



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 J.
NANCY HITSCHFELD K.

Este trabajo ha sido parcialmente financiado por NIC Chile Research Labs y el Área de Transportes del Departamento de Ingeniería Civil de la Universidad de Chile

SANTIAGO DE CHILE
JUNIO 2017

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

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

El presente trabajo de memoria presenta el diseño, implementación y validación de un *framework* de integración de un simulador de transporte, *Quadstone Paramics* con un simulador de redes comunicaciones inalámbricas, *OMNeT++*, para la simulación y estudio de Sistemas Inteligentes de Transporte.

Los Sistemas Inteligentes de Transporte surgen como una respuesta a la necesidad de optimización, modernización y mejoramiento de los actuales sistemas de transporte. Los Sistemas de Transporte Inteligente pretenden proveer servicios innovadores que otorguen información a los usuarios y les permitan utilizar el sistema de transporte de manera más segura, coordinada e inteligente. Resulta fundamental la recopilación y transmisión de información en estos sistemas, lo cual se realiza mediante implementación de redes comunicación inalámbrica, tanto entre vehículos como entre vehículos e infraestructura. Es necesario entonces el desarrollo de entornos de software de modelamiento y simulación de estos sistemas, para su estudio previo a su implementación en el mundo real.

Este trabajo de memoria presenta un *framework* que posibilita la simulación y análisis de los Sistemas Inteligentes de Transporte. PVEINS, como se denomina el software desarrollado, permite el estudio de la integración bidireccional de un sistema de transporte con un sistema de comunicaciones inalámbricas. En ese sentido, el *framework* permite determinar tanto el impacto de la comunicación entre vehículos sobre el modelo de transporte, como el impacto del movimiento de los vehículos sobre el medio de comunicación entre estos.

El software consiste principalmente en un *plugin* para Paramics, escrito en C++, el cual permite la integración transparente de este simulador con un *framework* de simulación de Sistemas Inteligentes de Transporte ya existente y de gran renombre en la academia. Se desarrolló utilizando una estrategia iterativa, siguiendo buenas prácticas de ingeniería de software.

Finalmente, el presente trabajo de memoria presenta además un análisis de eficiencia del software desarrollado, y un estudio para verificar su validez para la simulación de sistemas de transporte de alta complejidad. Los resultados son altamente positivos, y demuestran que PVEINS tiene el potencial para posicionarse como una opción competitiva para la simulación de Sistemas Inteligentes de Transporte en la academia.

*Den här är för Aros och Skatt,
som alltid stöttat mig
ovillkorligt som bara hundar kan.*

Agradecimientos

Son muchos, tal vez demasiados, a quienes debo agradecer por su apoyo, amistad y/o cariño en el camino al obtener mi título.

En primer lugar, quiero agradecer a mi familia, quienes siempre han estado presentes y me inculcaron desde el primer momento el valor del estudio y el esfuerzo. A mi padre, Gabriel Olguín, por ser mi ejemplo a seguir en el ámbito académico - me lleva un doctorado y un postdoc de ventaja, pero eventualmente lo alcanzaré. A mi madre, Valeria Muñoz, por ser una constante de estabilidad emocional en mi vida y que también desde siempre me ha impulsado a alcanzar las estrellas. Me ha tenido que soportar 24 años, pero finalmente llegó este mi primer paso en mi independización. A mi hermana, Paola Olguín, por soportar mis estupideces todos los días – me llena de orgullo saber que ella sigue mis pasos y que se dedicará a la misma carrera que yo. Y a mis abuelos, quienes siempre me han dado apoyo incondicional. En especial, a mi Tata Caupolicán Muñoz, con quien siempre compartí profundas discusiones sobre ciencia y quien me inculcó el amor por la lectura. Yo sé que estaría hinchado de orgullo al verme recibirme de ingeniero civil.

También quiero darle mis profundas gracias a los profesores que me han acompañado en este largo proceso. A la profesora Sandra Céspedes, por ser probablemente la mejor profesora guía que podría haber pedido – le pido disculpas por todos esos correos y mensajes en fines de semana, que ella respondió de manera muy cordial. A Javier Bustos, por haberme dado la oportunidad en su momento de realizar mi práctica en NICLabs a pesar de que no tenía las mejores calificaciones, y por haberme aceptado más tarde como miembro permanente del equipo. Espero haberle demostrado que tomó la decisión correcta. A Cristián Cortés, por su invaluable apoyo durante mi trabajo de memoria, y tremendo entusiasmo con los resultados que le he presentado. A la profesora Nancy Hitschfeld, quien fue la primera en confiar en mis habilidades para enseñar, y me aceptó como profesor auxiliar en su curso de computación gráfica. Me abrió el mundo de la docencia, el cual me fascinó y por lo cual le estoy eternamente agradecido. Al profesor Jérémie Barbay, por haber sido tanto un tutor como un amigo en estos años – gracias por los infinitos consejos, los buenos momentos y la oportunidad de desarrollar algo tan novedoso como lo fue Moulinette.

A mis amigos, quienes me han soportado durante todos estos años, en las buenas y en las malas, gracias. En especial quiero nombrar a JP, Juanjo, Negro y Varas – gracias amigos por haberme apoyado cuando necesitaba apoyo, haberme corregido cuando necesitaba ser corregido y por haberme insultado cuando necesitaba ser insultado. Un abrazo también para los chiquillos del “*lolcito*”; Diego, George y el resto del equipo – no sé que hubiese hecho estos

años sin esas partidas nocturnas que me alejaban del estrés diario de los estudios. Quiero además agradecer a mi *polola*, Maria Collin, por su paciencia y su enorme corazón – su cariño y apoyo me han impulsado a trabajar más duro que nunca.

Last, but not least, como dicen en inglés, quiero agradecer a todo el resto del mundo que no cupo en los párrafos anteriores – no me he olvidado de ustedes. Gracias por el apoyo, las palabras de ánimo, la buena onda. A todos, de todo mi corazón: a mis amigos del colegio, a mis amigos de la universidad, a los profesores del gimnasio, al tío del kiosko a quien semana tras semana le compré bebidas energéticas para seguir adelante. Gracias!

Manuel Osvaldo J. Olguín Muñoz

Junio 2017

Tabla de Contenido

1. Introducción	1
1.1. Motivación	1
1.2. Resumen	2
1.3. Organización del documento	4
2. Marco Teórico y Estado del Arte	5
2.1. Marco teórico	5
2.1.1. Sistemas Inteligentes de Transporte	5
2.1.2. Tecnologías de comunicaciones para ITS	6
2.1.3. Simulación de redes de comunicaciones	7
2.1.4. Simulación de tráfico	8
2.1.5. Simulación bidireccional	8
2.2. Estado del arte	9
2.2.1. Simuladores de tráfico	9
2.2.2. Simuladores de redes inalámbricas	11
2.2.3. Entornos de simulación bidireccional	13
3. Especificación del Problema y Objetivos	16
3.1. Especificación del problema	16
3.2. Elección de solución a implementar	17
3.3. Objetivos	17
3.3.1. Objetivo general	17
3.3.2. Objetivos particulares	18
3.4. Metodología	19
4. Diseño e Implementación	20
4.1. Diseño y Metodología de Desarrollo	20
4.1.1. Diseño arquitectural	20
4.1.2. Metodología de desarrollo	26
4.2. Funcionalidad implementada	27
4.2.1. Comandos Implementados	27
4.3. Implementación	29
4.3.1. plugin.c	31
4.3.2. TraCIServer	32
4.3.3. Simulation	42
4.3.4. VehicleManager	44

4.3.5. Otros módulos	51
4.4. Pruebas preliminares	54
4.4.1. Ejemplo de script de prueba: cambio de ruta	55
5. Validación	57
5.1. Escenario y Análisis	57
5.1.1. Escenario modelado	57
5.2. Eficiencia Computacional	60
5.2.1. Mediciones Realizadas	60
5.2.2. Resultados	61
5.2.3. Conclusiones	64
5.3. Modelo Vehicular	65
5.3.1. Mediciones Realizadas	65
5.3.2. Resultados	65
5.3.3. Conclusiones	68
6. Conclusiones	69
6.1. Conclusiones generales	69
6.2. Cumplimiento de objetivos	70
6.3. Trabajo futuro	71
Bibliografía	75
A. TraCI	79
A.0.1. Diseño	79
B. Paramics API	83
B.1. Categorías de Funciones	83
B.1.1. Funciones QPO	83
B.1.2. Funciones QPX	84
B.1.3. Funciones QPG	84
B.1.4. Funciones QPS	84
B.2. Dominios	85
C. Códigos	86

Índice de Tablas

2.1. Tabla comparativa simuladores de tráfico.	9
5.1. Especificaciones técnicas del entorno de simulación.	59
5.2. Factor de demanda vs. cantidad promedio de vehículos	60
5.3. Cantidad de vehículos vs. tiempo real de simulación	61
5.4. Comparación simulaciones de 15 y 120 minutos de duración	66

Índice de Ilustraciones

1.1. Integración bidireccional de Paramics con OMNeT++.	3
2.1. Entorno de simulación gráfica de OMNeT++.	12
2.2. Evolución de simulaciones integradas.	14
4.1. Visión macroscópica del framework	21
4.2. Arquitectura preliminar.	23
4.3. Arquitectura final del <i>framework</i>	25
4.4. Estructura de archivos del código fuente del framework.	29
4.5. Gráfico de dependencia entre los componentes del <i>framework</i> .	30
4.6. Diagrama de herencia, VariableSubscription	38
4.7. Definición del preprocesador vs. variable constante estática	52
4.8. Red de transporte utilizada para las pruebas preliminares.	54
4.9. <i>Test</i> de cambio de ruta	55
5.1. Escenario modelado, Paramics vs. “vida real”.	58
5.2. Cantidad de vehículos vs. tiempo real de simulación	62
5.3. Evolución temporal de la cantidad de vehículos en la simulación.	63
5.4. Carga sobre el sistema durante una simulación	63
5.5. I/O en disco durante simulación	64
5.6. Comparación cantidad de vehículos que alcanzaron su destino	66
5.7. Distancia vs. tiempo total	67
5.8. Distancia vs. CO ² total	68
A.1. Formatos de mensajes TraCI	80
A.2. Ejemplo solicitud de variable TraCI	81
A.3. Flujo de comunicación TraCI	82

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 surgen como una respuesta a la necesidad de optimización, modernización y mejoramiento de los sistemas de transporte ya existentes. La Unión Europea define a los ITS como aplicaciones que pretenden proveer servicios innovadores relacionados con distintos modos de transporte y de administración de tráfico, que además otorgan información a los usuarios y les permiten utilizar el sistema de transporte de manera más segura, coordinada e inteligente [1]. Esta amplia definición abarca una gran cantidad de aplicaciones: desde sistemas de alerta temprana a sistemas de entretenimiento en ruta, pasando incluso por aplicaciones tan avanzadas como sistemas de coordinación y control de vehículos autónomos.

El factor común entre todas estas aplicaciones es la necesidad de extraer información en tiempo 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 estas tecnologías en contextos de comunicaciones inalámbricas en redes vehiculares. Por otro lado, existe también la necesidad de modelar el impacto de la comunicación inalámbrica en un sistema de transporte, y cómo esta puede contribuir a optimizar la operación del mismo [5], [6]. 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. Resumen

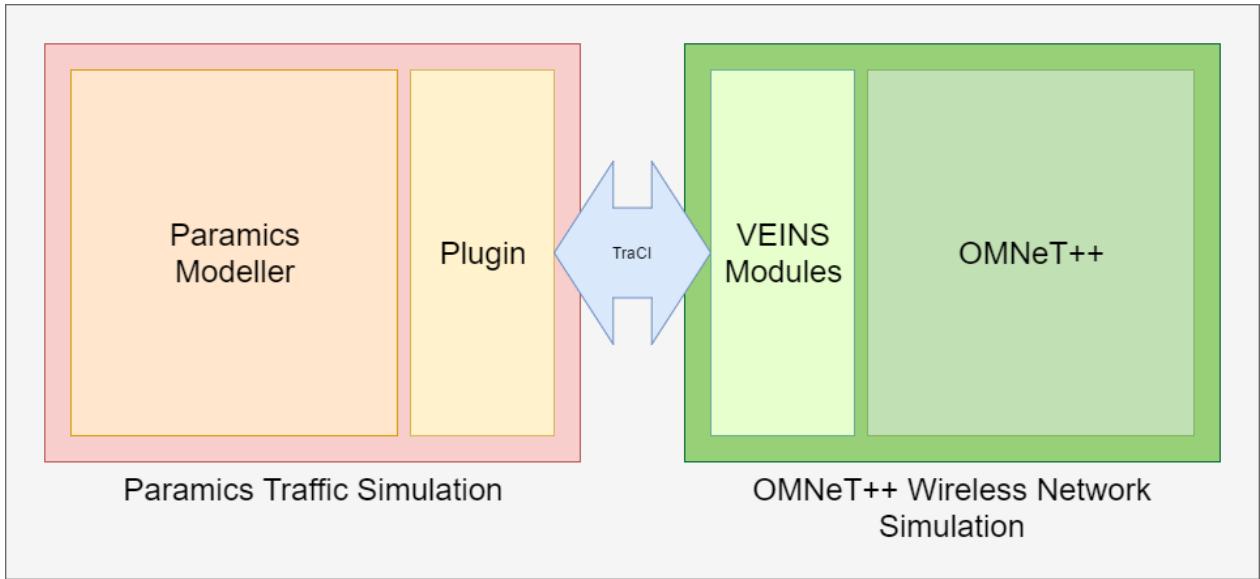
A raíz de la problemática expuesta en la sección anterior, el presente trabajo de memoria presenta un *framework* de integración bidireccional entre un simulador de redes de comunicaciones inalámbricas – OMNeT++ – con un simulador de redes de transporte – Quadstone Paramics – como una herramienta de apoyo para el estudio de las problemáticas previamente señaladas y la evaluación de nuevos modelos y soluciones para transporte inteligente.

La elección de éstos simuladores en particular tiene razones bien fundadas. En particular, OMNeT++ fue escogido principalmente por particularidades que se verán más adelante, relacionadas con la elección de solución al problema planteado. Por otro lado, Paramics es el simulador de transporte de preferencia del Departamento de Transportes de la Universidad de Chile, parte interesada en el presente proyecto de memoria dadas las aplicaciones que puede tener para la investigación que se realiza en el Departamento. La extensión de Paramics para su funcionamiento en un sistema de simulación integrado bidireccional permitiría a su vez la extensión de modelos ya existentes que maneja dicha área de investigación, para sus estudios en contextos de Sistemas de Transporte Inteligente.

El *framework* desarrollado para la memoria se implementó en base a una adaptación de un trabajo previamente realizado por Sommer *et al.* [7], [8] para la integración de OMNeT++ con SUMO, otro simulador de redes de transporte. Este trabajo previo, un *framework* denominado VEINS – por sus siglas en inglés, *Vehicles In Network Simulation* – utiliza una novedosa arquitectura en la que ambos simuladores se comunican a través de un *socket*, en una configuración cliente-servidor. La implementación presentada en este trabajo de memoria, denominada PVEINS¹, pretende reemplazar, de manera absolutamente transparente para OMNeT++, a SUMO en dicha arquitectura, para así poder aprovechar todo el trabajo en modelos de comunicación ya existentes para esta configuración.

El trabajo consistió principalmente en el desarrollo de un *plugin* de extensión de Paramics, el cual se encarga de recibir e interpretar comandos desde el simulador de redes, realizando las operaciones necesarias en el modelo de Paramics. Este se desarrolló en C++, en su estándar

¹PVEINS – Paramics VEINS



PVEINS Framework

Figura 1.1: Integración bidireccional de Paramics con OMNeT++.

2011, utilizando la API del software (ver apéndice B).

Este *plugin* permite la integración de Paramics con el entorno de simulación de OMNeT++, posibilitando la simulación de Sistemas Inteligentes de Transporte – el *framework* permite al simulador de redes construir una red inalámbrica equivalente a la red de transporte simulada, donde cada vehículo en Paramics se asocia a un nodo dotado de capacidades de comunicación inalámbrica en OMNeT++. El estado de ambas simulaciones evoluciona de manera sincronizada, y los nodos de OMNeT++ están dotados de lógica y pueden modificar el comportamiento de sus respectivos vehículos. De esta manera, es posible analizar tanto el impacto de la movilidad de los vehículos sobre el canal de comunicaciones, como el impacto de la transmisión de información en el modelo de transporte.

Este nuevo *framework* PVEINS se validó luego mediante una serie de experimentos destinados a medir su rendimiento y su aptitud para uso en investigación y modelación de Sistemas Inteligentes de Transporte de una complejidad considerable. El Área de Transportes del Departamento de Ingeniería Civil de la Universidad de Chile proporcionó un escenario realista que modela un sector de la ciudad de Santiago en hora punta, el cual se utilizó para la validación del *software*. Este escenario es de alta complejidad, elaborado por Zúñiga [9] en 2010 para su memoria de título, y presenta un promedio de aprox. 1400 vehículos presentes en la red en cada instante de la simulación.

Los resultados de los análisis realizados sobre el *framework* en el contexto de un escenario tan complejo como el proporcionado fueron altamente positivos. El software es eficiente, y permite modelar de manera precisa y realista el comportamiento de los Sistemas Inteligentes de Transporte.

Sin embargo, este trabajo de todas maneras identifica factores y detalles perfeccionables en la implementación del sistema, los cuales se presentan al final de este documento como

sugerencias para trabajo futuro.

Finalmente, el producto de este trabajo de memoria se encuentra disponible en su totalidad en el repositorio *git* personal del autor [10], bajo una licencia *BSD 3-Clause* [11]. Ahí se podrá encontrar tanto el código fuente del *framework* como del escenario de validación avanzado y los resultados de las pruebas realizadas.

1.3. Organización del documento

El presente documento de memoria se estructura como sigue; el presente capítulo expone la motivación tras el proyecto desarrollado, y se presenta un resumen del trabajo realizado.

El capítulo 2 expone el marco teórico que sustenta el trabajo de memoria, además de presentar una extensa revisión del estado del arte en los ámbitos de simulación de sistemas de transporte, de redes de comunicaciones y de simulación bidireccional entre las dos categorías anteriores.

El capítulo 3 define claramente el problema a solucionar, las opciones consideradas y los objetivos principales a lograr.

En el capítulo 4 se detallan primero las decisiones de diseño macroscópico que se tomaron al elaborar la arquitectura del proyecto, y las evoluciones por las cuales este diseño pasó. Se presenta además la metodología de trabajo utilizada para el desarrollo del *software* y las funcionalidades implementadas en términos generales. Luego, se describe en detalle la implementación en código del *framework*, además de entregarse una breve descripción de las pruebas que se realizaron durante el desarrollo.

El capítulo 5, expone el escenario avanzado de prueba que se utilizó para evaluar el rendimiento y la efectividad del proyecto. Se presentan además los resultados obtenidos de las pruebas realizadas, y se realiza un análisis de éstos.

Finalmente, el capítulo 6 concluye la presente memoria, realizando un análisis general de los resultados obtenidos, verificando que se cumplieron los objetivos establecidos en la sección 3.3 y presentando trabajo futuro a realizar.

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 [12] 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 [13]; 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 [14].

V2I

Vehicle-to-Infrastructure, 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, junto con la familia de estándares IEEE 1609.X, también conocida 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 autenticarse, 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 [15].

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 [16]:

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* aquél 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 [16]-[18].

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.

Simulador	Ocurrencias en Literatura	Tipo de Licencia
VISSIM	15	Comercial
PARAMICS	12	Comercial
CORSIM	10	Libre
AIMSUN	9	Comercial
SUMO	5	Libre

Tabla 2.1: Los cinco simuladores más prominentes en la literatura (tabla adaptada de [19]).

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. Ratnout y Rahman listan 14 de estos en su análisis comparativo del año 2009 [17], 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 [18].

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 [19].

VISSIM

VISSIM es un entorno de simulación discreto y microscópico, desarrollado propietariamente por el Grupo PVT en Alemania [20]. 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 [21]. 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 [22].

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 [23]), y a su necesidad de meticulosa calibración para obtener resultados realistas [17], [23].

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 [24]. 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 [23].

SUMO

SUMO, *Simulation of Urban MObility* [25], es un simulador de sistemas de transporte desarrollado por el Instituto de Sistemas de Transporte Alemán [26]. 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 [27], alcanzando el primer lugar en cantidad de publicaciones relacionadas con comunicaciones vehiculares [28]. 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 Bowes [29] 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 [30], trabajo basado parcialmente en el estudio realizado en 2009 por Weingartner, vom Lehn y Wehrle sobre la eficiencia de estos simuladores [31]. 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 [32].

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 estos 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 [33]. 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 [34]. 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.

OMNeT++

OMNeT++, *Objective Modular Network Testbed in C++*, es un entorno modular y basado en componentes para la simulación de sistemas de eventos discretos [35]. 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

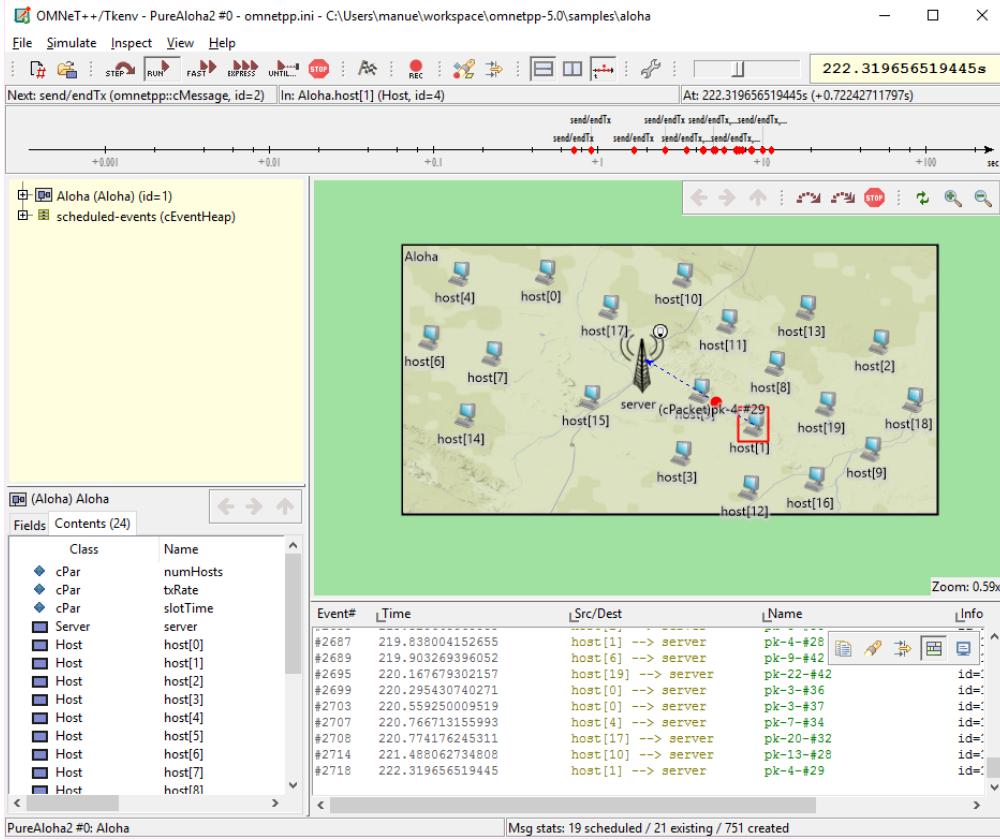


Figura 2.1: Entorno de simulación gráfica de OMNeT++.

