



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE UN FRAMEWORK INTEGRADO PARA
SIMULACIONES DE SISTEMAS INTELIGENTES DE TRANSPORTE EN OMNET ++
Y PARAMICS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MANUEL OSVALDO J. OLGUÍN MUÑOZ

PROFESOR GUÍA:
SANDRA CÉSPEDES U.

MIEMBROS DE LA COMISIÓN:
JAVIER BUSTOS
NANCY HITSCHFELD

Este trabajo ha sido parcialmente financiado por NIC Chile Research Labs

SANTIAGO DE CHILE
JULIO 2017

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: MANUEL OSVALDO J. OLGUÍN MUÑOZ
FECHA: JULIO 2017
PROF. GUÍA: SANDRA CÉSPEDES U.

DISEÑO E IMPLEMENTACIÓN DE UN FRAMEWORK INTEGRADO PARA
SIMULACIONES DE SISTEMAS INTELIGENTES DE TRANSPORTE EN OMNET ++
Y PARAMICS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Den här är för Aros och Skatt.

Agradecimientos

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Tabla de Contenido

1. Introducción	1
1.1. Motivación	1
1.2. Estado del arte	2
2. Marco Teórico y Estado del Arte	3
2.1. Marco Teórico	3
2.1.1. Sistemas de Transporte	3
2.1.2. Simulaciones de Eventos Discretos	4
2.1.3. Simulación de Redes de Comunicaciones	4
2.1.4. Simulación de Tráfico	5
2.1.5. Simulación Bidireccional	5
2.2. Estado del Arte	6
2.2.1. Simuladores de Tráfico	6
2.2.2. Simuladores de Redes de Comunicaciones	6
2.2.3. Entornos de Simulación Bidireccional	6
3. Conceptos clave	9
3.1. Paramics	9
3.1.1. API Plugins	9
3.2. OMNeT++	9
3.3. TraCI	9
3.3.1. Diseño	10
4. Comunicando Paramics con OMNeT++ mediante TraCI	14
4.1. Diseño Arquitectural	14
4.2. Funcionalidad Implementada	14
4.2.1. Comandos Implementados	15
4.2.2. Comandos de modificación de estado	17
4.3. Módulos Principales	18
4.3.1. plugin.c	18
4.3.2. TraCIServer	19
4.3.3. Simulation	29
4.3.4. VehicleManager	30
A. Códigos	36

Índice de Tablas

Índice de Ilustraciones

2.1. Evolución de simulaciones integradas para ITS (fuente: [15]).	7
3.1. Formatos de mensajes TraCI	10
3.2. Ejemplo solicitud de variable TraCI.	11
3.3. Flujo de comunicación TraCI.	12
4.1. Arquitectura del Framework	15
4.2. Estructura de archivos del código fuente del framework.	16
4.3. Diagrama de herencia, VariableSubscription	24

Capítulo 1

Introducción

1.1. Motivación

Los sistemas de transporte conforman la columna vertebral de nuestras ciudades, contribuyendo directamente al desarrollo de la sociedad urbana. Un sistema de transporte bien diseñado y eficiente permite el desplazamiento rápido y cómodo de personas y bienes; en cambio, uno ineficiente genera grandes problemas, alargando los tiempos de viaje y aumentando la contaminación atmosférica. Los Sistemas de Transporte Inteligente (ITS, por sus siglas en inglés: *Intelligent Transportation Systems*) surgen como una respuesta a la necesidad de optimización y modernización de los sistemas de transporte existentes. La Unión Europea define a los ITS como aplicaciones avanzadas que, sin incorporar inteligencia como tal, pretenden proveer servicios innovadores relacionados con distintos modos de transporte y de administración de tráfico, que además otorgan información a los usuarios, permitiéndoles utilizar el sistema de transporte de manera más segura, coordinada e inteligente [1]. De acuerdo al Departamento de Transportes de los EEUU¹, estos sistemas se pueden dividir en dos grandes categorías: sistemas de infraestructura inteligente y sistemas de vehículos inteligentes.

Los sistemas de infraestructura inteligente tienen como enfoque el manejo de los sistemas de transporte a niveles macro, y la transmisión de información oportuna a los usuarios. Esta categoría incluye, entre otros, sistemas de advertencia y señalización dinámica en ruta (ya sea a través de pantallas o sistemas de comunicación inalámbrica), sistemas de pago electrónico y de coordinación del flujo de tráfico.

Por otro lado, la categoría de sistemas de vehículos inteligentes engloba todo aquello relacionado con la automatización y optimización de la operación de un vehículo. Dentro de esta categoría se incluyen sistemas de advertencia y prevención de colisiones, de asistencia al conductor — por ejemplo, sistemas de navegación — y control autónomo de vehículos.

El factor común entre ambas categorías es la necesidad de extraer información en tiempo

¹Office of the Assistant Secretary for Research and Technology (OST-R), <http://www.itsoverview.its.dot.gov/>

real desde el entorno, la cual debe procesarse y en muchos casos generar una respuesta a transmitir al usuario. Para este fin, se ha propuesto la implementación de tecnologías que posibiliten esta comunicación, principalmente utilizando redes inalámbricas, tanto de área local (los estándares incluidos en la familia WLAN, IEEE 802.11), como de área personal (WPAN, IEEE 802.15) [2]-[4]. Sin embargo, estas tecnologías fueron diseñadas originalmente para su uso en redes estáticas o con patrones de movimiento muy limitados, y es necesaria la evaluación de su desempeño en entornos altamente dinámicos como lo son los sistemas de transporte vehicular. Parámetros críticos para el funcionamiento óptimo de la red, como la potencia de transmisión, las condiciones del canal de transmisión y la distancia óptima entre nodos, deben establecerse teniendo en cuenta las particularidades que presentan los sistemas de transporte — por ejemplo, la alta congestión de nodos en intersecciones con semáforos.

Existe entonces hoy en día la necesidad de modelar de manera realista y precisa el comportamiento de estas tecnologías en contextos de comunicaciones inalámbricas en redes vehiculares. Por otro lado, existe también la necesidad de modelar el impacto de la comunicación inalámbrica en un sistema de transporte, y cómo esta puede contribuir a optimizar la operación del mismo [5]. Un ejemplo de esto son los Sistemas Avanzados de Información al Viajero (*ATIS*, por sus siglas en inglés; *Advanced Traveller Information System*) los cuales proveen información en tiempo real sobre las condiciones del tránsito a conductores, permitiéndoles elegir la ruta más óptima para alcanzar su destino. Este *feedback* inmediato sin duda tiene efectos importantes en el flujo vehicular de un sistema de transportes, los cuales deben ser tomados en consideración al momento de modelar y simular el funcionamiento del mismo.

1.2. Estado del arte

Capítulo 2

Marco Teórico y Estado del Arte

2.1. Marco Teórico

En esta sección se detallarán los conceptos esenciales para la comprensión del presente trabajo de memoria.

2.1.1. Sistemas de Transporte

Cascetta define en [6] a los sistemas de transporte como aquella combinación de elementos que generan demanda de viaje en un cierta área geográfica, y que otorgan los servicios de transporte para suplir dicha demanda. Esta definición es amplia y otorga una visión general del concepto. En la práctica, en la presente memoria se denominará como sistema de transporte a aquel conjunto de infraestructura vial que permite el flujo de vehículos desde uno o más puntos de origen a uno o más puntos de destino.

Sistemas Inteligentes de Transporte

Los Sistemas Inteligentes de Transporte (en adelante *ITS*, por sus siglas en inglés – *Intelligent Transportation Systems*) surgen como una respuesta a la necesidad de optimización y modernización de sistemas de transporte existentes. La Unión Europea define a los ITS como aplicaciones avanzadas que, sin incorporar inteligencia como tal, pretenden proveer servicios innovadores relacionados con distintos modos de transporte y de administración de tráfico, que además otorgan información a los usuarios, permitiéndoles utilizar el sistema de transporte de manera más segura, coordinada e inteligente [1]. De acuerdo al Departamento de Transportes de los EEUU, estos sistemas se pueden dividir en dos grandes categorías [7]:

Sistemas de Infraestructura Inteligente Tienen como enfoque el manejo de los sistemas de transporte a niveles macro, y la transmisión de información oportuna a los usuarios. Esta categoría incluye, entre otros, sistemas de advertencia y señalización dinámica en

ruta (ya sea a través de pantallas o sistemas de comunicación inalámbrica), sistemas de pago electrónico y de coordinación del flujo de tráfico.

Sistemas de Vehículos Inteligentes Engloba todo aquello relacionado con la automatización y optimización de la operación de un vehículo. Dentro de esta categoría se incluyen sistemas de advertencia y prevención de colisiones, de asistencia al conductor — por ejemplo, sistemas de navegación — y control autónomo de vehículos.

2.1.2. Simulaciones de Eventos Discretos

Se denominan *simulaciones de eventos discretos* a la categoría de simulaciones en las cuales el estado del modelo cambia en instantes de tiempo discreto [8]. Este tipo de simulaciones tienen diversos usos, siendo dos de los principales (y de interés para el presente trabajo de memoria) las simulaciones de redes de comunicaciones y de tráfico.

2.1.3. Simulación de Redes de Comunicaciones

Las simulaciones de redes de comunicaciones tienen como fin modelar el comportamiento de sistemas interconectados mediante tecnologías de comunicaciones, sean estas cableadas o no. Para el fin del presente trabajo, por razones evidentes ligadas a la naturaleza de las comunicaciones dentro de un sistema altamente dinámico como lo son los sistemas de transporte, se consideraron únicamente sistemas de comunicación inalámbrica.

Comunicación Inalámbrica

En el contexto de la presente memoria, se entenderá por *comunicación inalámbrica* todo acto de transmisión de información entre dos o más entidades mediante la interacción con un campo electromagnético, sin otra conexión física entre dichas entidades (*e.g.* cables). Estas entidades denominarán *nodos*, y al establecerse una configuración que permita la comunicación inalámbrica entre múltiples nodos cercanos, se hablará de una *red inalámbrica*.

La simulación de una red de comunicaciones inalámbrica consiste en tres etapas principales [9]:

1. El ingreso de parámetros del funcionamiento de la red (potencia de transmisión, nivel de ruido, etc).
2. Un sistema de emulación del movimiento de información en la red, a través de la simulación del funcionamiento físico de las radios.
3. Finalmente, la obtención de resultados y métricas que indiquen la eficiencia de la red en términos de pérdidas de paquetes, el *throughput* (cantidad de datos correctamente transmitidos), etc.

2.1.4. Simulación de Tráfico

Se entenderá por *Simulación de Tráfico* aquel entorno virtual que permita la emulación y estudio del comportamiento de un sistema de transporte ficticio o real, mediante el modelamiento de éste utilizando herramientas computacionales. Estas simulaciones puede ser tanto discretas como continuas.

A continuación, se describirán brevemente las tres principales categorías de modelos de tráfico utilizados actualmente en academia; microscópicos, macroscópicos y mesoscópicos [9]-[11].

Microscópicos Los modelos microscópicos de tráfico modelan de manera particular cada entidad (vehículo, peatón, etc) en la red. Cada entidad tiene su propio origen, destino, velocidad y posición (y otras propiedades adicionales), y su comportamiento se modela de manera individual con respecto al resto de la red.

Macroscópicos En contraste con los modelos microscópicos, los modelos macroscópicos simulan el movimiento de entidades dentro de una red de tráfico como flujos en vez de movimientos particulares.

Mesoscópicos Finalmente, los modelos mesoscópicos consideran aspectos de ambos modelos anteriormente mencionados, simulando particularmente el comportamiento de las entidades pero también considerando su movimiento dentro de un flujo general.

La presente memoria considera únicamente la integración de una simulación de tipo microscópica, dada su fácil adaptación al modelo utilizado por las simulaciones de comunicaciones inalámbricas – un nodo en la red de comunicaciones corresponde directamente a un vehículo en el sistema de transporte.

2.1.5. Simulación Bidireccional

En el contexto de integración de simuladores de comunicaciones y de tráfico para el estudio de Sistemas Inteligentes de Transporte, se entenderá por *Simulación Bidireccional* aquél entorno de simulación en que un simulador de redes de comunicación y otro de tráfico se ejecuten de manera paralela, cada uno obteniendo *feedback* continuo del otro.

