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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer 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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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

Índice de Tablas

Índice de Ilustraciones

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 EEUU[1], 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

[1]Office 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 incluídos 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 éstas 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 Inteligentes de Transporte

Cascetta define en [6] 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.

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, los ITS se pueden dividir en dos grandes categorías [7]; Sistemas de Infraestructura Inteligente y Sistemas de Vehículos Inteligentes.

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 a través de sistemas de comunicación vehículo-infraestructura (*V2I*). 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. Esta categoría se caracteriza por el extenso uso de comunicaciones vehículo-vehículo (*V2V*), es decir, redes de comunicaciones distribuidas *ad-hoc* (ver sección 2.1.2).

2.1.2. Tecnologías de Comunicaciones para ITS

Una de las principales características de los ITS es la capacidad del sistema de otorgar información a los usuarios para la optimización del sistema. Con este fin, se han establecido distintos tipos de categorías de tecnologías de transmisión inalámbrica para el uso en variados escenarios [8].

V2I

Vehicle-to-Infraestructure, V2I, se refiere a toda comunicación en un ITS que ocurra entre un vehículo y la infraestructura, por ejemplo, para la transmisión de información del estado de la ruta, velocidad máxima, etc.

V2V

Vehicle-to-Vehicle, V2V, es el nombre otorgado a la categoría de tecnologías que posibilitan la comunicación directa entre vehículos en un ITS. Este tipo de comunicaciones tiende a ser de índole crítico (por ejemplo, transmisión de advertencias por accidente), y deben poder funcionar en ausencia de un sistema centralizado, por lo que generalmente se utilizan redes *ad-hoc*; es decir, redes descentralizadas en las cuales cada vehículo conforma un nodo que se comunica directamente con sus vecinos.

V2X

Finalmente, *Vehicle-to-Any*, V2X, se refiere a la combinación de las dos categorías anteriores.

IEEE 802.11p/WAVE

El estándar más común para las tres categorías de comunicaciones mencionadas anteriormente es hoy en el IEEE 802.11p, también conocido como WAVE (*Wireless Access for Vehicular Applications*).

IEEE 802.11p es una modificación al estándar 802.11 de la IEEE – el cual define el funcionamiento de redes inalámbricas de área local (popularmente conocidas como *Wi-Fi*) – para adaptarlo al funcionamiento en Sistemas Inteligentes de Transporte [4]. Su principal modificación es la habilidad de nodos en la red de comunicarse directamente sin antes tener que asociarse y autentificarse, ya que esto es costoso en términos de tiempo y las conexiones en un ITS son extremadamente efímeras.

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. Generalmente, esto se hace a través del empleo de modelos de eventos discretos, es decir, simulaciones en las cuales el estado del modelo cambia en instantes de tiempo discreto [9].

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

La simulación de una red de comunicaciones inalámbrica consiste en tres etapas principales [10]:

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 [10]-[12].

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.

De esta manera es posible no sólo estudiar el efecto que tiene el movimiento de los vehículos sobre la red de comunicaciones en un Sistema Inteligente de Transporte, sino también se hace posible estudiar las repercusiones de la diseminación de información en la red vehicular. Por ejemplo, permite analizar el efecto de que un grupo de conductores cambien su ruta dentro de la red de transporte en respuesta a la recepción de una notificación de un accidente más adelante en su ruta original.

Simulator Name	*Number of Occurrences in Literature*	*License Type*
VISSIM	15	Commercial Software
PARAMICS	12	Commercial Software
CORSIM	10	FOSS
AIMSUN	9	Commercial Software
SUMO	5	FOSS

Figura 2.1: Los cinco simuladores más prominentes en la literatura (fuente: Mubasher y ul Qounain [13]).

2.2. Estado del Arte

2.2.1. Simuladores de Tráfico

Actualmente, existe una gran oferta de simuladores de tráfico, ya sean de fuente abierta o propietarios. Ratrout y Rahman listan 14 de estos en su análisis comparativo del año 2009 [11], mientras que Boxill y Yu, ya en el año 2000 presentaban 8 simuladores distintos en su estudio de simuladores para el desarrollo de Sistemas Inteligentes de Transporte [12].

En esta sección se presentará brevemente el estado del arte de los más prominentes de estos simuladores, basándose en la revisión de literatura realizada por Mubasher y ul Qounain en [13].

VISSIM

VISSIM es un entorno de simulación discreto y microscópico, desarrollado propietariamente por el Grupo PVT en Alemania [14]. Es un simulador generalista, capaz de modelar sistemas de transporte multi-modales, en los que interactúan tanto vehículos "convencionales" como bicicletas, tranvías y hasta trenes pesados [15]. Modela el movimiento de cada entidad dinámica- y estocásticamente, en instantes discretos de tiempo.

VISSIM es considerado actualmente el líder en términos de popularidad y número de publicaciones en estudios de sistemas de transporte.

Aimsun

Aimsun es un simulador de tráfico con una larga trayectoria, desarrollado por *TSS - Trasport Simulation Systems*, una empresa basada en Barcelona. El desarrollo del simulador comenzó en el año 1989, y actualmente se encuentra en su versión 8.2 [16].

Aimsun cuenta con la particularidad de ser un entorno integrado micro- y mesoscópico de simulación de tráfico. Esto le da adaptabilidad a los problemas; para redes que requieran detalle del movimiento de sus entidades, se utiliza el modelo macroscópico, mientras que para redes de mayor escala se puede utilizar el modelo mesoscópico.

Este simulador es muy popular en la literatura académica dada su extensibilidad y adaptabilidad a un gran número de escenarios. Sin embargo, existe una crítica común a su complicado nivel de programación de sus redes (se estima un complejidad hasta 8(**!**) veces mayor que para otros simuladores [17]), y a su necesidad de meticulosa calibración para obtener resultados realistas [11], [17].

CORSIM

TSIS-CORSIM, actualmente en su versión 6.3, es un simulador de tráfico de tipo microscópico desarrollado por el *Centro McTrans* del Instituto de Transportes de la Universidad de Florida [18]. Al igual que Aimsun, CORSIM es muy popular en la literatura académica, y destaca por ser más apto para el modelamiento de redes de transporte complejas.

El simulador incluye dos modelos de simulación microscópica distintos - NETSIM para entornos urbanos, y FRESIM para tráfico en carreteras y zonas rurales. Si bien esto significa una mayor especialización y modelos más precisos para cada uno de estos casos, esto viene en desmedro de la posibilidad de simular de manera integrada un entorno que incluya ambas categorías [17].

SUMO

SUMO, ***S****imulation of* ***U****rban* ***MO****bility* [19], es un simulador de sistemas de transporte desarrollado por el Instituto de Sistemas de Transporte Alemán [20]. Es de fuente abierta, y utiliza un modelo microscópico para la simulación de redes de transporte.

Comparado con los simuladores presentados anteriormente, SUMO es relativamente nuevo, y todavía no cuenta con el mismo nivel de soporte y renombre que CORSIM o Aimsun, especialmente en el área de investigación de sistemas de transporte. Sin embargo, su popularidad ha aumentado de manera exponencial los últimos años, y ha ganado relevancia especialmente en estudios de Sistemas Inteligentes de Transporte [21], alcanzando el primer lugar en cantidad de publicaciones relacionadas con comunicaciones vehiculares [22]. Se especula que esto se debe a su naturaleza abierta, lo cual lo hace más accesible a investigadores, y además significa que es naturalmente extensible y adaptable a nuevos desafíos.

Paramics

Paramics, desarrollado por Quadstone Paramics, un subsidiary de Pitney Bowess [23] es un simulador microscópico de redes de transporte. Es capaz de simular el espectro completo de tamaño de redes de transporte - desde intersecciones aisladas a redes de transporte a escala nacional.

El simulador cuenta también con una API de extensión para la implementación de *plugins* enfocados a la integración de aplicaciones de Sistemas Inteligentes de Transporte. Esta API permite interactuar con todos los aspectos de la simulación, desde la simple obtención de datos desde las entidades internas hasta la modificación de los modelos de movilidad internos.

Finalmente, Paramics es además el simulador de preferencia del Área de Transportes del Departamento de Ingeniería Civil de la Universidad de Chile.

2.2.2. Simuladores de Redes Inalámbricas

Kumar *et al.* realizaron en 2012 un estudio comparativo en el ámbito de Simuladores para Redes Inalámbricas [24], trabajo basado parcialmente en el estudio realizado en 2009 por Weingartner, vom Lehn y Wehrle sobre la eficiencia de estos simuladores [25]. Paralelamente, investigadores de la Universidad de Malasia y la Universidad Carlos III de Madrid, España, publicaron también en 2012 un estudio enfocado específicamente en aquellos entornos de software de fuente abierta para la simulación de redes inalámbricas de sensores [26].

A continuación se discutirán brevemente las particularidades de los siguientes cuatro entornos de simulación, destacados en los artículos previamente mencionados: GloMoSim/-QualNet, OMNeT++, ns-2 y ns-3. Se escogieron específicamente esto simuladores dada su prominencia en dichos estudios y en la literatura académica en general.

GloMoSim

En primer lugar, GloMoSim es un simulador de fuente abierta desarrollado por investigadores de la Universidad de California, Los Ángeles [27]. El simulador utiliza las capacidades de simulación de eventos discretos y paralelos otorgadas por el lenguaje de programación Parsec, desarrollado en el Laboratorio de Computación Paralela de la UCLA [28]. QualNet es un derivado comercial de este mismo software, basado en C++ en vez de Parsec.

Las desventajas de GloMoSim y QualNet son varias. Para nombrar un par, no presentan soporte para un número considerable de implementaciones de TCP, y su interfaz gráfica es deficiente. Finalmente, además GloMoSim ya no se encuentra en desarrollo activo, por lo que es poco probable que esto se solucione en el futuro.