(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 [31] 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.* [6], 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 maneras: *offline*, es decir, aisladamente en el simulador de transporte, para luego ser exportadas en un formato que el simulador de red sea capaz de interpretar, y *decoupled online*, de manera que el simulador de transporte genere las trazas en tiempo real y el simulador de red simplemente las “consume”. Sin embargo, si bien este método permite analizar el efecto del modelo de movimiento de un sistema de transporte en las comunicaciones inalámbricas, es incapaz de reflejar el impacto de la propagación de información del estado del tráfico en el modelo mismo. Es decir, esta metodología no es útil para la simulación de, por ejemplo, sistemas de advertencia de accidentes o de asistencia al conductor, puesto que las trazas de movimiento están predefinidas o se generan sin considerar los resultados de esta comunicación. Este tipo de simulación, si bien es útil para ciertos análisis específicos, no constituye una simulación bidireccional y no abarca la totalidad del problema.

Los trabajos realizados por investigadores de la Universidad Jiao Tong de Shanghai en [36], [37] 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 [38]. En este trabajo, los investigadores utilizaron SUMO para la generación de trazas vehiculares realistas, las cuales luego fueron importadas en OMNeT++ para el estudio del impacto de la movilidad vehicular en comunicaciones celulares.

Entornos integrados

La necesidad de un entorno integrado para la simulación de Sistemas Inteligentes de Transportes es un tema que ha estado presente en la comunidad académica hace casi más de una

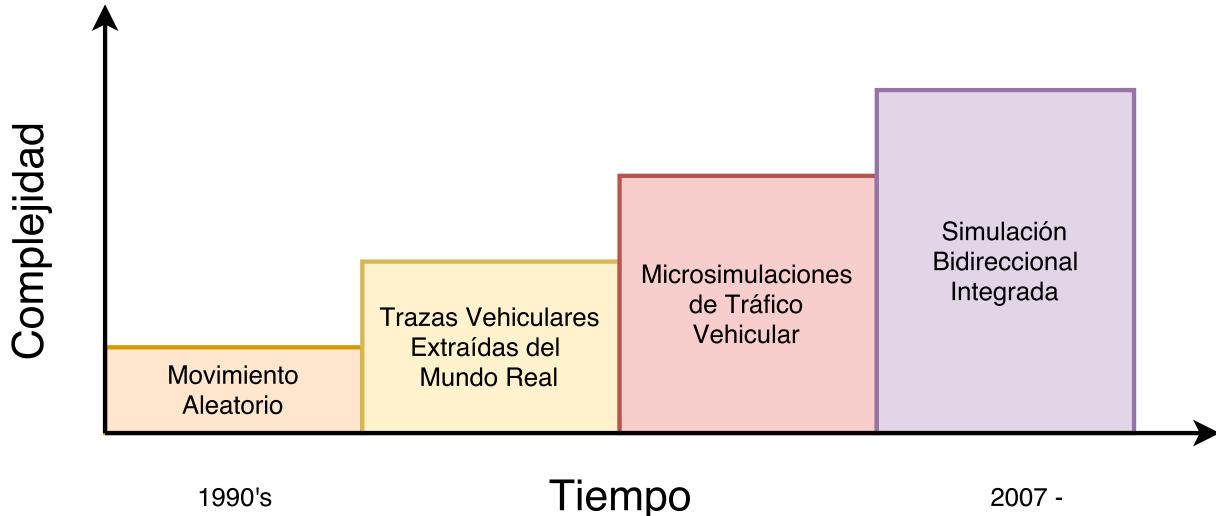


Figura 2.2: Evolución de simulaciones integradas para ITS (gráfico adaptado de [8]).

década ya. En particular, Sommer *et al.* argumentaron fuertemente a favor de la idea en [6] y [8]; 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 [39], [40], 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 [40]), 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 [41], 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 [42] y de la Universidad de Düsseldorf [43]. 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 máster [16] 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 [7]: VEINS, un *framework* de integración entre OMNeT++ y SUMO. Ambos simuladores se escogieron específicamente por su adopción en el mundo académico, y por sus naturalezas abiertas y fáciles de adaptar y modificar.

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

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

¹<http://sumo.dlr.de/wiki/FAQ>

Capítulo 3

Especificación del Problema y Objetivos

3.1. Especificación del problema

El problema abordado en este trabajo de memoria puede definirse como:

“La inexistencia de un *framework* que permita la integración bidireccional del simulador de transporte Quadstone Paramics con un simulador de redes de comunicación inalámbrica, para el modelamiento y análisis de Sistemas Inteligentes de Transporte.”

El problema se enfoca en Quadstone Paramics dada su importancia para el Área de Transporte del Departamento de Ingeniería Civil de la Universidad de Chile, quienes cuentan con una gran cantidad de modelos y simulaciones de transporte de gran complejidad ya implementados en Paramics. Para ellos, es de gran interés contar con un sistema que permita el estudio de nuevas tecnologías y tendencias en el ámbito de transporte, y una adaptación de Paramics para funcionar en un entorno de simulación de comunicaciones les permitiría ampliar su espectro de investigación a incluir, por ejemplo, sistemas de alarma temprana para conductores, o directamente sistemas de vehículos autónomos.

Esta integración, o *framework*, debe entonces ser altamente eficiente para operar sobre sistemas de transporte con miles de vehículos. Un escenario puede llegar a necesitar horas de tiempo simulado para poder analizar sus efectos y consecuencias de manera íntegra, y es esencial que esto sea factible de realizar en el producto final.

Finalmente, debe además ser flexible, y poder adaptarse sin mayores dificultades a diversos escenarios de simulación – no es lo mismo simular un sistema comunicación crítica para vehículos autónomos que un sistema de coordinación de vehículos de transporte público. Esto quiere decir que idealmente debe existir una separación entre el *framework* y los escenarios a simular, posiblemente a través de una API.

3.2. Elección de solución a implementar

Luego de realizar el exhaustivo estudio del estado del arte presentado en la sección 2.2 y de definir claramente la problemática a abordar, se tomó la decisión de desarrollar el *framework* propuesto para el presente trabajo de memoria basándose en la implementación de VEINS [7], [8]. Esta solución implica reemplazar a SUMO por Paramics, de manera totalmente transparente para OMNeT++ y VEINS, de tal manera de poder reutilizar la amplia oferta de módulos, modelos y esquemas de comunicación inalámbrica ya existentes para OMNeT++ sin necesidad de adaptarlos específicamente para el nuevo simulador de transporte.

Esto se puede lograr gracias al diseño modular y extensible de VEINS. Dado que ambos simuladores se comunican a través de un protocolo bien definido, el cual abstrae el funcionamiento de ambos extremos de la comunicación, ninguno de los dos simuladores necesita saber detalles de la implementación del otro. TraCI, el protocolo de comunicación, fue justamente diseñado con este tipo de escenarios en mente ([44]).

El alcance de la solución propuesta abarca entonces la implementación de una capa de interfaz para Paramics, que le permite interpretar y funcionar con TraCI – para esto se utilizó el API de extensión del *software*, detallado en el apéndice B.

Se tomó la decisión de optar por esta solución ya que ninguna de las demás opciones de simulación bidireccional presentaba la madurez y flexibilidad necesaria para el tipo de escenarios para los cuales se pretende utilizar el producto final. Como se comentó en la sección 2.2.3, si bien existen una serie de alternativas de entornos integrados para la simulación bidireccional, ninguna es tan avanzada y potente como VEINS.

Cabe notar además que los modelos de OMNeT++ y VEINS se implementan mediante módulos autocontenidos, los cuales gobiernan el comportamiento de la simulación. Estos módulos se comunican con el *framework* mediante una serie de APIs, y no existe la necesidad de modificar el código de la integración para cada escenario distinto.

Finalmente, una implementación utilizando VEINS necesariamente cumple con los requisitos de diseño descritos en la sección 3.3.2, ya que éstas son funcionalidades fundamentales del *framework* y del protocolo TraCI.

3.3. Objetivos

3.3.1. Objetivo general: Integración de un simulador de redes de comunicaciones y un simulador de tráfico.

El principal objetivo de este trabajo de memoria fue el desarrollo de un *framework* de integración entre un simulador de redes, OMNeT++ y un microsimulador de tráfico, Quadstone Paramics, de tal manera que exista comunicación bidireccional entre ambos. Dicho framework debía permitir la comunicación e interacción entre los simuladores. Esto quiere decir que el

simulador de redes de comunicaciones debía recibir información periódica desde el simulador de tráfico en base a la cual construir su topología de red interna. A su vez, el simulador de tráfico debía recibir instrucciones del modelo de comunicaciones, y poder modificar su modelo con base en estas. Además, la implementación de este software debía considerar conceptos y buenas prácticas de ingeniería de software, poniendo especial énfasis en la eficiencia necesaria para simular grandes redes de comunicación vehicular.

Cabe destacar que el simulador de transporte en cuestión, Paramics, fue escogido ya que es el simulador de preferencia del Área de Transporte del Departamento de Ingeniería Civil de la Universidad de Chile, quienes tienen gran interés en el presente proyecto de memoria.

3.3.2. Objetivos particulares

El objetivo general discutido anteriormente se desglosó en los siguientes objetivos particulares que debían haberse cumplido al final del desarrollo del trabajo de memoria:

1. Establecer el estado del arte en cuanto a simulación bidireccional en comunicaciones inalámbricas y sistemas de transporte, tomando en cuenta herramientas tanto de código abierto como cerrado (ver sección 2.2).
2. Escoger una solución viable a el problema particular presentado, basándose en el estado del arte previamente establecido (ver sección 3.2).
3. Diseñar el mecanismo que permita la comunicación entre ambos simuladores.
El diseño deberá considerar las siguientes funcionalidades:
 - i. Construcción de la topología del modelo de comunicaciones a partir de la topología del modelo de tráfico.
 - ii. Actualización dinámica de los nodos en OMNeT++, siguiendo los movimientos de los elementos de Paramics.
 - iii. Modificación del comportamiento de los nodos del modelo de transporte a partir de eventos en OMNeT++.
4. Implementar dicho mecanismo, siguiendo patrones y buenas prácticas de ingeniería de software.
5. Probar y validar el funcionamiento de la integración de los simuladores mediante la simulación de un modelo de transporte simple pero dinámico.
6. Implementar un modelo avanzado de transporte y definir métricas de desempeño e impacto de la red de comunicaciones en la operación del modelo.

Para el punto 6, se consideraron modelos desarrollados por el Área de Transportes del Departamento de Ingeniería Civil de la Universidad de Chile, de los cuales se escogió uno específico, detallado en el capítulo 5.

3.4. Metodología

La metodología seguida en el desarrollo del trabajo de memoria se detalla en los siguientes puntos.

1. **Elección de la solución a implementar:** En primera instancia, se debió escoger la solución a elaborar, basándose en una comparación exhaustiva de los puntos a favor y en contra de cada una.
2. **Diseño a escala macro de la solución:** Independiente de la solución escogida en el punto 1, se debió hacer un diseño macroscópico de la implementación a seguir, con el fin de establecer parámetros y guías a seguir durante el proceso de desarrollo.
3. **Desarrollo iterativo:** Se buscó seguir una metodología ágil en el desarrollo del software, acumulativamente agregando funcionalidades. Por ejemplo, para la primera iteración se buscó contar con una implementación básica de la comunicación entre los simuladores, la que simplemente permitiese la obtención de las posiciones iniciales de los nodos.
4. **Validación de la integración:** Se validó el *framework* primero utilizando un modelo básico y simple, y luego con un escenario más complejo, dinámico y realista.

Capítulo 4

Diseño e Implementación

Este capítulo detalla los detalles de diseño e implementación del proyecto de memoria. Se estructura como se explica a continuación.

La sección 4.1 detalla la evolución del diseño arquitectural del software, y la metodología de desarrollo que se utilizó.

La sección 4.2 expone brevemente la funcionalidad TraCI implementada en el *framework*.

La sección 4.3 describe en detalle la implementación de cada uno de los módulos que componen el *plugin* de Paramics. Se presta especial atención al detalle de las decisiones de diseño que se tomaron durante el desarrollo.

Finalmente, la sección 4.4 expone brevemente la validación preliminar que se le realizó al *framework* durante su desarrollo.

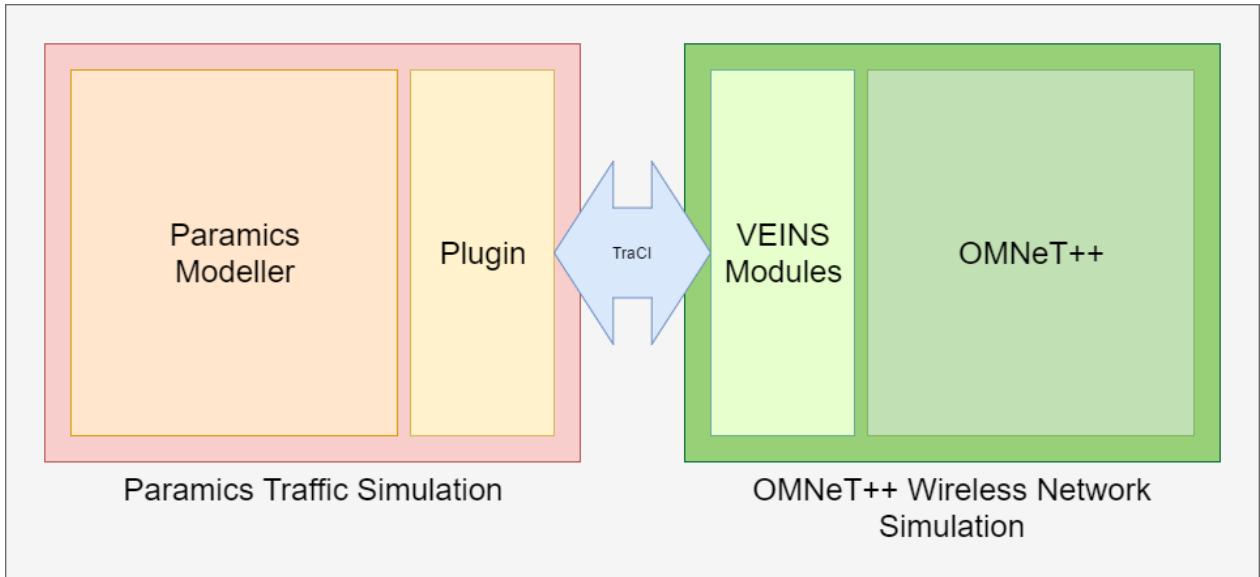
4.1. Diseño y Metodología de Desarrollo

4.1.1. Diseño arquitectural

El software desarrollado consiste en un *plugin* que extiende la funcionalidad de Paramics, agregándole la capacidad de comportarse como un servidor TraCI, y que permite a Paramics integrarse de manera transparente con el *framework* VEINS [7]. Este *framework* modificado, en el cual se reemplazó a SUMO por Paramics, se denominó **PVEINS**, por “Paramics VEINS”.

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 (figura 4.1).

A nivel más microscópico, la arquitectura del *framework* se desarrolló en dos versiones distintas, la primera de éstas siendo descartada al realizar las pruebas de validación del pro-



PVEINS Framework

Figura 4.1: Visión macroscópica del framework; el plugin desarrollado actúa como una interfaz entre TraCI y Paramics.

yecto. 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 4.2. Al iniciar Paramics, el *plugin* inicializa el servidor en un *thread* paralelo; el servidor luego se enlaza a un *socket* TCP y se bloquea en espera de mensajes entrantes desde un cliente TraCI (en nuestro caso, VEINS). Al recibir una serie de comandos TraCI, el servidor los interpreta, comunicándose con Paramics a través de su API, obteniendo datos y modificando el estado de la simulación. El servidor interpreta todos los comandos en un mensaje TraCI antes de enviar todos los mensajes de respuesta correspondientes en un único mensaje TraCI.

Esta arquitectura funciona de manera eficiente y permite 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 radica en la función de avance de simulación definido en la API de Paramics, `qps_GUI_runSimulation()`, la cual, como se descubrió, también actualiza la interfaz gráfica

de el modelador de Paramics a través de llamados a la librería Qt4 [45]. 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 genera corrupción de memoria en el motor gráfico (principalmente, lecturas de direcciones inválidas de memoria), lo cual causa 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*. 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 4.3. 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, y el ciclo comienza nuevamente.

Mediante esta arquitectura se logró eliminar por completo el problema presentado por `qps_GUI_runSimulation()`, y 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*.

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

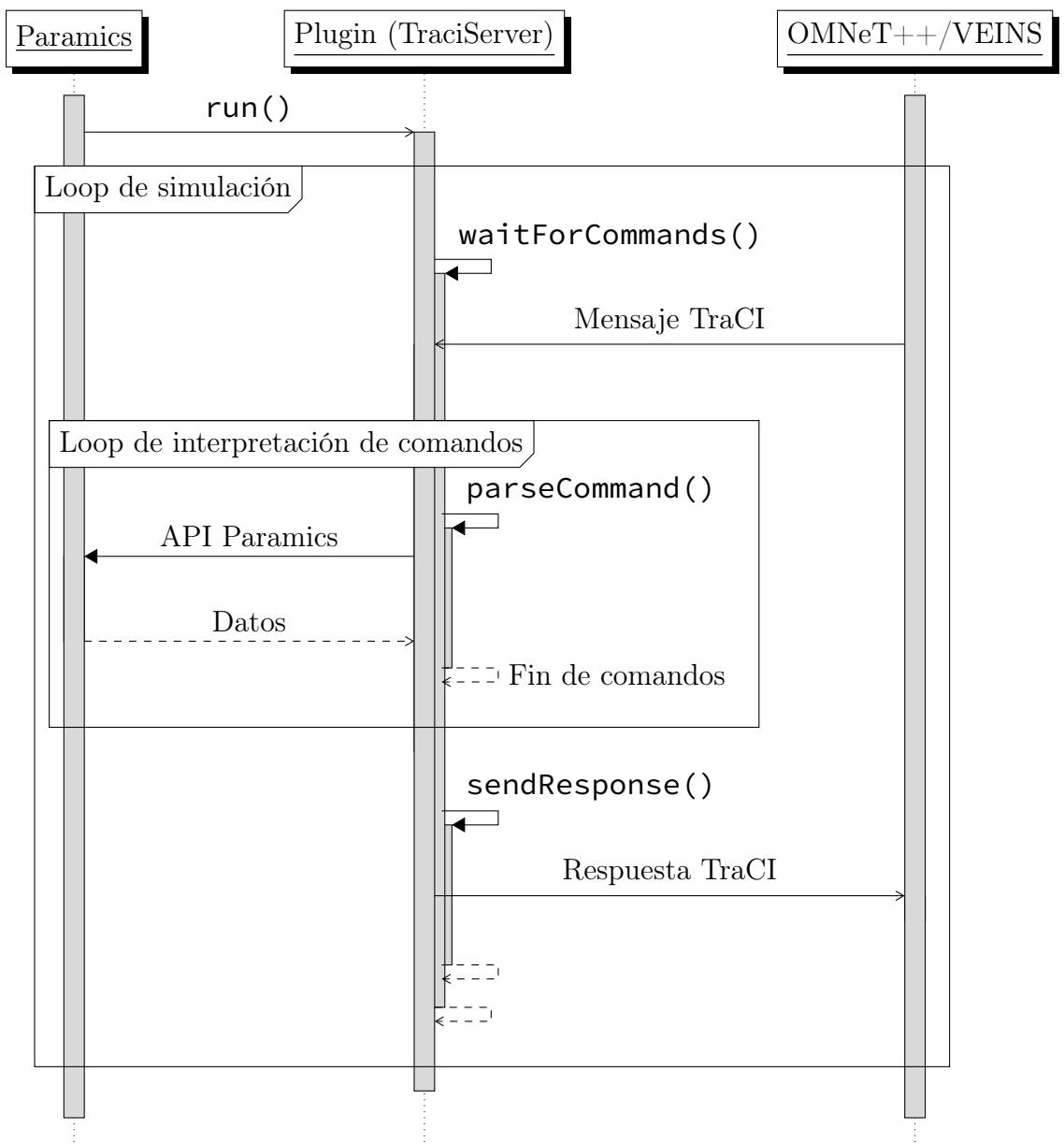


Figura 4.2: Arquitectura preliminar.

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 ΔT , el instante de tiempo en que se recibe el comando de avance como T_i y el instante de tiempo objetivo T_o , la simulación se avanzará un número $n \in \mathbb{N}$ de pasos, tal que

$$T_i + (n \times \Delta T) = T_f$$

$$T_f \geq T_o$$

$$T_i + ((n - 1) \times \Delta T) = T'_f$$

$$T'_f < T_o$$

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.