2.2. Estado del Arte

2.2.1. Simuladores de Tráfico

SUMO

Quadstone Paramics

2.2.2. Simuladores de Redes de Comunicaciones

NS2

OMNeT++

2.2.3. Entornos de Simulación Bidireccional

A continuación se resumirá brevemente el estado del arte en el tema de simulación bidireccional para simulaciones de Sistemas Inteligentes de Transporte.

Simulaciones unidireccionales

De acuerdo a Sommer *et al.*, gran parte de las simulaciones de comunicaciones inalámbricas en ITS se hacen a través de la importación de trazas de movimiento reales desde simuladores de transporte, de manera unidireccional. Dichas trazas se pueden generar de dos maneras: *offline*, es decir, aisladamente en el simulador de transporte, para luego ser exportadas en un formato que el simulador de red sea capaz de interpretar, y *decoupled online*, de manera que el simulador de transporte genere las trazas en tiempo real y el simulador de red simplemente las “consume”. Sin embargo, si bien este método permite analizar el efecto del modelo de movimiento de un sistema de transporte en las comunicaciones inalámbricas, es incapaz de reflejar el impacto de la propagación de información del estado del tráfico en el modelo mismo. Es decir, esta metodología no es útil para la simulación de, por ejemplo, sistemas de advertencia de accidentes o de asistencia al conductor, puesto que las trazas de movimiento están predefinidas o se generan sin considerar los resultados de esta comunicación. Este tipo de simulación, si bien es útil para ciertos análisis específicos, no constituye una simulación bidireccional y no abarca la totalidad del problema.

Los trabajos realizados por investigadores de la Universidad Jiao Tong de Shanghai en [12], [13] son ejemplos de esta modalidad. Para estas investigaciones, los autores obtuvieron trazas reales de movimiento de *SUVnet*, una red vehicular compuesta por aproximadamente 4000 taxis en la ciudad de Shanghai. Estas trazas luego fueron simplemente utilizadas en simulaciones de red de comunicaciones para la validación de los modelos desarrollados.

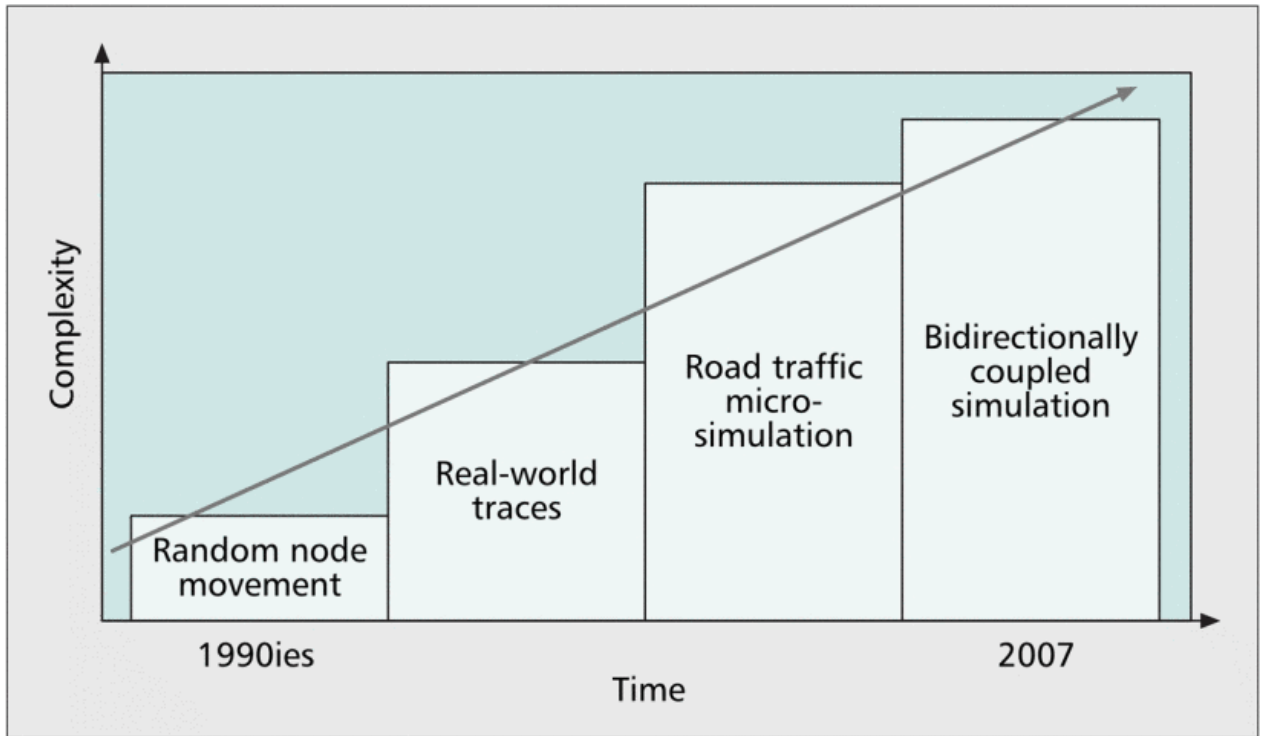


Figura 2.1: Evolución de simulaciones integradas para ITS (fuente: [15]).

Otro ejemplo de esto es la investigación presentada por Goebel *et al.* en [14]. En este trabajo, los investigadores utilizaron SUMO para la generación de trazas vehiculares realistas, las cuales luego fueron importadas en OMNeT++ para el estudio del impacto de la movilidad vehicular en comunicaciones celulares.

Entornos integrados

La necesidad de un entorno integrado para la simulación de Sistemas Inteligentes de Transportes es un tema que ha estado presente en la comunidad académica hace casi más de una década ya. En particular, Sommer *et al.* argumentaron fuertemente a favor de la idea en [5] y [15]; el siguiente análisis se basa principalmente en ambos documentos, con algunas fuentes adicionales que se mencionarán oportunamente.

En primer lugar, los autores destacan la existencia de un sistema de simulación bidireccional desarrollado por la Universidad Nacional de Chiao Tung, Taiwan [16], [17], el cual permite la simulación íntegra de un sistema de transportes dotado de capacidades de comunicación inalámbrica.

NCTUns, actualmente en su versión 6.0 (publicada en junio del 2010 [17]), es un simulador para el estudio de Sistemas Inteligentes de Transporte. Su principal particularidad es que presenta un entorno totalmente integrado para la ejecución de dichas simulaciones; es decir, es tanto un simulador de redes de comunicaciones como de tráfico. Incluye capacidades para simular comportamiento tanto autónomo como predefinido (*rutas*) de vehículos, e implementa un *stack* de protocolo completo en cada vehículo.

No obstante, Sommer *et al.* critican la incompatibilidad de dicho sistema (en su versión 4.0) con los modelos de protocolos de comunicación y transporte ya desarrollados para los simuladores más prominentes, limitando su utilidad práctica en la investigación. Además, si bien NCTUns es capaz de simular un número capacidad de capas físicas, todavía se encuentra muy limitado en ese aspecto en comparación con otros simuladores de redes.

Los investigadores mencionan también la existencia de TraNS [18], un *framework* para la integración de ns-2 con SUMO. Este sistema implementa un *loop* de control y *feedback* activo entre ambos simuladores, estableciendo así una simulación bidireccional que permite la emulación de un ITS.

TraNS integra dos simuladores de renombre en la academia, y ha sido muy bien recibido. Sin embargo, los autores destacan que carece de ciertas funcionalidades – principalmente, la capacidad de sincronizar y controlar el tiempo de simulación entre ambos simuladores.

Se debe destacar también los trabajos realizados por investigadores en la Universidad Estatal de Nueva York en Buffalo [19] y de la Universidad de Düsseldorf [20]. Ambos constituyen ejemplos de simulaciones bidireccionales – no obstante se ven limitados por su especificidad, y dificultad de adaptación a escenarios más diversos. El trabajo de Shalaby en su tesis de magíster [9] también sufre este mismo problema, además de temas relacionados a la eficiencia del *framework* desarrollado por la autora, principalmente ligado a la elección de mecanismo de comunicación entre los simuladores (archivos en disco).

Finalmente, Sommer, German y Dressler presentan su solución en [21]: VEINS, un *framework* de integración entre OMNeT++ y SUMO. Ambos simuladores se escogieron específicamente por su adopción en el mundo académico, y por sus naturalezas abiertas y fáciles de adaptar y modificar.

A través de VEINS, ambos simuladores se ejecutan en paralelo, comunicándose en tiempo real mediante un *socket* utilizando el protocolo TCP; SUMO proporciona las trazas de movimiento de los elementos en la simulación a la vez que OMNeT++ simula el comportamiento de la red de comunicaciones. Además, mediante este esquema, OMNeT++ también puede modificar directamente el comportamiento del modelo de transporte, por ejemplo alterando la velocidad de un vehículo en respuesta a un mensaje específico obtenido a través de la red de comunicaciones. De esta manera, el *framework* en cuestión permite modelar sistemas complejos y dinámicos, que reflejan de buena manera la realidad.

Sin embargo, VEINS sufre por su elección de simulador de transporte; SUMO todavía se encuentra en una temprana etapa de desarrollo, lo cual implica que frecuentemente sufre de problemas de estabilidad y de falta de características y documentación. Por ejemplo, hasta diciembre del 2015 (versión 0.25.0), SUMO no contaba con un editor gráfico de redes de transporte¹, lo cual dificultaba mucho el diseño de redes originales. Además, la curva de aprendizaje de SUMO es bastante pronunciada, y todas sus configuraciones son a través de archivos; es por esto que en muchos departamentos de ingeniería de transporte se opta por otros simuladores más avanzados y estables.

¹<http://sumo.dlr.de/wiki/FAQ>

Capítulo 3

Conceptos clave

3.1. Paramics

3.1.1. API Plugins

3.2. OMNeT++

3.3. TraCI

TraCI (**Traffic Control Interface**) es una arquitectura para la interacción con simuladores de redes de transporte, cuyo principal propósito es facilitar el diseño y la implementación de simulaciones de Sistemas de Transporte Inteligente [22]. Proporciona una interfaz unificada que permite no sólo la obtención de datos desde la simulación de transporte, sino que también permite el control directo sobre la ejecución de ésta y provee métodos para la modificación del comportamiento de sus componentes. Así, TraCI permite a un agente externo (como, por ejemplo, un simulador de redes) comunicarse de manera bidireccional con la simulación de la red de transporte, posibilitando un desarrollo dinámico de dicha simulación en reacción a estímulos externos.

Hoy en día, dicha arquitectura se encuentra integrada en SUMO, y se utiliza en conjunto con simuladores de redes de comunicación inalámbrica como OMNeT++ y NS2 para la simulación y estudio de Sistemas de Transporte Inteligente.