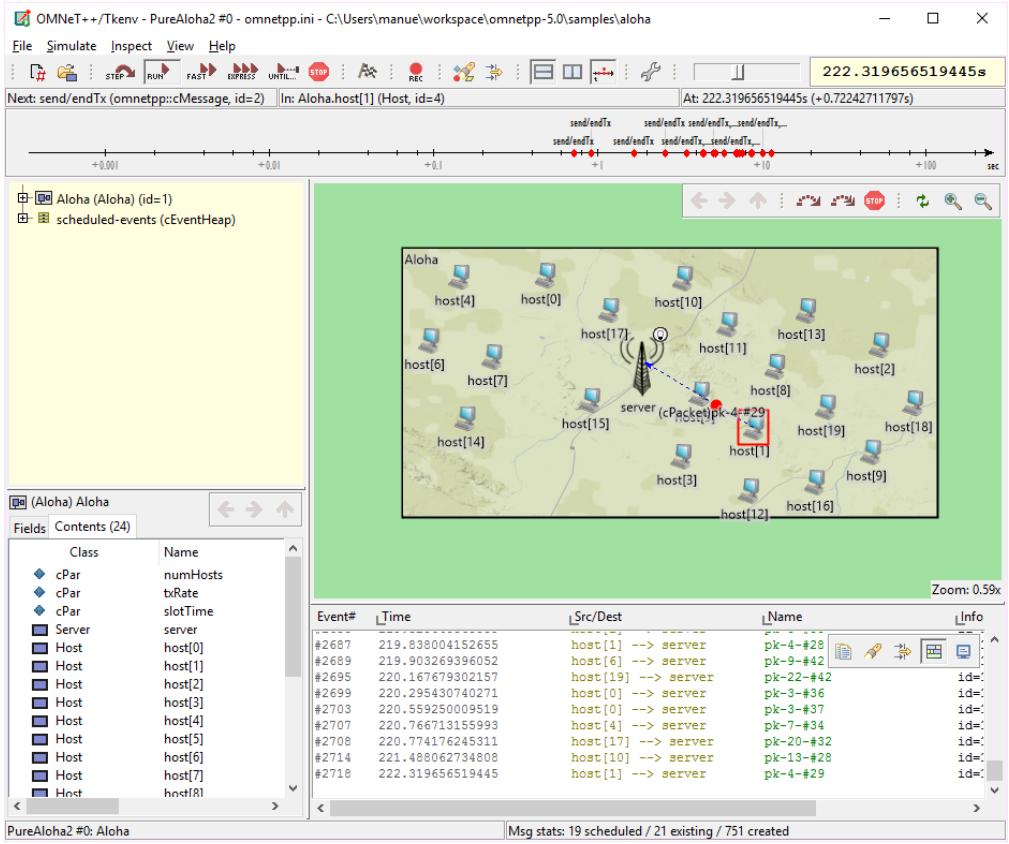


Figura 2.2: Entorno de simulación gráfica de OMNeT++.

OMNeT++

OMNeT++, ***O**bjective **M**odular **Net**work **T**estbed in C++*, es un entorno modular y basado en componentes para la simulación de sistemas de eventos discretos [29]. Está escrito en C++, y si bien en estricto rigor OMNeT++ en sí sólo conforma el *framework* genérico para la definición de modelos, la distribución incluye además múltiples extensiones para el modelamiento de redes de comunicación – siendo la principal de éstas INET.

INET incluye modelos para la simulación de múltiples *stacks* de protocolos para la comunicación tanto cableada como inalámbrica, a través de una gran cantidad de protocolos (IPV6, WSN, etc.). Finalmente, INET incluye además modelos de movilidad, para el estudio de redes con nodos en movimiento.

OMNeT++ presenta una gran ventaja en su diseño modular y extensible, y se posiciona como una excelente opción para simulaciones que requieran un alto nivel de flexibilidad.

ns-2 y ns-3

ns-2 es un simulador de eventos discretos para la simulación de redes de comunicación, cuyo desarrollo comenzó en 1989 y que a lo largo de los años ha recibido grandes contribuciones tanto de la comunidad científico como de corporaciones como DARPA, Xerox, etc. Gracias a

su larga trayectoria y extenso soporte, actualmente cuenta con un gran renombre en academia.

El simulador y sus módulos en sí están escritos en C++, pero se utiliza una extensión del lenguaje Tcl para su configuración y la definición de topologías de red. Esta decisión de diseño fue producto de un deseo de evitar la recompilación del simulador al realizar cambios en algún escenario, lo cual tenía mucho sentido en un tiempo en que la compilación implicaba extensos tiempos de espera. Hoy en día sin embargo, con los avances en potencia computacional, es más una desventaja, perjudicando la escalabilidad del sistema [25] a cambio de una marginal mejora en tiempos de recompilación.

ns-3 es considerado el sucesor de ns-2, llevando el exitoso simulador al siglo XXI. A diferencia de su ancestro, ns-3 está escrito completamente en C++, y opcionalmente algunos módulos pueden definirse en Python. Además, ns-3 incluye extensas optimizaciones en términos de escalabilidad y paralelismo, a cambio de una incompatibilidad con antiguos modelos desarrollados para ns-2

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.* [5], la mayor 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 manera: *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 [30], [31] 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.

Otro ejemplo de esto es la investigación presentada por Goebel *et al.* en [32]. En este

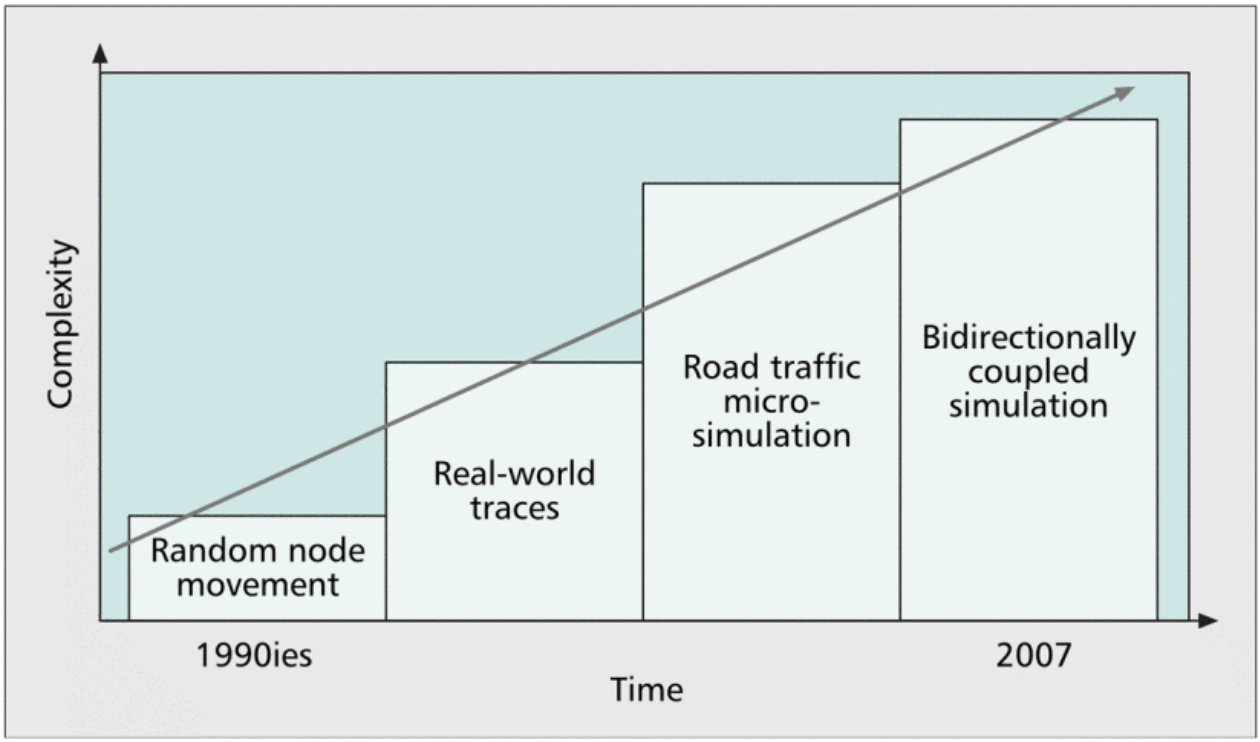


Figura 2.3: Evolución de simulaciones integradas para ITS (fuente: Sommer y Dressler [33]).

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 [33]; 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 [34], [35], 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 [35]), 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 [36], 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 [37] y de la Universidad de Düsseldorf [38]. 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 [10] también sufre este mismo problema, además de temas relacionados a la eficiencia del *framework* desarrollado por la autora, principalmente ligados a la elección de mecanismo de comunicación entre los simuladores (archivos en disco).

Finalmente, Sommer, German y Dressler presentan su solución en [39]: 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 transporte[1], 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

Arquitectura General y Funcionalidad Implementada

3.1. 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 interpreta mensajes entrantes a través de un *socket* TCP, ejecuta las acciones solicitadas, y responde a través del mismo medio.

La arquitectura general del *framework* se desarrolló en dos versiones distintas, la primera de éstas siendo descartada al realizar las pruebas de validación del proyecto. A continuación, se describirán brevemente estas dos iteraciones del diseño del software, destacando principalmente las razones del descarte de la versión preliminar.

Arquitectura Preliminar

Originalmente, el *framework* se implementó como un hilo de ejecución (un *thread*) paralelo a Paramics. La principal ventaja de este diseño era evitar el bloqueo de la interfaz del simulador al encontrarse el servidor TraCI bloqueado esperando mensajes entrantes en el *socket*.

