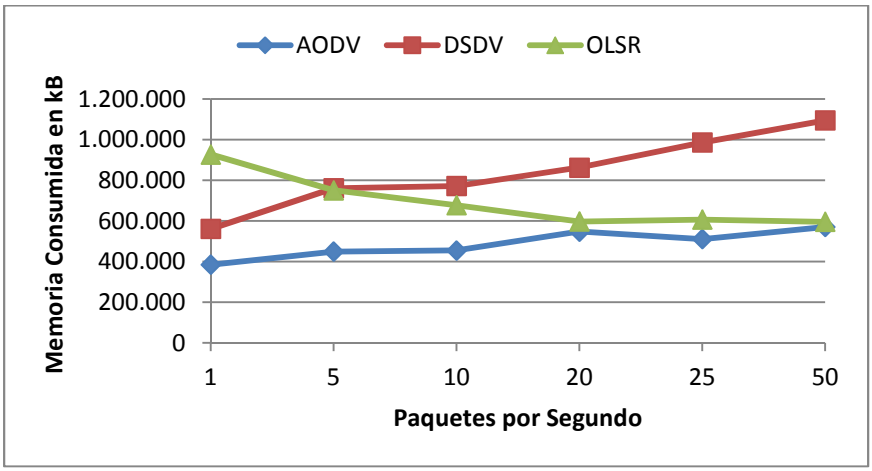


Figura 12.54: Memoria Consumida en ns-3 sin RSUs

12.1.10 Desempeño de Protocolos de Enrutamiento en ns-3 con RSUs

Al igual que en la Prueba para el Desempeño de Protocolos de Enrutamiento en ns-3 sin RSUs, esta prueba consiste en evaluar el desempeño de los protocolos de enrutamiento MANETs en este simulador de red pero ahora con la presencia de RSUs. Se ubican 10 RSUs siguiendo lo realizado en la prueba City Benchmark sin RSUs, utilizando la misma traza. Los parámetros más relevantes de valores constantes configurados para este benchmark son:

- Tiempo de duración de la simulación: 180 segundos.
- Número de vehículos: 400.
- Número de RSUs: 10.
- Estándar WiFi: 802.11a.

- Radio frecuencia: 5 GHz.
- Rango de propagación de la señal: 300 metros.
- Tamaño del payload de UDP: 512 bytes.
- Bitrate: 12 Mbps.
- Intervalo de envío de mensajes para los vehículos: 0.1 segundos.
- Intervalo de envío de mensajes para los RSUs: 0.01 segundos.
- Protocolo de enrutamiento: AODV, DSDV y OLSR.

El parámetro a variar en esta prueba es la probabilidad de enviar un mensaje de un vehículo a otro (car-to-car messages). De esta manera se realizaron corridas con las siguientes probabilidades: 0.1, 0.25, 0.4, 0.5, 0.75, 0.9.

OLSR vuelve a reportar un alto PDR que va en aumento a medida que la probabilidad aumenta, como se puede apreciar en la Figura 12.55. Esto se debe a que los vehículos tienen más chance de estar rodeados por otros vehículos que estar cerca de un RSU. Esta característica no es aprovechada ni por DSDV ni por AODV dado que el primero presenta un alto delay que está afectando en su PDR (como se puede observar en la Figura 12.56) y AODV por el bug que este modelo posee y que ocasiona un alto número de mensajes de control del protocolo.