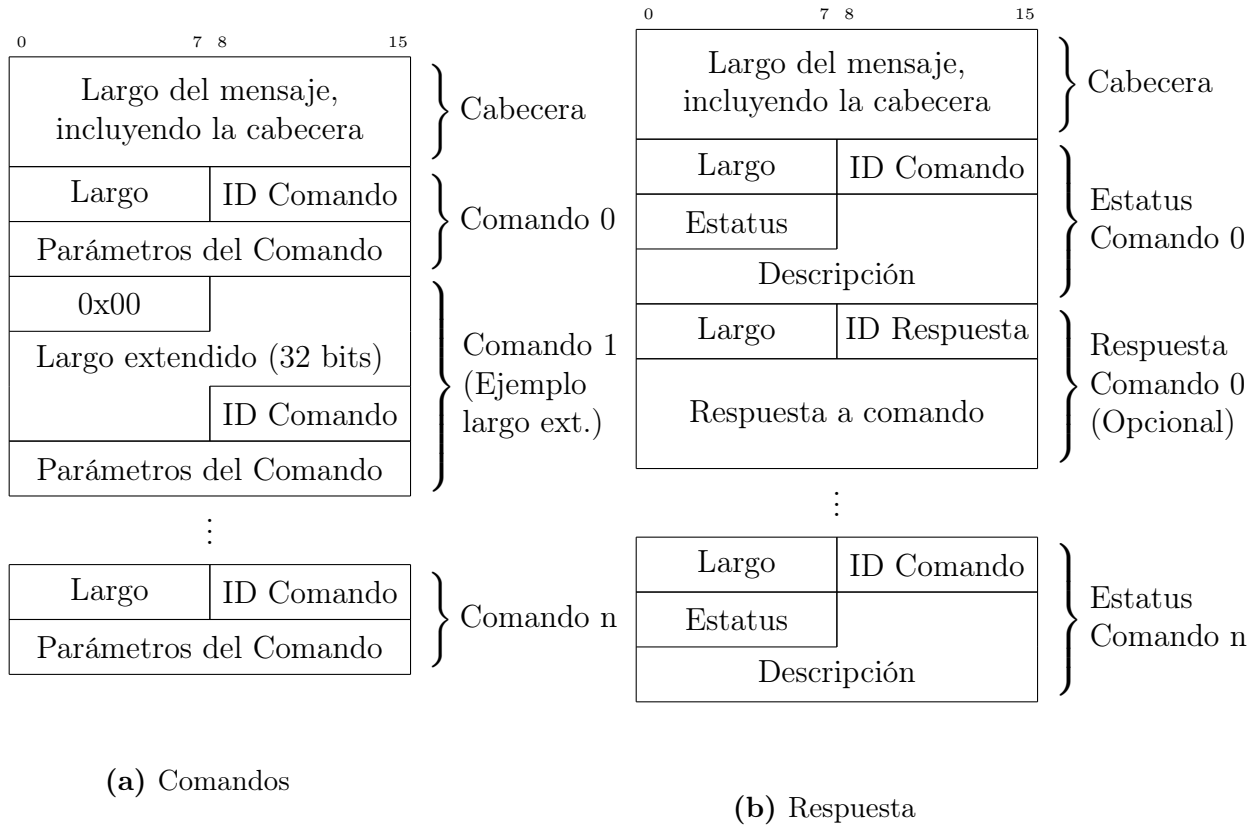


Figura 3.1: Formatos de mensajes TraCI

3.3.1. Diseño

Mensajes

TraCI se basa en una arquitectura cliente-servidor, en la cual el simulador de redes de transporte asume el rol de un servidor pasivo que espera comandos desde un cliente activo. Define además un protocolo de comunicaciones de capa de aplicación para la transmisión de comandos e información entre servidor y cliente mediante un *socket* TCP.

La figura 3.1a ilustra la estructura básica de un mensaje TraCI enviado desde un cliente al servidor. Consiste en una cadena de comandos TraCI consecutivos que deben ser ejecutados por este último; cada comando tiene un largo y un identificador, y puede incluir información adicional – por ejemplo, en el caso de que se trate de un comando que asigne algún valor a una variable de la simulación. En caso de que el valor del largo exceda 255, se agrega un campo de 32 bits para almacenar dicho valor y el campo original se fija en **0x00**.

Por otro lado, la figura 3.1b ilustra un ejemplo de respuesta del servidor, el cual debe responder a cada uno de éstos mensajes con una notificación del estado de la solicitud (“OK”, “ERROR” o “NO IMPLEMENTADO”) y, en caso de que corresponda, con información adicional de acuerdo a parámetros específicos definidos para cada comando. Finalmente, la figura 3.2 ilustra el flujo de mensajes para la solicitud de una variable de la simulación.

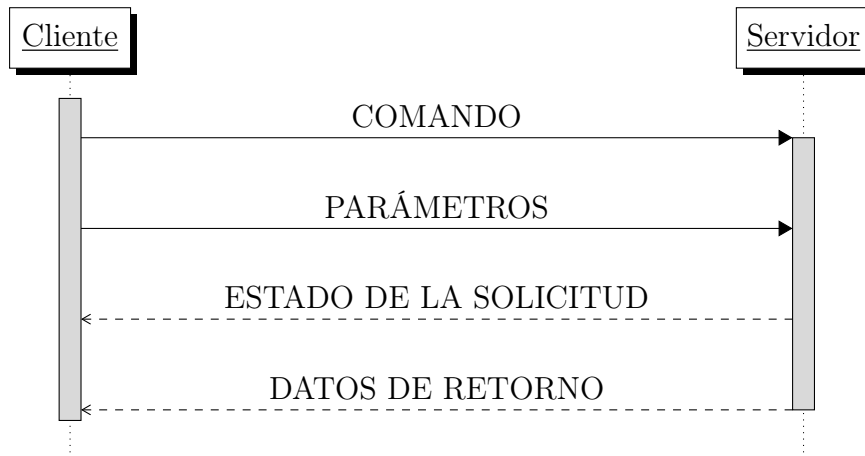


Figura 3.2: Ejemplo solicitud de variable TraCI.

Comandos

El protocolo define tres categorías de comandos disponibles:

- **Control de Simulación:** Esta categoría abarca en total tres comandos distintos, relacionados directamente con el control de la ejecución de la simulación:

0x00 GET VERSION Por diseño, es el primer mensaje en ser enviado por el cliente al iniciar una sesión TraCI – esto para asegurar versiones compatibles del protocolo con el servidor. Este último debe retornar un byte indicando la versión implementada de la API de TraCI y un *string* opcional de descripción del software.

0x02 SIMULATION STEP Corresponde al comando de control de simulación fundamental del protocolo, a través del cual el cliente controla la ejecución de cada paso de la simulación en el servidor.

Este comando tiene dos modos de operación; *single step* y *target time step*. En el modo *single step*, el servidor ejecuta exactamente un único instante de tiempo en la simulación, mientras que en el *target time step* el cliente le indica un instante de tiempo “objetivo” T , y el servidor ejecuta cuantos pasos sean necesarios tal que la simulación alcance el menor instante de tiempo t tal que $t \geq T$. En ambos modos, luego de avanzar la simulación, el servidor debe retornar las “suscripciones” que el cliente haya solicitado con anterioridad. Estas consisten en conjuntos de datos que el cliente puede requerir luego de cada ejecución de la simulación (por ejemplo, las posiciones de todos los vehículos de la simulación).

0x7f CLOSE Este mensaje es enviado por el cliente cuando desee cerrar la conexión y finalizar la simulación. El servidor entonces anuncia la recepción del comando y procede a cerrar el socket.

- **Obtención de Valores:** Esta categoría abarca una gran cantidad de comandos asociados a variables internas de la simulación vehicular o de sus componentes (vehículos, cruces, etc.). Cada comando representa a un conjunto de variables específicas; por ejem-

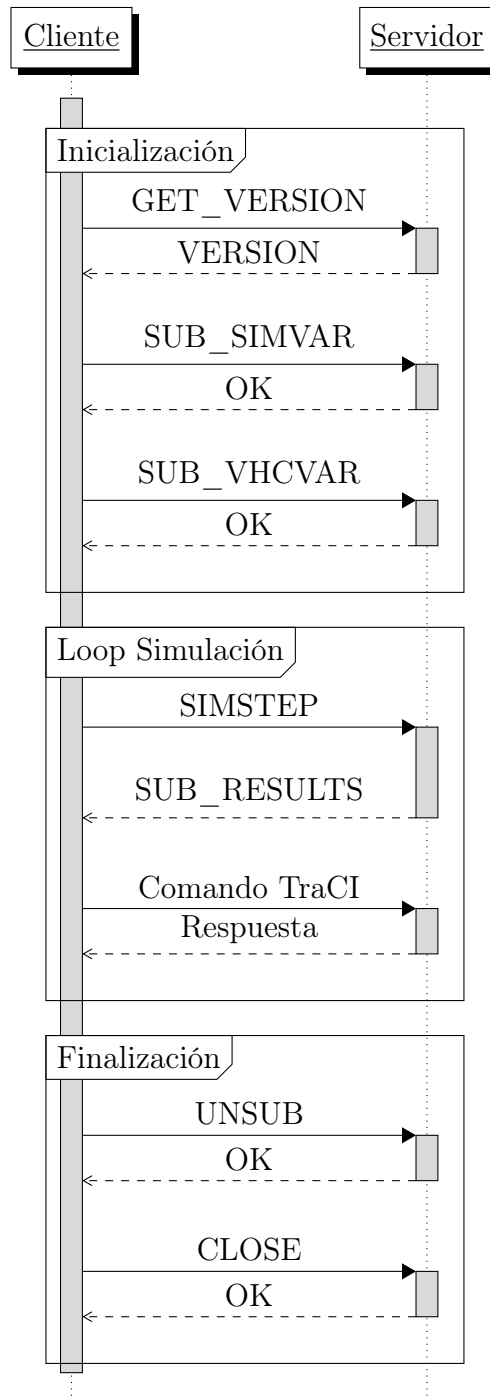


Figura 3.3: Flujo de comunicación TraCI.

por ejemplo, `0xa2 GET TRAFFIC LIGHTS VARIABLE` agrupa y obtiene los valores asociados a las variables propias de los semáforos en la red simulada, mientras que `0xa4 GET VEHICLE VARIABLE` está relacionado exclusivamente con los valores de los vehículos presentes en la red.

- **Modificación de Estados:** Finalmente, aquí se agrupan aquellos comandos que modifican valores y parámetros de la simulación y al igual que en la categoría anterior, cada comando de esta categoría está asociado a un conjunto de variables. Estos comandos tienden a ser más complejos que aquellos de categorías anteriores, ya que por razones obvias incluyen más información que debe ser interpretada por el servidor.

Capítulo 4

Comunicando Paramics con OMNeT++ mediante TraCI

4.1. Diseño Arquitectural

El software desarrollado consiste en un *plugin* que extiende la funcionalidad de Paramics, agregándole la capacidad de comportarse como un servidor TraCI. Específicamente, el *plugin* consiste en una implementación parcial de un servidor TraCI, el cual se ejecuta en un *thread* paralelo a Paramics; este se encuentra a su vez simulando en modo discreto, esperando instrucciones para avanzar la simulación. La comunicación entre ambos se efectúa a través de la API de extensión de Paramics.

La figura 4.1 ilustra esta arquitectura. A pesar de que se encuentra implementado como un *plugin* de Paramics, el servidor TraCI es prácticamente un programa independiente, y su interacción con el simulador de transporte se limita a un conjunto acotado de llamados a su API.

En términos de implementación, el código del *plugin* se separó en una serie de módulos lógicos que encapsulan y abstraen cada uno una categoría de funcionalidades de la interfaz con TraCI. De esta manera, se logró una separación lógica de las funcionalidades implementadas, y se simplifican futuras extensiones al código.

Describir separación de archivos, namespaces.