Un diagrama de la arquitectura general de esta implementación puede observarse en la figura 3.1. Al iniciarse Paramics, el *plugin* inicializaba el servidor en un *thread* paralelo; el servidor luego se enlazaba a un *socket* TCP y se bloqueaba en espera de mensajes entrantes desde un cliente TraCI (en nuestro caso, VEINS). Al recibir una serie de comandos TraCI, el servidor los interpretaba, comunicándose con Paramics a través de su API, obteniendo datos y modificando el estado de la simulación. El servidor interpretaba todos los comandos en un mensaje TraCI antes de enviar todos los mensajes de respuesta correspondientes en un único

mensaje TraCI.

Esta arquitectura funcionaba de manera eficiente y permitía la ejecución de la simulación de Paramics completamente sin la intervención del usuario (ya que el *thread* mismo del *plugin* era capaz de llamar el método de inicio de simulación).

Sin embargo, al comenzar a realizar pruebas con redes de tamaño más extenso se presentó un problema imprevisto, y – a la larga – irreparable sin acceso al código fuente de Paramics. El problema radicaba en la función de avance de simulación definido en la API de Paramics, `qps_GUI_runSimulation()`, la cual, como se descubrió más tarde, también actualiza la interfaz gráfica de el modelador de Paramics a través de llamados a la librería Qt4 [40]. Estos llamados no son *thread-safe*[1], y en redes grandes de Paramics generan *data races* al ser invocados desde un *thread* paralelo al principal del simulador. Esto finalmente generaba corrupción de memoria en el motor gráfico (principalmente, lecturas de direcciones inválidas de memoria), lo cual causaba un error fatal en la simulación.

Se estudiaron múltiples maneras de resolver este problema manteniendo la estructura paralela del *plugin* sin éxito, ya que la única manera confiable de forzar un avance de la simulación desde el *plugin* es a través de la función anteriormente mencionada. Se decidió entonces abordar el problema desde un ángulo distinto, enfoque que se discutirá en la siguiente sección.

Arquitectura Final

El problema presentado por la incompatibilidad de `qps_GUI_runSimulation()`, función de avance de simulación de la API de Paramics, con múltiples *threads* implicó la necesidad de reevaluar la arquitectura general del *framework* en su totalidad. Se decidió descartar la idea de un *thread* paralelo para el servidor, y se implementó un esquema secuencial de interpretación de mensajes TraCI, utilizando *loops* bloqueantes para controlar la ejecución de pasos de simulación.

Esta arquitectura puede visualizarse en la figura 3.2. Al principio de cada paso de simulación, el simulador invoca la función `qpx_CLK_startOfSimLoop()`, definida en el *plugin*, antes de realizar cualquier otra acción. Esta función a su vez invoca el método `preStep()` del servidor, dentro del cual se ejecuta un *loop* de interpretación de comandos TraCI (y de envío de respuestas a éstos). Este *loop* se interrumpe al recibir un mensaje de paso de simulación, retornando así de `preStep()` y `qpx_CLK_startOfSimLoop()`, y liberando al simulador para que realice su procedimiento interno de avance de simulación.

Luego de realizar el avance de simulación, Paramics invoca la función `qpx_CLK_endOfSimLoop()`, también definida en el *plugin*. Esta invoca a su vez el método `postStep()` del servidor, el cual se encarga de realizar la recolección de datos post-paso de simulación, de terminar de interpretar eventuales comandos recibidos previo al paso de simulación y de enviar respuestas pendientes al cliente. Finalmente, esta función retorna el control de la ejecución a Paramics,

[1]Es decir, no incluyen medidas para asegurar el acceso exclusivo de recursos a un sólo *thread*.

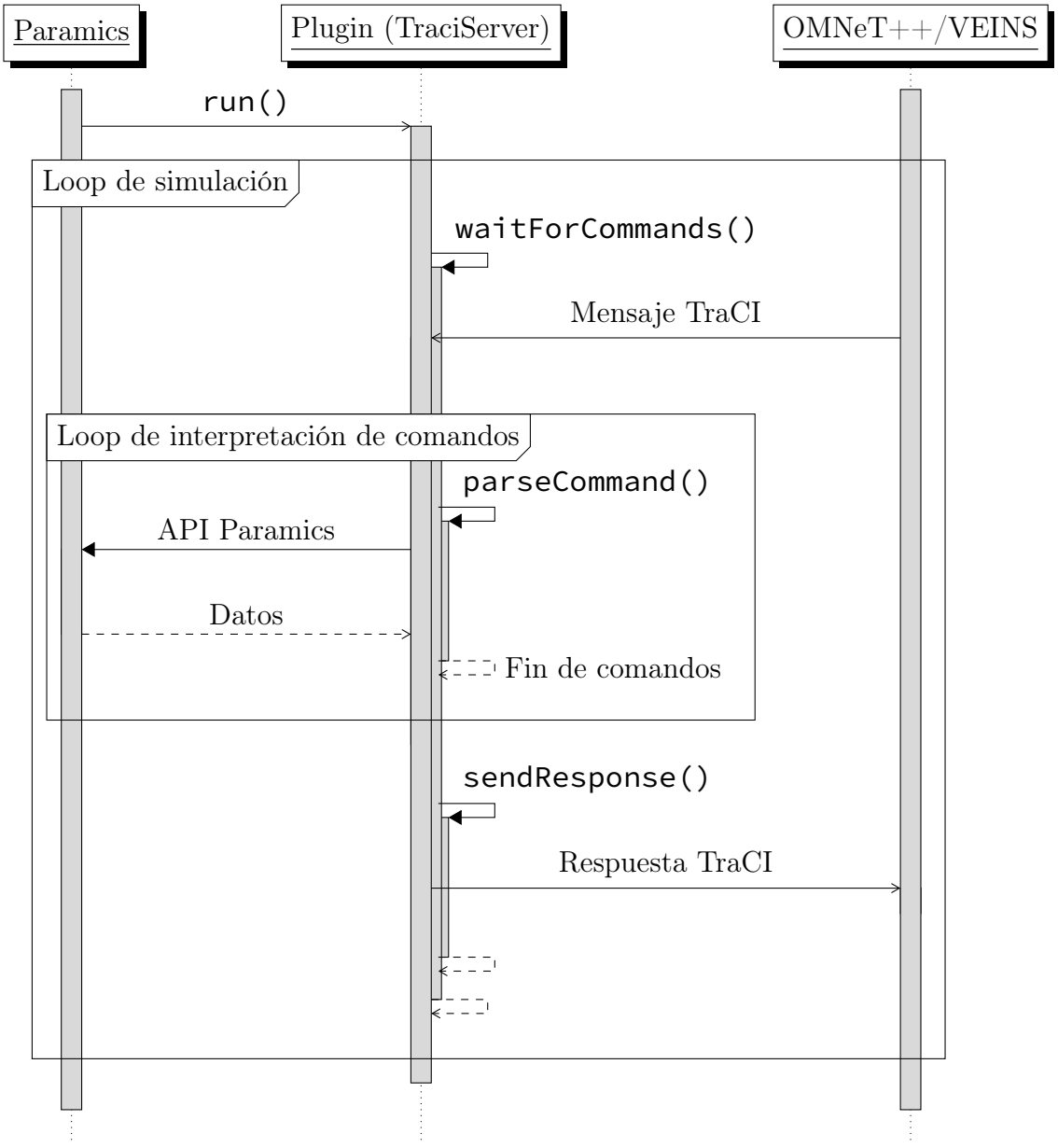


Figura 3.1: Arquitectura preliminar

y el ciclo comienza nuevamente.

Mediante esta arquitectura se logró eliminar por completo el problema presentado por `qps_GUI_runSimulation()`, obteniendo incluso una leve mejora en rendimiento respecto al diseño antiguo, dado que, ya que todo corre en un sólo *thread*, se evita el uso de elementos de sincronización, los cuales pueden agregar *overhead* al procedimiento.

Este nuevo diseño presenta una única desventaja: es necesario el inicio de la simulación de manera manual por parte del usuario, luego de lo cual funciona de manera autónoma. Esto ya que no existe manera de iniciar el *loop* de simulación de Paramics a través de la API sin recurrir a *threads*.

Finalmente, 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 $\triangle T$, 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

$$
\begin{gathered}
T_\mathrm{i} + (n \times \triangle T) = T_f \\
T_\mathrm{i} + ((n-1) \times \triangle T) = T'_f \\
T_f \geq T_o \\
T'_f < T_o
\end{gathered}
$$

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 que dependan del tiempo de simulación por lo menos hasta dicho instante.

3.2. Funcionalidad Implementada

El protocolo TraCI define más de 30 comandos distintos, cada uno con una gran cantidad de variables y parámetros asociados (ver apéndice A para una descripción más detallada del funcionamiento de este protocolo). 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 simulaciones de ITS.