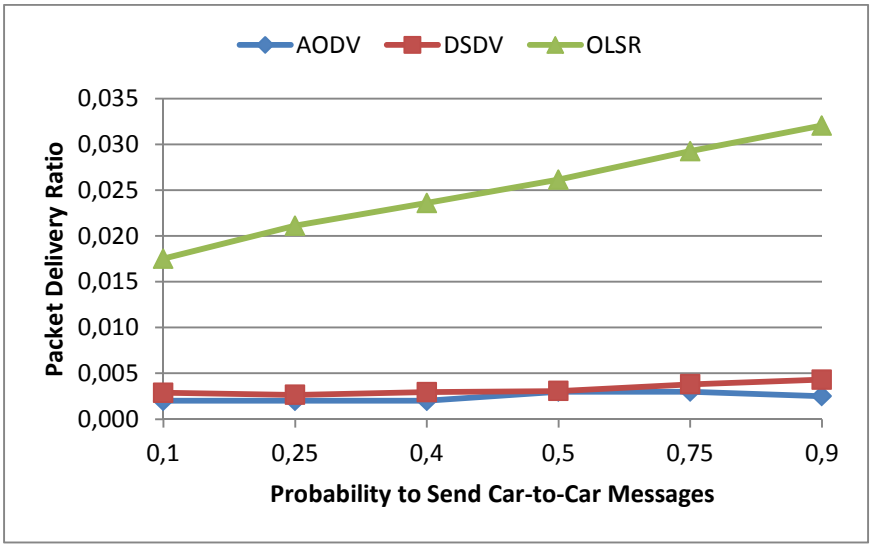


Figura 12.55: Packet Delivery Ratio en ns-3 con RSUs

Como ya se mencionó, nuevamente DSDV presenta un delay superior respecto a los otros dos protocolos (Figura 12.56). AODV presenta un delay bajo pero esto es ocasionado a que solo los destinatarios que están a un salto son los que están recibiendo mensajes de datos provocado por el alto NRL que posee el modelo de AODV en ns-3.

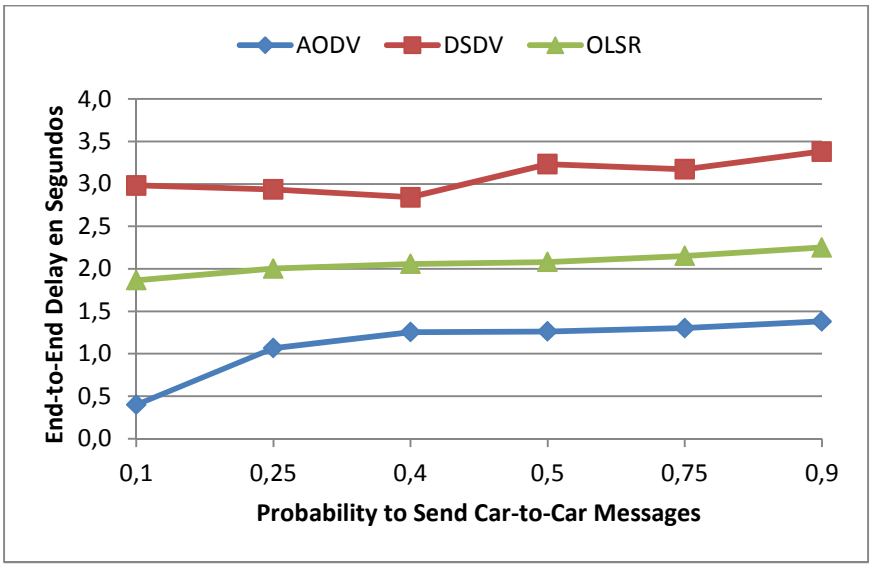


Figura 12.56: End-to-End Delay Promedio en ns-3 con RSUs

OLSR es el protocolo que mejor funciona en este escenario dado que asegura que los mensajes de datos puedan llegar a destinos que estén a más de dos saltos. En cambio, AODV y DSDV en líneas generales solo aseguran la entrega de mensajes a destinos que estén a un salto, como se observa en la Figura 12.57.

AODV reporta de nuevo el mayor NRL debido al bug que presenta este modelo en ns-3, como se puede apreciar en la Figura 12.58. Nuevamente OLSR reporta un NRL demasiado bajo que hace dudar del realismo de este modelo en ns-3 (en promedio 5 mensajes de control por cada mensaje de datos entregado).