4.2. Funcionalidad Implementada

Como se detalló en la sección 3.3, el protocolo TraCI define más de 30 comandos distintos, cada uno con una gran cantidad de variables y parámetros asociados. Implementar esta gran cantidad de funcionalidades no hubiese sido factible, por lo que se escogió un subconjunto acotado de éstas a implementar, considerando en específico aquellos comandos esenciales para

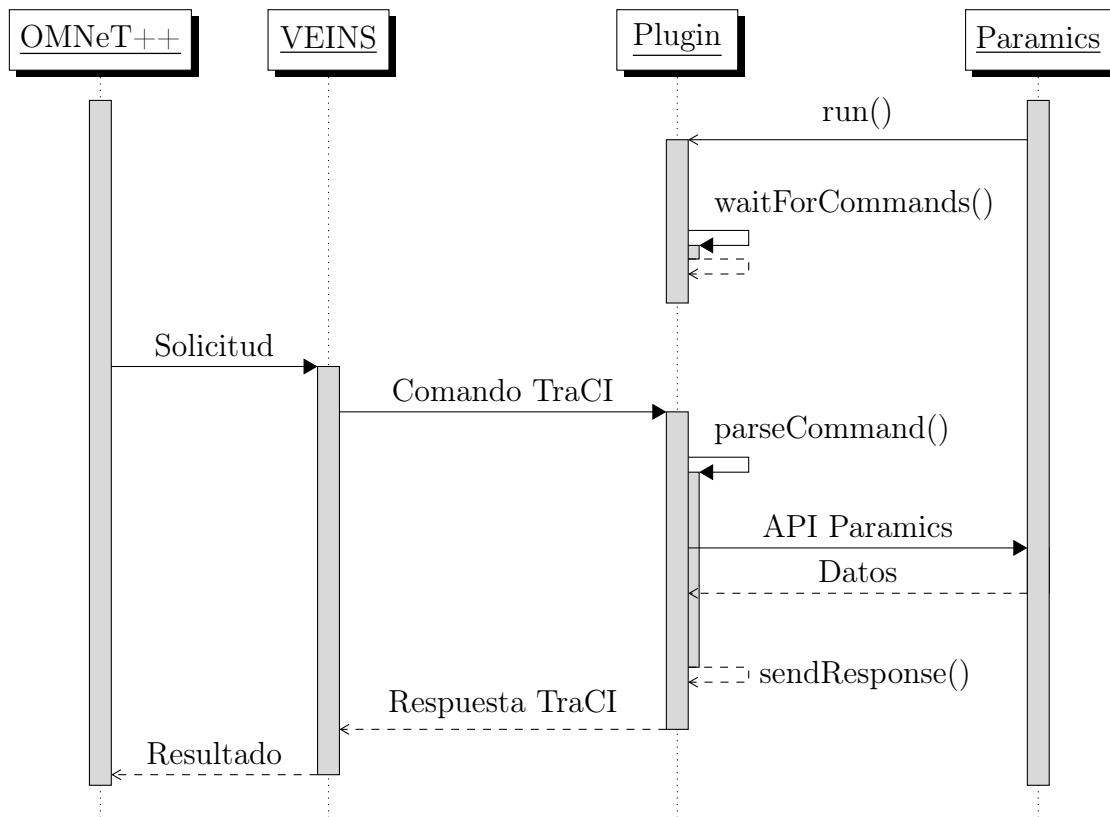


Figura 4.1: Arquitectura del Framework

simulaciones de ITS.

4.2.1. Comandos Implementados

Comandos de Control de Simulación

- **0x00** Obtención de Versión
- **0x02** Avance de Simulación
- **0xff** Cierre de Conexión

Comandos de Obtención de Variables

0xa4 Variables de vehículos

- **0x00** Lista de vehículos activos en la red
- **0x01** Número de vehículos activos en la red
- **0x36** Inclinación actual
- **0x39** Posición actual (3D)
- **0x40** Velocidad actual
- **0x42** Posición actual (2D)
- **0x43** Ángulo actual

```

src/
├── plugin.c.....Archivo de inicio del plugin
├── TraCIAPI/.....Implementación de la API
│   ├── Constants.h.....Constantes utilizadas en el sistema
│   ├── Exceptions.h.....Excepciones propias del framework
│   ├── Network.{cpp/h}.....Métodos de interacción con la red vehicular
│   ├── Simulation.{cpp/h}.....Métodos de interacción con la simulación vehicular
│   ├── Subscriptions.{cpp/h}.....Suscripciones TraCI
│   ├── TraCIServer.{cpp/h}.....Módulo principal, manejo de conexiones TraCI
│   ├── Triggers.{cpp/h}.....Operaciones disparadas (temporales o situacionales)
│   ├── Utils.{cpp/h}.....Funciones auxiliares y de conveniencia
│   └── VehicleManager.{cpp/h}.....Métodos de interacción con vehículos
└── shawn/.....Archivos externos
    ├── socket.{cpp/h}.....Manejo simplificado de sockets TCP
    └── storage.{cpp/h}.....Manejo simplificado de paquetes de datos

```

Figura 4.2: Estructura de archivos del código fuente del framework.

- 0x44 Largo
- 0x4d Ancho
- 0x4f Tipo de vehículo
- 0x50 Calle actual
- 0x51 Identificador de pista actual
- 0x52 Índice de pista actual
- 0xbc Altura

0xa5 Variables de tipos de vehículos

- 0x00 Lista de tipos definidos
- 0x01 Número de tipos definidos
- 0x41 Velocidad máxima
- 0x44 Largo
- 0x46 Aceleración máxima
- 0x47 Deceleración máxima
- 0x4d Ancho
- 0xbc Altura

0xa6 Variables de rutas

- 0x00 Lista de rutas definidas
- 0x01 Número de rutas definidas
- 0x54 Arcos (calles) componentes de la ruta

0xa8 Variables de polígonos (edificios y estructuras)

- 0x00 Lista de polígonos
- 0x01 Número de polígonos

Cabe notar que Paramics no maneja edificios en sus simulaciones, al menos no edificios accesibles a través de la API de programación, por lo que estos métodos se implementaron de manera que reportan siempre 0 polígonos en la simulación.

0xa9 Variables de nodos (intersecciones) de la red

- **0x00** Lista de intersecciones
- **0x01** Número de intersecciones
- **0x42** Posición de la intersección

0xaa Variables de arcos (calles) de la red

- **0x00** Lista de calles
- **0x01** Número de calles

0xab Variables de Simulación

- **0x70** Tiempo de simulación
- **0x73** Número de vehículos liberados a la red en el último paso de simulación
- **0x74** Lista de vehículos liberados a la red en el último paso de simulación
- **0x79** Número de vehículos que han llegado a su destino en el último paso de simulación
- **0x7a** Lista de vehículos que han llegado a su destino en el último paso de simulación
- **0x7b** Tamaño del paso de simulación
- **0x7c** Coordenadas de los límites de la red vehicular

Las variables **0x75**, **0x76**, **0x77** y **0x78**, correspondientes a los números y listas de vehículos que comenzaron y terminaron de teletransportarse en el último paso de simulación, así como las variables **0x6c**, **0x6d**, **0x6e** y **0x6f**, las cuales corresponden a números y listas de vehículos que comenzaron y terminaron de estar estacionados, fueron implementadas “parcialmente”. En estricto rigor, los mecanismos subyacentes no se implementaron porque no se consideraron críticos, pero se implementó una respuesta *dummy* de 0 vehículos para asegurar su funcionamiento con VEINS.

4.2.2. Comandos de modificación de estado

0xc4 Variables de vehículo

- **0x13** Cambio de pista
- **0x14** Cambio de velocidad (lineal)
- **0x40** Cambio de velocidad (instantáneo)
- **0x41** Cambio de velocidad máxima
- **0x45** Coloreado
- **0x57** Cambio de ruta (a una lista de arcos otorgada por el cliente)

4.3. Módulos Principales

4.3.1. plugin.c

Si bien en estricto rigor no es un módulo del *framework*, merece ser mencionado al ser el archivo principal del *plugin* desarrollado. En este archivo se definen las funciones de extensión (prefijo QPX, ver sección 3.1.1) a ser invocadas por Paramics al inicializar el *plugin*. A continuación se describirán brevemente las más importantes de estas funciones, mientras que el archivo `plugin.c` puede estudiarse en su totalidad en el código A.1 en los anexos.

void qpx_NET_postOpen(void)

Invocada inmediatamente luego de que Paramics carga la red y el *plugin*, esta función cambia el modo de ejecución de Paramics a su modo discreto e inicializa el servidor TraCI. Para esto, crea un *thread* donde corre una función auxiliar `runner_fn()`, la cual se encarga de:

1. Obtener el puerto en el cual esperar conexiones entrantes desde los parámetros de ejecución de Paramics. De no haberse especificado puerto, utiliza uno por defecto.
2. Inicializar un objeto `TraCIServer` (ver sección 4.3.2) encargado de las conexiones entrantes en el puerto anteriormente definido.

void qpx_VHC_release(VEHICLE* vehicle)

Función invocada por Paramics cada vez que un vehículo es liberado a la red de transporte. Simplemente se encarga de notificar al `VehicleManager` (ver sección 4.3.4) para su inclusión en el modelo interno del *plugin*.

void qpx_VHC_arrive(VEHICLE* vehicle, LINK* link, ZONE* zone)

Invocada cuando un vehículo alcanza su destino final, esta función notifica al `VehicleManager` para eliminar el vehículo en cuestión de la representación interna.

void qpx_VHC_transfer(VEHICLE* vehicle, LINK* link1, LINK* link2)

Este método es ejecutado por Paramics cada vez que un vehículo pasa de una calle a otra. También se utiliza en el *plugin* para notificar a `VehicleManager` para que actualice su estado interno.

4.3.2. TraCIServer

Implementa el funcionamiento base del servidor TraCI. Es el primer módulo como tal en inicializarse, y tiene como funciones:

1. Asociarse a un *socket* TCP, y esperar una conexión de un cliente TraCI.
2. Mientras exista una conexión abierta, recibir e interpretar comandos TraCI entrantes.
 - En el caso de los comandos de obtención de versión y cierre de la conexión, estos son ejecutados por el módulo mismo.
 - El comando de avance de simulación es ejecutado parcialmente por este módulo y el módulo **Simulation** (ver sección 4.3.3). Específicamente, se delega el avance del estado de la simulación a **Simulation**, luego del cual se realiza la evaluación de las suscripciones en **TraCIServer**.
 - Demás comandos son delegados a los módulos pertinentes.
3. Enviar mensajes de estado y respuesta a comandos TraCI.
4. Al recibir un comando de cierre, finalizar la simulación y cerrar el *socket*.

El módulo en cuestión se implementó como una clase de C++ en los archivos `src/-TraCIAPI/TraCIServer.h` y `src/TraCIAPI/TraCIServer.cpp`, y se instancia en el archivo `plugin.c`. Se implementó considerando que se ejecutaría en un *thread* paralelo al principal de Paramics, y por lo tanto incluye elementos de sincronización.

Cabe destacar que para facilitar el uso de *sockets* y la obtención y envío de datos a través de éstos, se utilizaron las clases `tcpip::Socket` y `tcpip::Storage`, definidas en los archivos `src/shawn/socket.{cpp/h}` y `src/shawn/storage.{cpp/h}`. `tcpip::Socket` abstrae el funcionamiento de un *socket* TCP, y provee métodos de conveniencia que permiten leer y escribir mensajes TraCI completos como objetos `tcpip::Storage`. Estos a su vez proveen métodos para escribir y leer todo tipo de variables en dichos mensajes, sin la necesidad de hacer la conversión manual a bytes.

Estos archivos no fueron desarrollados por el memorista, sino que fueron obtenidos desde el código fuente de SUMO ¹, distribuidos bajo una licencia BSD².

A continuación se detalla la implementación de las funcionalidades anteriormente mencionadas.

¹Fuente SUMO: <https://github.com/planetsumo/sumo/tree/master/sumo/src/foreign/tcpip>. Debe notarse que, a su vez, los creadores de SUMO originalmente obtuvieron dichos archivos del código fuente del simulador de eventos discretos para redes de sensores *SHAWN* [23]. Su fuente original se encuentra en <https://github.com/itm/shawn/tree/master/src/apps/tcpip>

²Licencia clases `tcpip::Socket` y `tcpip::Storage`: http://sumo.dlr.de/wiki/Libraries_Licenses#tcpip_-_TCP.2FIP_Socket_Class_to_communicate_with_other_programs

```

1 void traci_api::TraCIServer::run()
2 {
3     running = true;
4     std::string version_str = "Paramics TraCI plugin v" +
        std::string(PLUGIN_VERSION) + " on Paramics v" +
        std::to_string(qpg_UTL_parentProductVersion());
5     infoPrint(version_str);
6     debugPrint("Simulation start time: " +
        std::to_string(qpg_CFG_simulationTime()));
7     infoPrint("Awaiting connections on port " + std::to_string(port));
8     ssocket.accept();
9     infoPrint("Accepted connection");
10    this->waitForCommands();
11 }

```

Código 4.1: Rutina de inicio de conexión.

Inicio de conexión TraCI

Como se mencionó en la sección 4.3.1, al iniciarse el *plugin* se crea un nuevo *thread*, en el cual se instancia un objeto de la clase **TraCIServer**, al cual se le invoca su método **run()** (código 4.1).

Este método es simple: imprime información pertinente sobre el *plugin* en la ventana de información de Paramics, y luego procede a bloquear el *thread* mientras espera una conexión entrante en el *socket*. Al recibirla, resume la ejecución y realiza un llamado al método **waitForCommands()** para comenzar a recibir e interpretar comandos TraCI.