3.2.1. Comandos Implementados

Comandos de Control de Simulación

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

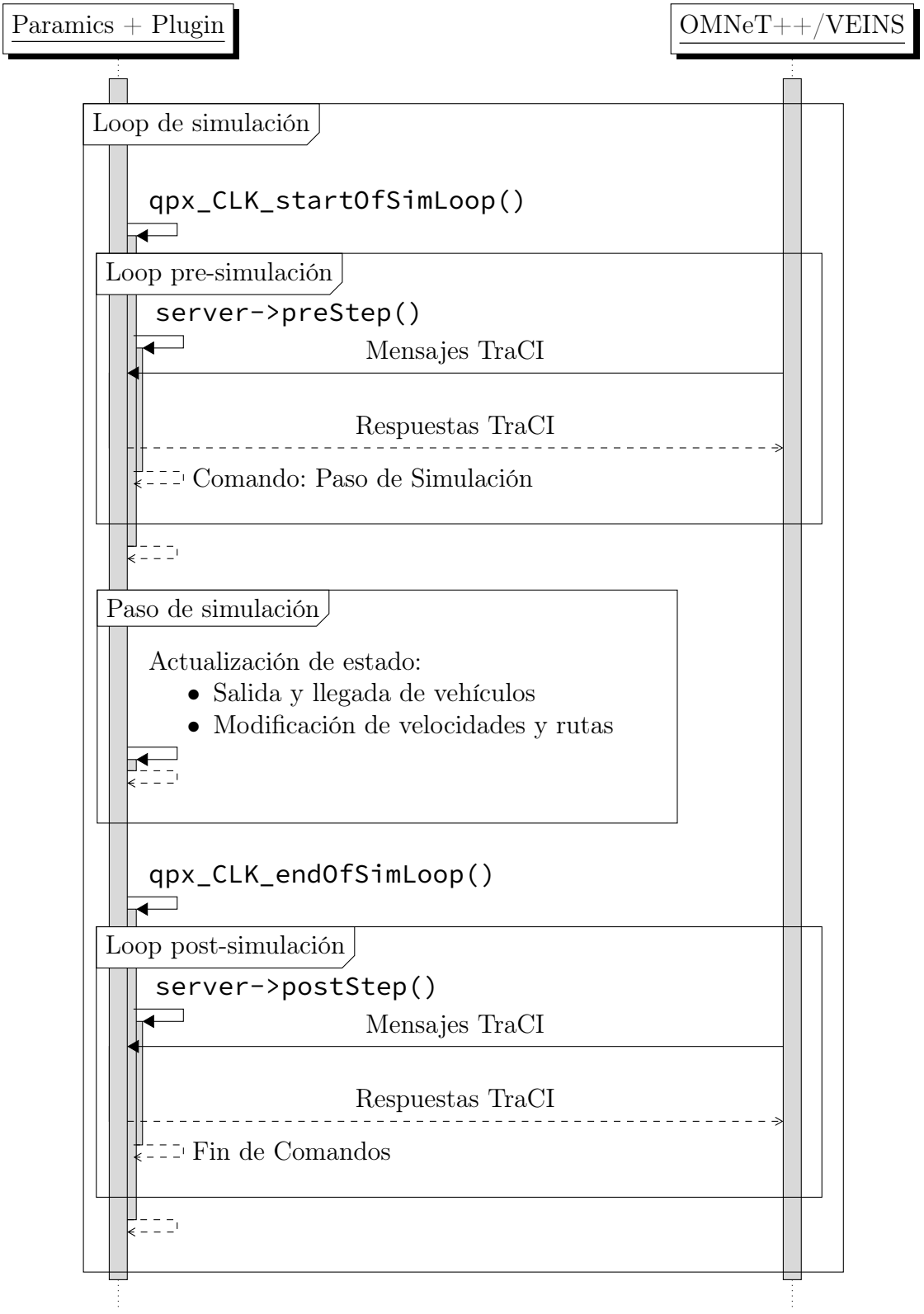


Figura 3.2: Arquitectura final del *framework*

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
- `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 comienzaron 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.

3.2.2. 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)

Capítulo 4

Implementación

4.1. Organización Lógica del Código

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. La organización en archivos de éstos módulos puede observarse en la figura 4.1.

Cabe notar también que se utilizó el *namespace* `traci_api` para agrupar los elementos propios del framework.

4.2. 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 y *override* (prefijos `QPX` y `QPO`, ver apéndice B para un detalle sobre la API de Paramics) a ser invocadas por Paramics. 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 C.1 en los anexos.

`void qpx_NET_postOpen()`

Invocada inmediatamente luego de que Paramics carga la red y el *plugin*, esta función 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

```
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 4.1: Estructura de archivos del código fuente del framework.

ejecución de Paramics. De no haberse especificado puerto, utiliza uno por defecto.

2. Inicializar un objeto `TraCIServer` (ver sección 4.3) encargado de las conexiones entrantes en el puerto anteriormente definido.

`void qpx_CLK_startOfSimLoop()` y `void qpx_CLK_endOfSimLoop()`

Estas funciones se ejecutan antes y después de cada paso de simulación respectivamente, y llaman a los procedimientos correspondientes en el servidor, los método `preStep()` y `postStep()`. Ver sección 4.3 para más detalle sobre estos métodos y el avance de simulación en general.

`void qpx_VHC_release(...)` y `void qpx_VHC_arrive(...)`

`qpx_VHC_release(VEHICLE* vehicle)` es invocada por Paramics cada vez que un vehículo es liberado a la red de transporte. Simplemente se encarga de notificar al `VehicleManager` (ver sección 4.5) para su inclusión en el modelo interno del *plugin*.

Por otro lado, `qpx_VHC_arrive(VEHICLE* vehicle, LINK* link, ZONE* zone)` es invocada cuando un vehículo alcanza su destino final, y notifica al `VehicleManager` para eliminar el vehículo en cuestión de la representación interna.

`int qpo_RTM_decision(...)`

Esta función de *override* sobrescribe la elección de rutas de vehículos que hayan recibido un mensaje de cambio de ruta a través de TraCI.

`void qpx_VHC_transfer(...)`

Este método es ejecutado por Paramics cada vez que un vehículo pasa de una calle a otra, y se utiliza para determinar si es necesario recalcular la ruta del vehículo en cuestión.

`float qpo_CFM_leadSpeed(...)` y **`float qpo_CFM_followSpeed(...)`**

Sobrescriben las velocidades de vehículos que hayan recibido modificaciones a su comportamiento desde TraCI.`leadSpeed()` realiza esta modificación para vehículos que no tienen otro vehículo delante, y `followSpeed()` lo hace para aquellos que se encuentran detrás de otro vehículo.

4.3. 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.
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`.

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 BSD[2].

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

4.3.1. Inicio de conexión TraCI

Como se mencionó en la sección 4.2, 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 `waitForConnection()` (código 4.1).

Este método es simple: imprime información pertinente sobre el *plugin* en la ventana de información de Paramics, y luego espera a recibir una conexión entrante. Este método es además el único del *framework* que corre en un *thread* paralelo a Paramics en la arquitectura final. Se decidió implementarlo de esta manera para que el inicio de Paramics fuera más fluido y no se bloqueara la interfaz mientras el servidor espera una conexión desde un cliente TraCI.

4.3.2. Recepción e Interpretación de Comandos Entrantes

Como se explicó en la sección 3.1, la interpretación de comandos entrantes y el avance del *loop* de simulación se realizan en dos etapas; una previa al paso de simulación y una posterior a éste. Los métodos encargados de esto son `preStep()` y `postStep()`, los cuales se detallarán a continuación.

`preStep()`

El método `preStep()` es invocado por Paramics al principio de cada iteración del *loop* de simulación, antes de ejecutar cualquier otra función. En este método se encarga de recibir mensajes nuevos entrante a través del *socket* desde el cliente TraCI, e interpreta los comandos dentro de un *loop*.

```
1 void traci_api::TraCIServer::preStep()
2 {
3    std::lock_guard<std::mutex> lock(socket_lock);
```

[1]Fuente 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* [41]. Su fuente original se encuentra en `https://github.com/itm/shawn/tree/master/src/apps/tcpip`

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

```
/**
 * \brief Starts this instance, binding it to a port and awaiting
    connections.
 */
void traci_api::TraCIServer::waitForConnection()
{
   running = true;
   std::string version_str = "Paramics TraCI plugin v" +
      std::string(PLUGIN_VERSION) + " on Paramics v" +
      std::to_string(qpg_UTL_parentProductVersion());
   infoPrint(version_str);
   infoPrint("Timestep size: " +
      std::to_string(static_cast<int>(qpg_CFG_timeStep() * 1000.0f)) +
      "ms");
   infoPrint("Simulation start time: " +
      std::to_string(Simulation::getInstance()
      ->getCurrentTimeMilliseconds()) + "ms");
   infoPrint("Awaiting connections on port " + std::to_string(port));

   {
      std::lock_guard<std::mutex> lock(socket_lock);
      ssocket.accept();
   }

   infoPrint("Accepted connection");
}
```

Código 4.1: Rutina de inicio de conexión.

```
   if (multiple_timestep &&
      Simulation::getInstance()->getCurrentTimeMilliseconds() <
      target_time)
   {
      VehicleManager::getInstance()->reset();
      return;
   }

   multiple_timestep = false;
   target_time = 0;

   tcpip::Storage cmdStore; // individual commands in the message

   debugPrint("Waiting for incoming commands from the TraCI
      client...");

   // receive and parse messages until we get a simulation step
      command
   while (running && ssocket.receiveExact(incoming))
```

```
19      {
20          incoming_size = incoming.size();
21
22          debugPrint("Got message of length " +
                std::to_string(incoming_size));
23          //debugPrint("Incoming: " + incoming.hexDump());
24
25
26          /* Multiple commands may arrive at once in one message,
27          * divide them into multiple storages for easy handling */
28          while (incoming_size > 0 && incoming.valid_pos())
29          {
30              uint8_t cmdlen = incoming.readUnsignedByte();
31              cmdStore.writeUnsignedByte(cmdlen);
32
33              debugPrint("Got command of length " +
                    std::to_string(cmdlen));
34
35
36              for (uint8_t i = 0; i < cmdlen - 1; i++)
37                  cmdStore.writeUnsignedByte(incoming.readUnsignedByte());
38
39              bool simstep = this->parseCommand(cmdStore);
40              cmdStore.reset();
41
42              // if the received command was a simulation step command,
                    return so that
43              // Paramics can do its thing.
44              if (simstep)
45              {
46                  VehicleManager::getInstance()->reset();
47                  return;
48              }
49          }
50
51          this->sendResponse();
52          incoming.reset();
53          outgoing.reset();
54      }
55  }
```

Código 4.2: Método `preStep()` en `TraCIServer`

Nótese que `preStep()` continuamente interpreta, ejecuta y responde a comandos, y sólo retorna al recibir un comando de avance de simulación. De esta manera, retorna el control de la ejecución a Paramics, y el simulador mismo se encarga de realizar el paso de simulación.

En términos de la interpretación de los mensajes, 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* interno.

postStep()

Al recibir un comando de avance de simulación, `preStep()` inmediatamente retorna el control del flujo del programa a Paramics. El simulador entonces avanza la simulación, y luego ejecuta el método `postStep()` del servidor.

Este método tiene como fin la recopilación de las eventuales suscripciones existentes (ver sección 4.3.2), la interpretación de los comandos restantes en el último mensaje recibido antes del comando de avance de simulación y el envío de eventuales respuestas al cliente. Finalmente, este método retorna, y Paramics vuelve a iniciar una nueva iteración del *loop* de simulación y a ejecutar `preStep()`.

Cabe notar que `postStep()` sólo se ejecuta inmediatamente después de la ejecución de un paso de simulación por parte de Paramics, y por ende no se ejecuta si nunca se recibe un comando de avance de simulación.