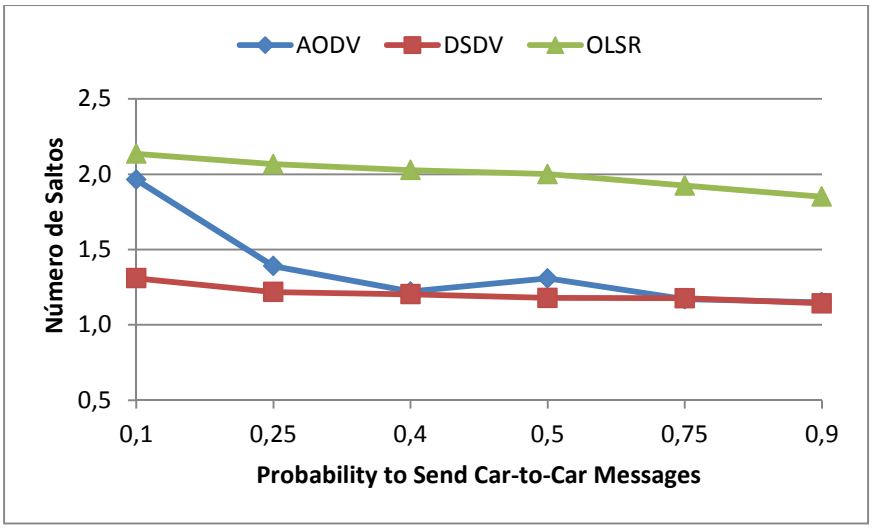


Figura 12.57: Número de Saltos Promedio en ns-3 con RSUs

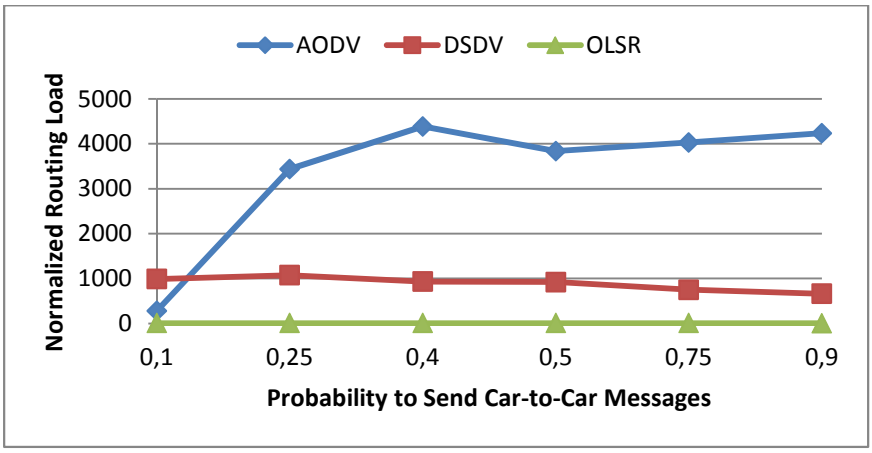


Figura 12.58: Normalized Routing Load en ns-3 con RSUs

En la Figura 12.59 se puede apreciar el tiempo real de ejecución para los protocolos estudiados. Debido al bug que presenta AODV, este protocolo reporta el mayor tiempo de ejecución comparado con los otros dos protocolos que prácticamente permanecen invariantes. Por último, en la Figura 12.60 se observa la memoria consumida, donde nuevamente AODV posee un menor consumo de memoria a pesar de su bug.