Recepción e Interpretación de Comandos Entrantes

El método **waitForCommands()**, presentado en el código 4.2, se encarga de recibir mensajes TraCI entrantes y separar los comandos para su posterior interpretación.

El código inicia un *loop* dentro del cual monitorea el *socket*. Al recibir datos entrantes, el *socket* retorna un objeto **tcpip::Storage** con el mensaje completo. Luego, en un *loop* adicional, este mensaje se separa en sus comandos TraCI constituyentes, copiando la información perteneciente a cada comando en otro objeto **tcpip::Storage** temporal. Este objeto se entrega como parámetro al método **parseCommand()** para la interpretación del comando, luego de lo cual se limpia y se vuelve a utilizar para el siguiente comando.

Finalmente, interpretados todos los comandos, se envía la respuesta al cliente a través del mismo *socket* y se limpian los objetos **tcpip::Storage** para su reutilización en una nueva iteración del *loop*.

```

1 void traci_api::TraCIServer::waitForCommands()
2 {

```

```

3   tcpip::Storage incoming; // the whole incoming message
4   tcpip::Storage cmdStore; // individual commands in the message
5
6   debugPrint("Waiting for incoming commands from TraCI client...");
7
8   /* While the connection is open, receive commands from the client
9      */
10  while (running && ssocket.receiveExact(incoming))
11  {
12      auto msize = incoming.size();
13      debugPrint("Got message of length " + std::to_string(msize));
14
15      /* Multiple commands may arrive at once in one message,
16         * divide them into multiple storages for easy handling */
17      while (msize > 0 && incoming.valid_pos())
18      {
19          uint8_t cmdlen = incoming.readUnsignedByte();
20          cmdStore.writeUnsignedByte(cmdlen);
21          debugPrint("Got command of length " +
22                     std::to_string(cmdlen));
23
24          for (uint8_t i = 0; i < cmdlen - 1; i++)
25              cmdStore.writeUnsignedByte(incoming.readUnsignedByte());
26
27          this->parseCommand(cmdStore);
28          cmdStore.reset();
29      }
30      this->sendResponse();
31      incoming.reset();
32      outgoing.reset();
33      debugPrint("----- waiting for commands -----");
34  }

```

Código 4.2: Recepción de mensajes desde el *socket* TCP.

Como se mencionó anteriormente, la interpretación de los comandos se lleva a cabo en el método `parseCommand()`, el cual recibe un único comando encapsulado en un objeto `tcpip::Storage`. Este método tiene una única misión; interpretar el identificador del comando recibido y delegar su ejecución al método correspondiente de la clase `TraCIServer`. Su implementación es simple, aunque un poco tediosa, y su esqueleto puede observarse en el código 4.3. En específico, el código del método se puede dividir en dos ramas de ejecución; en caso de comando de suscripción (cuyos identificadores se encuentran todos en el rango `[0xd0, 0xdb]`) se extraen los parámetros de la suscripción y se invoca el método `addSubscription()` para la subsecuente validación y activación de ésta. Por el contrario, en caso de recibir un comando con identificador fuera de dicho rango, se procede a verificar su tipo mediante un *switch*. Cada caso se relaciona con un comando y método específico a invocar, y en caso de no encontrarse el identificador en cuestión se notifica al cliente que el comando deseado no está implementado.

```

1 void traci_api::TraCIServer::parseCommand(tcpip::Storage& storage)
2 {
3     /* ... */
4     uint8_t cmdLen = storage.readUnsignedByte();
5     uint8_t cmdId = storage.readUnsignedByte();
6     tcpip::Storage state;
7     /* ... */
8
9     if (cmdId >= CMD_SUB_INDVAR && cmdId <= CMD_SUB_SIMVAR)
10    {
11        // subscription
12        // | begin Time | end Time | Object ID | Variable Number | The
13        //   list of variables to return
14        /* read subscription params */
15        /* ... */
16        addSubscription(cmdId, oID, btime, etime, vars);
17    }
18    else
19    {
20        switch (cmdId)
21        {
22            case CMD_GETVERSION:
23                /* ... */
24                /* ... */
25            default:
26                debugPrint("Command not implemented!");
27                writeStatusResponse(cmdId, STATUS_NIMPL, "Method not
28                implemented.");
29        }
30    }
31 }

```

Código 4.3: Esqueleto de `parseCommand()`

Se definieron una serie de comandos en `TraCIServer` encargados de obtener variables de la simulación o modificar el estado de esta. El funcionamiento de estos es idéntico en todos los casos (a excepción de `cmdGetPolygonVar()`), y se limita al siguiente procedimiento (ver ejemplo en el código 4.4):

1. Obtener el valor desde el módulo apropiado (por ejemplo, `VehicleManager` para variables de vehículos, `Simulation` para variables de la simulación, etc.).
2. En caso de error en la obtención de los datos (variable no implementada, objeto no existente, etc.), atrapar el error y determinar el curso de acción apropiado (por ejemplo, notificar al cliente).
3. Finalmente, enviar un mensaje de estado de la solicitud y, en caso de éxito, el valor de la variable, al cliente.

```

1 void traci_api::TraCIServer::cmdGetVhcVar(tcpip::Storage& input)

```

```

2 {
3     tcpip::Storage result;
4     try
5     {
6         VehicleManager::getInstance()->packVehicleVariable(input,
7             result);
8         this->writeStatusResponse(CMD_GETVHCVAR, STATUS_OK, "");
9         this->writeToOutputWithSize(result, false);
10    }
11    catch (NotImplementedError& e)
12    {
13        debugPrint("Variable not implemented");
14        debugPrint(e.what());
15        this->writeStatusResponse(CMD_GETVHCVAR, STATUS_NIMPL,
16            e.what());
17    }
18    catch (std::exception& e)
19    {
20        debugPrint("Fatal error???");
21        debugPrint(e.what());
22        this->writeStatusResponse(CMD_GETVHCVAR, STATUS_ERROR,
23            e.what());
24        throw;
25    }
26 }

```

Código 4.4: Ejemplo de método de obtención y empaquetado de variables en **TraCIServer**

El caso de `cmdGetPolygonVar()` es especial. En TraCI, un polígono representa un edificio o una estructura presente en las cercanías de la simulación vehicular, la cual puede interferir con el modelo de comunicación inalámbrica en OMNeT++. Sin embargo, el modelador de Paramics no maneja elementos externos a la simulación de transporte, por lo que se decidió, en el caso de comandos de obtención de variables relacionadas, simplemente reportar que no existen polígonos en la simulación para simplificar la integración.

Avance de Simulación y Evaluación de Suscripciones

En caso de recibir un comando de avance de simulación, **TraCIServer** ejecuta su método `cmdSimStep()`, dentro del cual extrae los parámetros del comando y delega la ejecución de dicho avance al módulo **Simulation**. Sin embargo, al terminar este procedimiento, se realiza la evaluación de las suscripciones activas en **TraCIServer**, mediante el método `processSubscriptions()`.

Como se detalló en la sección 3.3, el protocolo TraCI define 12 tipos de suscripciones a variables de objeto, las cuales comparten todas una estructura idéntica. Cada suscripción se caracteriza por su identificador de tipo y sus parámetros: tiempo de inicio, tiempo de fin,

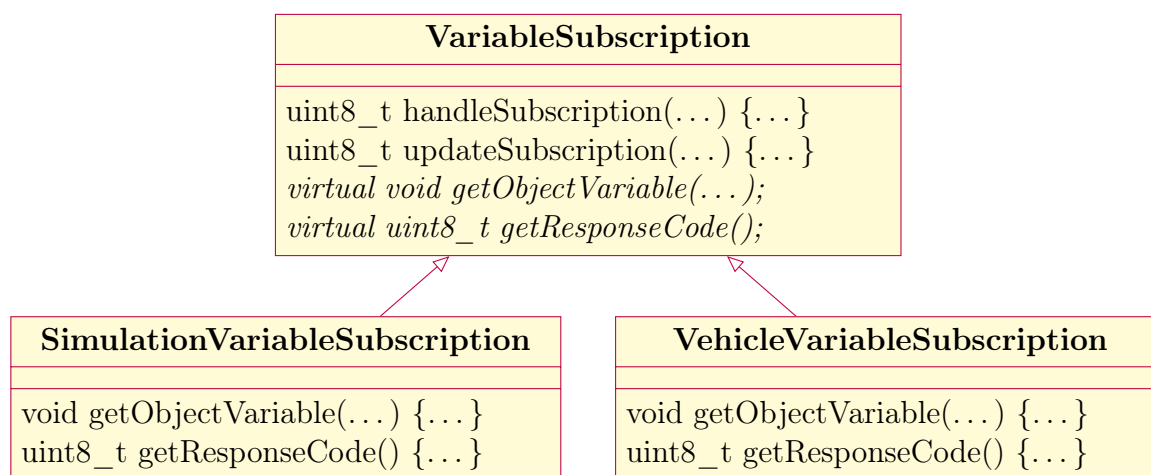


Figura 4.3: Diagrama de herencia, **VariableSubscription**

identificador del objeto y las variables a las cuales el cliente se ha suscrito. En la práctica, lo único que diferencia a las suscripciones entre sí son las categorías de objetos a las cuales están asociadas, y por ende, cómo obtener esos datos desde la implementación interna del *plugin*. A raíz de esto, se decidió implementar un árbol de clases de C++ para representar las suscripciones en memoria (declarada e implementada en los archivos `src/TraCI-API/Subscriptions.h` y `src/TraCI-API/Subscriptions.cpp` respectivamente).

La raíz de éste árbol, la clase **VariableSubscription**, implementa la funcionalidad completa de evaluación y actualización de una suscripción en los métodos `handleSubscription()` y `updateSubscription()` (implementación completa de estos métodos en A.2), abstrayendo la obtención de datos específicos a cada tipo en los métodos `getObjectVariable()` y `getResponseCode()`. Estos métodos son *virtuales* en la clase base, y son implementados por las clases derivadas, de manera que **TraCIServer** sólo necesita mantener un vector de variables del tipo base, las cuales se evalúan de manera polimórfica.

En términos más simples, lo único que debe implementar una clase derivada de **VariableSubscription** para definir un nuevo tipo de suscripción son versiones propias de los métodos `getObjectVariable()` y `getResponseCode()`, ya que toda la demás funcionalidad de evaluación de suscripciones está ya implementada en la clase base.

De esta manera, la evaluación en **TraCIServer** se simplifica, ya que la instancia sólo necesita mantener un vector con punteros a objetos de la clase base, dado que independiente de la implementación específica de los métodos `getObjectVariable()` y `getResponseCode()` de cada subclase, `handleSubscription()` es exactamente igual para todas (ver línea 7 en el código 4.5). Además, esto facilita la extensión futura del software.

```

1 void traci_api::TraCIServer::processSubscriptions(tcpip::Storage&
  sub_store)
2 {
3     /* ... */
4     for (auto i = subs.begin(); i != subs.end(); i++)
5     {

```

```

6      /* polymorphic evaluation of subscriptions; (*i) may be
       Vehicle or Sim subscription */
7      sub_res = (*i)->handleSubscription(temp, false, errors);
8      if (sub_res == VariableSubscription::STATUS_EXPIRED
9          || sub_res == VariableSubscription::STATUS_OBJNOTFOUND)
10     {
11         delete *i;
12         i = subs.erase(i);
13     }
14     else
15     {
16         writeToStorageWithSize(temp, sub_results, true);
17         count++;
18         ++i; // increment
19     }
20     temp.reset();
21 }
22 /* ... */
23 sub_store.writeInt(count);
24 sub_store.writeStorage(sub_results);
25 }

```

Código 4.5: Rutina de evaluación de suscripciones en **TraCIServer**. **subs** es una variable de instancia de **TraCIServer** correspondiente a un vector de punteros **VariableSubscription***, poblado de elementos de clases derivadas de **VariableSubscription**.