```
1  void traci_api::TraCIServer::postStep()
2  {
3     // after each step, have VehicleManager update its internal state
4     VehicleManager::getInstance()->handleDelayedTriggers();
5
6     if (multiple_timestep &&
         Simulation::getInstance()->getCurrentTimeMilliseconds() <
         target_time)
7        return;
8
9     // after a finishing a simulation step command (completely),
         collect subscription results and
10    // check if there are commands remaining in the incoming storage
11    this->writeStatusResponse(CMD_SIMSTEP, STATUS_OK, "");
12
13    // handle subscriptions after simstep command
14    tcpip::Storage subscriptions;
15    this->processSubscriptions(subscriptions);
16    outgoing.writeStorage(subscriptions);
17
18    // finish parsing the message we got before the simstep command
19    tcpip::Storage cmdStore;
20    /* Multiple commands may arrive at once in one message,
```

```
21      * divide them into multiple storages for easy handling */
22     while (incoming_size > 0 && incoming.valid_pos())
23     {
24        uint8_t cmdlen = incoming.readUnsignedByte();
25        cmdStore.writeUnsignedByte(cmdlen);
26
27        debugPrint("Got command of length " + std::to_string(cmdlen));
28
29        for (uint8_t i = 0; i < cmdlen - 1; i++)
30           cmdStore.writeUnsignedByte(incoming.readUnsignedByte());
31
32        bool simstep = this->parseCommand(cmdStore);
33        cmdStore.reset();
34
35        // weird, two simstep commands in one message?
36        if (simstep)
37        {
38           if(!multiple_timestep)
39           {
40              multiple_timestep = true;
41              Simulation* sim = Simulation::getInstance();
42              target_time = sim->getCurrentTimeMilliseconds() +
                   sim->getTimeStepSizeMilliseconds();
43           }
44           VehicleManager::getInstance()->reset();
45           return;
46        }
47     }
```

Código 4.3: Método `postStep()` en `TraCIServer`

Interpretación de comandos TraCI

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 4.4. 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.

```
void traci_api::TraCIServer::parseCommand(tcpip::Storage& storage)
{
   /* ... */
   uint8_t cmdLen = storage.readUnsignedByte();
   uint8_t cmdId = storage.readUnsignedByte();
   tcpip::Storage state;
   /* ... */

   if (cmdId >= CMD_SUB_INDVAR && cmdId <= CMD_SUB_SIMVAR)
   {
      // subscription
      // | begin Time | end Time | Object ID | Variable Number | The
         list of variables to return
      /* read subscription params */
      /* ... */
      addSubscription(cmdId, oID, btime, etime, vars);
   }
   else
   {
      switch (cmdId)
      {
      case CMD_GETVERSION:
      /* ... */
      /* ... */
      default:
         debugPrint("Command not implemented!");
         writeStatusResponse(cmdId, STATUS_NIMPL, "Method not
            implemented.");
      }
   }
}
```

Código 4.4: Esqueleto de `parseCommand()`

Se definieron una serie de métodos en `TraCIServer` encargados de obtener variables de la simulación o modificar el estado de ésta. El funcionamiento de éstos es idéntico en todos los casos (a excepción de `cmdGetPolygonVar()`), y se limita al siguiente procedimiento (ver ejemplo en el código 4.5):

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.

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

```
{
   tcpip::Storage result;
   try
   {
      VehicleManager::getInstance()->packVehicleVariable(input,
         result);
      this->writeStatusResponse(CMD_GETVHCVAR, STATUS_OK, "");
      this->writeToOutputWithSize(result, false);
   }
   catch (NotImplementedError& e)
   {
      debugPrint("Variable not implemented");
      debugPrint(e.what());
      this->writeStatusResponse(CMD_GETVHCVAR, STATUS_NIMPL,
         e.what());
   }
   catch (std::exception& e)
   {
      debugPrint("Fatal error???");
      debugPrint(e.what());
      this->writeStatusResponse(CMD_GETVHCVAR, STATUS_ERROR,
         e.what());
      throw;
   }
}
```

Código 4.5: 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.

Evaluación de suscripciones

Como se explicó en la sección 4.3.2, luego de realizar un paso de avance de simulación, en `postStep()` se realiza la evaluación de las suscripciones activas en `TraCIServer`, mediante un llamado al método `processSubscriptions()`.

Como se detalla en el apéndice A, 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, 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

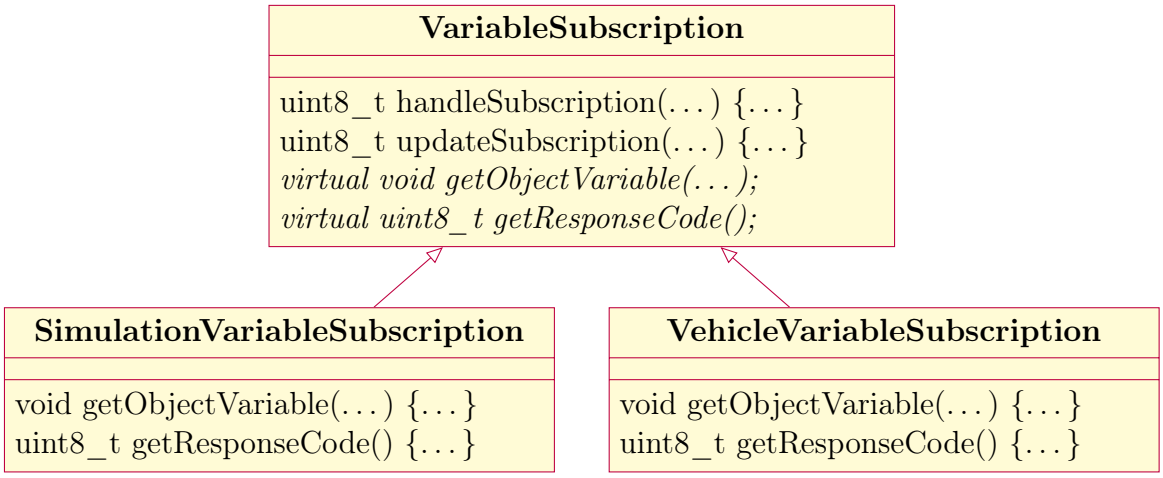


Figura 4.2: Diagrama de herencia, `VariableSubscription`

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/TraCIAPI/Subscriptions.h` y `src/TraCIAPI/Subscriptions.cpp` respectivamente).

La raíz de éste árbol, la clase `VariableSuscription`, implementa la funcionalidad completa de evaluación y actualización de una suscripción en los métodos `handleSubscription()` y `updateSubscription()` (implementación completa de estos métodos en C.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 todo 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 4.6). 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();)
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 4.6: Rutina de evaluación de suscripciones en `TraCIServer`. `subs` es una variable de instancia de `TraCIServer` correspondiente a un vector de punteros `VariableSuscription*`, poblado de elementos de clases derivadas de `VariableSuscription`.

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 4.4,línea 15). Este código puede estudiarse en su totalidad en el anexo C, código C.3, sin embargo, a continuación se explicará brevemente su funcionamiento con algunos extractos de código.

Como se comenta en la descripción del protocolo TraCI (apéndice A), 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 C.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 `addSubscription()` 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.

```
for (auto it = subs.begin(); it != subs.end(); ++it)
{
   uint8_t result = (*it)->updateSubscription(sub_type,
      object_id, start_time, end_time, variables, temp, errors);

   switch (result)
   {
   case VariableSubscription::STATUS_OK:
      // update ok, return now
      debugPrint("Updated subscription");
      writeStatusResponse(sub_type, STATUS_OK, "");
      writeToOutputWithSize(temp, true);
      return;
   case VariableSubscription::STATUS_UNSUB:
      // unsubscribe command, remove the subscription
      debugPrint("Unsubscribing...");
      delete *it;
      it = subs.erase(it);
      // we don't care about the deleted iterator, since we
         return from the loop here
      writeStatusResponse(sub_type, STATUS_OK, "");
      return;
   case VariableSubscription::STATUS_ERROR:
      // error when updating
      debugPrint("Error updating subscription.");
      writeStatusResponse(sub_type, STATUS_ERROR, errors);
      break;
   case VariableSubscription::STATUS_NOUPD:
      // no update, try next subscription
      continue;
   default:
      throw std::runtime_error("Received unexpected result " +
```