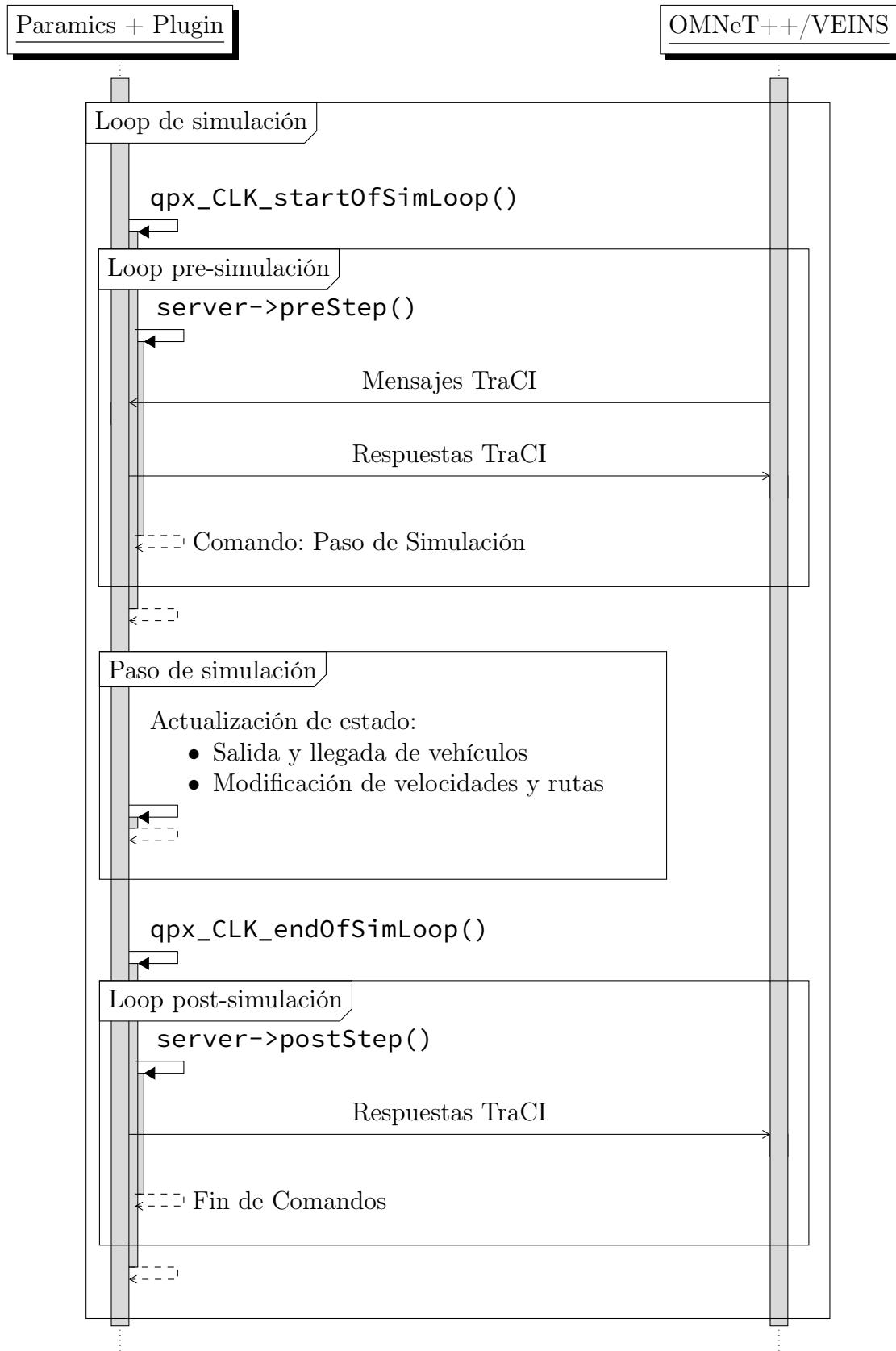


Figura 4.3: Arquitectura final del *framework*

4.1.2. Metodología de desarrollo

El desarrollo del *plugin* se llevó a cabo de manera iterativa, implementando funcionalidades esenciales en primera instancia, y luego construyendo sobre esta base, cuidando en cada paso de no pasar a llevar las funcionalidades previamente implementadas y perfeccionando implementaciones anteriores. El orden de desarrollo de las funcionalidades fue cuidadosamente planeado, tomando en cuenta que en muchos casos se requería un orden específico de implementación de funcionalidades; *e.g.* era imperativo el desarrollo de la funcionalidad de obtención de variables de vehículos antes de poder implementar suscripciones, ya estas últimas dependen de la funcionalidad de la primera. Las etapas generales de desarrollo que se siguieron fueron:

1. En primer lugar, se desarrolló la base de comunicaciones del *framework*, es decir, comunicación con el *socket*, recepción e interpretación de mensajes.
2. A continuación, se implementó la funcionalidad esencial de control de simulación (los comandos presentados en la sección 4.2.1), con el fin de poder establecer una primera conexión con un cliente TraCI y simplemente ejecutar una simulación sin otros comandos. El protocolo define un “*handshake*” consistente en la verificación de versiones de TraCI compatibles entre cliente y servidor, por lo que el correcto funcionamiento del comando de obtención de versión fue prioridad en esta etapa.
3. Se implementaron los comandos de obtención de variables de la simulación. Teniendo ya la base de comunicaciones funcionando, la implementación de éstos fue mucho más directa.
4. Sobre la implementación de los comandos de obtención de variables, se desarrollaron los distintos tipos de suscripciones disponibles.
5. Finalmente, en última instancia, se desarrollaron los comandos de modificación de estados, ya que éstos necesariamente requerían una fundación sólida dada su relativa complejidad.

Cabe destacar que, pese a la cuidadosa planificación realizada previa al desarrollo del *framework* (y como siempre sucede en el desarrollo de *software*), en muchas oportunidades fue necesario volver a un paso anterior para rediseñar o mejorar una implementación. El principal ejemplo de ésto es el rediseño de la arquitectura general del *plugin*, detallado en la sección 4.1.1, el cual implicó el rediseño y posterior reimplementación de gran parte de las etapas 1 (base de comunicaciones) y 2 (funcionalidad de control de simulación) del *plugin*.

En términos de control de versiones y manejo del historial del desarrollo se escogió utilizar el sistema *git* [46], dada su popularidad, extenso soporte y documentación y la familiaridad del memorista con este sistema. El servicio de *hosting* específico escogido para el sistema fue *GitHub* [47]; el código fuente del proyecto puede encontrarse en el perfil personal del autor [48], en el repositorio [10].

4.2. 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). La implementación de todas las funcionalidades está fuera del alcance de esta memoria, por lo que se escogió un subconjunto acotado de éstas a implementar, considerando en específico aquellos comandos esenciales para simulaciones de ITS que contribuyen al cumplimiento del objetivo general de esta memoria.

4.2.1. Comandos Implementados

Comandos de Control de Simulación

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

Comandos de Obtención de Variables

0xa4 Variables de vehículos

- **0x00** Lista de vehículos activos en la red
- **0x01** Número de vehículos activos en la red
- **0x36** Inclinación actual
- **0x39** Posición actual (3D)
- **0x40** Velocidad actual
- **0x42** Posición actual (2D)
- **0x43** Ángulo actual
- **0x44** Largo
- **0x4d** Ancho
- **0x4f** Tipo de vehículo
- **0x50** Calle actual
- **0x51** Identificador de pista actual
- **0x52** Índice de pista actual
- **0xbc** Altura

0xa5 Variables de tipos de vehículos

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

0xa6 Variables de rutas

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

0xa8 Variables de polígonos (edificios y estructuras)

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

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

0xa9 Variables de nodos (intersecciones) de la red

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

0xaa Variables de arcos (calles) de la red

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

0xab Variables de Simulación

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

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

Comandos de modificación de estado

0xc4 Variables de vehículo

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

4.3. Implementación

El código del *plugin* se separó en una serie de módulos lógicos que encapsulan y abstraen cada uno una categoría de funcionalidades de la interfaz con TraCI. De esta manera, se logró una separación lógica de las funcionalidades implementadas, y se simplifican futuras extensiones al código. La organización en archivos de estos módulos puede observarse en la figura 4.4, y puede explorarse en línea en el repositorio del proyecto [10].

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

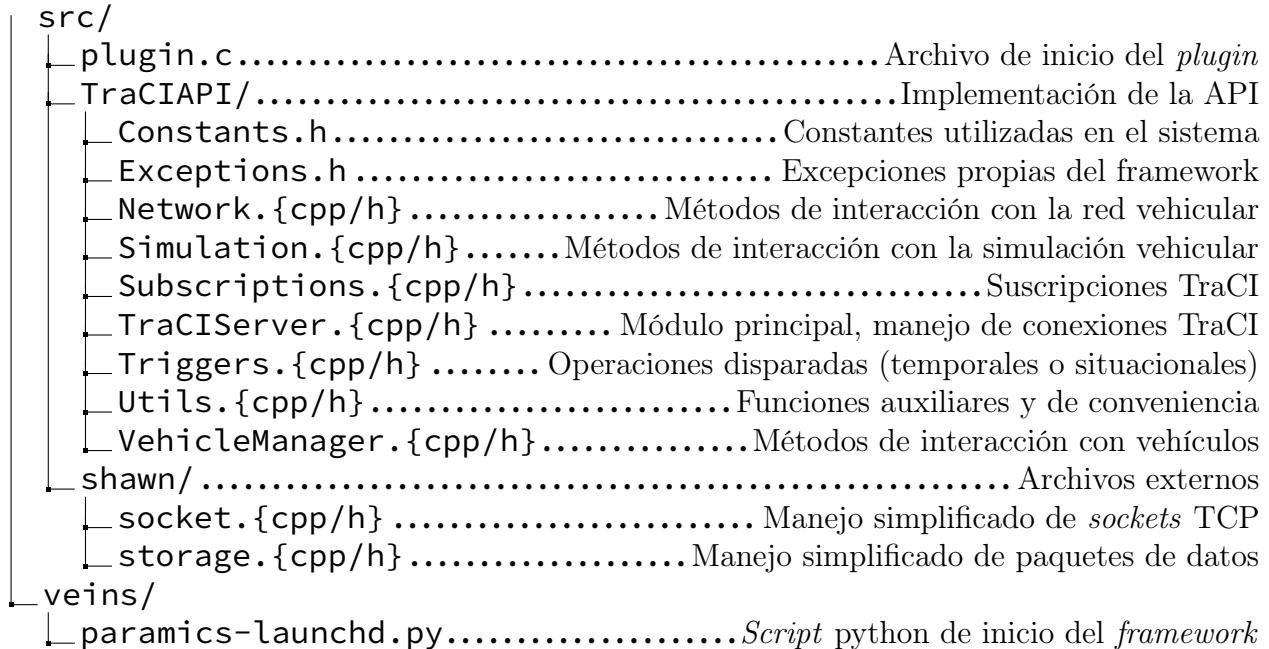


Figura 4.4: Estructura de archivos del código fuente del framework.

A continuación, se presenta en detalle la implementación de cada uno de los módulos del software. Se discutirán decisiones de diseño e implementación, además de consideraciones ligadas al funcionamiento de TraCI.

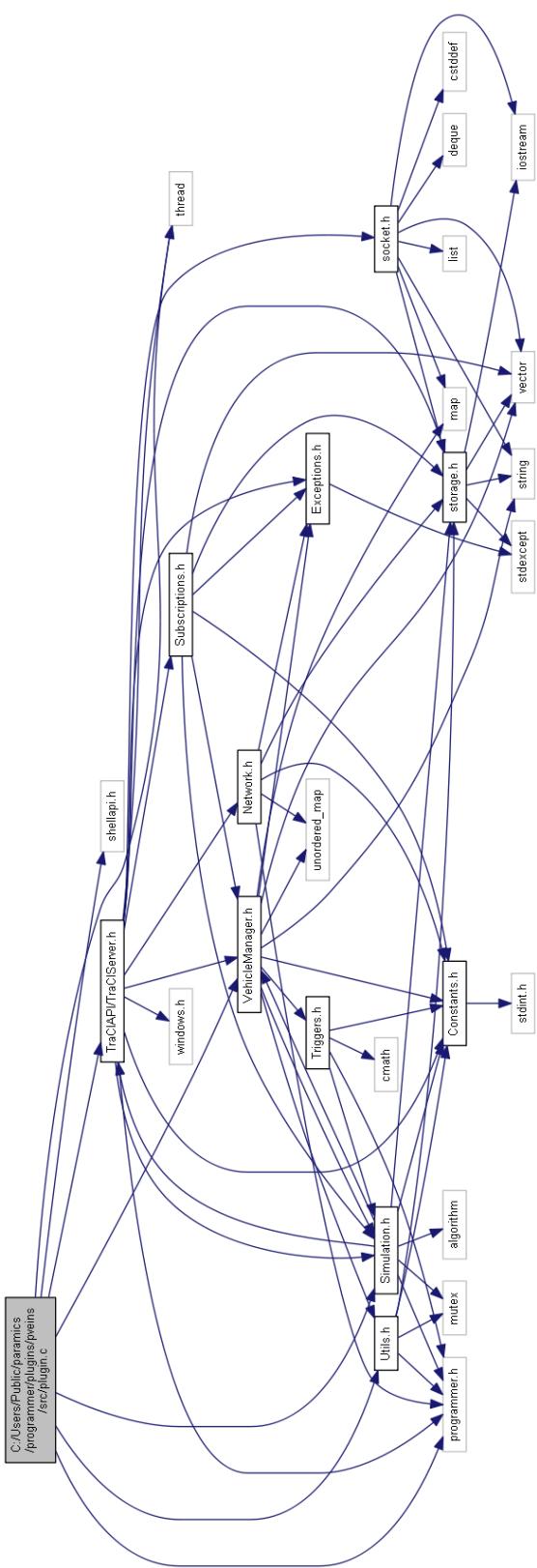


Figura 4.5: Gráfico de dependencia entre los componentes del *framework*.

4.3.1. plugin.c

Si bien en estricto rigor no es un módulo del *framework*, merece ser mencionado al ser el archivo principal del *plugin* desarrollado. En este archivo se definen las funciones de extensión 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 ejecución de Paramics. De no haberse especificado puerto, utiliza uno por defecto.
2. Inicializar un objeto `TraCIServer` (ver sección 4.3.2) encargado de las conexiones entrantes en el puerto anteriormente definido.

void qpx_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 4.3.2 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 4.3.4) 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* es llamada por el núcleo de simulación de Paramics cada vez que un vehículo necesita evaluar su elección de ruta, y debe retornar el índice de la siguiente

salida que el vehículo debe tomar (o 0 si se desea mantener la ruta por defecto). En el *plugin* se utiliza para aplicar rutas personalizadas otorgadas por el cliente 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(...)

Estas funciones se invocan en cada paso de simulación para cada vehículo en la simulación de tráfico de Paramics – **leadSpeed()** se invoca para aquellos vehículos que no tienen otro vehículo delante, mientras **followSpeed()** es invocada los vehículos que se encuentran detrás de otro.

Estas funciones deben retornar la rapidez que se le deberá aplicar al vehículo en cuestión en el siguiente paso de simulación. En el *framework*, se utilizan para aplicar cambios de velocidad dictados por comandos TraCI.

4.3.2. TraCIServer

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

1. Asociarse a un *socket* TCP, y esperar una conexión de un cliente TraCI.
2. Mientras exista una conexión abierta, recibir e interpretar comandos TraCI entrantes.
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 crea una 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 SUMO², distribuidos bajo una licencia BSD³.

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

Inicio de conexión TraCI

Como se mencionó en la sección 4.3.1, al iniciarse el *plugin* se crea un nuevo *thread*, en el cual se crea una instancia de 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. 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.

Recepción e interpretación de comandos entrantes

Como se explicó en la sección 4.1.1, 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. Este método se encarga de recibir mensajes nuevos entrantes a través del *socket* desde el cliente TraCI, e interpreta los comandos dentro de un *loop*. La implementación de éste método puede observarse en los apéndices, código C.2.

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.

²Fuente 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* [49]. Su fuente original se encuentra en <https://github.com/itm/shawn/tree/master/src/apps/tcpip>

³Licencia clases **tcpip::Socket** y **tcpip::Storage**: http://sumo.dlr.de/wiki/Libraries_Licenses#tcpip_-_TCP_2FIP_Socket_Class_to_communicate_with_other_programs

```

1  /**
2  * \brief Starts this instance, binding it to a port and awaiting
3  * connections.
4  */
5 void traci_api::TraCIServer::waitForConnection()
6 {
7     running = true;
8     std::string version_str = "Paramics TraCI plugin v" +
9         std::string(PLUGIN_VERSION) + " on Paramics v" +
10        std::to_string(qpg_UTL_parentProductVersion());
11    infoPrint(version_str);
12    infoPrint("Timestep size: " +
13        std::to_string(static_cast<int>(qpg_CFG_timeStep() * 1000.0f)) +
14        "ms");
15    infoPrint("Simulation start time: " +
16        std::to_string(Simulation::getInstance()
17            ->getCurrentTimeMilliseconds()) + "ms");
18    infoPrint("Awaiting connections on port " + std::to_string(port));
19
20    {
21        std::lock_guard<std::mutex> lock(socket_lock);
22        ssocket.accept();
23    }
24
25    infoPrint("Accepted connection");
26}

```

Código 4.1: Rutina de inicio de conexión.

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. Al igual que para **preStep()**, el código de éste método se puede encontrar en los anexos, código C.3.

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.

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, y su esqueleto puede observarse en el código 4.2. 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 subsiguiente 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.

Se definieron una serie de métodos en `TraCIServer` encargados de obtener variables de la simulación o modificar el estado de ésta. El funcionamiento de los métodos es idéntico en todos los casos (a excepción de `cmdGetPolygonVar()`), y se limita al siguiente procedimiento (ver ejemplo en el código 4.3):

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.

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 modulador 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.

```

1 void traci_api::TraCIServer::parseCommand(tcpip::Storage& storage)
2 {
3     /* ... */
4     uint8_t cmdLen = storage.readUnsignedByte();
5     uint8_t cmdId = storage.readUnsignedByte();
6     tcpip::Storage state;
7     /* ... */
8
9     if (cmdId >= CMD_SUB_INDOBJ && cmdId <= CMD_SUB_SIMOBJ)
10    {
11        // subscription
12        // | begin Time | end Time | Object ID | Variable Number | The
13        // list of variables to return
14        /* read subscription params */
15        /* ... */
16        addSubscription(cmdId, oID, btime, etime, vars);
17    }
18    else
19    {
20        switch (cmdId)
21        {
22            case CMD_GETVERSION:
23                /* ... */
24                /* ... */
25            default:
26                debugPrint("Command not implemented!");
27                writeStatusResponse(cmdId, STATUS_NIMPL, "Method not
28                                implemented.");
29        }
50    }
}

```

Código 4.2: Esqueleto de parseCommand()

```

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

```

Código 4.3: Ejemplo de método de obtención y empaquetado de variables en TraciServer

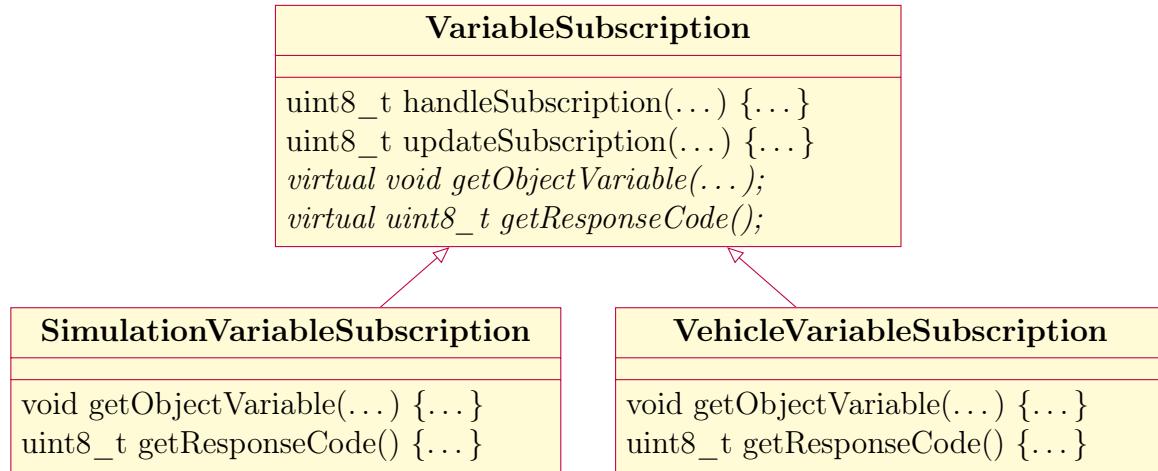


Figura 4.6: Diagrama de herencia, `VariableSubscription`

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 están asociadas, y por ende, cómo obtener esos datos desde la implementación interna del *plugin*. A raíz de esto, se decidió implementar un árbol de clases de C++ para representar las suscripciones en memoria (declarada e implementada en los archivos `src/TraCI API/Subscriptions.h` y `src/TraCI API/Subscriptions.cpp` respectivamente).

La raíz de éste árbol, la clase `VariableSubscription`, implementa la funcionalidad completa de evaluación y actualización de una suscripción en los métodos `handleSubscription()` y `updateSubscription()` (implementación completa de estos métodos en C.4), abstrayendo la obtención de datos específicos a cada tipo en los métodos `getObjectVariable()` y `getResponseBody()`. 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 `getResponseBody()`, 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 `getResponsibleCode()` de cada subclase, `handleSubscription()` es exactamente igual para todas (ver línea 7 en el código 4.4). Además, esto facilita la extensión futura del software.

```

1 void traci_api::TraCIServer::processSubscriptions(tcpip::Storage&
2   sub_store)
3 {
4   /* ... */
5   for (auto i = subs.begin(); i != subs.end();)
6   {
7     /* polymorphic evaluation of subscriptions; (*i) may be Vehicle
8       or Sim subscription */
9     sub_res = (*i)->handleSubscription(temp, false, errors);
10    if (sub_res == VariableSubscription::STATUS_EXPIRED
11      || sub_res == VariableSubscription::STATUS_OBJNOTFOUND)
12    {
13      delete *i;
14      i = subs.erase(i);
15    }
16    else
17    {
18      writeToStorageWithSize(temp, sub_results, true);
19      count++;
20      ++i; // increment
21    }
22    temp.reset();
23  }
24  /* ... */
25  sub_store.writeInt(count);
26  sub_store.writeStorage(sub_results);
}

```

Código 4.4: Rutina de evaluación de suscripciones en `TraCIServer`. `subs` es una variable de instancia de `TraCIServer` correspondiente a un vector de punteros `VariableSubscription*`, poblado de elementos de clases derivadas de `VariableSubscription`.