Creación de Nuevas Suscripciones

Por otro lado, para crear nuevas suscripciones, **TraCIServer** debe considerar la categoría de objetos a la cual se está suscribiendo, e insertar un puntero a un objeto con el tipo correspondiente en la variable de instancia **subs**. Esto sucede al recibir un comando con un identificador correspondiente a una suscripción; los parámetros de la suscripción son extraídos y delegados al método **addSubscription()** (código 4.3, línea 15). Este código puede estudiarse en su totalidad en el anexo A, código A.3, sin embargo, a continuación se explicará brevemente su funcionamiento con algunos extractos de código.

Como se comentó en la descripción del protocolo TraCI (sección 3.3), un comando de suscripción puede solicitar tanto la creación de una nueva suscripción como la actualización o cancelación de una ya existente. Esto se determina basado en si, al recibir el comando de suscripción, ya existe una suscripción asociada a dicha categoría y objeto. Esto es lo primero en verificarse en **addSubscription()**, mediante llamados al método **updateSubscription()** de cada suscripción ya existente en el servidor.

El funcionamiento de este método se detalla en el código A.2, a partir de la línea 70. Verifica si los parámetros recibidos corresponden a una suscripción ya existente, y retorna un byte cuyo valor representa el estado de la suscripción, valor interpretado por **addSubs-**

`cription()` para determinar el curso de acción a tomar:

1. **STATUS_NOUPD**: Los parámetros entregados no corresponden a esta suscripción. `addSubscription()` sigue recorriendo las suscripciones restantes para verificar si corresponde a alguna ya existente.
2. **STATUS_UNSUB**: Los parámetros corresponden a una solicitud de cancelación de esta suscripción (categoría e identificador de objeto son los mismos, número de variables a suscribir es 0). `addSubscription()` entonces procede a eliminar esta suscripción del vector `subs` en `TraCIServer` y dealocar la memoria asignada al puntero.
3. **STATUS_ERROR**: Los parámetros corresponden a una actualización de esta suscripción (categoría e identificador de objeto son los mismos), pero sucedió un error en la actualización. `addSubscription()` escribe un mensaje de notificación al cliente y retorna.
4. **STATUS_OK**: Los parámetros corresponden a una actualización de esta suscripción (categoría e identificador de objeto son los mismos), y la actualización fue exitosa. `addSubscription()` escribe un mensaje de notificación al cliente y retorna.

```
1  for (auto it = subs.begin(); it != subs.end(); ++it)
2  {
3      uint8_t result = (*it)->updateSubscription(sub_type,
4          object_id, start_time, end_time, variables, temp, errors);
5
6      switch (result)
7      {
8          case VariableSubscription::STATUS_OK:
9              // update ok, return now
10             debugPrint("Updated subscription");
11             writeStatusResponse(sub_type, STATUS_OK, "");
12             writeToOutputWithSize(temp, true);
13             return;
14         case VariableSubscription::STATUS_UNSUB:
15             // unsubscribe command, remove the subscription
16             debugPrint("Unsubscribing...");
17             delete *it;
18             it = subs.erase(it);
19             // we don't care about the deleted iterator, since we
20             // return from the loop here
21             writeStatusResponse(sub_type, STATUS_OK, "");
22             return;
23         case VariableSubscription::STATUS_ERROR:
24             // error when updating
25             debugPrint("Error updating subscription.");
26             writeStatusResponse(sub_type, STATUS_ERROR, errors);
27             break;
28         case VariableSubscription::STATUS_NOUPD:
29             // no update, try next subscription
30             continue;
```



```

29     default:
30         throw std::runtime_error("Received unexpected result " +
                                   std::to_string(result) + " when trying to update
                                   subscription.");
31     }
32 }

```

Código 4.6: Verificación de actualización en `addSubscription()`.

La segunda parte del método es más simple. De corresponder el comando a una solicitud de creación de una suscripción nueva, se verifica su tipo y se instancia dinámicamente un objeto de la clase apropiada (como se explicó anteriormente, derivada de `VariableSubscription`). Finalmente, se verifica el correcto funcionamiento de la nueva suscripción mediante un llamado a su método `handleSubscription()` y se notifica al cliente del resultado.

```

1  VariableSubscription* sub;
2  switch (sub_type)
3  {
4      case CMD_SUB_VHCVAR:
5          debugPrint("Adding VHC subscription.");
6          sub = new VehicleVariableSubscription(object_id, start_time,
          end_time, variables);
7          break;
8      case CMD_SUB_SIMVAR:
9          debugPrint("Adding SIM subscription.");
10         sub = new SimulationVariableSubscription(object_id,
            start_time, end_time, variables);
11         break;
12     default:
13         writeStatusResponse(sub_type, STATUS_NIMPL, "Subscription type
            not implemented: " + std::to_string(sub_type));
14         return;
15 }

```

Código 4.7: Creación de una nueva suscripción. Notar la instanciación polimórfica.

Envío de resultados al cliente

`TraCIServer` mantiene una variable de instancia `tcpip::Storage outgoing`, en la cual se almacenan los mensajes de estado y resultados de comandos TraCI. El envío de estos al cliente se efectúa al final de cada iteración del *loop* en `waitForCommands()`, enviando así conjuntamente las respuestas a todos los comandos obtenidos desde el cliente en el último paso de tiempo. Gracias a las clases `tcpip::Storage` y `tcpip::Socket` utilizadas, la operación de enviar los datos almacenados se reduce a una invocación del método `sendExact()` del objeto `tcpip::Socket`, la cual recibe un objeto `tcpip::Storage`, le adjunta una cabecera con su tamaño total y lo envía a través del *socket* al cliente.

La escritura de datos en el almacenamiento saliente se implementó en dos métodos de `TraCIServer`; `writeStatusResponse()`, método de conveniencia para la escritura de mensajes de estado, y `writeToOutputWithSize()`, el cual recibe otro objeto de tipo `tcpip::Storage` que contiene el resultado de algún comando y escribe sus contenidos en `outgoing`, junto con una cabecera que indique su tamaño. Esto implicó también una decisión de diseño en términos de la comunicación de `TraCIServer` con los demás módulos del sistema. Se optó por realizar la mayor parte de esta comunicación mediante objetos de tipo `tcpip::Storage`, delegando la estructuración de los resultados de cada comando específico a los módulos responsables. De esta manera se aumenta la modularidad, ya que cada módulo sabe como escribir sus resultados de manera correcta, y `TraCIServer` sólo necesita asumir que recibirá un `tcpip::Storage` bien formateado como respuesta a los comandos.

```
1 void traci_api::TraCIServer::writeStatusResponse(uint8_t cmdId,
2   uint8_t cmdStatus, std::string description)
3 {
4   debugPrint("Writing status response " + std::to_string(cmdStatus)
5     + " for command " + std::to_string(cmdId));
6   outgoing.writeUnsignedByte(1 + 1 + 1 + 4 +
7     static_cast<int>(description.length())); // command length
8   outgoing.writeUnsignedByte(cmdId); // command type
9   outgoing.writeUnsignedByte(cmdStatus); // status
10  outgoing.writeString(description); // description
11 }
12
13 void traci_api::TraCIServer::writeToOutputWithSize(tcpip::Storage&
14   storage, bool force_extended)
15 {
16   this->writeToStorageWithSize(storage, outgoing, force_extended);
17 }
18
19 void traci_api::TraCIServer::writeToStorageWithSize(tcpip::Storage&
20   src, tcpip::Storage& dest, bool force_extended)
21 {
22   uint32_t size = 1 + src.size();
23   if (size > 255 || force_extended)
24   {
25     // extended-length message
26     dest.writeUnsignedByte(0);
27     dest.writeInt(size + 4);
28   }
29   else
30     dest.writeUnsignedByte(size);
31   dest.writeStorage(src);
32 }
```

Código 4.8: Escritura de datos en almacenamiento saliente.

4.3.3. Simulation

La principal funcionalidad de este módulo es controlar el avance de la simulación, además de abstraer y encapsular el acceso a los parámetros de la simulación vehicular de Paramics. Se implementó como una clase de C++ utilizando el patrón de diseño *Singleton*; esto quiere decir que sólo se permite la instanciación de un único objeto de este tipo en la ejecución del programa. Esto ya que, por razones lógicas, cada ejecución del *plugin* está asociada a una única simulación en Paramics, y por ende no tiene sentido que pueda existir más de un objeto de acceso a ésta.

Avance de la Simulación

Al recibir un mensaje de avance de simulación, **TraCIServer** invoca el método **runSimulation()** del presente módulo, detallado en los anexos, código A.4. Este método recibe como parámetro un entero representando el instante de tiempo hasta cual se quiere avanzar la simulación (de ser 0 se interpreta como solicitud de avance de un único paso de simulación) y ejecuta los llamados necesarios a la API de Paramics, además de notificar a **VehicleManager** que debe actualizar sus listas de vehículos y ejecutar las acciones diferidas correspondientes (ver sección 4.3.4).

Se debe notar que dada la representación en tiempo discreto de los pasos de simulación, el avance de esta en muchos casos no alcanza exactamente el tiempo deseado. Si definimos el paso de simulación como ΔT , el instante de tiempo en que se recibe el comando de avance como T_i y el instante de tiempo objetivo T_o , la simulación se avanzará un número $n \in \mathbb{N}$ de pasos, tal que

$$\begin{aligned}T_i + (n \times \Delta T) &= T_f \\T_i + ((n - 1) \times \Delta T) &= T'_f \\T_f &\geq T_o \\T'_f &< T_o\end{aligned}$$

En otras palabras, la simulación se avanzará el mínimo número de pasos tal que el tiempo final es *igual o mayor* al instante de tiempo objetivo. Esto es para asegurar que se ejecuten todas las acciones diferidas por lo menos hasta dicho instante.

Obtención de variables

Las variables obtenibles desde este módulo son todas aquellas que se relacionan con la simulación como ente abstracto, enumeradas en el ítem **0xab** de la sección 4.2.1.

Sin embargo, algunas de éstas variables si bien se encuentran comprendidas en la categoría mencionada, están fuertemente ligadas a otras también. En específico, las variables referentes

a los vehículos que comenzaron o terminaron su viaje en el último paso de simulación son accesibles desde este módulo, pero su obtención fue implementada en el módulo **VehicleManager**. Esto ya que dicho módulo debe mantener una lista interna de todos los vehículos de la simulación en todo instante de tiempo, por lo que obtener estos valores era mucho más directo de implementar allá. Ver la sección sobre **VehicleManager**, 4.3.4, para más detalles.

De las variables efectivamente implementadas en este módulo, vale destacar un par de detalles. En primer lugar, existe una diferencia entre cómo VEINS y OMNeT++ manejan el tiempo de simulación, y cómo lo hace Paramics; los primeros ocupan mili-segundos, mientras que este último ocupa segundos. Esto implicó realizar las respectivas conversiones necesarias.

En segundo lugar se hablará del comando de obtención de las coordenadas de los límites de la simulación. Este es de extrema importancia para VEINS, ya que con estos valores se crea el escenario de comunicación inalámbrica en OMNeT++; de ser erróneos, tarde o temprano la posición de un vehículo (representado por un nodo de comunicación en OMNeT++) quedará fuera del escenario, gatillando un error fatal en la simulación. Desafortunadamente, si bien la API de Paramics cuenta con un comando para, supuestamente, obtener estas coordenadas, por razones que no se lograron dilucidar, este comando retorna valores altamente erróneos (esto se verificó con múltiples redes de transporte). Se debió entonces implementar el cálculo correcto de éstos límites en el módulo mismo, en el método, apropiadamente nombrado, **getRealNetworkBounds()** (expuesto en el código A.5 en los anexos). Este cálculo se hace prácticamente a fuerza bruta, recorriendo todos los elementos que definen el alcance de la red (calles, intersecciones y zonas de emisión de vehículos), obteniendo sus coordenadas y luego obteniendo el rectángulo que las contiene (más un cierto margen de error). Si bien este método no escala bien con redes más grandes, su impacto en la eficiencia del sistema se estimó como mínimo ya que se accede una única vez por simulación a este valor.

4.3.4. VehicleManager

El módulo más complejo y grande (en términos de líneas de código) del *framework*. **VehicleManager** tiene como función abstraer el acceso a variables directamente relacionadas con los vehículos presentes en la simulación, mantener registros de dichos vehículos, y encargarse de ejecutar los diversos cambios de estado de éstos que puede solicitar el cliente (ver 4.2.2). Además, varios de éstos cambios de estado requieren acciones en múltiples instantes de tiempo (por ejemplo, el cambio de velocidad lineal, el cual se ejecuta durante un periodo de tiempo determinado), por lo que adicionalmente el módulo mantiene colas de eventos diferidos a ejecutar en instantes determinados.

