Capítulo 1

Arquitectura General y Funcionalidad Implementada

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. 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 1.1).

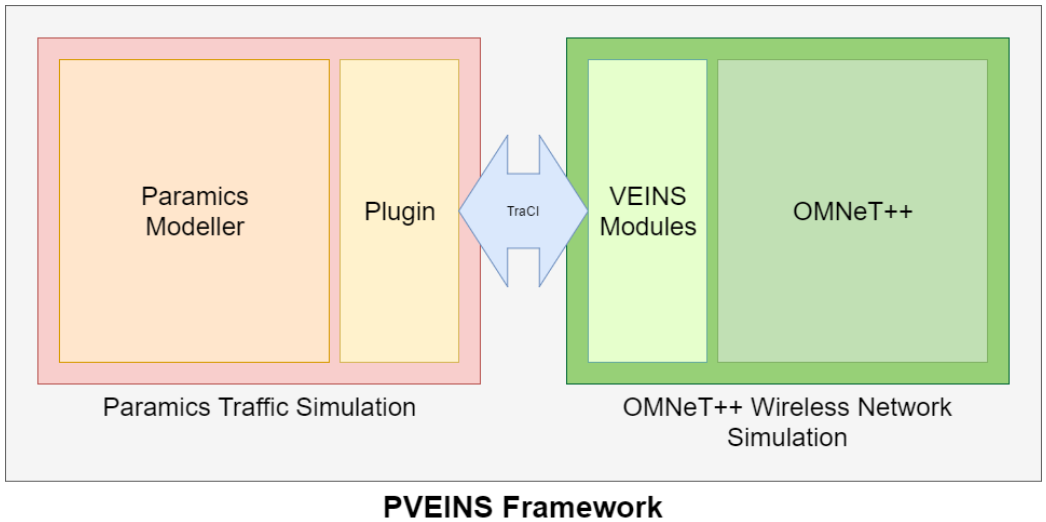


Figura 1.1: Visión macroscópica del framework; el plugin desarrollado actúa como una interfaz entre TraCI y Paramics.

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 proyecto. A continuación, se describirán brevemente estas dos iteraciones del diseño del software, destacando principalmente las razones del descarte de la versión preliminar.

Arquitectura Preliminar

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

Un diagrama de la arquitectura general de esta implementación puede observarse en la figura 1.2. Al iniciarse Paramics, el *plugin* inicializaba el servidor en un *thread* paralelo; el servidor luego se enlazaba a un *socket* TCP y se bloqueaba en espera de mensajes entrantes desde un cliente TraCI (en nuestro caso, VEINS). Al recibir una serie de comandos TraCI, el servidor los interpretaba, comunicándose con Paramics a través de su API, obteniendo datos y modificando el estado de la simulación. El servidor interpretaba todos los comandos en un mensaje TraCI antes de enviar todos los mensajes de respuesta correspondientes en un único mensaje TraCI.

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

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

Se estudiaron múltiples maneras de resolver este problema manteniendo la estructura paralela del *plugin* sin éxito, ya que la única manera confiable

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

de forzar un avance de la simulación desde el *plugin* es a través de la función anteriormente mencionada. Se decidió entonces abordar el problema desde un ángulo distinto, enfoque que se discutirá en la siguiente sección.

Arquitectura Final

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

Esta arquitectura puede visualizarse en la figura 1.3. 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()`, obteniendo incluso una leve mejora en rendimiento respecto al diseño antiguo, dado que, ya que todo corre en un sólo *thread*, se evita el uso de elementos de sincronización, los cuales pueden agregar *overhead* al procedimiento.