Creación de nuevas suscripciones

Por otro lado, para crear nuevas suscripciones, `TraCIServer` debe considerar la categoría de objetos a la cual se está suscribiendo, e insertar un puntero a un objeto con el tipo correspondiente en la variable de instancia `subs`. Esto sucede al recibir un comando con un identificador correspondiente a una suscripción; los parámetros de la suscripción son extraídos y delegados al método `addSubscription()` (código 4.2,línea 15). Este código puede estudiarse en su totalidad en el anexo C, código C.5, 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.4, 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 liberar 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.

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 dinámicamente se crea 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.

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.

```

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

```

Código 4.5: Verificación de actualización en `addSubscription()`.

```

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,
7             end_time, variables);
8         break;
9     case CMD_SUB_SIMVAR:
10        debugPrint("Adding SIM subscription.");
11        sub = new SimulationVariableSubscription(object_id, start_time,
12            end_time, variables);
13        break;
14    default:
15        writeStatusResponse(sub_type, STATUS_NIMPL, "Subscription type
           not implemented: " + std::to_string(sub_type));
16        return;
17    }

```

Código 4.6: Creación de una nueva suscripción. Notar la instanciación polimórfica.

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.

4.3.3. 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 creación de una única instancia de un 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 o función.

```

1 void traci_api::TraCIServer::writeStatusResponse(uint8_t cmdId,
2     uint8_t cmdStatus, std::string description)
3 {
4     debugPrint("Writing status response " + std::to_string(cmdStatus) +
5         " for command " + std::to_string(cmdId));
6     outgoing.writeUnsignedByte(1 + 1 + 1 + 4 +
7         static_cast<int>(description.length())); // command length
8     outgoing.writeUnsignedByte(cmdId); // command type
9     outgoing.writeUnsignedByte(cmdStatus); // status
10    outgoing.writeString(description); // description
11 }
12
13
14
15 void traci_api::TraCIServer::writeToOutputWithSize(tcpip::Storage&
16     storage, bool force_extended)
17 {
18     this->writeToStorageWithSize(storage, outgoing, force_extended);
19 }
20
21
22
23
24
25
26
27 }
```

Código 4.7: Escritura de datos en almacenamiento saliente.

Obtención de variables

Las variables obtenibles desde este módulo son todas aquellas que se relacionan con la simulación como ente abstracto, enumeradas en el ítem **0xab** de la sección 4.2.1. La implementación de los métodos **packSimulationVariable()** y **getSimulationVariable()**, encargados de facilitar el acceso a las variables representadas por este módulo, pueden observarse en el código C.6 en los apéndices. Cabe destacar que los módulos **VehicleManager** y **Network** cuentan con métodos análogos *muy* similares, por lo que no se incluirá el código de éstos últimos en el documento.

Se debe mencionar también la especial implementación de la obtención de algunas de las variables anteriormente mencionadas. En específico, las variables referentes a los vehículos que comenzaron o terminaron su viaje en el último paso de simulación son accesibles desde este

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

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

En segundo lugar se hablará del comando de obtención de las coordenadas de los límites de la simulación. Este es de extrema importancia para VEINS, ya que con estos valores se crea el escenario de comunicación inalámbrica en OMNeT++; de ser erróneos, tarde o temprano la posición de un vehículo (representado por un nodo de comunicación en OMNeT++) quedará fuera del escenario, gatillando un error fatal en la simulación. Infortunadamente, 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.7 en los anexos). Este cálculo se hace prácticamente a fuerza bruta, recorriendo todos los elementos que definen el alcance de la red (calles, intersecciones y zonas de emisión de vehículos), obteniendo sus coordenadas y luego obteniendo el rectángulo que las contiene (más un cierto margen de error). Si bien este método no escala bien con redes más grandes, su impacto en la eficiencia del sistema se estimó como mínimo ya que se accede una única vez por simulación a este valor.

4.3.4. VehicleManager

El módulo más complejo y grande (en términos de líneas de código) del *framework*. **VehicleManager** tiene como función abstraer el acceso a variables directamente relacionadas con los vehículos presentes en la simulación, mantener registros de dichos vehículos, y encargarse de ejecutar los diversos cambios de estado de éstos que puede solicitar el cliente (ver 4.2.1). Además, varios de éstos cambios de estado requieren acciones en múltiples instantes de tiempo (por ejemplo, el cambio de velocidad lineal, el cual se ejecuta durante un periodo de tiempo determinado), por lo que adicionalmente el módulo mantiene colas de eventos diferidos a ejecutar en instantes determinados.

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

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

Estado interno

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

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 4.3.1). 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 4.3.4), para efectuar cambios de velocidad dictados por el cliente Traci.

vhc_routes Mapa para el manejo de cambios de ruta desde Traci (ver sección 4.3.4).

lane_set_triggers *Hashmap* utilizado para relacionar vehículos con eventuales comandos de cambio de pista (ver sección 4.3.4).

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.

`getVehicleVariable()` y `getVhcTypesVariable()` son métodos relativamente simples, los cuales simplemente comparan el identificador de variable proporcionado como argumento y obtienen el valor solicitado mediante un llamado a alguno de los métodos auxiliares implementados para la obtención de variables. Dada su gran similitud con los métodos `packSimulationVariable()` y `getSimulationVariable()` ya presentados en la

sección 4.3.3, dedicada a la obtención de variables desde el módulo **Simulation**, no se presentará la implementación de los métodos propios del presente módulo en el documento (ver código C.6 para un acercamiento a la implementación real de éstos).

Modificación de estado de vehículos

La segunda función de **VehicleManager** es la de ejecutar los comandos de modificación de estado y comportamiento de los vehículos en la simulación (ver sección 4.2.1 para una lista de los comandos de este tipo que se implementaron). El método **setVehicleState()** es el encargado de la interpretación de comandos de cambio de estado, y su implementación es simple; determina el tipo de cambio de estado solicitado y si se encuentra implementado delega su ejecución al método correspondiente.

Dos de los comandos de cambio de estado implementados, **0x45 Coloreado** y **0x41 Cambio de velocidad máxima**, se ejecutan de manera directa a través de la API de Paramics. El resto requiere procedimientos más complejos, los cuales se describirán brevemente a continuación.

Cambios de velocidad lineal e instantáneo

Los comandos de cambio de velocidad de TraCI requieren un procedimiento especial ya que el efecto debe aplicarse por un periodo mayor a un paso de simulación, y por lo tanto es necesario un procedimiento que se encargue de mantener el efecto en el tiempo. Esto se implementó mediante la clase **traci_api::BaseSpeedController** y sus derivadas.

traci_api::BaseSpeedController define una clase compuesta únicamente de métodos virtuales, en base a la cual se construyen distintos tipos de controladores de velocidad. Como se comentó anteriormente, en la sección 4.3.4, **VehicleManager** mantiene un *hashmap* que relaciona vehículos con controladores derivados de la clase anteriormente mencionada. Este mapa es accedido para cada vehículo, en cada paso de simulación por el método **speedControlOverride()** (a su vez, invocado por **qpo_CFM_followSpeed()** y **qpo_CFM_leadSpeed()** – ver sección 4.3.1), el cual verifica si el vehículo en cuestión cuenta con un modificador de velocidad y aplica el cambio necesario. Además, cada controlador de velocidad cuenta con un método **repeat()** para verificar si debe seguir aplicándose en pasos de simulación futuros – de no ser así, se elimina de la representación interna.

En la implementación final del *framework* se definieron dos clases derivadas distintas de **traci_api::BaseSpeedController**: **traci_api::HoldSpeedController** y **traci_api::LinearSpeedChangeController**, las cuales implementan, respectivamente, los cambios inmediatos y lineales de velocidad definidos en el protocolo TraCI. La implementación de éstos puede revisarse en los apéndices, código C.8.

```

1  bool traci_api::VehicleManager::speedControlOverride(VEHICLE* vhc,
2  	float& speed)
3  {
4  	BaseSpeedController* controller;
5  	try
6  	{
7  		controller = speed_controllers.at(vhc);
8  		speed = controller->nextTimeStep();
9
10  	if (!controller->repeat())
11  	{
12  		speed_controllers.erase(vhc);
13  		delete controller;
14  	}
15
16  	return true;
17 }
18 catch (std::out_of_range& e)
19 {
20 	return false;
21 }
```

Código 4.8: Método de verificación de control de velocidad en VehicleManager. Verifica la existencia de un controlador personalizado de velocidad en `speed_controllers` y luego guarda el resultado de la evaluación en la variable `speed`.

Cambio de ruta

TraCI cuenta con un comando **0x57 Cambio de Ruta** mediante el cual un cliente puede proveer un número de arcos (calles) que el vehículo en cuestión deberá seguir antes de reencaminarse a su destino original. Este comando es especial en el sentido que requiere invalidar el ruteo interno de Paramics para dicho vehículo mientras esté siguiendo la ruta otorgada por el cliente, lo cual puede durar un tiempo indefinido.

Para esto se definió entonces un método **rerouteVehicle()** en **VehicleManager**, el cual recibe un puntero a un vehículo y su calle actual, y retorna el índice de la siguiente salida que debe tomar – en caso de tener una ruta personalizada, este método retornará el índice de la siguiente calle en la ruta, y de otro modo retorna 0, lo cual es interpretado por Paramics como una indicación a seguir la ruta dictada por el modelo interno.

Este método es invocado cada vez que un vehículo necesite evaluar su elección de ruta, a través de la función de extensión de la API de Paramics **int qpo_RTM_decision()** (ver sección 4.3.1).

Las rutas en sí se almacenan en la variable interna **vhc_routes**; un *hashmap* que relaciona vehículos con punteros a otro *hashmap* más. Este segundo mapa es de tipo **<LINK*, int>**, relacionando cada arco en la ruta con un índice a la siguiente salida que deberá tomar el vehículo al encontrarse sobre ese arco. De esta manera no fue necesaria la implementación de una estructura de datos adicional para el almacenamiento de las rutas.

Cambio de pista

Finalmente, el comando de cambio de pista de TraCI también debe aplicarse por un tiempo determinado. Infortunadamente, dadas ciertas limitaciones del modelo que utiliza Paramics para controlar la selección de pistas, este cambio no se pudo implementar como el cambio de ruta o el cambio de velocidad, dejando que la simulación de Paramics misma consultara la pista a tomar en el siguiente paso de simulación, sino que se debió implementar a “fuerza bruta”.

Esto se logró mediante la implementación de la clase de métodos virtuales **traci_api::BaseTrigger** y su clase derivada **traci_api::LaneSetTrigger**. **BaseTrigger** define una interfaz general para operaciones de ejecución periódica o diferida, y **LaneSetTrigger** representa una implementación de ésta interfaz para la ejecución constante de un cambio de pista por un tiempo definido.

La ejecución de estos *triggers* se maneja en el método **handleDelayedTriggers()** en **VehicleManager**, el cual es ejecutado al fin de cada paso de simulación. Cabe notar que si bien en la versión final del *framework* sólo se implementó una clase derivada de **BaseTrigger**, el diseño polimórfico de la evaluación de los *triggers* hace que en el futuro sea muy fácil la integración de nuevos procedimientos diferidos al sistema.

```

1 int traci_api::VehicleManager::rerouteVehicle(VEHICLE* vhc, LINK* lnk)
2 {
3     if (0 == qpg_VHC_uniqueID(vhc)) // dummy vhc
4         return 0;
5
6     // check if the vehicle has a special route
7     std::unordered_map<LINK*, int*>* exit_map;
8     try
9     {
10         exit_map = vhc_routes.at(vhc);
11     }
12     catch (std::out_of_range& e)
13     {
14         // no special route, return default
15         return 0;
16     }
17
18     int next_exit = 0;
19     try
20     {
21         next_exit = exit_map->at(lnk);
22     }
23     catch (std::out_of_range& e)
24     {
25         // outside route, clear
26         exit_map->clear();
27         delete exit_map;
28         vhc_routes.erase(vhc);
29     }
30
31     return next_exit;
32 }
```

Código 4.9: Método de reruteo en VehicleManager, para vehículos con rutas dictadas por un cliente TraCI.

```

1 void traci_api::LaneSetTrigger::handleTrigger()
2 {
3     int t_lane = target_lane;
4     // make sure we stay within maximum number of lanes
5     int maxlanes = qpg_LNK_lanes(qpg_VHC_link(vehicle));
6     if (t_lane > maxlanes)
7         t_lane = maxlanes;
8     else if (t_lane < 1)
9         t_lane = 1;
10
11    int current_lane = qpg_VHC_lane(vehicle);
12    if (current_lane > t_lane) // move outwards
13        qps_VHC_laneChange(vehicle, -1);
14    else if (current_lane < t_lane)
15        qps_VHC_laneChange(vehicle, +1); // move inwards
16    else
17        qps_VHC_laneChange(vehicle, 0); // stay in this lane
18 }
```

Código 4.10: Cambio de pista, implementado en LaneSetTrigger

```

1 void traci_api::VehicleManager::handleDelayedTriggers()
2 {
3     // handle lane set triggers
4     debugPrint("Handling vehicle triggers: lane set triggers");
5     for (auto kv = lane_set_triggers.begin(); kv != lane_set_triggers.end();)
6     {
7         kv->second->handleTrigger();
8
9         /* check if need repeating */
10        if (!kv->second->repeat())
11        {
12            delete kv->second;
13            kv = lane_set_triggers.erase(kv);
14        }
15        else
16            ++kv;
17    }
18    debugPrint("Handling vehicle triggers: done");
19 }
```

Código 4.11: Manejo de *triggers* para operaciones diferidas en VehicleManager

4.3.5. Otros módulos

Network

El módulo **Network** encapsula el acceso a variables de elementos de la red, en particular, calles, intersecciones y rutas. Al igual que **VehicleManager** y **Simulation**, se implementó utilizando un *singleton*.

La implementación del módulo es muy simple, ya que sólo otorga acceso a elementos no modificables por el usuario. Sus métodos de acceso a variables, **getLinkVariable()**, **getJunctionVariable()** y **getRouteVariable()** son altamente similares al ya presentado **getSimulationVariable()** (código C.6), y las únicas variables de instancia que mantiene son dos *hashmaps*, las cuales se inicializan al momento de instanciarse el módulo:

route_name_map De tipo `<std::string, BUSROUTE*>`, relaciona nombres de rutas con punteros a éstas, para un acceso más directo y eficiente.

route_links_map De tipo `<BUSROUTE*, std::vector<std::string>>`, asocia cada ruta con sus arcos constituyentes.

```
1 traci_api::Network::Network()
2 {
3     int routes = qpg_NET_busroutes();
4     for (int i = 1; i <= routes; i++)
5     {
6         BUSROUTE* route = qpg_NET_busrouteByIndex(i);
7         std::string name = qpg_BSR_name(route);
8
9         route_name_map[name] = route;
10
11        int link_n = qpg_BSR_links(route);
12        std::vector<std::string> link_names;
13
14        LINK* current_link = qpg_BSR_firstLink(route);
15        link_names.push_back(qpg_LNK_name(current_link));
16
17        for (int link_i = 0; link_i < link_n - 1; link_i++)
18        {
19            current_link = qpg_BSR_nextLink(route, current_link);
20            link_names.push_back(qpg_LNK_name(current_link));
21        }
22
23        route_links_map[route] = link_names;
24    }
25 }
```

Código 4.12: Constructor del módulo **Network**

Utils

En `Utils.{cpp/h}` se implementaron una serie de funciones de conveniencia:

- `debugPrint()` e `infoPrint()`, para la escritura de mensajes a la ventana de información de Paramics, además de la salida de error y estándar respectivamente.
- Las funciones `readTypeChecking<tipo>()`, las cuales reciben un elemento de tipo `tcpip::Storage` y leen el primer elemento contenido ahí, verificando que sea del tipo deseado. Estas funciones no fueron implementadas por el memorista, sino obtenidas del código fuente de SUMO.
- Las funciones `RGB2HEX()` y `HEX2RGB()`, para la conversión de colores entre ambas representaciones.

Constants

En el archivo de cabecera `Constants.h` se declararon una serie de constantes globales al sistema. No obstante, cada módulo maneja además un conjunto de constantes propias. Cabe notar que las constantes del *framework* fueron definidas como *variables constantes estáticas*, y no como *definiciones del preprocesador*.

<code>#define DUMMY_CONST 0x42</code>	<code>static const</code> <code>DUMMY_CONST = 0x42;</code>
---------------------------------------	---

Figura 4.7: Definición del preprocesador (izq.) *vs* variable constante estática (der.).

La diferencia entre ambos métodos de definición radica en la interpretación que el *toolchain* de compilación les da. Las *definiciones del preprocesador* son interpretadas por el *preprocesador*, antes de pasar por el compilador, y se ejecutan como simples reemplazos textuales en el código por el valor definido. Por otro lado, las variables constantes son tratadas como cualquier otra variable, y por ende cuentan con todas las propiedades de éstas. La decisión de utilizar este segundo método se tomó en base a que las variables constantes tienen la particularidad de estar restringidas a su *scope* – es decir, si se declaran por ejemplo dentro de un *namespace* (como es el caso en `Constants.h`), su identificador no queda definido fuera de dicho entorno. Esto es altamente deseable para futuras extensiones del *framework*, *e.g.* en el caso que se desee integrar con algún otro *plugin* que ya cuente con sus propias constantes, ya que de esta manera se facilita la distinción de cual valor pertenece a qué parte del software. Por otro lado, las *definiciones de preprocesador* tienen la ventaja de que no ocupan memoria en el programa final compilado (ya que los identificadores en el código se reemplazan directamente por el valor antes de compilarse el código); no obstante, dado que el número de constantes definidas es altamente acotado, el impacto en memoria de declararlas como variables del lenguaje es negligible.

paramics-launchd.py

El archivo **paramics-launchd.py** corresponde a una versión modificada del *script* de Python 2.7 **sumo-launchd.py** incluído con la distribución de VEINS, modificado para su funcionamiento con Paramics en vez de SUMO.

Este archivo funciona como un *daemon* de ejecución del *framework*, cuya labor es la de recibir conexiones entrantes desde clientes TraCI y preparar la simulación de Paramics para dar inicio a la simulación bidireccional. Su funcionamiento se detalla a continuación:

1. El usuario inicia el *script* en el *host* donde se desea correr la simulación vehicular de Paramics. Gracias a la arquitectura cliente-servidor de VEINS (y por extensión, del presente proyecto), ambos simuladores pueden ejecutarse en equipos distintos (virtuales o físicos).
2. Por defecto, el *script* se asocia a un *socket* en el puerto 9999 y espera conexiones TraCI entrantes.
3. Por otro lado, el usuario inicia la simulación de VEINS en OMNeT++. Esta automáticamente se conecta con el puerto 9999 del *host*, y le transfiere los contenidos de un archivo XML **paramics-launchd.xml**, definido por el usuario (ver ejemplo en figura 4.13). Este archivo define parámetros de simulación como la red vehicular a utilizar y la *semilla* deseada para la generación de valores pseudoaleatorios.
4. Al recibir una conexión entrante junto con el archivo de configuración, **paramics-launchd.py** prepara el inicio de la simulación integrada siguiendo los siguientes pasos:
 - i. En primer lugar, encuentra un puerto de red disponible en el *host* y notifica al cliente de esta elección.
 - ii. Luego, prepara la red vehicular, copiando los archivos de definición y configuración de ésta a una ubicación temporal y modificándolos para incluir el valor de semilla especificado por el usuario y la dirección al *dll* del *plugin*.
 - iii. Inicia el modelador de Paramics con el *plugin*, especificando la red a simular y el puerto asignado.
5. Finalmente, al terminar la simulación bidireccional, el *script* finaliza la conexión entre ambos simuladores y limpia los archivos temporales generados (esta acción puede suprimirse mediante un parámetro de consola al ejecutar el *script*).

```

1  <?xml version="1.0"?>
2  <launch>
3      <basedir path="X:\PVEINS\pveins" />
4      <network name="example8_network" />
5      <seed value="1234" />
6  </launch>

```

Código 4.13: Ejemplo de archivo XML de inicialización de la simulación.

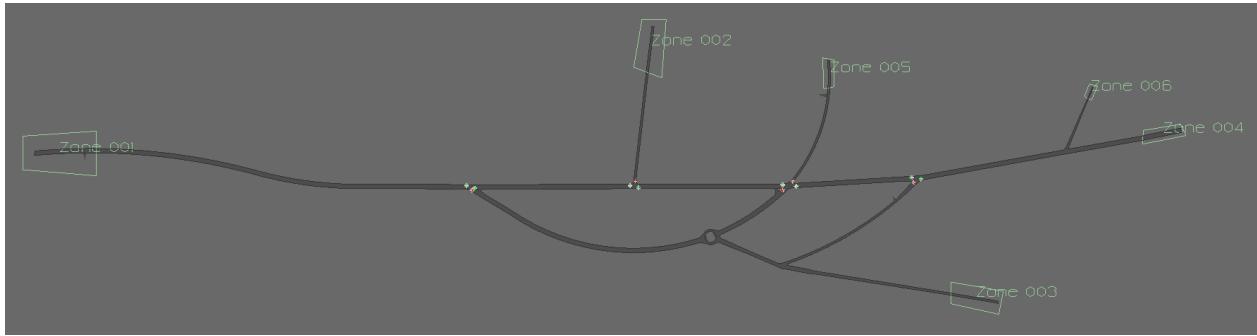


Figura 4.8: Red de transporte utilizada para las pruebas preliminares.

4.4. Pruebas preliminares

La validación preliminar del *framework* se realizó utilizando la implementación de TraCI en Python incluida en la distribución de SUMO. Esta consiste en una librería para Python 2.7+ y 3.0+, la cual implementa un cliente TraCI en su totalidad ([50], [51]), permitiendo así la validación del correcto funcionamiento de los comandos implementados en el *framework* PVeins.