```
            std::to_string(result) + " when trying to update
            subscription.");
31      }
32  }
```

Código 4.7: 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 4.8: Creación de una nueva suscripción. Notar la instanciación polimórfica.

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

Código 4.9: Escritura de datos en almacenamiento saliente.

4.4. Simulation

La principal funcionalidad de este módulo es 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. Este patrón de diseño tiene además la ventaja que simplifica el acceso a la instancia global de la clase en el sistema, desde cualquier otro objeto u función.

4.4.1. 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 3.2.1.

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`, 4.5, 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 C.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.

4.5. 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 3.2.2). 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.

4.5.1. 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 4.2.

Se utilizan las siguientes variables para almacenar información sobre el estado de la simulación en todo instante:

`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`.

`departed_vehicles` y **`arrived_vehicles`** Vectores de punteros a vehículos, actualizados por Paramics a través de las funciones de extensión de la API `qpx_VHC_release()` y `qpx_VHC_arrive()` en `plugin.c` (ver sección 4.2). Mantienen punteros a vehículos que iniciaron su viaje y que llegaron a su destino, respectivamente, en último paso de simulación. Se vacían al antes de cada paso.

`speed_controllers` Mapa que relaciona vehículos con controladores de velocidad (ver sección **??**), para efectuar cambios de velocidad dictados por el cliente TraCI.

`lane_set_triggers` *Hashmap* utilizado para relacionar vehículos con eventuales comandos de cambio de pista (ver sección **??**).

vhc_routes Mapa para el manejo de cambios de ruta desde TraCI (ver sección **??**).

4.5.2. 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.

[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.

[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.

[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. 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] (Abr. de 2017). U.S. Department of Transportation, Office of the Assistant Secretary for Research and Technology (OST-R), dirección: http://www.itsoverview.its.dot.gov/.

[8] K. Dar, M. Bakhouya, J. Gaber, M. Wack y P. Lorenz, «Wireless communication technologies for ITS applications [Topics in Automotive Networking]», *IEEE Communications Magazine*, vol. 48, n.º 5, págs. 156-162, 2010.

[9] 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.

[10] Y. Shalaby, «An Integrated Framework for Coupling Traffic and Wireless Network Simulations», Tesis de mtría., Department of Civil Engineering, University of Toronto, Canada, 2010.

[11] N. T. Ratrout 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.

[12] S. A. Boxill y L. Yu, «An evaluation of traffic simulation models for supporting ITS», *Houston, TX: Development Centre for Transportation Training and Research, Texas Southern University*, 2000.

[13] M. M. Mubasher y J. S. W. ul Qounain, «Systematic literature review of vehicular traffic flow simulators», en *2015 International Conference on Open Source Software Computing (OSSCOM)*, sep. de 2015, págs. 1-6. DOI: 10.1109/OSSCOM.2015.7372687.

[14] (Mayo de 2017). PVT VISSIM, dirección: http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/.

[15] M. Fellendorf y P. Vortisch, «Microscopic traffic flow simulator VISSIM», en *Fundamentals of traffic simulation*, Springer, 2010, págs. 63-93.

[16] (Mayo de 2017). Aimsun traffic modelling software, dirección: https://www.aimsun.com/aimsun/.

[17] S. L. Jones, A. J. Sullivan, N. Cheekoti, M. D. Anderson y D. Malave, «Traffic simulation software comparison study», *UTCA Report*, vol. 2217, 2004.

[18] (Mayo de 2017). TSIS-CORSIM™, dirección: https://mctrans.ce.ufl.edu/mct/index.php/tsis-corsim/.

[19] M. Behrisch, L. Bieker, J. Erdmann y D. Krajzewicz, «SUMO - Simulation of Urban MObility: An Overview», en *SIMUL 2011*, S. Ũ. of Oslo Aida Omerovic, R. I. .-.-. R. T. P. D. A. Simoni y R. I. .-.-. R. T. P. G. Bobashev, eds., ThinkMind, oct. de 2011. dirección: http://elib.dlr.de/71460/.

[20] (Mayo de 2017). Deutsches Zentrum für Luft- und Raumfahrt (DLR), dirección: http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-1221/1665_read-3070/.

[21] D. Krajzewicz, «Summary on Publications citing SUMO, 2002-2012», en *1st SUMO User Conference-SUMO 2013*, DLR, vol. 21, 2013, págs. 11-24.

[22] S. Joerer, C. Sommer y F. Dressler, «Toward reproducibility and comparability of IVC simulation studies: a literature survey», *IEEE Communications Magazine*, vol. 50, n.º 10, págs. 82-88, oct. de 2012, ISSN: 0163-6804. DOI: 10.1109/MCOM.2012.6316780.

[23] (Mar. de 2017). Quadstone Paramics website, dirección: http://www.paramics-online.com.

[24] A. Kumar, S. K. Kaushik, R. Sharma y P. Raj, «Simulators for Wireless Networks: A Comparative Study», en *2012 International Conference on Computing Sciences*, sep. de 2012, págs. 338-342. DOI: 10.1109/ICCS.2012.65.

[25] E. Weingartner, H. vom Lehn y K. Wehrle, «A Performance Comparison of Recent Network Simulators», en *2009 IEEE International Conference on Communications*, jun. de 2009, págs. 1-5. DOI: 10.1109/ICC.2009.5198657.

[26] A. R. Khan, S. M. Bilal y M. Othman, «A performance comparison of open source network simulators for wireless networks», en *2012 IEEE International Conference on Control System, Computing and Engineering*, nov. de 2012, págs. 34-38. DOI: 10.1109/ICCSCE.2012.6487111.

[27] X. Zeng, R. Bagrodia y M. Gerla, «GloMoSim: a library for parallel simulation of large-scale wireless networks», en *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, mayo de 1998, págs. 154-161. DOI: 10.1109/PADS.1998.685281.

[28] R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin y H. Y. Song,

«Parsec: a parallel simulation environment for complex systems», *Computer*, vol. 31, n.º 10, págs. 77-85, oct. de 1998, ISSN: 0018-9162. DOI: 10.1109/2.722293.

[29] A. Varga y R. Hornig, «An overview of the OMNeT++ simulation environment», en *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, ICST (Institute for Computer Sciences, Social-Informatics y Telecommunications Engineering), 2008, pág. 60.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] (Mayo de 2017). Qt | Cross-platform software development for embedded & desktop, dirección: https://www.qt.io/.

[41] A. Kröller, D. Pfisterer, C. Buschmann, S. P. Fekete y S. Fischer, «Shawn: A new approach to simulating wireless sensor networks», *arXiv preprint cs/0502003*, 2005.

[42] 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.

Apéndice A

TraCI

TraCI (**Tra**ffic **C**ontrol **I**nterface) 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 [42]. 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.

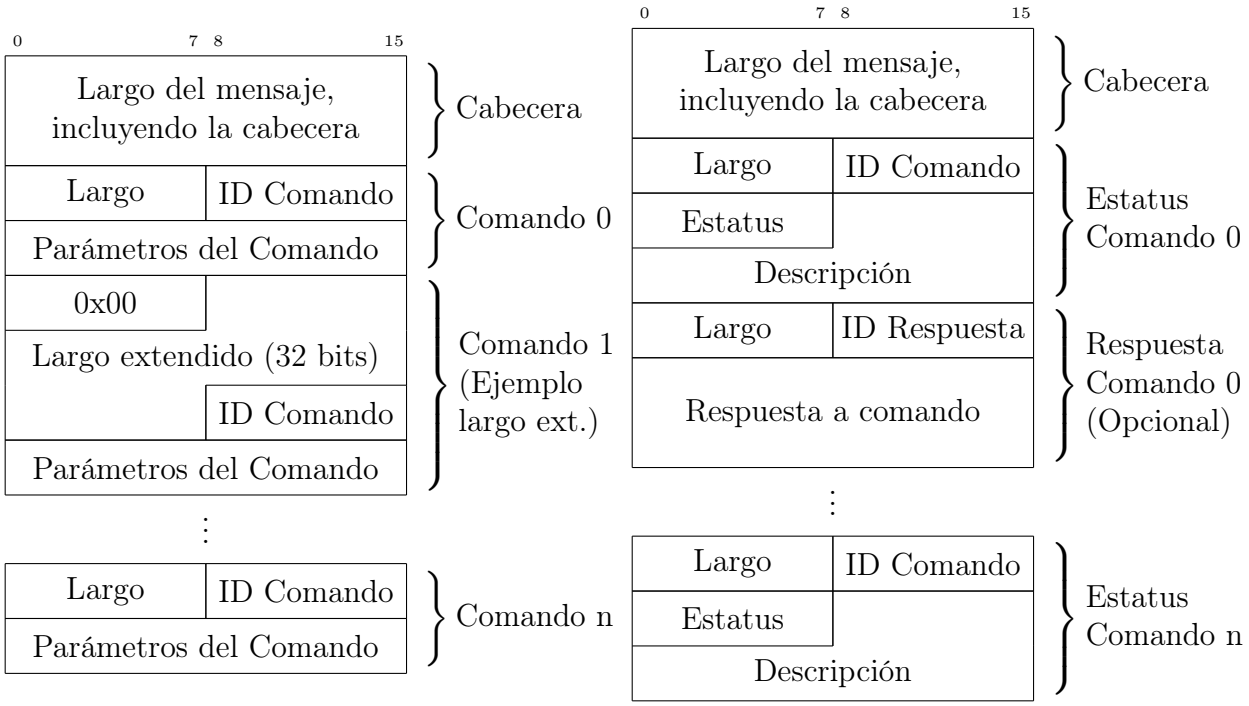
Hoy en día, dicha arquitectura se encuentra integrada en SUMO, y se utiliza en conjunto con simuladores de redes de comunicación inalámbrica como OMNeT++ y NS2 para la simulación y estudio de Sistemas de Transporte Inteligente.

A.0.1. 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 A.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`.



(a) Comandos

(b) Respuesta

Figura A.1: Formatos de mensajes TraCI

Por otro lado, la figura A.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 A.2 ilustra el flujo de mensajes para la solicitud de una variable de la simulación.

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.

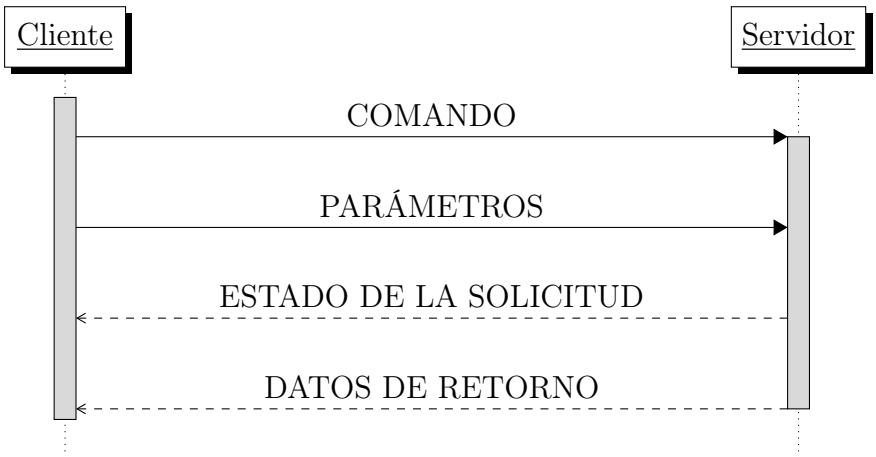


Figura A.2: Ejemplo solicitud de variable TraCI.

Este comando tiene dos modos de operación; *single step* y *target time step*. En 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 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.

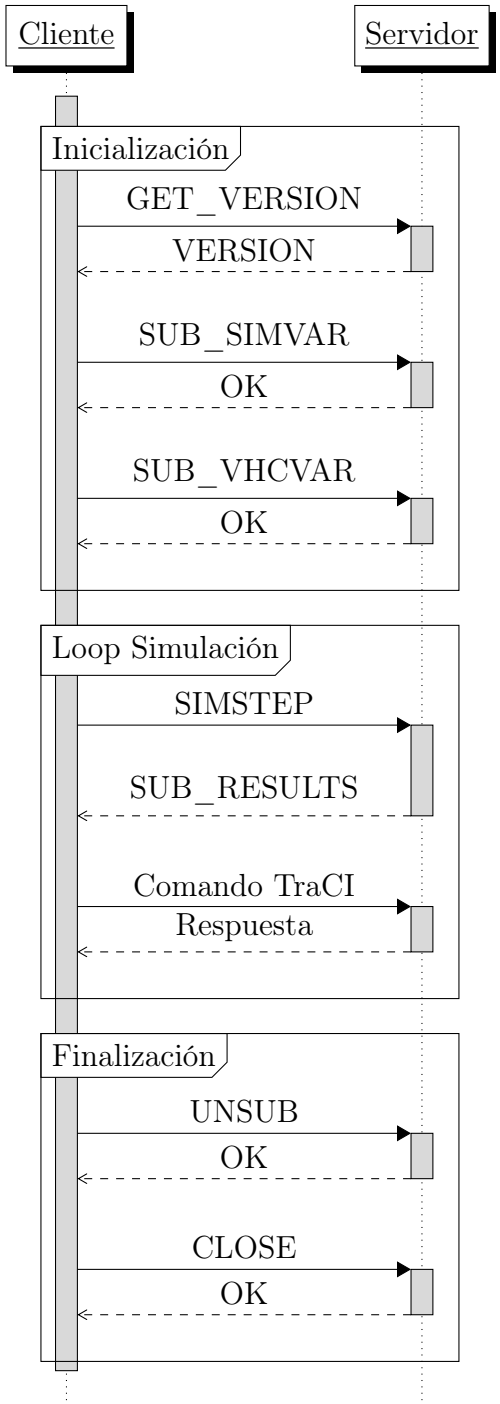


Figura A.3: Flujo de comunicación TraCI.

Apéndice B

Paramics API

La API de Paramics consiste en un conjunto de *headers* de código `C`, los cuales definen un conjunto de funciones accesibles (o incluso, que pueden ser sobrescritas) por *plugins* para el simulador. A continuación se describirán de manera resumida las distintas categorías de funciones expuestas por el *software*.

En primer lugar, los nombres de los métodos de la API siguen el siguiente patrón:

`CATEGORIA_DOMINIO_nombre_de_funcion()`

`CATEGORIA` y `DOMINIO` corresponden a identificadores de tres caracteres, los cuales indican el tipo de función (por ejemplo, de obtención de valores) y su dominio (*e.g.* vehículos o calles).

B.1. Categorías de Funciones

Se definen cuatro categorías de funciones:

- Funciones de *override*, prefijo `QPO`
- Funciones de extensión, prefijo `QPX`
- Funciones de obtención de valores, prefijo `QPG`
- Funciones de modificación de valores, prefijo `QPS`

B.1.1. Funciones `QPO`

Las funciones de *override*, con prefijo `QPO`, corresponden a funciones que controlan algún comportamiento clave del modelo interno de Paramics y que son sobrescribibles por el usua-

rio. Por ejemplo, la función `float qpo_CFM_leadSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])` se utiliza para modificar las velocidades de cada vehículo que no tiene otro vehículo delante, en cada paso de simulación; esta función es sobrescrita en el *plugin* desarrollado para retornar velocidades distintas a las que retornaría Paramics por defecto, para los casos de vehículos que han recibido comandos de modificación de velocidad desde OMNeT++.

B.1.2. Funciones `QPX`

Las funciones de prefijo `QPX`, correspondiente a funciones de extensión de funcionalidad, son funciones definibles por el usuario que extienden algún funcionamiento de Paramics. Por lo general, estos métodos están ligados a eventos disparados o periódicos en Paramics; *e.g.*, la función `void qpx_NET_postOpen()` se ejecuta una única vez inmediatamente luego de terminar de cargar la red de Paramics, y en el *plugin* se utiliza para inicializar el servidor TraCI. Por otro lado, la función `void qpx_CLK_startOfSimLoop()` se ejecuta antes de cada paso de simulación, y se utiliza en el *framework* para ejecutar un *loop* de recepción y interpretación de mensajes desde el cliente TraCI.

B.1.3. Funciones `QPG`

La obtención de valores desde la simulación se realiza a través de estas funciones con prefijo `QPG`. Ejemplos de estas son `int qpg_VHC_uniqueID(VEHICLE* V)`, utilizada para obtener el identificador único de algún vehículo, y `float qpg_CFG_simulationTime()`, la cual retorna el tiempo de simulación actual (en segundos).

B.1.4. Funciones `QPS`

Finalmente, las funciones `QPS` sobrescriben valores internos de la simulación, o modifican comportamientos puntuales. Por ejemplo, `void qps_DRW_vehicleColour(VEHICLE* vehicle, int colour)` cambia el color de un vehículo, y `void qps_VHC_changeLane (VEHICLE*, int direction)` fuerza un cambio de pista.

B.2. Dominios

El segundo trío de caracteres en el nombre de cada función de la API indica el dominio de esta, es decir, a qué categoría de objetos o funcionalidades dentro del modelo de Paramics está asociada. Dada la gran cantidad de dominios definidos en la API, no se detallarán aquí. Sin embargo, cabe notar que los principales dominios de funciones utilizados en el desarrollo del presente trabajo corresponden a los dominios `VHC`, ligado a los vehículos presentes en la

red, LNK, ligado a los arcos (calles) de la red y NET, ligado a propiedades de la red en su totalidad.

Apéndice C

Códigos

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