Para la implementación de éste módulo, se utilizó nuevamente el paradigma de *singleton*, por las mismas razones esgrimidas que para **Simulation**.

A continuación se tratará de detallar los aspectos más importantes de este módulo.

Estado interno

Para simplificar muchas de las operaciones de obtención de variables y modificación de estados, el módulo mantiene un estado interno congruente con el estado de la simulación en Paramics. Para este fin se ocupan los llamados de la API de Paramics mencionados en la sección 4.3.1.

En específico, se utilizan las siguientes variables para almacenar información sobre el estado de la simulación en todo instante:

std::unordered_map<int, VEHICLE*> vehicles_in_sim *Hashmap* que almacena el ID y un puntero a cada vehículo presente en la simulación. Se utiliza ya que Paramics no provee un método directo para obtener un puntero a un vehículo dada su ID, sino que es necesario buscarlo en la red. Este método elimina esa búsqueda y facilita además el conteo de vehículos en la simulación (basta con obtener la cantidad de pares {llave, valor} en el *hashmap*). Se actualiza dinámicamente cada vez que ingresa un vehículo nuevo a la red, a través del llamado al método `vehicleDepart()` del presente módulo desde `plugin.c`.

Obtención de variables

La función más básica de `VehicleManager` es la de abstraer el acceso a las variables de simulación directamente relacionadas con vehículos y tipos de vehículos. Los principales métodos encargados de estas funcionalidades son `getVehicleVariable()` y `getVhcTypesVariable()`, respectivamente, aunque éstos por lo general son invocados por `packVehicleVariable()` y `packVhcTypesVariable()`, respectivamente, métodos que empaquetan los resultados en un `tcpip::Storage` para su fácil manejo.

Bibliografía

- [1] *Directive 2010/40/EU of the European Parliament and of the Council on the framework for the deployment of Intelligent Transport Systems in the field of road transport and for interfaces with other modes of transport*, 2010 O.J. L 207/1, European Parliament, 2010.
- [2] D. J. Dailey, K. McFarland y J. L. Garrison, «Experimental study of 802.11 based networking for vehicular management and safety», en *2010 IEEE Intelligent Vehicles Symposium*, jun. de 2010, págs. 1209-1213. DOI: [10.1109/IVS.2010.5547955](https://doi.org/10.1109/IVS.2010.5547955).
- [3] W. Xiong, X. Hu y T. Jiang, «Measurement and Characterization of Link Quality for IEEE 802.15.4-Compliant Wireless Sensor Networks in Vehicular Communications», *IEEE Transactions on Industrial Informatics*, vol. 12, n.º 5, págs. 1702-1713, oct. de 2016, ISSN: 1551-3203. DOI: [10.1109/TII.2015.2499121](https://doi.org/10.1109/TII.2015.2499121).
- [4] D. Jiang y L. Delgrossi, «IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments», en *VTC Spring 2008 - IEEE Vehicular Technology Conference*, mayo de 2008, págs. 2036-2040. DOI: [10.1109/VETECS.2008.458](https://doi.org/10.1109/VETECS.2008.458).
- [5] C. Sommer, Z. Yao, R. German y F. Dressler, «On the Need for Bidirectional Coupling of Road Traffic Microsimulation and Network Simulation», en *Proceedings of the 1st ACM SIGMOBILE Workshop on Mobility Models*, ép. MobilityModels '08, Hong Kong, Hong Kong, China: ACM, 2008, págs. 41-48, ISBN: 978-1-60558-111-8. DOI: [10.1145/1374688.1374697](https://doi.org/10.1145/1374688.1374697). dirección: <http://doi.acm.org/10.1145/1374688.1374697>.
- [6] E. Cascetta, *Transportation systems engineering: theory and methods*. Springer Science & Business Media, 2013, vol. 49.
- [7] *U.S. Department of Transportation, Office of the Assistant Secretary for Research and Technology (OST-R)*, <http://www.itsoverview.its.dot.gov/>, Visitado 04/2017.
- [8] T. J. Schriber, D. T. Brunner y J. S. Smith, «How Discrete-event Simulation Software Works and Why It Matters», en *Proceedings of the Winter Simulation Conference*, ép. WSC '12, Berlin, Germany: Winter Simulation Conference, 2012, 3:1-3:15. dirección: <http://dl.acm.org/citation.cfm?id=2429759.2429763>.
- [9] Y. Shalaby, «An Integrated Framework for Coupling Traffic and Wireless Network Simulations», Tesis de mtría., Department of Civil Engineering, University of Toronto, Canada, 2010.
- [10] N. T. Ratrouy y S. M. Rahman, «A comparative analysis of currently used microscopic and macroscopic traffic simulation software», *The Arabian Journal for Science and Engineering*, vol. 34, n.º 1B, págs. 121-133, 2009.
- [11] S. A. Boxill y L. Yu, «An evaluation of traffic simulation models for support-

- ting its», *Houston, TX: Development Centre for Transportation Training and Research, Texas Southern University*, 2000.
- [12] D. Li, H. Huang, X. Li, M. Li y F. Tang, «A Distance-Based Directional Broadcast Protocol for Urban Vehicular Ad Hoc Network», en *2007 International Conference on Wireless Communications, Networking and Mobile Computing*, sep. de 2007, págs. 1520-1523. DOI: **10.1109/WICOM.2007.383**.
 - [13] H. Y. Huang, P. E. Luo, M. Li, D. Li, X. Li, W. Shu y M. Y. Wu, «Performance Evaluation of SUVnet With Real-Time Traffic Data», *IEEE Transactions on Vehicular Technology*, vol. 56, n.º 6, págs. 3381-3396, nov. de 2007, ISSN: 0018-9545. DOI: **10.1109 / TVT . 2007.907273**.
 - [14] N. Goebel, R. Bialon, M. Mauve y K. Graffi, «Coupled simulation of mobile cellular networks, road traffic and V2X applications using traces», en *2016 IEEE International Conference on Communications (ICC)*, mayo de 2016, págs. 1-7. DOI: **10.1109/ICC.2016.7511126**.
 - [15] C. Sommer y F. Dressler, «Progressing toward realistic mobility models in VANET simulations», *IEEE Communications Magazine*, vol. 46, n.º 11, págs. 132-137, nov. de 2008, ISSN: 0163-6804. DOI: **10.1109 / MCOM . 2008 . 4689256**.
 - [16] S. Y. Wang, C. L. Chou, Y. H. Chiu, Y. S. Tzeng, M. S. Hsu, Y. W. Cheng, W. L. Liu y T. W. Ho, «NCTUns 4.0: An Integrated Simulation Platform for Vehicular Traffic, Communication, and Network Researches», en *2007 IEEE 66th Vehicular Technology Conference*, sep. de 2007, págs. 2081-2085. DOI: **10.1109/VETECF.2007.437**.
 - [17] S. Y. Wang y C. C. Lin, «NCTUns 6.0: A Simulator for Advanced Wireless Vehicular Network Research», en *2010 IEEE 71st Vehicular Technology Conference*, mayo de 2010, págs. 1-2. DOI: **10.1109/VETECS.2010.5494212**.
 - [18] M. Piorkowski, M. Raya, A. L. Lugo, P. Papadimitratos, M. Grossglauser y J.-P. Hubaux, «TraNS: realistic joint traffic and network simulator for VANETs», *ACM SIGMOBILE mobile computing and communications review*, vol. 12, n.º 1, págs. 31-33, 2008.
 - [19] Y. Zhao, A. Wagh, Y. Hou, K. Hulme, C. Qiao y A. W. Sadek, «Integrated traffic-driving-networking simulator for the design of connected vehicle applications: eco-signal case study», *Journal of Intelligent Transportation Systems*, vol. 20, n.º 1, págs. 75-87, 2016.
 - [20] C. Lochert, A. Barthels, A. Cervantes, M. Mauve y M. Caliskan, «Multiple simulator interlinking environment for IVC», en *Proceedings of the 2nd ACM international workshop on Vehicular ad hoc networks*, ACM, 2005, págs. 87-88.
 - [21] C. Sommer, R. German y F. Dressler, «Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis», *IEEE Transactions on Mobile Computing*, vol. 10, n.º 1, págs. 3-15, ene. de 2011, ISSN: 1536-1233. DOI: **10.1109 / TMC . 2010 . 133**.
 - [22] A. Wegener, M. Piórkowski Michałand Raya, H. Hellbrück, S. Fischer y J.-P. Hubaux, «TraCI: An Interface for Coupling Road Traffic and Network Simulators», en *Proceedings of the 11th Communications and Networking Simulation Symposium*, ép. CNS '08, Ottawa, Canada: ACM, 2008, págs. 155-163, ISBN: 1-56555-318-7. DOI: **10.1145 / 1400713 . 1400740**. dirección: <http://doi.acm.org/10.1145/1400713.1400740>.
 - [23] A. Kröller, D. Pfisterer, C. Buschmann, S. P. Fekete y S. Fischer, «Shawn: A new approach to simulating wire-

less sensor networks», *arXiv preprint*
cs/0502003, 2005.

Apéndice A

Códigos

Código A.1: Archivo `src/plugin.c` en su totalidad.

```
1 #include "programmer.h"
2 #include <thread>
3 #include "TraCIAPI/TraCIServer.h"
4 #include <shellapi.h>
5 #include "TraCIAPI/VehicleManager.h"
6 #include "TraCIAPI/Utils.h"
7
8 #define DEFAULT_PORT 5000
9 #define CMDARG_PORT "--traci_port="
10
11 std::thread* runner;
12 traci_api::TraCIServer* server;
13
14
15 /* checks a string for a matching prefix */
16 bool starts_with(std::string const& in_string,
17                 std::string const& prefix)
18 {
19     return prefix.length() <= in_string.length() &&
20         std::equal(prefix.begin(), prefix.end(), in_string.begin());
21 }
22
23 void runner_fn()
24 {
25     try {
26         //try to get port from command line arguments
27         int argc;
28         LPWSTR* argv = CommandLineToArgvW(GetCommandLineW(), &argc);
29         std::string prefix(CMDARG_PORT);
30
31         int port = DEFAULT_PORT; // if it fails, use the default port
32         for (int i = 0; i < argc; i++)
```

```

33     {
34         // convert from wstring to normal string
35         std::wstring temp(argv[i]);
36         std::string str(temp.begin(), temp.end());
37
38         // check if argument prefix matches
39         if (starts_with(str, prefix))
40         {
41             std::string s_port = str.substr(prefix.length(),
42                 str.npos);
43             try
44             {
45                 port = std::stoi(s_port);
46             }
47             catch (...)
48             {
49                 traci_api::infoPrint("Invalid port identifier -
50                     Falling back to default port");
51                 port = DEFAULT_PORT;
52             }
53         }
54
55         server = new traci_api::TraCIServer(port);
56         server->run();
57         server->close();
58         delete(server);
59     }
60     catch (std::exception& e)
61     {
62         traci_api::debugPrint("Uncaught exception in server thread.");
63         traci_api::debugPrint(e.what());
64         traci_api::debugPrint("Exiting...");
65         throw;
66     }
67
68     // Called once after the network is loaded.
69     void qpx_NET_postOpen(void)
70     {
71         qps_GUI_singleStep(PTRUE);
72         traci_api::infoPrint("TraCI support enabled");
73         runner = new std::thread(runner_fn);
74     }
75
76     void qpx_NET_reload()
77     {
78         server->close();
79         runner->join();
80         qpx_NET_postOpen();

```

```
81 }
82
83 void qpx_VHC_release(VEHICLE* vehicle)
84 {
85     traci_api::VehicleManager::getInstance()->vehicleDepart(vehicle);
86 }
87
88 void qpx_VHC_arrive(VEHICLE* vehicle, LINK* link, ZONE* zone)
89 {
90     traci_api::VehicleManager::getInstance()->vehicleArrive(vehicle);
91 }
92 }
```