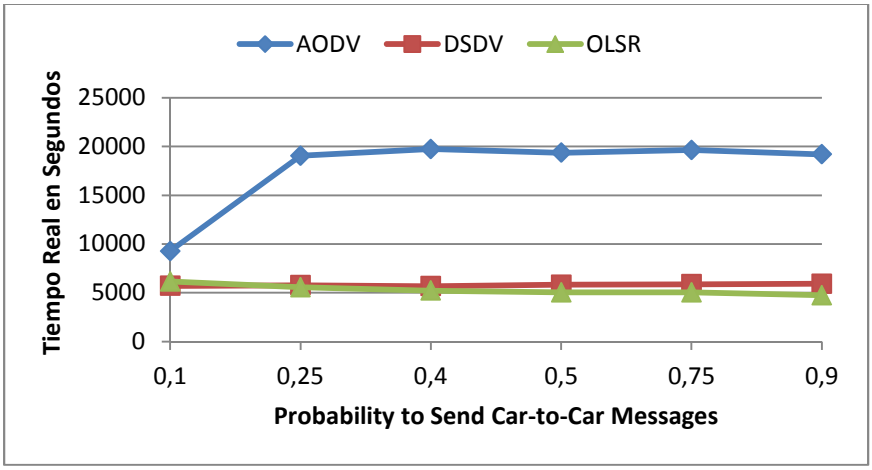


Figura 12.59: Tiempo Real de Simulación en ns-3 con RSUs

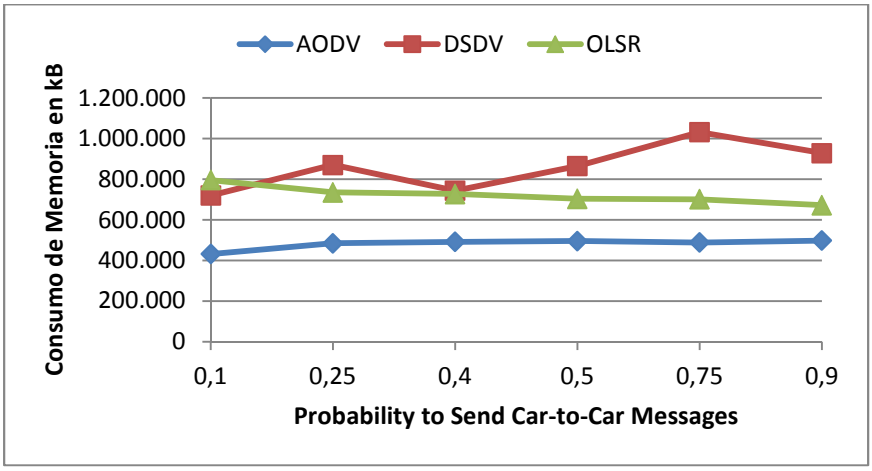


Figura 12.60: Memoria Consumida en ns-3 con RSUs

13. Especificaciones Técnicas

El conjunto de benchmarks propuestos fue implementado para los simuladores de red ns-3, OMNeT++ y JiST/SWANS bajo la distribución GNU/Linux basada en Debian 6 (Squeeze) para arquitectura AMD64.

Las pruebas fueron realizadas en equipos HP xw4600 con procesadores Intel Core 2 Duo E6750 CPU de 2.67 GHz y 4 GB de RAM.

Para la ejecución correcta de los benchmarks es necesario contar con la versión adecuada de los simuladores de red, frameworks y aplicaciones adicionales como es descrito en la Tabla 13.1:

Simuladores	Versiones, Frameworks y Aplicaciones Adicionales
OMNeT++	OMNeT++ versión 4.2.2 INETMANET Framework versión 2.0.0 top versión 3.2.8 beep-1.2.2 (opcional)
JiST/SWANS	js-uulm-allinone-1.2 jist-swans-1.0.6 (JiST versión 1.06 y SWANS versión 1.0.6) jdk1.6.0_34 top versión 3.2.8 beep-1.2.2 (opcional)
ns-3	ns-3 versión 3.16 NetAnim 3.0 top versión 3.2.8 beep-1.2.2 (opcional)

Tabla 13.1: Especificaciones Técnicas Simuladores de Red

Las trazas generadas en formato ns-2 fueron creadas utilizando SUMO versión 0.15.0 y VanetMobiSim versión 1.1.

Para la herramienta ns-2 Trace Toolkit se utilizó el IDE NetBeans en su versión 6.9.1, jdk versión 1.6.0_26 y el paquete JFreeChart versión 1.0.14.

14. Conclusiones y Trabajos Futuros

La tecnología de las redes vehiculares es prometedora y fascinante debido a la gran cantidad de aplicaciones y soluciones que se lograrían obtener con respecto a diversas problemáticas inherentes al tráfico vehicular. Dado que la experimentación real es muy costosa para este tipo de redes, la simulación es una solución práctica para estudiar y analizar redes de comunicaciones gracias a su disponibilidad y reducción de costos económicos.

Las herramientas de simulación son cada vez más realistas y representan un punto de partida para experimentar con nuevas tecnologías. Dada la variedad de simuladores que se encuentran a disposición fue realizado un estudio de sus características, ventajas y desventajas. Es importante conocer el alcance de un simulador para determinar su escalabilidad para un escenario específico a simular.

Las redes vehiculares representan un desafío por su alta escalabilidad y variabilidad de la conectividad en espacio y tiempo. Por lo tanto, colocar en práctica experimentos relacionados con las tecnologías a utilizarse en las redes vehiculares no es una tarea fácil. La simulación de este tipo de redes representa la solución ideal para el estudio de las redes vehiculares. Gracias a la utilización de herramientas para la generación de trazas vehiculares fue posible simular el tráfico sobre mapas de carretas existentes en los simuladores de red ns-3, OMNeT++ y JiST/SWANS.

Debido a las características que tienen las redes vehiculares, es de suma importancia realizar simulaciones de escenarios con gran cantidad de nodos móviles, gran movilidad, obstáculos, etc. Para ello, fueron desarrollados un conjunto de benchmarks con los cuales se hizo una evaluación del desempeño de los simuladores de red ns-3, OMNeT++ y JiST/SWANS logrando determinar sus capacidades y limitaciones en lo que respecta a la simulación de redes vehiculares.