```
/*
   ------------------------------------------------------------------------
 * Paramics Programmer API (paramics-support@quadstone.com)
 * Quadstone Ltd.      Tel: +44 131 220 4491
 * 16 Chester Street   Fax: +44 131 220 4492
 * Edinburgh, EH3 7RA, UK WWW: http://www.paramics-online.com
 *
   ------------------------------------------------------------------------
   */

#include "programmer.h"
#include <thread>
#include "TraCIAPI/TraCIServer.h"
#include <shellapi.h>
#include "TraCIAPI/VehicleManager.h"
#include "TraCIAPI/Utils.h"

#define DEFAULT_PORT 5000
#define CMDARG_PORT "--traci_port="

std::thread* runner;
traci_api::TraCIServer* server;

/* checks a string for a matching prefix */
bool starts_with(std::string const& in_string,
             std::string const& prefix)
{
   return prefix.length() <= in_string.length() &&
      std::equal(prefix.begin(), prefix.end(), in_string.begin());
}

void runner_fn()
```

```
{
   try {
      //try to get port from command line arguments
      int argc;
      LPWSTR* argv = CommandLineToArgvW(GetCommandLineW(), &argc);
      std::string prefix(CMDARG_PORT);

      int port = DEFAULT_PORT; // if it fails, use the default port
      for (int i = 0; i < argc; i++)
      {
         // convert from widestring to normal string
         std::wstring temp(argv[i]);
         std::string str(temp.begin(), temp.end());

         // check if argument prefix matches
         if (starts_with(str, prefix))
         {
            std::string s_port = str.substr(prefix.length(),
               str.npos);
            try
            {
               port = std::stoi(s_port);
            }
            catch (...)
            {
               traci_api::infoPrint("Invalid port identifier -
                  Falling back to default port");
               port = DEFAULT_PORT;
            }
         }
      }

      server = new traci_api::TraCIServer(port);
      server->waitForConnection();
   }
   catch (std::exception& e)
   {
      traci_api::debugPrint("Uncaught while initializing server.");
      traci_api::debugPrint(e.what());
      traci_api::debugPrint("Exiting...");
      throw;
   }
}

// Called once after the network is loaded.
void qpx_NET_postOpen(void)
{
   qps_GUI_singleStep(PFALSE);
   traci_api::infoPrint("TraCI support enabled");
   runner = new std::thread(runner_fn);
```