Código A.2: Métodos base de todas las suscripciones.

```
1 int traci_api::VariableSubscription::checkTime() const
2 {
3     int current_time =
4         Simulation::getInstance()->getCurrentTimeMilliseconds();
5     if (beginTime > current_time) // begin time in the future
6         return -1;
7     else if (beginTime <= current_time && current_time <= endTime) //
8         within range
9         return 0;
10    else // expired
11        return 1;
12 }
13
14 uint8_t
15 traci_api::VariableSubscription::handleSubscription(tcpip::Storage&
16 output, bool validate, std::string& errors)
17 {
18     int time_status = checkTime();
19     if (!validate && time_status < 0) // not yet (skip this check if
20         validating, duh)
21         return STATUS_TIMESTEPNOTREACHED;
22     else if (time_status > 0) // expired
23         return STATUS_EXPIRED;
24
25     // prepare output
26     output.writeUnsignedByte(getResponseCode());
27     output.writeString(objID);
28     output.writeUnsignedByte(vars.size());
29
30     bool result_errors = false;
31
32     // get ze vahriables
33     tcpip::Storage temp;
34     for (uint8_t sub_var : vars)
35     {
36         // try getting the value for each variable,
37         // recording errors in the output storage
38         try {
39             output.writeUnsignedByte(sub_var);
40             getObjectVariable(sub_var, temp);
41             output.writeUnsignedByte(traci_api::STATUS_OK);
42             output.writeStorage(temp);
43         }
44         // ReSharper disable once CppEntityNeverUsed
45         catch (NoSuchObjectError& e)
46         {
47             // no such object
48             errors = "Object " + objID + " not found in simulation.";
49         }
50     }
51 }
```

```

44         return STATUS_OBJNOTFOUND;
45     }
46     catch (std::runtime_error& e)
47     {
48         // unknown error
49         result_errors = true;
50         output.writeUnsignedByte(traci_api::STATUS_ERROR);
51         output.writeUnsignedByte(VTYPE_STR);
52         output.writeString(e.what());
53         errors += std::string(e.what()) + "; ";
54     }
55
56     temp.reset();
57 }
58
59 if (validate && result_errors)
60     // if validating this subscription, report the errors.
61     // that way the subscription is not added to the sub
62     // vector in TraCIServer
63     return STATUS_ERROR;
64 else
65     // else just return the subscription to the client,
66     // and let it decide what to do about the errors.
67     return STATUS_OK;
68 }
69
70 uint8_t traci_api::VariableSubscription::updateSubscription(uint8_t
    sub_type, std::string obj_id, int begin_time, int end_time,
    std::vector<uint8_t> vars, tcpip::Storage& result_store,
    std::string& errors)
71 {
72     if (sub_type != this->sub_type || obj_id != objID)
73         // we're not the correct subscription,
74         // return NO UPDATE
75         return STATUS_NOUPD;
76
77     if (vars.size() == 0)
78         // 0 vars => cancel this subscription
79         return STATUS_UNSUB;
80
81     // backup old values
82     int old_start_time = this->beginTime;
83     int old_end_time = this->endTime;
84     std::vector<uint8_t> old_vars = this->vars;
85
86     // set new values and try
87     this->beginTime = begin_time;
88     this->endTime = end_time;
89     this->vars = vars;
90

```

```

91 // validate
92 uint8_t result = this->handleSubscription(result_store, true,
93     errors);
94
95 if (result == STATUS_EXPIRED)
96     // if new time causes subscription to expire, just unsub
97     return STATUS_UNSUB;
98 else if (result != STATUS_OK)
99 {
100     // reset values if the new values
101     // cause errors on evaluation
102     this->beginTime = old_start_time;
103     this->endTime = old_end_time;
104     this->vars = old_vars;
105 }
106
107 return result;
108 }

```

Código A.3: Método de actualización y creación de suscripciones en TraCIServer en su totalidad.

```
1 void traci_api::TraCIServer::addSubscription(uint8_t sub_type,
2     std::string object_id, int start_time, int end_time,
3     std::vector<uint8_t> variables)
4 {
5     std::string errors;
6     tcpip::Storage temp;
7
8     // first check if this corresponds to an update for an existing
9     // subscription
10    for (auto it = subs.begin(); it != subs.end(); ++it)
11    {
12        uint8_t result = (*it)->updateSubscription(sub_type,
13            object_id, start_time, end_time, variables, temp, errors);
14
15        switch (result)
16        {
17            case VariableSubscription::STATUS_OK:
18                // update ok, return now
19                debugPrint("Updated subscription");
20                writeStatusResponse(sub_type, STATUS_OK, "");
21                writeToOutputWithSize(temp, true);
22                return;
23            case VariableSubscription::STATUS_UNSUB:
24                // unsubscribe command, remove the subscription
25                debugPrint("Unsubscribing...");
26                delete *it;
27                it = subs.erase(it);
28                // we don't care about the deleted iterator, since we
29                // return from the loop here
30                writeStatusResponse(sub_type, STATUS_OK, "");
31                return;
32            case VariableSubscription::STATUS_ERROR:
33                // error when updating
34                debugPrint("Error updating subscription.");
35                writeStatusResponse(sub_type, STATUS_ERROR, errors);
36                break;
37            case VariableSubscription::STATUS_NOUPD:
38                // no update, try next subscription
39                continue;
40            default:
41                throw std::runtime_error("Received unexpected result " +
42                    std::to_string(result) + " when trying to update
43                    subscription.");
44        }
45    }
46 }
```



```

40 // if we reach here, it means we need to add a new subscription.
41 // note: it could also mean it's an unsubscribe command for a car
    that reached its
42 // destination. Check number of variables and do nothing if it's
    0.
43
44 if(variables.size() == 0)
45 {
46     // unsub command that didn't match any of the currently
        running subscriptions, so just
47     // tell the client it's ok, everything's alright
48
49     debugPrint("Unsub from subscription already removed.");
50     writeStatusResponse(sub_type, STATUS_OK, "");
51     return;
52 }
53
54
55 debugPrint("No update. Adding new subscription.");
56 VariableSubscription* sub;
57 switch (sub_type)
58 {
59     case CMD_SUB_VHCVAR:
60         debugPrint("Adding VHC subscription.");
61         sub = new VehicleVariableSubscription(object_id, start_time,
            end_time, variables);
62         break;
63     case CMD_SUB_SIMVAR:
64         debugPrint("Adding SIM subscription.");
65         sub = new SimulationVariableSubscription(object_id,
            start_time, end_time, variables);
66         break;
67     default:
68         writeStatusResponse(sub_type, STATUS_NIMPL, "Subscription type
            not implemented: " + std::to_string(sub_type));
69         return;
70 }
71
72 uint8_t result = sub->handleSubscription(temp, true, errors); //
    validate
73
74 if (result == VariableSubscription::STATUS_EXPIRED)
75 {
76     debugPrint("Expired subscription.");
77
78     writeStatusResponse(sub_type, STATUS_ERROR, "Expired
        subscription.");
79     return;
80 }
81 else if (result != VariableSubscription::STATUS_OK)

```

```
82 {  
83     debugPrint("Error adding subscription.");  
84  
85     writeStatusResponse(sub_type, STATUS_ERROR, errors);  
86     return;  
87 }  
88  
89 writeStatusResponse(sub_type, STATUS_OK, "");  
90 writeToOutputWithSize(temp, true);  
91 subs.push_back(sub);  
92 }
```

Código A.4: Avance de simulación en el módulo `Simulation`.

```
1 int traci_api::Simulation::runSimulation(uint32_t target_timems)
2 {
3     auto current_simtime = this->getCurrentTimeSeconds();
4     auto target_simtime = target_timems / 1000.0;
5     int steps_performed = 0;
6
7     traci_api::VehicleManager::getInstance()->reset();
8
9     if (target_timems == 0)
10    {
11        debugPrint("Running one simulation step...");
12
13        qps_GUI_runSimulation();
14        traci_api::VehicleManager::getInstance()
15            ->handleDelayedTriggers();
16        steps_performed = 1;
17    }
18    else if (target_simtime > current_simtime)
19    {
20        debugPrint("Running simulation up to target time: " +
21            std::to_string(target_simtime));
22        debugPrint("Current time: " + std::to_string(current_simtime));
23
24        while (target_simtime > current_simtime)
25        {
26            qps_GUI_runSimulation();
27            steps_performed++;
28            traci_api::VehicleManager::getInstance()
29                ->handleDelayedTriggers();
30
31            current_simtime = this->getCurrentTimeSeconds();
32
33            debugPrint("Current time: " +
34                std::to_string(current_simtime));
35        }
36    }
37    else
38    {
39        debugPrint("Invalid target simulation time: " +
40            std::to_string(target_timems));
41        debugPrint("Current simulation time: " +
42            std::to_string(current_simtime));
43        debugPrint("Doing nothing");
44    }
45
46    stepcnt += steps_performed;
47    return steps_performed;
48 }
```

Código A.5: Obtención de los límites del escenario de transporte.

```
1 void traci_api::Simulation::getRealNetworkBounds(double& llx,  
2     double& lly, double& urx, double& ury)  
3 {  
4     /*  
5      * Paramics qpg_POS_network() function, which should return the  
6      * network bounds, does not make sense.  
7      * It returns coordinates which leave basically the whole network  
8      * outside of its own bounds.  
9      *  
10     * Thus, we'll have to "bruteforce" the positional data for the  
11     * network bounds.  
12     */  
13     // get all relevant elements in the network, and all their  
14     // coordinates  
15  
16     std::vector<float> x;  
17     std::vector<float> y;  
18  
19     int node_count = qpg_NET_nodes();  
20     int link_count = qpg_NET_links();  
21     int zone_count = qpg_NET_zones();  
22  
23     float tempX, tempY, tempZ;  
24  
25     for (int i = 1; i <= node_count; i++)  
26     {  
27         NODE* node = qpg_NET_nodeByIndex(i);  
28         qpg_POS_node(node, &tempX, &tempY, &tempZ);  
29  
30         x.push_back(tempX);  
31         y.push_back(tempY);  
32     }  
33  
34     for (int i = 1; i <= zone_count; i++)  
35     {  
36         ZONE* zone = qpg_NET_zone(i);  
37         int vertices = qpg_ZNE_vertices(zone);  
38         for (int j = 1; j <= vertices; j++)  
39         {  
40             qpg_POS_zoneVertex(zone, j, &tempX, &tempY, &tempZ);  
41  
42             x.push_back(tempX);  
43             y.push_back(tempY);  
44         }  
45     }  
46  
47     for (int i = 1; i <= link_count; i++)
```

```

44 {
45     // links are always connected to zones or nodes, so we only
46     // need
47     // to get position data from those that are curved
48     LINK* lnk = qpg_NET_linkByIndex(i);
49     if (!qpg_LNK_arc(lnk) && !qpg_LNK_arcLeft(lnk))
50         continue;
51
52     // arc are perfect sections of circles, thus we only need the
53     // start, end and middle point (for all lanes)
54     float len = qpg_LNK_length(lnk);
55     int lanes = qpg_LNK_lanes(lnk);
56
57     float g, b;
58
59     for (int j = 1; j <= lanes; j++)
60     {
61         // start points
62         qpg_POS_link(lnk, j, 0, &tempX, &tempY, &tempZ, &b, &g);
63
64         x.push_back(tempX);
65         y.push_back(tempY);
66
67         // middle points
68         qpg_POS_link(lnk, j, len / 2.0, &tempX, &tempY, &tempZ, &b,
69             &g);
70
71         x.push_back(tempX);
72         y.push_back(tempY);
73
74         // end points
75         qpg_POS_link(lnk, j, len, &tempX, &tempY, &tempZ, &b, &g);
76
77         x.push_back(tempX);
78         y.push_back(tempY);
79     }
80
81     // we have all the coordinates, now get maximums and minimums
82     // add some wiggle room as well, just in case
83     urx = *std::max_element(x.begin(), x.end()) + 100;
84     llx = *std::min_element(x.begin(), x.end()) - 100;
85     ury = *std::max_element(y.begin(), y.end()) + 100;
86     lly = *std::min_element(y.begin(), y.end()) - 100;
87 }

```