Por otro lado, la red de transporte utilizada para las pruebas corresponde a una red simple, incluida por defecto en la instalación de Paramics. Esta red consiste en un corredor central y conjunto de calles que lo intersectan (ver figura 4.8). El flujo de vehículos en la red es medio-bajo, manteniéndose bajo los 500 vehículos activos en toda la red en cualquier momento dado.

A lo largo del desarrollo de este trabajo, se utilizó la librería anteriormente mencionada, junto con el entorno de *debugging* de Visual Studio y la red de transporte, para probar la correcta implementación de cada funcionalidad que se le agregó al *framework*. Se implementaron simples *scripts* en Python para probar cada una de las funcionalidades desarrolladas; sólo se utilizará uno de éstos como ejemplo a continuación, ya que no es factible ni interesante exponer todas las pruebas realizadas en este documento, dada la gran cantidad de éstas que se efectuaron y el alto grado de similitud que existe entre las mismas.

Además, implementado ya el *framework* en su totalidad, se realizaron pruebas de validación de mayor envergadura, midiendo la eficiencia y la efectividad del sistema para la simulación de grandes redes de transporte. Los resultados de éstas pruebas se presentan en el capítulo 5.

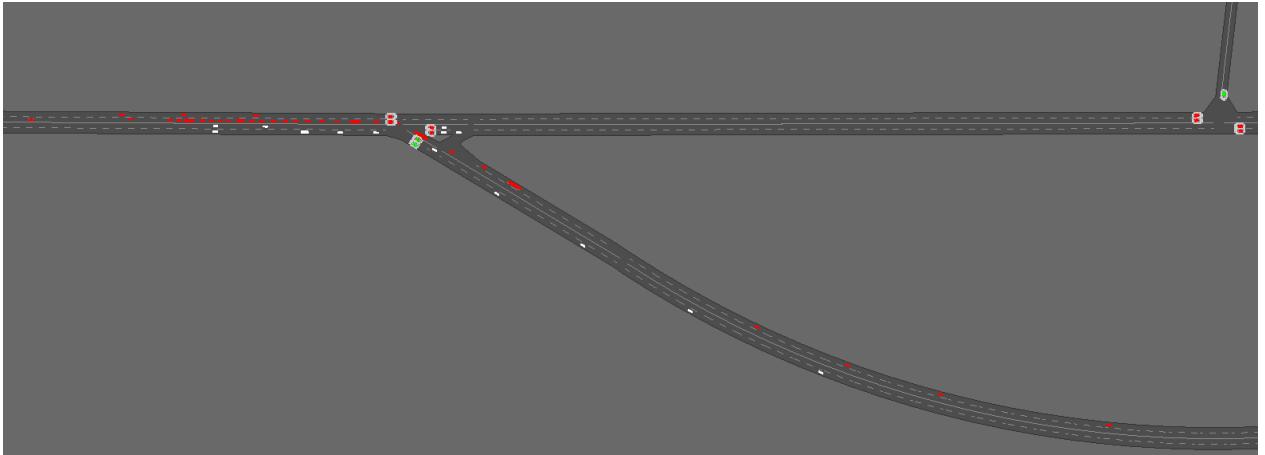


Figura 4.9: Visualización del *test* de cambio de ruta en curso. Los vehículos pintados de rojo son aquellos afectados por el cambio.

4.4.1. Ejemplo de script de prueba: cambio de ruta

El código 4.14 expone el *script* utilizado para una prueba de la funcionalidad del cambio de ruta en TraCI, la cual fue implementada en la última etapa de desarrollo del software por lo que ya se contaba con una base con más funcionalidades sobre la cual construir (*e.g.*, obtención de valores mediante suscripciones).

El procedimiento es simple; el *script* avanza la simulación en un *loop*, obteniendo luego de cada iteración la lista de vehículos en la red. De estos vehículos, encuentra aquellos que se encuentran en la primera calle de una ruta predefinida y procede a cambiar su ruta original por una nueva, al mismo tiempo pintándolos de un color rojo para poder distinguirlos del resto. El resultado puede observarse en la figura 4.9.

Este código expone de manera clara la estructura del *loop* de simulación TraCI, estructura que se replica en VEINS (aunque de manera mucho más compleja); el cliente es quien controla la ejecución de los pasos de simulación, avanzando el escenario a medida que va realizando sus propios cálculos y análisis. También demuestra las razones por la cual se llevó a cabo el desarrollo en etapas comentado en la sección 4.1.2 – si bien el enfoque de esta prueba es la funcionalidad de cambio de ruta, es necesario también el uso de otras funcionalidades de TraCI como el *handshake* de inicio de conexión (`traci.init(...)`), la suscripción a variables de vehículo (`traci.vehicle.subscribe(...)` y `.getSubscriptionResults(...)`), el avance de la simulación (`traci.simulationStep()`) y la obtención de variables de vehículo (`traci.getRoadID(...)`).

```

1 PORT = 8245
2 new_route = ["2:6c", "6c:25", "25:15"]
3 affected_cars = []
4
5 def run():
6     """execute the TraCI control loop"""
7     traci.init(PORT)
8     print("Server version: " + str(traci.getVersion()))
9     traci.vehicle.subscribe("x",[0]) # sub to list of vehicles in sim
10    for i in range(0, 10000):
11        traci.simulationStep()
12        car_list = traci.vehicle.getSubscriptionResults("x")[0]
13        for car in car_list:
14            current_road = traci.vehicle.getRoadID(car) # get road
15            if (current_road == new_route[0]) and (car not in
16                affected_cars):
17                print("route change for " + str(car))
18                traci.vehicle.setColor(car, (255, 0, 0, 0))
19                traci.vehicle.setRoute(car, new_route)
20                affected_cars.append(car)
21
22 traci.close()

```

Código 4.14: *Script* para la prueba de cambio de ruta.

Capítulo 5

Validación

La implementación del *plugin* se sometió a un conjunto de diversas pruebas para verificar su correcto funcionamiento y la eficiencia del *framework*. Estas pruebas consistieron en la simulación de escenarios realistas, de idéntica complejidad a aquellos escenarios para los que fue concebido y diseñado.

5.1. Escenario y Análisis

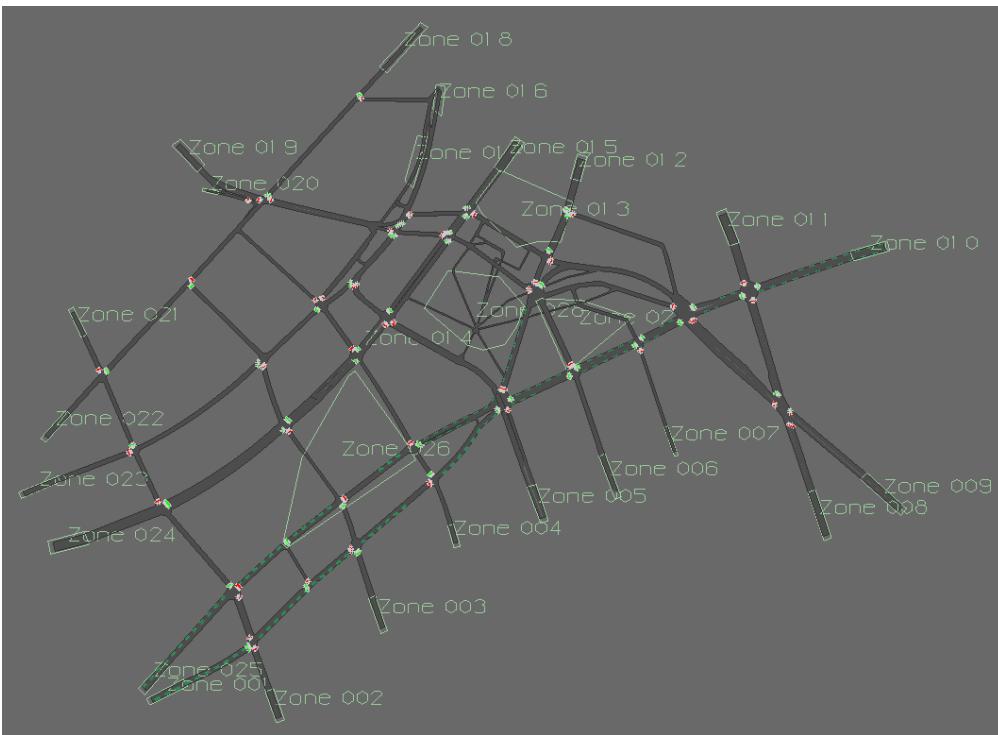
5.1.1. Escenario modelado

Como escenario de transporte para la validación del *framework* se utilizó un modelo de un sector de la ciudad de Santiago de Chile, el cual consiste en una simulación detallada del flujo vehicular en la comuna de Providencia. Este modelo fue creado en 2010 por Víctor Zúñiga para el desarrollo de su memoria de grado [9], y simula el impacto sobre el sector entre las avenidas Providencia, Tobalaba, Andrés Bello y Santa María proyectado en ese entonces por la construcción de un nuevo centro comercial (ver figura 5.1).

Sobre este escenario vehicular se construyó un modelo de ITS, utilizando el *plugin* desarrollado y el *framework* VEINS en OMNeT++. Este consistió en un escenario en que un vehículo sufre un desperfecto en una cierta calle de la simulación y emite *beacons* de advertencia en *broadcast* a todos los demás vehículos que se encuentran dentro del alcance de la transmisión. A aquellos vehículos que reciben un *beacon*, y que se puede predecir ingresarán a la calle en que se encuentra el vehículo averiado, se les modifica luego su ruta mediante VEINS y TraCI.

De manera más detallada, el escenario funciona de la siguiente manera:

1. Al iniciarse el *framework*, OMNeT++ (utilizando VEINS) inicializa una conexión TraCI con el *plugin* en Paramics.
2. Por cada vehículo que ingresa a la red de transporte, OMNeT++ crea un módulo dotado



(a) Mapa en Paramics del escenario simulado.



(b) Escenario simulado en la “vida real”, Google Maps.

Figura 5.1: Escenario modelado, Paramics vs. “vida real”.

Sistema Operativo	Windows 10 v.10.0.14393
Procesador	Intel Core i7 4720HQ @ 2.60 GHz
Nº de Núcleos / Threads	4 núcleos / 8 threads
Arquitectura	x86_64
RAM	12 GB DDR3L 1600 MHz
Tarjeta de Video	NVIDIA GeForce GTX 960M
Memoria Video	2 GB GDDR5

Tabla 5.1: Especificaciones técnicas del entorno de simulación.

de lógica y capacidades comunicaciones en su simulación de red, y asocia el movimiento de este módulo al vehículo en Paramics. En adelante, se entenderá por “vehículo” el par consistente en el vehículo en Paramics y su módulo asociado en OMNeT++.

3. Periódicamente se verifica la posición de cada vehículo, y al detectar el primero en ingresar a la calle del “accidente”, este es detenido. La calle en cuestión para esta simulación fue el arco “40:5”, el cual corresponde a Avenida Vitacura, frente al centro comercial. El vehículo además se colorea rojo para su fácil identificación.
4. El módulo OMNeT++ del vehículo accidentado emite luego, cada 5 segundos, un mensaje WAVE [4] en *broadcast*.
5. Aquellos vehículos que reciban el *beacon* de emergencia, se encuentren en alguna de las calles aledañas al accidente y que tengan un destino que probablemente los haga pasar por la calle afectada, cambian su ruta a utilizar el arco “40:7”, correspondiente a la calle Holanda, entre Avda. Vitacura y Avda. Providencia. Además, se cambia su color a púrpura para visualizarlos de manera más fácil en Paramics.

Esta simulación dura un total de **15 minutos de tiempo simulado**, y se ejecutó con múltiples configuraciones del sistema de transporte y del sistema de comunicaciones, las cuales se verán a continuación en las secciones 5.2 y 5.3. Las especificaciones técnicas del equipo en donde se realizaron estos experimentos pueden observarse en la tabla 5.1.

Las mediciones realizadas se enfocaron a probar el funcionamiento del *framework* en dos categorías de análisis; **eficiencia computacional** de la implementación e **impacto sobre el modelo de transporte**. De esta manera se pretende demostrar que PVEINS es una opción viable para la investigación en Sistemas Inteligentes de Transporte, tanto en términos de los recursos que utiliza el *framework* para la ejecución de las simulaciones como en términos de la validez de los resultados obtenidos.

Los resultados obtenidos de cada simulación fueron exportados desde OMNeT++ a archivos CSV, los cuales luego se analizaron utilizando Python 3.6. Se utilizaron las librerías Pandas [52], para el manejo de los datos de manera eficiente, Numpy [53], para cálculos, y Matplotlib [54] y matplotlib2tikz [55] para la generación de gráficos que permitiesen analizar los datos de manera más efectiva e intuitiva.

Factor de Demanda	Ctdad. Prom. Vehículos
100 %	1379.9
75 %	868.75
50 %	514.5825
25 %	246.5675

Tabla 5.2: Tabla de relación entre factor de demanda y cantidad promedio de vehículos por instante de tiempo en el escenario de prueba.

5.2. Eficiencia Computacional

5.2.1. Mediciones Realizadas

El principal factor a medir en esta categoría es la relación entre la cantidad de vehículos en una simulación y el tiempo real que demora el *framework* en simular el escenario, para una duración en tiempo simulado específica. De esta manera, se pretende caracterizar el comportamiento del *software* para escenarios vehiculares de alta complejidad, en los cuales pueden llegar a interactuar miles de vehículos. Además, interesa también la carga en términos de recursos de sistema que genera la ejecución de la simulación sobre el equipo de prueba.

A través de éstos datos se pretende generar un perfil del *software* que indique los requisitos que impone sobre el entorno de simulación, y su factibilidad de uso para sistemas de transporte complejos tanto en entornos de simulación de alto rendimiento como de mediano y bajo.

Antes de detallar las simulaciones realizadas, se debe definir el término *factor de demanda*. Éste corresponde a un elemento de configuración de la simulación de transporte en Paramics, el cual caracteriza la carga vehicular sobre el sistema de transporte en cada instante de tiempo. En términos más simples, el factor de demanda regula la cantidad de vehículos que Paramics inserta a la red durante la simulación; es un valor porcentual cuya correspondencia en cantidad real de vehículos en la red dependerá de las características particulares de cada simulación. Sin embargo, los valores aproximados para el escenario utilizado, para distintos factores de demanda, pueden observarse en la tabla 5.2.

Se realizaron entonces 16 ejecuciones del escenario (de aquí en adelante, denominadas *runs*) con cuatro factores de demanda distintos (4 *runs* con 100 %, 4 con 75 %, 4 con 50 % y 4 con 25 %) y cada una con una *semilla* distinta para los generadores de números pseudoaleatorios. De éstos *runs* se extrajeron las siguientes estadísticas:

1. Timestamp de inicio del *run*.
2. Timestamp de fin del *run*.
3. Duración en tiempo real del *run*.
4. Cantidad de vehículos en la red de transporte, cada 1 minuto de tiempo simulado.

Factor de Demanda	Nro. Prom.	Vehículos	Tiempo Promedio [s]
100 %		1379.9	1471.5
75 %		868.75	683.5
50 %		514.5825	275.75
25 %		246.5675	113.25

Tabla 5.3: Promedio cantidad de vehículos en simulación (por instante de tiempo) vs. tiempo promedio de simulación, 15 minutos de tiempo simulado.

De manera adicional, para determinar el principal origen del *overhead* presente en las simulaciones, se ejecutaron también una serie de *runs* de la simulación *sin* OMNeT++, sólamente utilizando el *plugin* controlado por un *script* Python, y se midió su tiempo de ejecución.

Finalmente, se realizó una ejecución adicional de un *run* con factor de demanda 100 %, y se midió la carga sobre el equipo de prueba mientras se ejecutaba la simulación utilizando las herramientas de monitoreo de sistema de Microsoft Windows. En específico, se midió la carga porcentual sobre el procesador, la memoria RAM utilizada y la cantidad de operaciones de escritura y lectura del disco físico durante la simulación.

5.2.2. Resultados

Duración de la Simulación

La tabla 5.3 presenta los resultados obtenidos del tiempo de duración de la simulación versus la cantidad de vehículos promedio por instante de tiempo; estos datos se visualizan además de manera gráfica en la figura 5.2.

De estos resultados se puede concluir que el *framework*, como era de esperarse dada la naturaleza de la simulación, presenta una relación levemente exponencial entre la cantidad promedio de nodos en la red vehicular con el tiempo que efectivamente demora una simulación en completar su ejecución.

Se utilizó numpy para calcular un ajuste polinomial a los datos obtenidos, obteniendo la siguiente relación entre cantidad de vehículos promedio en la simulación N_v y tiempo de ejecución de ésta en segundos, $T(N_v)$:

$$T(N_v) = (5,829 \times 10^{-4})N_v^2 + (2,586 \times 10^{-1})N_v + 6,317$$

Esto indica que a pesar de ser exponencial, la relación presenta una curva bastante suavizada. Es factible entonces simular escenarios de escala aún mayor que la presentada en esta memoria (la cual, cabe notar, no es menor), sin mayores dificultades.

La figura 5.3 presenta por lo otro lado la evolución de la red vehicular en términos de

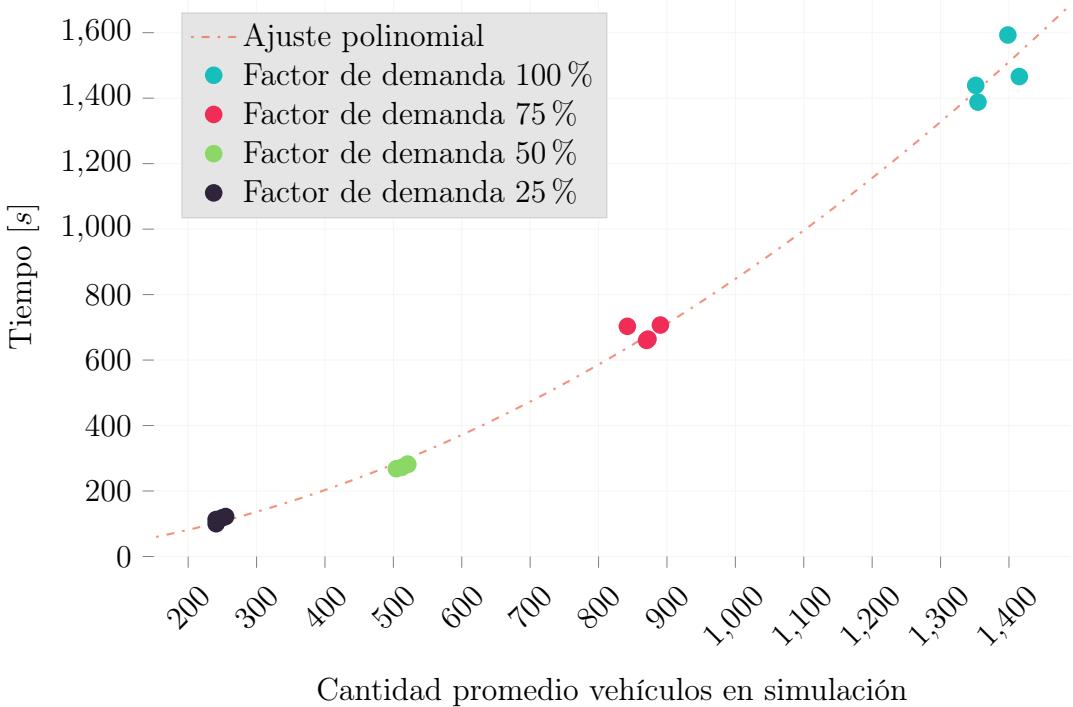


Figura 5.2: Gráfico de dispersión del promedio de vehículos en simulación por instante de tiempo vs. tiempo total de simulación, para una simulación de 15 minutos de tiempo simulado.

cantidad de vehículos para un *run* con factor de demanda de 100 %, tanto en tiempo real como en tiempo simulado. Este gráfico permite visualizar además como la cantidad de vehículos en la red afecta el tiempo de ejecución de cada paso de simulación.

Para determinar si la mayor parte del *overhead* en la ejecución de las simulaciones provenía desde el *plugin* en sí o desde OMNeT++, se realizaron 6 *runs* de la simulación, con factor de demanda 100 %, omitiendo la integración con VEINS. Es decir, se ejecutó el escenario dentro de Paramics, con el *plugin* activado, pero sin accidente y sin comunicación entre los nodos, controlado por un *script* Python. Estas mediciones arrojaron como promedio 63.37 segundos de tiempo real por 15 minutos de tiempo simulado, indicando claramente que la mayor parte del *overhead* proviene de OMNeT++, y de la simulación de la comunicación inalámbrica.

Carga computacional

En términos de carga sobre el entorno de simulación, se pueden observar los resultados obtenidos en la figura 5.4. Esta figura ilustra la carga sobre el sistema en términos porcentuales, para el escenario con factor de demanda 100 %. En específico, se puede observar como el uso promedio del procesador aumenta en aproximadamente un 20 % durante la simulación, situación fácilmente manejable para cualquier procesador moderno. Además, el uso de memoria aumenta en menos de un 5 % – en términos numéricos, el sistema utiliza menos de 600 MB para simular un escenario con un promedio de 1400 nodos presentes en cualquier instante, lo cual es un valor muy razonable si se considera que el estándar de memoria RAM

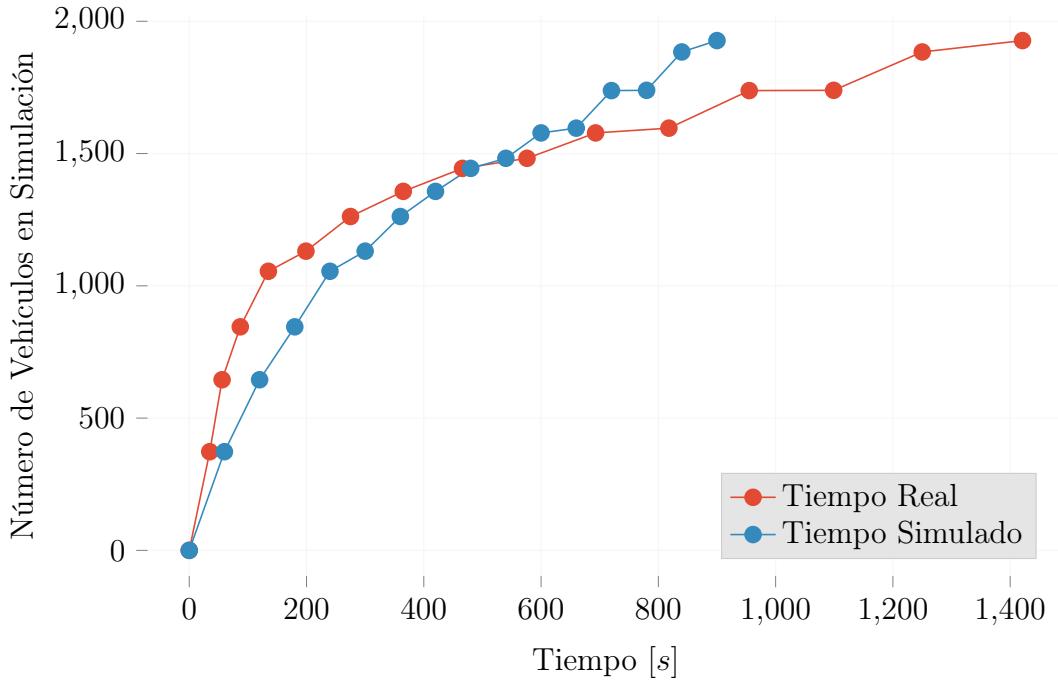


Figura 5.3: Evolución de la cantidad de vehículos en una simulación con factor de demanda 100 %, para tiempo real y simulado.