Basado en el estudio realizado, se puede concluir que OMNeT++ es un simulador que a nivel de consumo de memoria y tiempo de ejecución es muy similar a JiST/SWANS, simulador que maneja muy bien la memoria y es considerado como uno de los más rápidos en ejecución. La implementación de DYMO y AODV fueron las que arrojaron mejores resultados, siendo DYMO superior a AODV. BATMAN mostró un desempeño muy pobre. Los otros modelos de protocolos de enrutamiento soportados (OLSR, DSR, DSDV, etc) fueron probados y descartados debido a su baja escalabilidad, con más de 200 nodos por ejemplo, estos modelos fallan en la realización de las pruebas planteadas y no permiten que la simulación termine.

Para JiST/SWANS se concluye que es un simulador eficiente en cuanto a utilización de memoria y tiempo de ejecución pero bastante limitado a nivel de modelos para la simulación de redes VANETs. El desempeño de DSR se mostró más adecuado para redes vehiculares que el mostrado por AODV, AODV solo supera a DSR en el uso de la memoria siendo bastante elevado para DSR a pesar del buen uso con el cual JiST/SWANS realiza el manejo de memoria.

En el caso de ns-3 se concluye que es un simulador bastante completo a nivel de modelos para simular redes vehiculares y que también maneja bien el uso de la memoria y el tiempo de ejecución con los modelos de OLSR y DSDV. El modelo de AODV contiene bugs que hacen que sea uno de los modelos que peores resultados arrojó junto con la implementación de BATMAN en OMNeT++. OLSR fue el modelo del protocolo de enrutamiento con el que se obtuvieron mejores resultados y al igual que en OMNeT++, el modelo de DSR no es escalable.

Con los experimentos realizados, queda demostrado que los protocolos de enrutamiento para MANETs no son adecuados para ser utilizados en redes vehiculares. Ya sea porque les toma tiempo

establecer una ruta, por inconsistencia en la tabla de ruta de nodos intermedios, porque si existen muchos nodos la información de la ruta en la cabecera va a generar mucha sobrecarga simplemente porque cuando estos protocolos fueron diseñados no estaban pensados para operar en redes con alto dinamismo como lo son las redes vehiculares.

Como trabajos futuros, se propone implementar este conjunto de benchmarks en otros simuladores populares como ns-2, OPNET o NCTUns. También es posible desarrollar implementaciones más completas de capa de aplicación como por ejemplo para el manejo de tráfico vehicular o de seguridad mediante la simulación de accidentes las cuales se ejecuten sobre este mismo conjunto de benchmarks. Adicionalmente, el desarrollo de nuevos protocolos de enrutamiento adecuados a las redes vehiculares es también una opción viable de trabajo futuro.

Referencias Bibliográficas

[1] K. Beck and C. Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional. Segunda Edición. Noviembre 2004.

[2] A. Cockburn. Agile Software Development. Addison-Wesley Professional. Octubre 2001.

[3] S. Ali and A. Ali. Performance Analysis of AODV, DSR and OLSR in MANET. Department of Computing at Blekinge Institute of Technology Sweden. 2009.

[4] J. Bernsen and D. Manivannan. Greedy Routing Protocols for Vehicular Ad Hoc Networks. University of Kentucky. Agosto 2008.

[5] CAN Specification Versión 2.0. Bosch. Septiembre 1991.

[6] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol. RFC 3626. Octubre 2003

[7] A. Ehliar and D. Liu. Benchmarking Network Processors. Dept. of Electrical Engineering. Linköping University. 2006.

[8] R. Fujimoto, K. Perumalla, A. Park, H. Wu, M. Ammar, and G. Riley. Large-Scale Network Simulation: How Big? How Fast?. Georgia Institute of Technology, Atlanta. 2003.

[9] E. Gamess. Arquitectura para la Construcción Orientada por Rendimiento de Metasistemas. Tesis Doctoral. Universidad Central de Venezuela. Junio 2000.

[10] C. Gawron. Simulation-Based Traffic Assignment: Computing User Equilibria in Large Street Networks. Febrero 1999.

[11] M. Guizani, A. Rayes, B. Khan, and A. Al-Fuqaha. Network Modeling and Simulation. A Practical Perspective. Wiley. Primera Edición. Abril 2010.

[12] H. Guo. Automotive Informatics and Communicative Systems: Principles in Vehicular Networks and Data Exchange. Information Science Reference. Primera Edición. Marzo 2009.