```
}

void qpx_CLK_startOfSimLoop(void)
{
   if (runner->joinable())
      runner->join();

   server->preStep();
}

void qpx_CLK_endOfSimLoop(void)
{
   server->postStep();
}

void close()
{
   server->close();
   delete server;
   delete runner;
}

void qpx_NET_complete(void)
{
   close();
}

void qpx_NET_close()
{
   close();
}

void qpx_VHC_release(VEHICLE* vehicle)
{
   traci_api::VehicleManager::getInstance()->vehicleDepart(vehicle);
}

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

{
   traci_api::VehicleManager::getInstance()->vehicleArrive(vehicle);
}

// routing through TraCI
Bool qpo_RTM_enable(void)
{
   return PTRUE;
}

int qpo_RTM_decision(LINK *linkp, VEHICLE *Vp)
```

```
{
   return
      traci_api::VehicleManager::getInstance()->rerouteVehicle(Vp,
      linkp);
}

void qpx_VHC_timeStep(VEHICLE* vehicle)
{
   //traci_api::VehicleManager::getInstance()->routeReEval(vehicle);
}

void qpx_VHC_transfer(VEHICLE* vehicle, LINK* link1, LINK* link2)
{
   traci_api::VehicleManager::getInstance()->routeReEval(vehicle);
}

// speed control override
float qpo_CFM_followSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])
{
   float speed = 0;
   if
      (traci_api::VehicleManager::getInstance()->speedControlOverride(v,
      speed))
      return speed;
   else
      return qpg_CFM_followSpeed(link, v, ahead);
}

float qpo_CFM_leadSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])
{
   float speed = 0;
   if
      (traci_api::VehicleManager::getInstance()->speedControlOverride(v,
      speed))
      return speed;
   else
      return qpg_CFM_leadSpeed(link, v, ahead);
}
```

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

```
int traci_api::VariableSubscription::checkTime() const
{
   int current_time =
      Simulation::getInstance()->getCurrentTimeMilliseconds();
   if (beginTime > current_time) // begin time in the future
      return -1;
   else if (beginTime <= current_time && current_time <= endTime) //
      within range
      return 0;
   else // expired
      return 1;
}

uint8_t
   traci_api::VariableSubscription::handleSubscription(tcpip::Storage&
   output, bool validate, std::string& errors)
{
   int time_status = checkTime();
   if (!validate && time_status < 0) // not yet (skip this check if
      validating, duh)
      return STATUS_TIMESTEPNOTREACHED;
   else if (time_status > 0) // expired
      return STATUS_EXPIRED;

   // prepare output
   output.writeUnsignedByte(getResponseCode());
   output.writeString(objID);
   output.writeUnsignedByte(vars.size());

   bool result_errors = false;

   // get ze vahriables
   tcpip::Storage temp;
   for (uint8_t sub_var : vars)
   {
      // try getting the value for each variable,
      // recording errors in the output storage
      try {
         output.writeUnsignedByte(sub_var);
         getObjectVariable(sub_var, temp);
         output.writeUnsignedByte(traci_api::STATUS_OK);
         output.writeStorage(temp);
      }
      // ReSharper disable once CppEntityNeverUsed
      catch (NoSuchObjectError& e)
      {
         // no such object
         errors = "Object " + objID + " not found in simulation.";
```

```
            return STATUS_OBJNOTFOUND;
        }
        catch (std::runtime_error& e)
        {
            // unknown error
            result_errors = true;
            output.writeUnsignedByte(traci_api::STATUS_ERROR);
            output.writeUnsignedByte(VTYPE_STR);
            output.writeString(e.what());
            errors += std::string(e.what()) + "; ";
        }

        temp.reset();
    }

    if (validate && result_errors)
        // if validating this subscription, report the errors.
        // that way the subscription is not added to the sub
        // vector in TraCIServer
        return STATUS_ERROR;
    else
        // else just return the subscription to the client,
        // and let it decide what to do about the errors.
        return STATUS_OK;
}

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)
{
    if (sub_type != this->sub_type || obj_id != objID)
        // we're not the correct subscription,
        // return NO UPDATE
        return STATUS_NOUPD;

    if (vars.size() == 0)
        // 0 vars => cancel this subscription
        return STATUS_UNSUB;

    // backup old values
    int old_start_time = this->beginTime;
    int old_end_time = this->endTime;
    std::vector<uint8_t> old_vars = this->vars;

    // set new values and try
    this->beginTime = begin_time;
    this->endTime = end_time;
    this->vars = vars;

```

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

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

```
void traci_api::TraCIServer::addSubscription(uint8_t sub_type,
   std::string object_id, int start_time, int end_time,
   std::vector<uint8_t> variables)
{
   std::string errors;
   tcpip::Storage temp;

   // first check if this corresponds to an update for an existing
      subscription
   for (auto it = subs.begin(); it != subs.end(); ++it)
   {
      uint8_t result = (*it)->updateSubscription(sub_type,
         object_id, start_time, end_time, variables, temp, errors);

      switch (result)
      {
      case VariableSubscription::STATUS_OK:
         // update ok, return now
         debugPrint("Updated subscription");
         writeStatusResponse(sub_type, STATUS_OK, "");
         writeToOutputWithSize(temp, true);
         return;
      case VariableSubscription::STATUS_UNSUB:
         // unsubscribe command, remove the subscription
         debugPrint("Unsubscribing...");
         delete *it;
         it = subs.erase(it);
         // we don't care about the deleted iterator, since we
            return from the loop here
         writeStatusResponse(sub_type, STATUS_OK, "");
         return;
      case VariableSubscription::STATUS_ERROR:
         // error when updating
         debugPrint("Error updating subscription.");
         writeStatusResponse(sub_type, STATUS_ERROR, errors);
         break;
      case VariableSubscription::STATUS_NOUPD:
         // no update, try next subscription
         continue;
      default:
         throw std::runtime_error("Received unexpected result " +
            std::to_string(result) + " when trying to update
            subscription.");
      }
   }

```

```
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)
```

```
    {
        debugPrint("Error adding subscription.");

        writeStatusResponse(sub_type, STATUS_ERROR, errors);
        return;
    }

    writeStatusResponse(sub_type, STATUS_OK, "");
    writeToOutputWithSize(temp, true);
    subs.push_back(sub);
}
```

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

```
int traci_api::Simulation::runSimulation(uint32_t target_timems)
{
   auto current_simtime = this->getCurrentTimeSeconds();
   auto target_simtime = target_timems / 1000.0;
   int steps_performed = 0;

   traci_api::VehicleManager::getInstance()->reset();

   if (target_timems == 0)
   {
      debugPrint("Running one simulation step...");

      qps_GUI_runSimulation();
      traci_api::VehicleManager::getInstance()
         ->handleDelayedTriggers();
      steps_performed = 1;
   }
   else if (target_simtime > current_simtime)
   {
      debugPrint("Running simulation up to target time: " +
         std::to_string(target_simtime));
      debugPrint("Current time: " + std::to_string(current_simtime));

      while (target_simtime > current_simtime)
      {
         qps_GUI_runSimulation();
         steps_performed++;
         traci_api::VehicleManager::getInstance()
            ->handleDelayedTriggers();

         current_simtime = this->getCurrentTimeSeconds();

         debugPrint("Current time: " +
            std::to_string(current_simtime));
      }
   }
   else
   {
      debugPrint("Invalid target simulation time: " +
         std::to_string(target_timems));
      debugPrint("Current simulation time: " +
         std::to_string(current_simtime));
      debugPrint("Doing nothing");
   }

   stepcnt += steps_performed;
   return steps_performed;
}
```

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

```
void traci_api::Simulation::getRealNetworkBounds(double& llx,
   double& lly, double& urx, double& ury)
{
   /*
    * Paramics qpg_POS_network() function, which should return the
       network bounds, does not make sense.
    * It returns coordinates which leave basically the whole network
       outside of its own bounds.
    *
    * Thus, we'll have to "bruteforce" the positional data for the
       network bounds.
    */

   // get all relevant elements in the network, and all their
      coordinates

   std::vector<float> x;
   std::vector<float> y;

   int node_count = qpg_NET_nodes();
   int link_count = qpg_NET_links();
   int zone_count = qpg_NET_zones();

   float tempX, tempY, tempZ;

   for (int i = 1; i <= node_count; i++)
   {
      NODE* node = qpg_NET_nodeByIndex(i);
      qpg_POS_node(node, &tempX, &tempY, &tempZ);

      x.push_back(tempX);
      y.push_back(tempY);
   }

   for (int i = 1; i <= zone_count; i++)
   {
      ZONE* zone = qpg_NET_zone(i);
      int vertices = qpg_ZNE_vertices(zone);
      for (int j = 1; j <= vertices; j++)
      {
         qpg_POS_zoneVertex(zone, j, &tempX, &tempY, &tempZ);

         x.push_back(tempX);
         y.push_back(tempY);
      }
   }

   for (int i = 1; i <= link_count; i++)
```

```
44      {
45          // links are always connected to zones or nodes, so we only
                need
46          // to get position data from those that are curved
47
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
                start, end and middle point (for all lanes)
53          float len = qpg_LNK_length(lnk);
54          int lanes = qpg_LNK_lanes(lnk);
55
56          float g, b;
57
58          for (int j = 1; j <= lanes; j++)
59          {
60              // start points
61              qpg_POS_link(lnk, j, 0, &tempX, &tempY, &tempZ, &b, &g);
62
63              x.push_back(tempX);
64              y.push_back(tempY);
65
66              // middle points
67              qpg_POS_link(lnk, j, len / 2.0, &tempX, &tempY, &tempZ, &b,
                    &g);
68
69              x.push_back(tempX);
70              y.push_back(tempY);
71
72              // end points
73              qpg_POS_link(lnk, j, len, &tempX, &tempY, &tempZ, &b, &g);
74
75              x.push_back(tempX);
76              y.push_back(tempY);
77          }
78      }
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  }
```