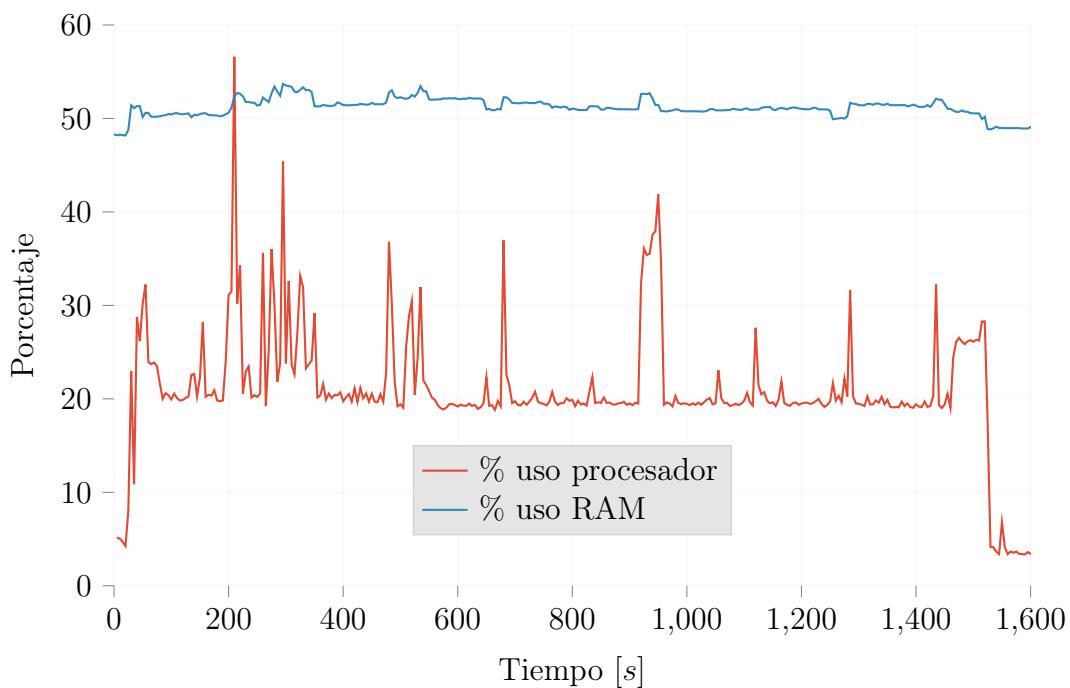


Figura 5.4: Carga sobre el sistema durante una simulación con factor de demanda 100 %.

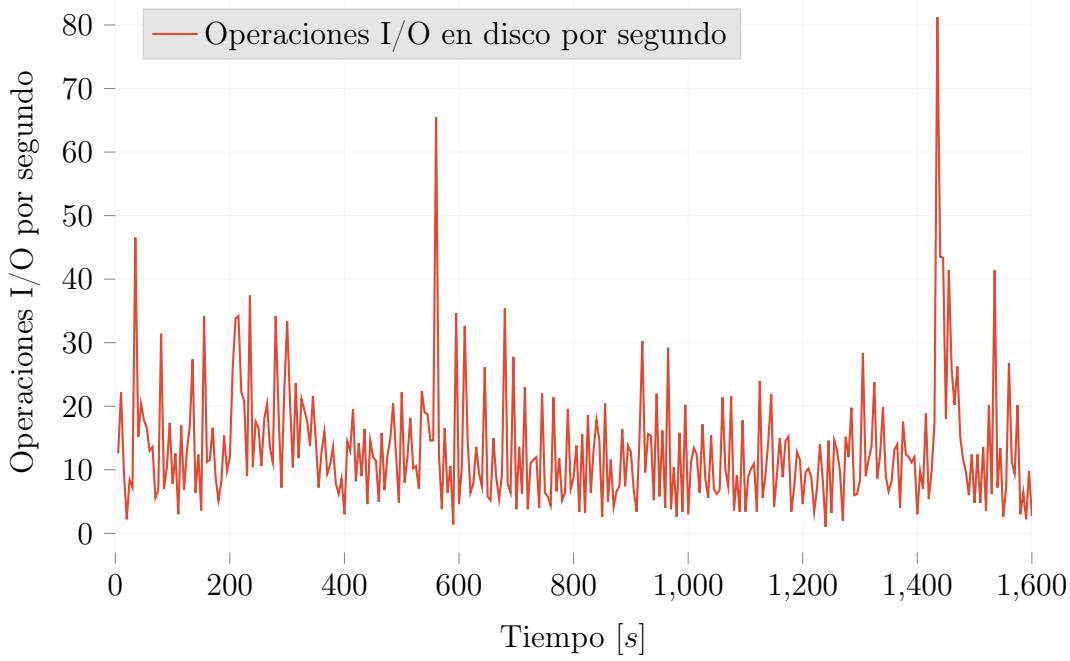


Figura 5.5: Lecturas y escrituras de disco por segundo durante una simulación con factor de demanda 100 %.

para computadores personales hoy en día es por lo menos 4 GB [56], [57].

Finalmente, la figura 5.5 ilustra el uso del disco duro del sistema durante la ejecución de la simulación. Se debe destacar que el uso del disco variará fuertemente dependiendo de el escenario simulado, y de la configuración de OMNeT++. Esto ya que el simulador de redes por defecto periódicamente almacena estadísticas del funcionamiento de la simulación en archivos en disco, lo cual evidentemente genera un impacto sobre la eficiencia del sistema. De ser necesario, es posible desactivar esta funcionalidad para obtener un mejor rendimiento. De todas maneras, la figura 5.5 demuestra que a pesar de esto, los accesos al disco del *framework* son muy razonables y no generan un mayor impacto en el rendimiento de éste.

5.2.3. Conclusiones

De los experimentos realizados para la medición de la eficiencia computacional del *framework* desarrollado, se puede concluir que este alcanza el grado de eficiencia deseado para un software destinado al uso en la investigación de modelos complejos de sistemas de transporte. Es capaz de simular eficientemente sistemas con 1400 nodos presentes en cada de simulación, manteniendo un uso de procesador razonable y una huella en memoria casi despreciable.

Además, destaca que la mayor parte del *overhead* evidenciado en las simulaciones proviene de fuentes externas al *plugin* desarrollado; en particular, una gran parte del retraso proviene de la simulación de comunicación inalámbrica de OMNeT++, y de ser necesario esto puede de ser evitado mediante la omisión de la integración con VEINS, realizando la simulación directamente con, por ejemplo, Python.

5.3. Modelo Vehicular

5.3.1. Mediciones Realizadas

Esta categoría de experimentos y mediciones pretende verificar el correcto funcionamiento del *framework* para la simulación de Sistemas Inteligentes de Transporte, utilizando el escenario de simulación descrito al principio del presente capítulo, en la sección 5.1. Cabe destacar que en ningún caso se pretende argumentar que el escenario en cuestión es óptimo, ni que los parámetros del sistema de transporte son los correctos para este escenario, sino simplemente que PVEINS permite de manera precisa y confiable comparar y medir las ventajas que otorga un sistema de transporte dotado de comunicación intervehicular.

Para este fin se realizaron 6 *runs* del escenario en cuestión, la mitad sin comunicación alguna entre vehículos y la otra mitad con comunicación perfecta¹, para tres factores de demanda distintos. Estos *runs* se contrastaron principalmente en términos de la cantidad de vehículos que alcanzaron su destino dentro de los 15 minutos de simulación, factor que de manera intuitiva permite evaluar el desempeño del sistema. Para *runs* con el mismo factor de demanda, una cantidad menor de vehículos que alcanzan su destino en un lapso dado indica una menor eficiencia del sistema y un mayor retardo en los viajes realizados dentro del escenario. De esta manera, se pretende mostrar que el “accidente” modelado causa una cierta congestión en el sistema de transporte, y que, utilizando la comunicación intervehicular, es posible disminuir dicho impacto sobre la red.

Además, se realizaron dos análisis un poco más avanzados para los casos con factor de demanda 100 %; uno contrastando distancia y tiempos de recorrido en ambas simulaciones (con y sin comunicación), y un segundo comparando la emisión de dióxido de carbono en ambos casos.

Todos los valores analizados en estos experimentos fueron obtenidos desde OMNeT++ – el simulador de redes de comunicación a su vez los obtiene desde el simulador de transporte o los calcula en base a datos proporcionados por este último.

5.3.2. Resultados

Los resultados del conjunto de 6 *runs* con distintos factores de demanda puede observarse en la figura 5.6, representados en un diagrama de barras. Puede observarse que aquellos *runs* dotados de capacidades de comunicación intervehicular constantemente presentan una mayor cantidad de vehículos que alcanzan su destino final en el escenario dentro del tiempo de simulación. Si bien esta mejora en términos porcentuales es de apenas un 2.33 % en promedio, en la realidad se traduce a una considerable cantidad de vehículos que en el escenario *sin comunicación* se ven atascados en una congestión que no les permite alcanzar su destino, pero que en el escenario *con comunicación* logran evitar esta situación.

¹Es decir, con un factor de pérdida de paquetes en el medio de transmisión de 0 %

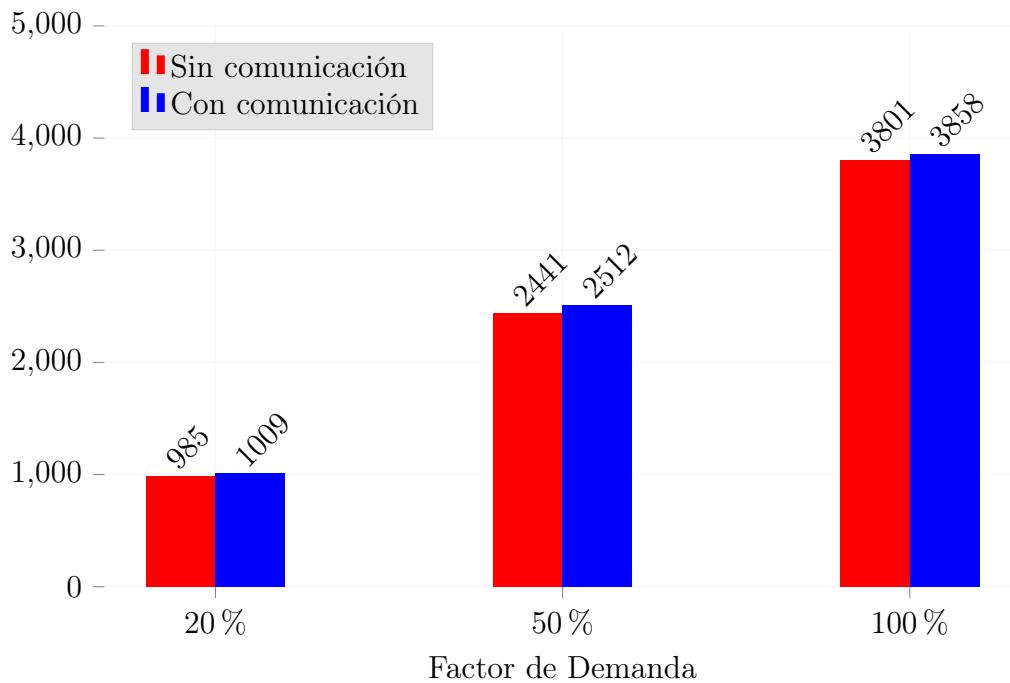


Figura 5.6: Comparación cantidad de vehículos que alcanzaron su destino en 15 minutos de tiempo simulado con tres factores de demanda distintos.

Duración	Alcanzaron destino con comunicación	Alcanzaron destino sin comunicación	△	% de mejora
15 min		3858	3801	57
120 min	11488	11014	474	4.3 %

Tabla 5.4: Comparación cantidad de vehículos que alcanzaron su destino con y sin comunicación intervehicular, para simulaciones de 15 minutos y 2 horas, con factor de demanda 100 %.

Con el fin de ilustrar de mejor manera el impacto de la comunicación en la cantidad de vehículos que logran evitar congestión y trasladarse exitosamente a su destino, se realizaron dos simulaciones adicionales al conjunto mencionado anteriormente. Esta simulaciones fueron de una duración extendida de dos horas de tiempo simulado, y los resultados de éstas se pueden visualizar en la tabla 5.4, comparados con los resultados obtenidos de la simulación de 15 minutos con factor de demanda 100 %. Se puede evidenciar que la diferencia estudiada se acentúa con el tiempo – mientras que para el escenario de 15 minutos la diferencia entre la cantidad de vehículos que alcanzan su destino para ambas configuraciones es de “apenas” 57, cuando el escenario se extiende a dos horas, esta diferencia alcanza casi los 500 vehículos. Esto indica que la comunicación tiene un efecto no-despreciable en el flujo vehicular.

Por otro lado, los análisis de eficiencia del sistema de transporte en términos de distancia total y emisión total de dióxido de carbono pueden estudiarse a continuación, en las figuras 5.7 y 5.8. Estas figuras corresponden a gráficos de dispersión para la comparación de variables claves para los escenarios *con* y *sin* comunicación, y cada punto en el gráfico representa un vehículo en la simulación.

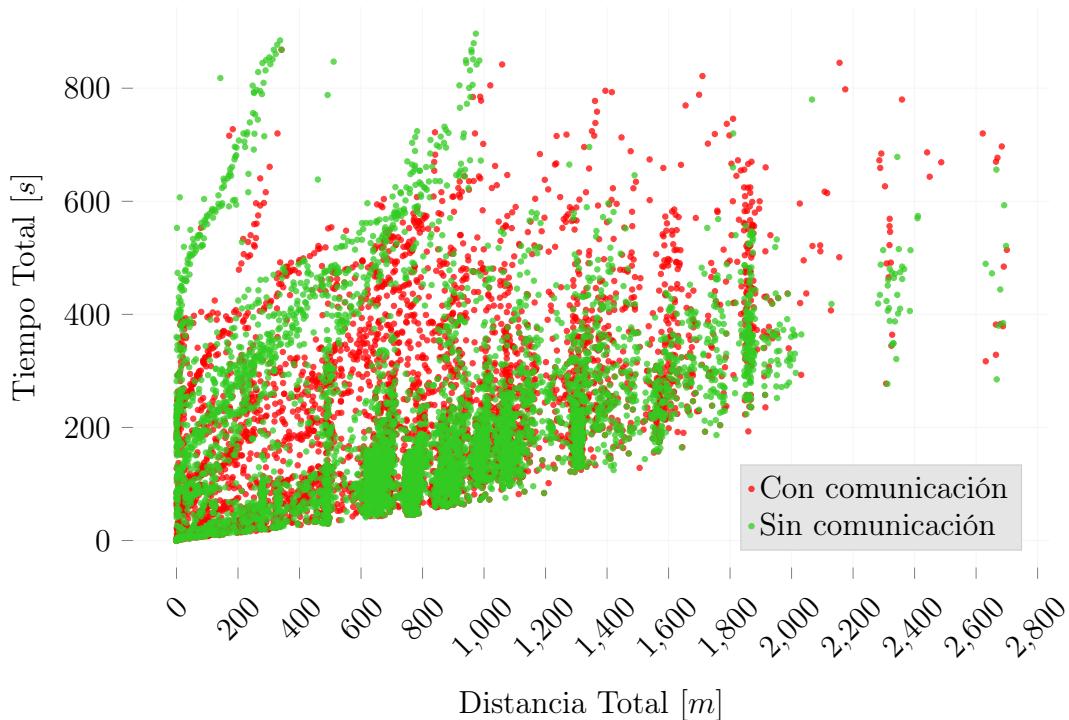


Figura 5.7: Gráfico de dispersión, distancia total vs. tiempo total en la simulación, para escenarios con factor de demanda 100 %, 15 minutos de tiempo simulado.

El primero de éstos, el gráfico 5.7, ilustra la relación entre distancia total recorrida y tiempo total de viaje para cada uno de los vehículos presentes en las simulaciones de 15 minutos de duración con un factor de demanda del 100 %. Puede notarse que si bien ambos *runs* por lo general presentan un comportamiento similar, en el *run* sin comunicación existen dos grupos con tendencia a presentar un mayor tiempo de viaje pero con distancias totales recorridas más bajas de lo esperado. En particular, es muy clara la presencia de un grupo de vehículos que están presentes en prácticamente la duración total de la simulación pero que sin embargo recorren distancias menores a 300 metros. Por otro lado, si bien el escenario con comunicación perfecta presenta una mayor dispersión de sus tiempos totales, no presenta mayores tendencias a largos tiempos de viaje asociados a distancias cortas.

La interpretación de estos resultados es directa; aquellas “ramas” del escenario sin comunicación que tienden hacia tiempos mayores para distancias comparativamente más cortas, corresponden a aquellos vehículos que se ven atascados en la congestión de tráfico generada por el “accidente”, y que por ende pasan mucho tiempo detenidos o con velocidades muy bajas. Por otro lado, la mayor dispersión de los puntos asociados al escenario con comunicación perfecta se asocia al hecho de que se “redirige” a todo vehículo que se estima potencialmente pudiese pasar por la calle afectada – esto aumenta la distancia y tiempo de viaje para mucho de ellos. Estos es un *tradeoff* para asegurar un mayor flujo vehicular.

El gráfico 5.8 presenta la relación entre la distancia total recorrida vs. la cantidad total de dióxido de carbono emitido por cada vehículo en los escenarios. Este gráfico presenta resultados muy similares al anterior; existen dos grupos de vehículos, en el caso sin comunicación, que tienden a exhibir mayores cantidades totales de emisiones para distancias comparativa-

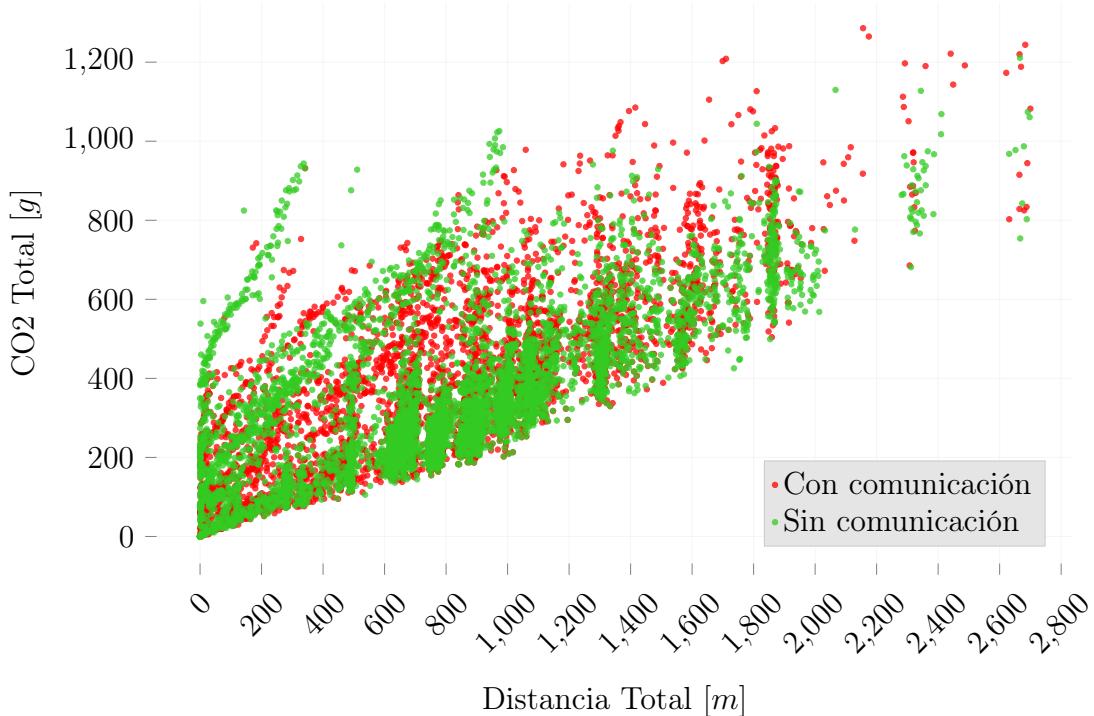


Figura 5.8: Gráfico de dispersión, distancia total vs. gramos totales de CO² emitidos en la simulación, para escenarios con factor de demanda 100 %, 15 minutos de tiempo simulado.

mente más cortas, mientras que el *run* con comunicación perfecta presenta una dispersión un poco mayor pero sin tendencia a extremos.

Nuevamente se asocian los extremos en el *run* sin comunicación a aquellos vehículos que se ven atrapados en el atoamiento producido por el accidente. Estos vehículos se encontraron la mayor del tiempo desplazándose a muy bajas velocidades o detenidos con el motor encendido, lo cual causa mayores emisiones de dióxido de carbono a la atmósfera. Por otro lado, en el *run* con comunicación perfecta, nuevamente se debe considerar que muchos vehículos son redirigidos y deben extender su trayectoria, lo cual genera una mayor dispersión en los valores obtenidos

5.3.3. Conclusiones

El análisis del modelo vehicular arrojó los resultados esperados, y se evidencia claramente el efecto mitigante en la congestión vehicular producto de un accidente que puede tener la comunicación intervehicular en un sistema de transporte inteligente.