[13] J. Härri, F. Filali, and C. Bonnet. VanetMobiSim: Generating Realistic Mobility Patterns for VANETs. Institut Eurécom. Department of Mobile Communications. Septiembre 2006.

[14] J. Härri and F. Filali. VanetMobiSim – Vehicular Ad hoc Network Mobility Extension to the CanuMobiSim Framework. Institut Eurécom/Politecnico di Torino. 2006.

[15] J. Härri, M. Flore, F. Filali, and C. Bonnet. Vehicular Mobility Simulation with VanetMobiSim. The Society for Modeling and Simulation International. 2009.

[16] T. Henderson, G. Riley, and M. Lacage**.** Network Simulations with the ns-3 Simulator. Agosto 2008.

[17] T. Henderson, G. Riley, and M. Lacage**.** ns-3 Overview. nsam.org. Mayo 2011.

[18] T. Henderson, G. Riley, and S. Floyd**.** ns-3 Project Goals. Junio 2006.

[19] T. Henderson, G. Riley, and S. Floyd**.** Project Description. Agosto 2005.

[20] C. Huang and Y. Chen. Telematics Communication Technologies and Vehicular Networks: Wireless Architectures and Applications. Information Science Reference. Primera Edición. Diciembre 2009.

[21] IEEE 1609 - Family of Standards for Wireless Access in Vehicular Environments (WAVE). Septiembre 2009.

[22] IEEE 802.11p - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Amendment 6: Wireless Access in Vehicular Environments. Julio 2010.

[23] T. Issariyakul and E. Hossain. Introduction to Network Simulator ns-2. Springer. Julio 2008.

[24] H. Kuan Ong, H. Loong Ong, E. Tee, and R. Sureswaran. Scalability Study of Ad Hoc Wireless Mobile Network Routing Protocol in Sparse and Dense Networks. University Sains Malaysia. Mayo 2006.

[25] M. Lacage. Experimentation with ns-3. Agosto 2009.

[26] M. Lusheng and D. Karim. A Survey of IEEE 802.11p MAC Protocol. Tshwane University of Technology. Septiembre 2011.

[27] F. Martinez, C. Keong Toh, J. Cano, C. Calafate, and P. Manzoni. A Survey and Comparative Study of Simulators for Vehicular Ad Hoc Networks (VANETs). John Wiley & Sons, Ltd. Octubre 2009.

[28] H. Moustafa and Y. Zhang. Vehicular Networks: Techniques, Standards and Applications. CRC Press. Primera Edición. Abril 2009.

[29] T. Murray, M. Cojocari, and H. Fu. Measuring the Performance of IEEE 802.11p Using ns-2 Simulator for Vehicular Networks. Oakland University, Rochester Hills. Mayo 2008.

[30] V. Naoumov and T. Gross. Simulation of Large Ad Hoc Networks. Departement Informatik. ETH Zürich. Septiembre 2003.

[31] D. Nicol. Comparison of Network Simulators Revisited. Dartmouth College. Mayo 2002.

[32] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561. Julio 2003.

[33] R. Popescu-Zeletin, I. Radusch, and A. Rigani. Vehicular-2-X Communication, State-of-the-Art and Research in Mobile Vehicular Ad Hoc Networks. Springer. Primera Edición. Mayo 2010.

[34] E. Schoch, M. Feiri, F. Kargl, and M. Weber. Simulation of Ad Hoc Networks: ns-2 Compared to JiST/SWANS. Ulm University, Germany. Marzo 2008.

[35] TIGER/Line Files - Technical Documentation. First Edition. US Census Bureau. Enero 2005.

[36] A. Vargas. OMNeT++ User Manual version 4.2. OpenSim Ltd. Noviembre 2011.

[37] A. Vargas and R. Hornig. An Overview of the OMNeT++ Simulation Enviroment. Febrero 2008.

[38] K. Wehrle, M. Günes, and J. Gross. Modeling and Tools for Network Simulation. Springer. Febrero 2010.

[39] E. Weingärtner, H. vom Lehn, and K. Wehrle. A Performance Comparison of Recent Network Simulators. Aachen University, Germany. Junio 2009.

[40] R. Barr. JiST - Java in Simulation Time User Guide. Septiembre 2003.

[41] Rimon Barr. SWANS - Scalable Wireless Ad hoc Network Simulator User Guide. Marzo 2004.