En particular, los experimentos demostraron que PVEINS es capaz de simular escenarios viales realistas, y que permite analizar los resultados de manera correcta y precisa. La integración permite la obtención de datos claves del sistema, como la distancia total recorrida por cada vehículo, el tiempo que permaneció en el escenario y el total de su emisión de dióxido de carbono.

Capítulo 6

Conclusiones

6.1. Conclusiones generales

Las conclusiones y resultados obtenidos al final del presente trabajo de memoria son altamente positivos. Cumple a cabalidad con lo propuesto al principio de este documento – ver sección 6.2 para un desglose de cumplimiento de objetivos. El *software* desarrollado es eficiente, y logra su propósito completamente – se demostró que se puede utilizar para simular no solo sistemas de transportes pequeños y de poco flujo vehicular, sino que también es una herramienta potente para el estudio de sistemas de gran complejidad.

En términos de desarrollo e implementación, el trabajo conllevó la adquisición de experiencia en trabajar con software propietario, el cual no siempre cuenta con documentación completa y actualizada.

Si bien el API de Paramics era lo suficientemente potente y flexible como para realizar la propuesta de memoria, su documentación en muchas ocasiones no incluía la información necesaria, y en reiterados casos el memorista se vio obligado a reimplementar funcionalidades al descubrir – por accidente – maneras más eficientes de realizar la acción en cuestión que no se encontraban documentadas. Así mismo, ocurrió también lo inverso. Existieron situaciones en que un método de la API de Paramics parecía ser perfecto para la implementación de alguna funcionalidad, y al utilizarlo se descubría que tenía un desperfecto o que simplemente no funcionaba. El mejor ejemplo de ésto fue la función de avance de simulación de Paramics, que en ningún lugar en la documentación se advertía no funcionaba de manera correcta con *threads* paralelos. Al descubrirse, esto implicó, como se detalló en la sección 4.1.1, un rediseño total de la arquitectura del *framework*. De todas maneras, se logró sobrelevar estas dificultades y llevar a cabo el proyecto en su totalidad.

En términos personales, el desarrollo del trabajo presentó grandes oportunidades de aprendizaje, y el acercamiento a un área de investigación un poco alejada de lo común en ciencias de la computación. Implicó la familiarización con temas relacionados con el modelamiento de flujo vehicular, con sistemas de transporte y temas relacionados con las tecnologías de comunicaciones, y los estándares en uso hoy en día. Permitió al memorista además ponerse

en contacto con investigadores de nivel mundial como los doctores Falko Dressler y Christoph Sommer, autores de [7] y [8] y reconocidos investigadores en el área de comunicaciones intervehiculares. De hecho, cabe destacar que quien originalmente propuso la idea de resolver la problemática original mediante la adaptación de VEINS fue el mismo profesor Dressler en una visita a la Universidad de Chile.

6.2. Cumplimiento de objetivos

El objetivo general del presente trabajo de memoria, “**el desarrollo de un framework de integración entre un simulador de redes, OMNeT++ y un microsimulador de tráfico, Quadstone Paramics, de tal manera que exista comunicación bidireccional entre ambos**”, fue logrado en su totalidad. PVEINS, el *framework* desarrollado, permite la integración totalmente transparente de Paramics con el *framework* VEINS, desarrollado por Sommer *et al.* en [7], y OMNeT++, posibilitando así la simulación de Sistemas de Transporte Inteligentes complejos, además del uso de la gran cantidad de modelos de comunicación ya desarrollados para dicho simulador de redes de comunicaciones. Se logró esta integración mediante la implementación de un *plugin* para Paramics, el cual actúa como una interfaz entre el simulador y el protocolo TraCI, permitiendo así el intercambio estandarizado de información y comandos con OMNeT++ y VEINS a través de un *socket* TCP.

Se demostró además que esta implementación es altamente eficiente en términos de la relación entre tiempo de duración real y tiempo simulado para simulaciones de gran tamaño, del orden de cientos o hasta miles de nodos presentes simultáneamente en la red. La implementación es también austera en recursos de sistema, utilizando menos de 600 MB de memoria y aumentando la actividad del procesador en un 20 % para una simulación con un promedio de 1400 nodos activos en cualquier instante dado.

En términos de los objetivos particulares enumerados en la sección 3.3.2, se puede concluir que:

1. **Se estableció el estado del arte en cuanto a simulación bidireccional en comunicaciones inalámbricas y sistemas de transporte**, tomando en cuenta herramientas tanto de código abierto como cerrado. Esto se expuso de manera extensa y profunda en la sección 2.2.
2. **Se escogió la opción de adaptar el framework VEINS como solución al problema presentado**, dado que correspondía a la opción más madura y flexible en términos de simulación bidireccional. El razonamiento completo detrás de esta decisión se encuentra en la sección 3.2.
3. **Se adaptó Paramics**, mediante la implementación de un *plugin*, **para que pudiese comunicarse de manera bidireccional con OMNeT++**. Se implementaron además todas las funcionalidades enumeradas en la sección 4.2, las cuales incluyen:
 - i. Construcción de la topología del modelo de comunicaciones a partir de la topología del modelo de tráfico.

- ii. Actualización dinámica de los nodos en OMNeT++, siguiendo los movimientos de los elementos de Paramics.
 - iii. Modificación del comportamiento de los nodos del modelo de transporte a partir de eventos en OMNeT++.
4. **Se implementó el framework PVEINS siguiendo patrones y buenas prácticas de ingeniería de software** – el detalle de esto se encuentra en el capítulo 4.
 5. **Se probó y validó el funcionamiento de la integración de los simuladores** mediante la simulación de un modelo de transporte simple pero dinámico (ver sección 4.4).
 6. Finalmente, **se implementó un modelo avanzado de transporte**, y se utilizó para **la evaluación del rendimiento del software implementado y su aplicabilidad a modelos realistas de Sistemas de Transporte Inteligentes**. El detalle del modelo desarrollado y los resultados obtenidos se expusieron en el capítulo 5.

6.3. Trabajo futuro

El *framework* desarrollado cumple con todos los objetivos que se expusieron al principio del presente documento, y se evalúa como altamente aplicable para el estudio de modelos de Sistemas Inteligentes de Transporte; sin embargo, aún así queda trabajo futuro por desarrollar.

Lo más evidente, en primer lugar, es la implementación de los comandos TraCI restantes, los cuales no fueron considerados dentro del alcance de la presente memoria. Estos comandos incluyen diversas acciones no esencialmente necesarias para la simulación de ITS, pero que en un futuro pudiesen llegar a ser requeridos. Comandos como para, por ejemplo, modificar el comportamiento de peatones, semáforos y hasta detectores de bucle de inducción, se encuentran definidos en el protocolo y, de llegar a necesitarse, deberán ser implementados en el *framework*. Se debe también tener en consideración la constante evolución de TraCI – el protocolo se actualiza periódicamente en conjunto con VEINS, y cada nueva versión puede introducir comandos nuevos o deprecar o modificar totalmente comandos antiguos (reemplazándolos con nuevas versiones). Será necesario entonces mantener actualizado PVEINS con estos cambios constantes si se desea mantener compatibilidad con VEINS en el futuro. De esta misma manera, se deberá actualizar el *plugin* de Paramics en el caso de actualizaciones que rompan compatibilidad con el API de extensión actual.

En segundo lugar, se plantea la necesidad a futuro de implementar el inicio automático de la simulación de Paramics al recibir una conexión entrante por el *socket* TCP. Esta funcionalidad estaba presente en la primera versión de la arquitectura del *software*, sin embargo fue necesario descartarla al realizar las modificaciones necesarias para el correcto funcionamiento *single-thread* del *plugin* (ver sección 4.1.1). Contar con esta funcionalidad significaría poder ejecutar *batches* (conjuntos) de *runs* de simulaciones sin ninguna intervención del usuario – permitiendo así realizar análisis mucho más acabados de manera más simple.

El *plugin* todavía admite optimización, a pesar de que se trató de implementar de la manera más óptima posible y se realizaron revisiones y reimplementaciones de funcionalidades al descubrir nuevas maneras de optimizar el funcionamiento de éstas. Por ejemplo, la funcionalidad de cambio de pista utiliza una colección aparte para almacenar los comandos de cambios recibidos, sobre la cual se itera luego de cada paso de simulación. Esto significa que Paramics, aparte de actualizar el estado de los N vehículos presentes en la simulación en cualquier instante de acuerdo a su modelo interno, deberá recorrer un máximo de otros N elementos adicionales (máximo un cambio de pista por vehículo) y ejecutar los cambios correspondientes después de cada paso de simulación. De lograrse implementar entonces una versión del cambio de pista similar a las implementaciones de cambio de ruta y velocidad que se llevaron a cabo en el *framework*, es decir, utilizando el mismo modelo de Paramics para realizar los cambios, podría mejorarse el rendimiento, tal vez de manera considerable para escenarios con una gran cantidad de cambios de pista.

También existen leves ineficiencias en los procesos de recepción e interpretación de mensajes TraCI, las cuales fueron introducidas intencionalmente para aumentar la modularidad y extensibilidad del código. Decisiones de diseño como la encapsulación de cada comando en un objeto `tcpip::Storage` aparte agregan *overhead* al *software* a cambio de código más legible y flexible, factores que en el futuro pueden ser menos importantes que el rendimiento del *framework*. En ese caso, será necesaria una completa refactorización de la estructura interna de paso de parámetros del código, ya esto fue una decisión de diseño fundamental en la implementación actual.

Finalmente, el *framework* podría ser evaluado y analizado por alguien con más experiencia en el área de transporte. En específico, sería valioso un estudio de la validez de los resultados de PVEINS con un escenario que implemente un modelo de re-enrutamiento más avanzado y realista que el empleado para el escenario de validación del presente trabajo de memoria.

Bibliografía

- [1] *Directive 2010/40/EU of the European Parliament and of the Council on the framework for the deployment of Intelligent Transport Systems in the field of road transport and for interfaces with other modes of transport*, 2010 O.J. L 207/1, European Parliament, 2010.
- [2] D. J. Dailey, K. McFarland y J. L. Garrison, «Experimental study of 802.11 based networking for vehicular management and safety», en *2010 IEEE Intelligent Vehicles Symposium*, jun. de 2010, págs. 1209-1213. DOI: [10.1109/IVS.2010.5547955](https://doi.org/10.1109/IVS.2010.5547955).
- [3] W. Xiong, X. Hu y T. Jiang, «Measurement and Characterization of Link Quality for IEEE 802.15.4-Compliant Wireless Sensor Networks in Vehicular Communications», *IEEE Transactions on Industrial Informatics*, vol. 12, n.º 5, págs. 1702-1713, oct. de 2016, ISSN: 1551-3203. DOI: [10.1109/TII.2015.2499121](https://doi.org/10.1109/TII.2015.2499121).
- [4] D. Jiang y L. Delgrossi, «IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments», en *VTC Spring 2008 - IEEE Vehicular Technology Conference*, mayo de 2008, págs. 2036-2040. DOI: [10.1109/VETECS.2008.458](https://doi.org/10.1109/VETECS.2008.458).
- [5] C. Sommer, Z. Yao, R. German y F. Dressler, «On the need for bidirectional coupling of road traffic microsimulation and network simulation», en *Proceedings of the 1st ACM SIGMOBILE workshop on Mobility models*, ACM, 2008, págs. 41-48.
- [6] ——, «On the Need for Bidirectional Coupling of Road Traffic Microsimulation and Network Simulation», en *Proceedings of the 1st ACM SIGMOBILE Workshop on Mobility Models*, ép. MobilityModels '08, Hong Kong, Hong Kong, China: ACM, 2008, págs. 41-48, ISBN: 978-1-60558-111-8. DOI: [10.1145/1374688.1374697](https://doi.org/10.1145/1374688.1374697). dirección: <http://doi.acm.org/10.1145/1374688.1374697>.
- [7] 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](https://doi.org/10.1109/TMC.2010.133).
- [8] 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](https://doi.org/10.1109/MCOM.2008.4689256).
- [9] V. Zúñiga Alarcón, «Uso de Herramientas de Microsimulación para la Definición de Estrategias de Control de Tránsito para la Ciudad de Santiago», memoria de Ing. Civil, Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, 2010. dirección: <http://www.repositorio.uchile.cl/handle/2250/103923>.
- [10] (Jun. de 2017). Repositorio PVeins en GitHub, dirección: https://github.com/molguin92/paramics_traci.

- [11] (Jun. de 2017). BSD 3-Clause License, dirección: <https://opensource.org/licenses/BSD-3-Clause>.
- [12] E. Cascetta, *Transportation systems engineering: theory and methods*. Springer Science & Business Media, 2013, vol. 49.
- [13] (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/>.
- [14] 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.
- [15] 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>.
- [16] Y. Shalaby, «An Integrated Framework for Coupling Traffic and Wireless Network Simulations», Tesis de mtría., Department of Civil Engineering, University of Toronto, Canada, 2010.
- [17] 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.
- [18] 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.
- [19] 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](https://doi.org/10.1109/OSSCOM.2015.7372687).
- [20] (Mayo de 2017). PVT VISSIM, dirección: <http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/>.
- [21] M. Fellendorf y P. Vortisch, «Microscopic traffic flow simulator VISSIM», en *Fundamentals of traffic simulation*, Springer, 2010, págs. 63-93.
- [22] (Mayo de 2017). Aimsun traffic modelling software, dirección: <https://www.aimsun.com/aimsun/>.
- [23] S. L. Jones, A. J. Sullivan, N. Cheekoti, M. D. Anderson y D. Malave, «Traffic simulation software comparison study», *UTCA Report*, vol. 2217, 2004.
- [24] (Mayo de 2017). TSIS-CORSIM™, dirección: <https://mctrans.ce.ufl.edu/mct/index.php/tsis-corsim/>.
- [25] M. Behrisch, L. Bieker, J. Erdmann y D. Krajzewicz, «SUMO - Simulation of Urban MObility: An Overview», en *SIMUL 2011*, S. Ū. 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/>.
- [26] (Mayo de 2017). Deutsches Zentrum für Luft- und Raumfahrt (DLR), dirección: http://www.dlr.de/ts/en/desktopdefault.aspx/tabcid-1221/1665_read-3070/.
- [27] D. Krajzewicz, «Summary on Publications citing SUMO, 2002-2012», en *1st SUMO User Conference-SUMO 2013*, DLR, vol. 21, 2013, págs. 11-24.

- [28] 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](https://doi.org/10.1109/MCOM.2012.6316780).
- [29] (Mar. de 2017). Quadstone Paramics website, dirección: <http://www.paramics-online.com>.
- [30] 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](https://doi.org/10.1109/ICCS.2012.65).
- [31] 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](https://doi.org/10.1109/ICC.2009.5198657).
- [32] 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](https://doi.org/10.1109/ICCSCE.2012.6487111).
- [33] 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](https://doi.org/10.1109/PADS.1998.685281).
- [34] 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](https://doi.org/10.1109/2.722293).
- [35] 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.
- [36] 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](https://doi.org/10.1109/WICOM.2007.383).
- [37] 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](https://doi.org/10.1109/TVT.2007.907273).
- [38] 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](https://doi.org/10.1109/ICC.2016.7511126).
- [39] 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](https://doi.org/10.1109/VETECF.2007.437).
- [40] 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](https://doi.org/10.1109/VETECS.2010.5494212).
- [41] 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 SIG-*

MOBILE mobile computing and communications review, vol. 12, n.º 1, págs. 31-33, 2008.

- [42] 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.
- [43] 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.
- [44] 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](https://doi.acm.org/10.1145/1400713.1400740). dirección: <http://doi.acm.org/10.1145/1400713.1400740>.
- [45] (Mayo de 2017). Qt | Cross-platform software development for embedded & desktop, dirección: <https://www.qt.io/>.
- [46] (Jun. de 2017). Git, a FOSS distributed version control system, dirección: <https://git-scm.com/>.
- [47] (Jun. de 2017). GitHub development platform, dirección: <https://github.com/>.
- [48] (Jun. de 2017). Perfil personal del autor en GitHub, dirección: <https://github.com/molguin92>.
- [49] 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.
- [50] (Mayo de 2017). Python TraCI Library Documentation, dirección: <http://www.sumo.dlr.de/pydoc/traci.html>.
- [51] (Mayo de 2017). Código fuente cliente TraCI Python, GitHub, dirección: <https://github.com/planetsumo/sumo/tree/master/sumo/tools/traci>.
- [52] (Jun. de 2017). Pandas: Python Data Analysis Library, dirección: <http://pandas.pydata.org/>.
- [53] (Jun. de 2017). Numpy, the fundamental package for scientific computing with Python, dirección: <http://www.numpy.org/>.
- [54] (Jun. de 2017). Matplotlib, Python plotting, dirección: <https://matplotlib.org/>.
- [55] (Jun. de 2017). matplotlib2tikz, a Python tool for converting matplotlib figures into PGFPlots (TikZ) figures, dirección: <https://github.com/nschloe/matplotlib2tikz>.
- [56] (Jun. de 2017). Steam Hardware Survey, dirección: <http://store.steampowered.com/hwsurvey/>.
- [57] (Jun. de 2017). Unity Hardware Stats, dirección: <https://hwstats.unity3d.com/>.

Apéndice A

TraCI

TraCI (**T**raffic **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 [44]. 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**.

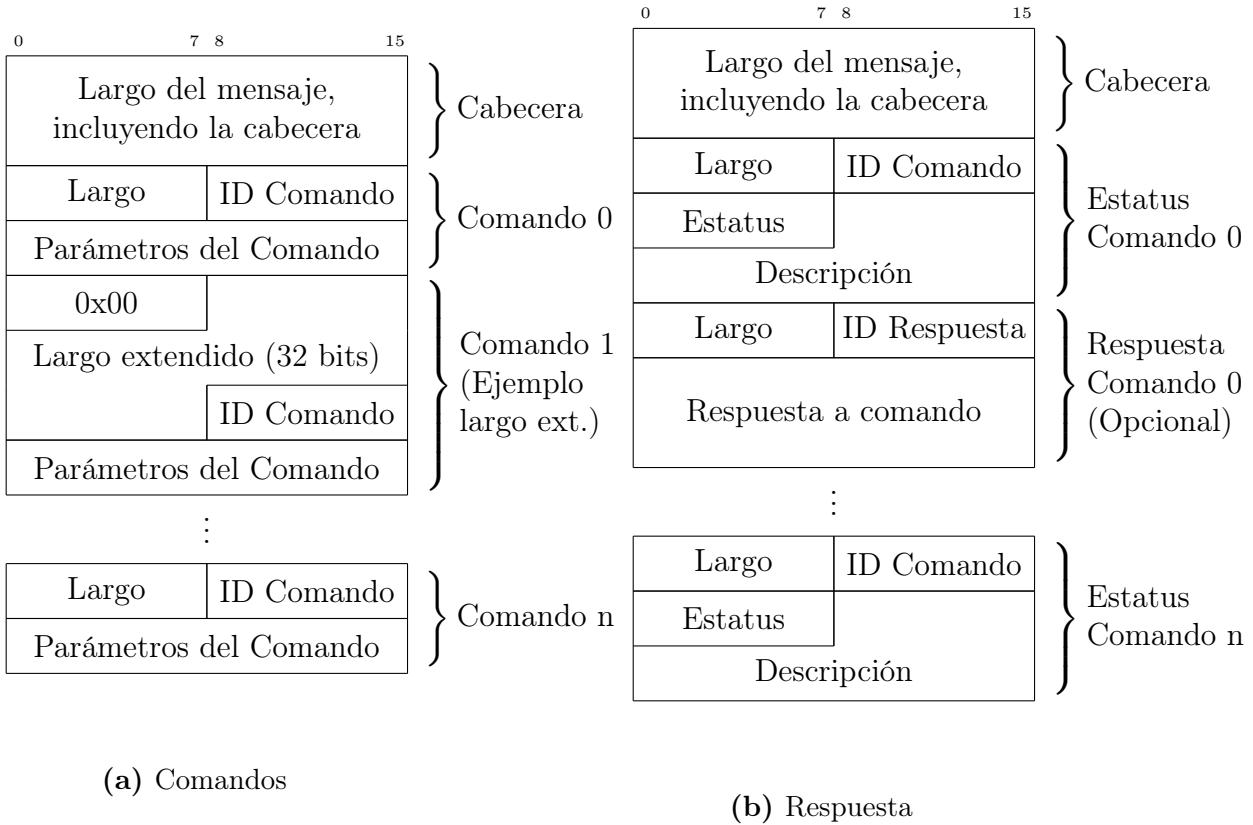


Figura A.1: Formatos de mensajes TraCI

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.

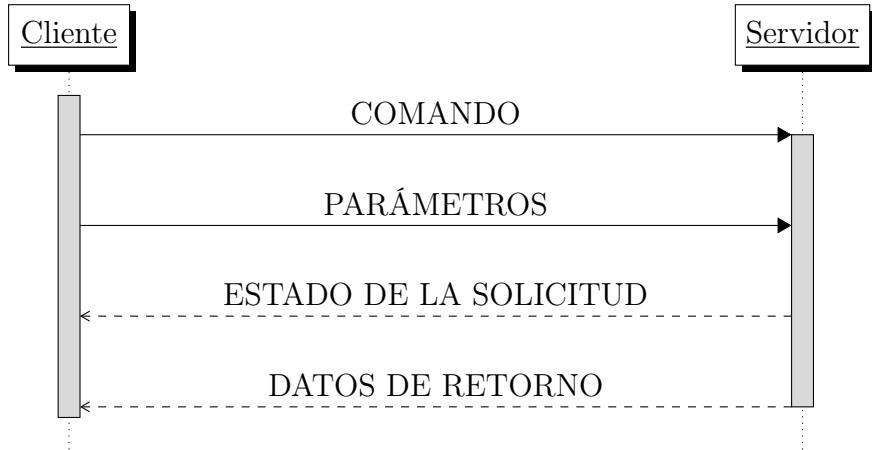


Figura A.2: Ejemplo solicitud de variable TraCI.

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

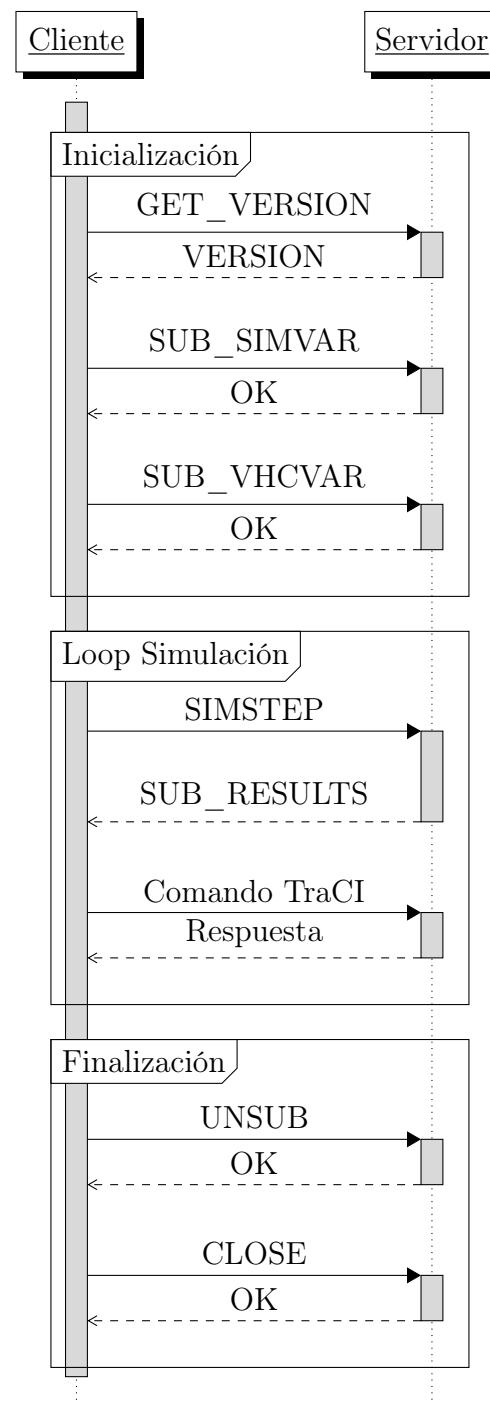


Figura A.3: Flujo de comunicación TraCI.

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 sobreescritas) 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 sobreescribibles por el usuau-

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 sobreescrita 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 e 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 sobre escriben 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.

```
1 #include "programmer.h"
2 #include <thread>
3 #include "TraCI API/TraCIServer.h"
4 #include <shellapi.h>
5 #include "TraCI API/VehicleManager.h"
6 #include "TraCI API/Utils.h"
7 #include "TraCI API/Simulation.h"

8
9 #define DEFAULT_PORT 5000
10#define CMDARG_PORT "--traci_port="

11
12 std::thread* runner;
13 traci_api::TraCIServer* server;

14
15 /* checks a string for a matching prefix */
16 bool starts_with(std::string const& in_string,
17                   std::string const& prefix)
18 {
19     return prefix.length() <= in_string.length() &&
20           std::equal(prefix.begin(), prefix.end(), in_string.begin());
21 }

22
23 void runner_fn()
24 {
25     try {
26         //try to get port from command line arguments
27         int argc;
28         LPWSTR* argv = CommandLineToArgvW(GetCommandLineW(), &argc);
29         std::string prefix(CMDARG_PORT);

30
31         int port = DEFAULT_PORT; // if it fails, use the default port
32         for (int i = 0; i < argc; i++)
```

```

33     {
34         // convert from widestring to normal string
35         std::wstring temp(argv[i]);
36         std::string str(temp.begin(), temp.end());
37
38         // check if argument prefix matches
39         if (starts_with(str, prefix))
40         {
41             std::string s_port = str.substr(prefix.length(),
42                                             strnpos);
43             try
44             {
45                 port = std::stoi(s_port);
46             }
47             catch (...)
48             {
49                 traci_api::infoPrint("Invalid port identifier - "
50                                     "Falling back to default port");
51                 port = DEFAULT_PORT;
52             }
53         }
54
55         server = new traci_api::TraCIServer(port);
56         server->waitForConnection();
57     }
58     catch (std::exception& e)
59     {
60         traci_api::debugPrint("Uncaught while initializing server.");
61         traci_api::debugPrint(e.what());
62         traci_api::debugPrint("Exiting...");
63         throw;
64     }
65
66     // Called once after the network is loaded.
67     void qpx_NET_postOpen(void)
68     {
69         qps_GUI_singleStep(PFALSE);
70         std::string version_str = "Paramics TraCI plugin v" +
71             std::string(PVEINS_VERSION) + " on Paramics v" +
72             std::to_string(qpg_UTL_parentProductVersion());
73         traci_api::infoPrint(version_str);
74         traci_api::infoPrint(PVEINS_COPYRIGHT);
75         traci_api::infoPrint(PVEINS_LICENSE);
76         traci_api::infoPrint("----");
77         traci_api::infoPrint("Timestep size: " +
78             std::to_string(static_cast<int>(qpg_CFG_timeStep() * 1000.0f)) +
79             "ms");

```

```

76     traci_api::infoPrint("Simulation start time: " +
77         std::to_string(traci_api::Simulation::getInstance()
78                         ->getCurrentTimeMilliseconds()) + "ms");
79     runner = new std::thread(runner_fn);
80 }
81
82 void qpx_CLK_startOfSimLoop(void)
83 {
84     if (runner->joinable())
85         runner->join();
86
87     server->preStep();
88 }
89
90 void qpx_CLK_endOfSimLoop(void)
91 {
92     server->postStep();
93 }
94
95 void close()
96 {
97     server->close();
98     delete server;
99     delete runner;
100 }
101
102 void qpx_NET_complete(void)
103 {
104     close();
105 }
106
107 void qpx_NET_close()
108 {
109     close();
110 }
111
112 void qpx_VHC_release(VEHICLE* vehicle)
113 {
114     traci_api::VehicleManager::getInstance()->vehicleDepart(vehicle);
115 }
116
117 void qpx_VHC_arrive(VEHICLE* vehicle, LINK* link, ZONE* zone)
118 {
119     traci_api::VehicleManager::getInstance()->vehicleArrive(vehicle);
120 }
121
122 // routing through TraCI
123 Bool qpo_RTM_enable(void)
124 {

```

```

125     return PTRUE;
126 }
127
128 int qpo_RTM_decision(LINK *linkp, VEHICLE *Vp)
129 {
130     return
131         traci_api::VehicleManager::getInstance()->rerouteVehicle(Vp,
132         linkp);
133
134 void qpx_VHC_timeStep(VEHICLE* vehicle)
135 {
136     //traci_api::VehicleManager::getInstance()->routeReEval(vehicle);
137
138 void qpx_VHC_transfer(VEHICLE* vehicle, LINK* link1, LINK* link2)
139 {
140     traci_api::VehicleManager::getInstance()->routeReEval(vehicle);
141
142 // speed control override
143 float qpo_CFM_followSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])
144 {
145     float speed = 0;
146     if (traci_api::VehicleManager::getInstance()
147         ->speedControlOverride(v, speed))
148         return speed;
149     else
150         return qpg_CFM_followSpeed(link, v, ahead);
151
152
153 float qpo_CFM_leadSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])
154 {
155     float speed = 0;
156     if (traci_api::VehicleManager::getInstance()
157         ->speedControlOverride(v, speed))
158         return speed;
159     else
160         return qpg_CFM_leadSpeed(link, v, ahead);
161
162 }
```

Código C.2: Método preStep() en TraCIServer

```
1 void traci_api::TraCIServer::preStep()
2 {
3     std::lock_guard<std::mutex> lock(socket_lock);
4     if (multiple_timestep
5         && Simulation::getInstance()->getCurrentTimeMilliseconds() <
6             target_time)
7     {
8         VehicleManager::getInstance()->reset();
9         return;
10    }
11
12    multiple_timestep = false;
13    target_time = 0;
14
15    tcPIP::Storage cmdStore; // individual commands in the message
16
17    debugPrint("Waiting for incoming commands from the TraCI
18        client...");
19
20    // receive and parse messages until we get a simulation step
21    // command
22    while (running && ssocket.receiveExact(incoming))
23    {
24        incoming_size = incoming.size();
25
26        debugPrint("Got message of length " +
27            std::to_string(incoming_size));
28        //debugPrint("Incoming: " + incoming.hexDump());
29
30
31        /* Multiple commands may arrive at once in one message,
32         * divide them into multiple storages for easy handling */
33        while (incoming_size > 0 && incoming.valid_pos())
34        {
35            uint8_t cmdlen = incoming.readUnsignedByte();
36            cmdStore.writeUnsignedByte(cmdlen);
37
38            debugPrint("Got command of length " +
39                std::to_string(cmdlen));
40
41            for (uint8_t i = 0; i < cmdlen - 1; i++)
42                cmdStore.writeUnsignedByte(incoming
43                    .readUnsignedByte());
44
45            bool simstep = this->parseCommand(cmdStore);
46            cmdStore.reset();
47        }
48    }
49}
```

```
44 // if the received command was a simulation step command,
45 // return so that
46 // Paramics can do its thing.
47 if (simstep)
48 {
49     VehicleManager::getInstance()->reset();
50     return;
51 }
52
53 this->sendResponse();
54 incoming.reset();
55 outgoing.reset();
56 }
57 }
```

Código C.3: Método postStep() en TraCIServer

```
1 void traci_api::TraCIServer::postStep()
2 {
3     // after each step, have VehicleManager update its internal state
4     VehicleManager::getInstance()
5         ->handleDelayedTriggers();
6     if (multiple_timestep
7         && Simulation::getInstance()->getCurrentTimeMilliseconds() <
8             target_time)
9         return;
10    // after a finishing a simulation step command (completely),
11    // collect subscription results and
12    // check if there are commands remaining in the incoming storage
13    this->writeStatusResponse(CMD_SIMSTEP, STATUS_OK, "");
14    // handle subscriptions after simstep command
15    tcpip::Storage subscriptions;
16    this->processSubscriptions(subscriptions);
17    outgoing.writeStorage(subscriptions);
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        debugPrint("Got command of length " + std::to_string(cmdlen));
27
28        for (uint8_t i = 0; i < cmdlen - 1; i++)
29            cmdStore.writeUnsignedByte(incoming
30                .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() +
43                    sim->getTimeStepSizeMilliseconds();
44            }
45            VehicleManager::getInstance()->reset();
46            return;
47        }
48    }
49 }
```

Código C.4: Métodos base de todas las suscripciones.

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

```

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

```

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

Código C.5: Método de actualización y creación de suscripciones en **TraCIServer** en su totalidad.

```
1 void traci_api::TraCIServer::addSubscription(uint8_t sub_type,
2     std::string object_id, int start_time, int end_time,
3     std::vector<uint8_t> variables)
4 {
5     std::string errors;
6     tcpip::Storage temp;
7
8     // first check if this corresponds to an update for an existing
9     // subscription
10    for (auto it = subs.begin(); it != subs.end(); ++it)
11    {
12        uint8_t result = (*it)->updateSubscription(sub_type,
13            object_id, start_time, end_time, variables, temp, errors);
14
15        switch (result)
16        {
17            case VariableSubscription::STATUS_OK:
18                // update ok, return now
19                debugPrint("Updated subscription");
20                writeStatusResponse(sub_type, STATUS_OK, "");
21                writeToOutputWithSize(temp, true);
22                return;
23            case VariableSubscription::STATUS_UNSUB:
24                // unsubscribe command, remove the subscription
25                debugPrint("Unsubscribing...");
26                delete *it;
27                it = subs.erase(it);
28                // we don't care about the deleted iterator, since we
29                // return from the loop here
30                writeStatusResponse(sub_type, STATUS_OK, "");
31                return;
32            case VariableSubscription::STATUS_ERROR:
33                // error when updating
34                debugPrint("Error updating subscription.");
35                writeStatusResponse(sub_type, STATUS_ERROR, errors);
36                break;
37            case VariableSubscription::STATUS_NOUPD:
38                // no update, try next subscription
39                continue;
40            default:
41                throw std::runtime_error("Received unexpected result " +
42                    std::to_string(result) + " when trying to update
43                    subscription.");
44    }
45
46    // 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
42 // that reached its
43 // destination. Check number of variables and do nothing if it's
44 // 0.
45
46 if(variables.size() == 0)
47 {
48     // unsub command that didn't match any of the currently
49     // running subscriptions, so just
50     // tell the client it's ok, everything's alright
51
52     debugPrint("Unsub from subscription already removed.");
53     writeStatusResponse(sub_type, STATUS_OK, "");
54     return;
55 }
56
57 debugPrint("No update. Adding new subscription.");
58 VariableSubscription* sub;
59 switch (sub_type)
60 {
61     case CMD_SUB_VHCVAR:
62         debugPrint("Adding VHC subscription.");
63         sub = new VehicleVariableSubscription(object_id, start_time,
64                                             end_time, variables);
65         break;
66     case CMD_SUB_SIMVAR:
67         debugPrint("Adding SIM subscription.");
68         sub = new SimulationVariableSubscription(object_id,
69                                         start_time, end_time, variables);
70         break;
71     default:
72         writeStatusResponse(sub_type, STATUS_NIMPL, "Subscription type
73                             not implemented: " + std::to_string(sub_type));
74         return;
75 }
76
77 uint8_t result = sub->handleSubscription(temp, true, errors); // validate
78
79 if (result == VariableSubscription::STATUS_EXPIRED)
80 {
81     debugPrint("Expired subscription.");
82
83     writeStatusResponse(sub_type, STATUS_ERROR, "Expired
84                         subscription.");
85     return;
86 }
87 else if (result != VariableSubscription::STATUS_OK)
88 {
89
90 }
```

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

Código C.6: Obtención de variables en **Simulation**. **VehicleManager** y **Network** cuentan con métodos análogos a los presentados aquí, por lo que no se expondrán en este documento.

```
1  bool traci_api::Simulation::packSimulationVariable(uint8_t varID,
2          tcpip::Storage& result_store)
3  {
4      debugPrint("Fetching SIMVAR " + std::to_string(varID));
5
6      result_store.writeUnsignedByte(RES_GETSIMVAR);
7      result_store.writeUnsignedByte(varID);
8      result_store.writeString("");
9
10     try
11     {
12         getSimulationVariable(varID, result_store);
13     }
14     catch (...)
15     {
16         return false;
17     }
18     return true;
19 }
20
21 void traci_api::Simulation::getSimulationVariable(uint8_t varID,
22          tcpip::Storage& result)
23 {
24     VehicleManager* vhcm = traci_api::VehicleManager::getInstance();
25
26     switch (varID)
27     {
28     case VAR_SIMTIME:
29         result.writeUnsignedByte(VTYPE_INT);
30         result.writeInt(this->getCurrentTimeMilliseconds());
31         break;
32     case VAR_DEPARTEDVHC_CNT:
33         result.writeUnsignedByte(VTYPE_INT);
34         result.writeInt(vhcm->getDepartedVehicleCount());
35         break;
36     case VAR_DEPARTEDVHC_LST:
37         result.writeUnsignedByte(VTYPE_STRLST);
38         result.writeStringList(vhcm->getDepartedVehicles());
39         break;
40     case VAR_ARRIVEDVHC_CNT:
41         result.writeUnsignedByte(VTYPE_INT);
42         result.writeInt(vhcm->getArrivedVehicleCount());
43         break;
44     case VAR_ARRIVEDVHC_LST:
45         result.writeUnsignedByte(VTYPE_STRLST);
46         result.writeStringList(vhcm->getArrivedVehicles());
```

```

44     break;
45 case VAR_TIMESTEPSZ:
46     result.writeUnsignedByte(VTYPE_INT);
47     result.writeInt(getTimeStepSizeMilliseconds());
48     break;
49 case VAR_NETWORKBNDS:
50     result.writeUnsignedByte(VTYPE_BOUNDBOX);
51 {
52     double llx, lly, urx, ury;
53     this->getRealNetworkBounds(llx, lly, urx, ury);
54
55     result.writeDouble(llx);
56     result.writeDouble(lly);
57     result.writeDouble(urx);
58     result.writeDouble(ury);
59 }
60 break;
61 // we don't have teleporting vehicles in Paramics, nor parking
   (temporarily at least)
62 case VAR_VHCENDTELEPORT_CNT:
63 case VAR_VHSTARTTELEPORT_CNT:
64 case VAR_VHSTARTPARK_CNT:
65 case VAR_VHCENDPARK_CNT:
66     result.writeUnsignedByte(VTYPE_INT);
67     result.writeInt(0);
68     break;
69
70 case VAR_VHCENDTELEPORT_LST:
71 case VAR_VHSTARTTELEPORT_LST:
72 case VAR_VHSTARTPARK_LST:
73 case VAR_VHCENDPARK_LST:
74     result.writeUnsignedByte(VTYPE_STRLST);
75     result.writeStringList(std::vector<std::string>());
76     break;
77 default:
78     throw std::runtime_error("Unimplemented variable " +
   std::to_string(varID));
79 }
80 }
```

Código C.7: Obtención de los límites del escenario de transporte.

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

```

44 {
45     // links are always connected to zones or nodes, so we only
46     // need
47     // to get position data from those that are curved
48
49     LINK* lnk = qpg_NET_linkByIndex(i);
50     if (!qpg_LNK_arc(lnk) && !qpg_LNK_arcLeft(lnk))
51         continue;
52
53     // arc are perfect sections of circles, thus we only need the
54     // start, end and middle point (for all lanes)
55     float len = qpg_LNK_length(lnk);
56     int lanes = qpg_LNK_lanes(lnk);
57
58     float g, b;
59
60     for (int j = 1; j <= lanes; j++)
61     {
62         // start points
63         qpg_POS_link(lnk, j, 0, &tempX, &tempY, &tempZ, &b, &g);
64
65         x.push_back(tempX);
66         y.push_back(tempY);
67
68         // middle points
69         qpg_POS_link(lnk, j, len / 2.0, &tempX, &tempY, &tempZ, &b,
70             &g);
71
72         x.push_back(tempX);
73         y.push_back(tempY);
74
75         // end points
76         qpg_POS_link(lnk, j, len, &tempX, &tempY, &tempZ, &b, &g);
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 }

```

Código C.8: Implementación de los controladores de velocidad.

```
1 // Triggers.h
2 namespace traci_api
3 {
4     class BaseSpeedController
5     {
6         public:
7             virtual ~BaseSpeedController()
8             {
9             }
10            virtual float nextTimeStep() = 0;
11            virtual bool repeat() = 0;
12    };
13
14    class HoldSpeedController : public BaseSpeedController
15    {
16        private:
17            VEHICLE* vhc;
18            float target_speed;
19
20        public:
21            HoldSpeedController(VEHICLE* vhc, float target_speed) :
22                vhc(vhc), target_speed(target_speed){}
23            ~HoldSpeedController() override {}
24
25            float nextTimeStep() override;
26            bool repeat() override { return true; }
27    };
28
29    class LinearSpeedChangeController : public BaseSpeedController
30    {
31        private:
32            VEHICLE* vhc;
33            int duration;
34            bool done;
35
36            float acceleration;
37
38        public:
39            LinearSpeedChangeController(VEHICLE* vhc, float target_speed,
40                int duration);
41            ~LinearSpeedChangeController() override {};
42
43            float nextTimeStep() override;
44            bool repeat() override { return !done; }
45    };
46 } // Triggers.cpp
```

```

47 float traci_api::HoldSpeedController::nextTimeStep()
48 {
49     float current_speed = qpg_VHC_speed(vhc);
50     float diff = target_speed - current_speed;
51     if (abs(diff) < NUMERICAL_EPS)
52     {
53         if (target_speed < NUMERICAL_EPS && !qpg_VHC_stopped(vhc))
54             qps_VHC_stopped(vhc, PTRUE);
55         return current_speed;
56     }
57
58     /* find acceleration/deceleration needed to reach speed asap */
59     float accel = 0;
60     if (diff < 0)
61     {
62         /* decelerate */
63         accel = max(qpg_VTP_deceleration(qpg_VHC_type(vhc)), diff);
64     }
65     else
66     {
67         /* accelerate */
68         accel = min(qpg_VTP_acceleration(qpg_VHC_type(vhc)), diff);
69     }
70
71     return current_speed + (qpg_CFG_timeStep()*accel);
72 }
73
74 traci_api::LinearSpeedChangeController
75 ::LinearSpeedChangeController(VEHICLE* vhc, float target_speed, int
76 duration) : vhc(vhc), duration(0), done(false)
77 {
78     /*
79      * calculate acceleration needed for each timestep. if duration
80      * is too short, i.e.
81      * it causes the needed acceleration to be greater than the
82      * maximum allowed, we'll use
83      * the maximum for the duration, but we'll never reach the
84      * desired speed.
85      */
86
87     float current_speed = qpg_VHC_speed(vhc);
88     float diff = target_speed - current_speed;
89     // first, check if we actually need to change the speed
90     // this will do nothing if we don't
91     if (abs(diff) < NUMERICAL_EPS)
92     {
93         done = true;
94         acceleration = 0;
95         return;
96     }

```

```

93     float timestep_sz = qpg_CFG_timeStep();
94     float duration_s = duration / 1000.0f;
95     int d_factor = round(duration_s / timestep_sz);
96     this->duration = d_factor * (timestep_sz * 1000);
97
98     acceleration = diff / (duration / 1000.0f); // acceleration (m/s2)
99     if (diff < 0)
100    {
101        /* decelerate */
102        acceleration = max(qpg_VTP_deceleration(qpg_VHC_type(vhc)),
103                            acceleration);
104    }
105    else
106    {
107        /* accelerate */
108        acceleration = min(qpg_VTP_acceleration(qpg_VHC_type(vhc)),
109                            acceleration);
110    }
111}
112float traci_api::LinearSpeedChangeController::nextTimeStep()
113{
114    float timestep_sz = qpg_CFG_timeStep();
115    duration -= timestep_sz * 1000;
116    if (duration <= 0)
117        done = true;
118
119    return qpg_VHC_speed(vhc) + (timestep_sz * acceleration);
120}

```