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

Finalmente, se debe notar que dada la representación en tiempo discreto de los pasos de simulación, el avance de esta en muchos casos no alcanza exactamente el tiempo deseado. Si definimos el paso de simulación como

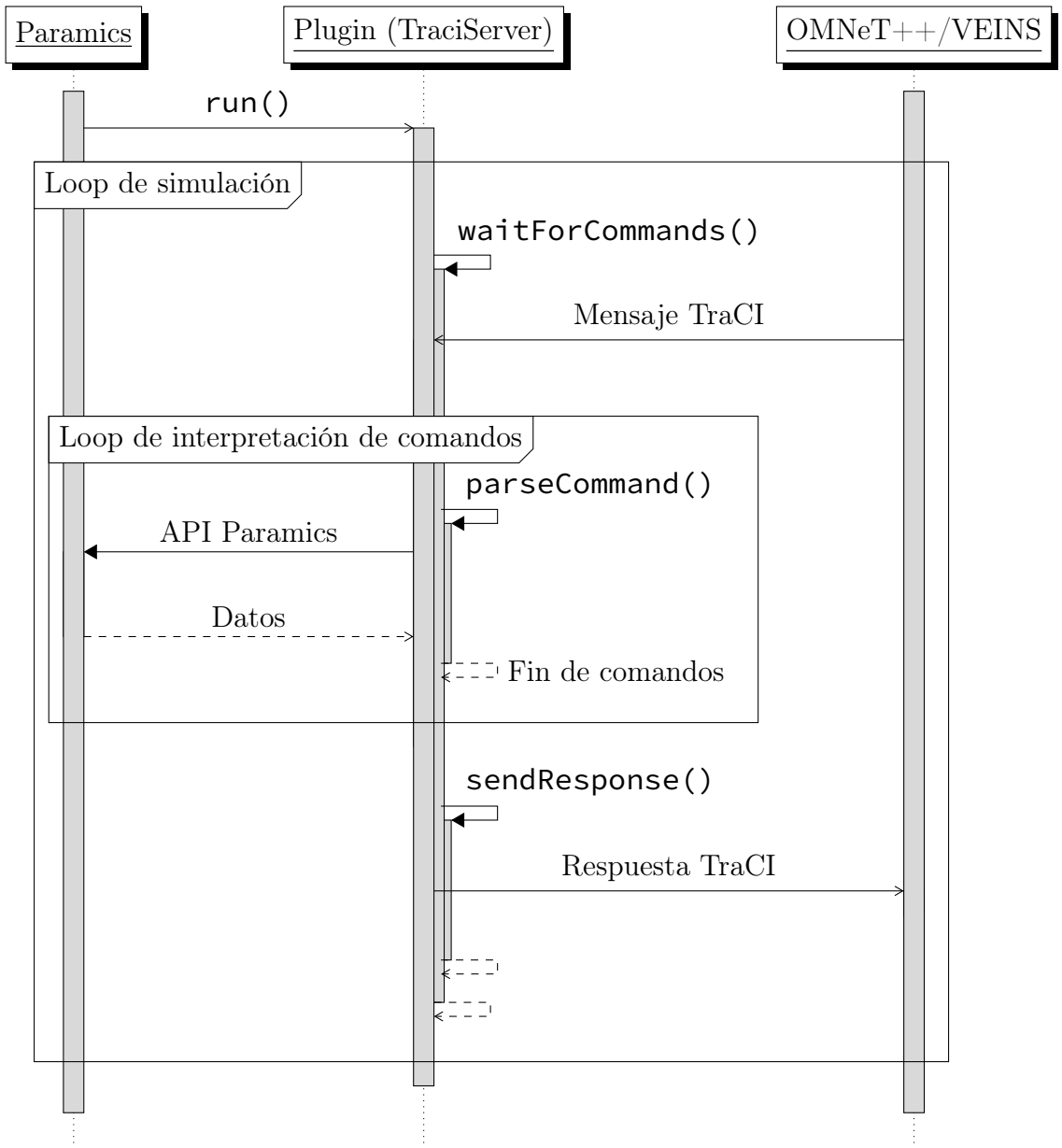


Figura 1.2: Arquitectura preliminar

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

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

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

$$T_f \geq T_o$$

$$T'_f < T_o$$

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.

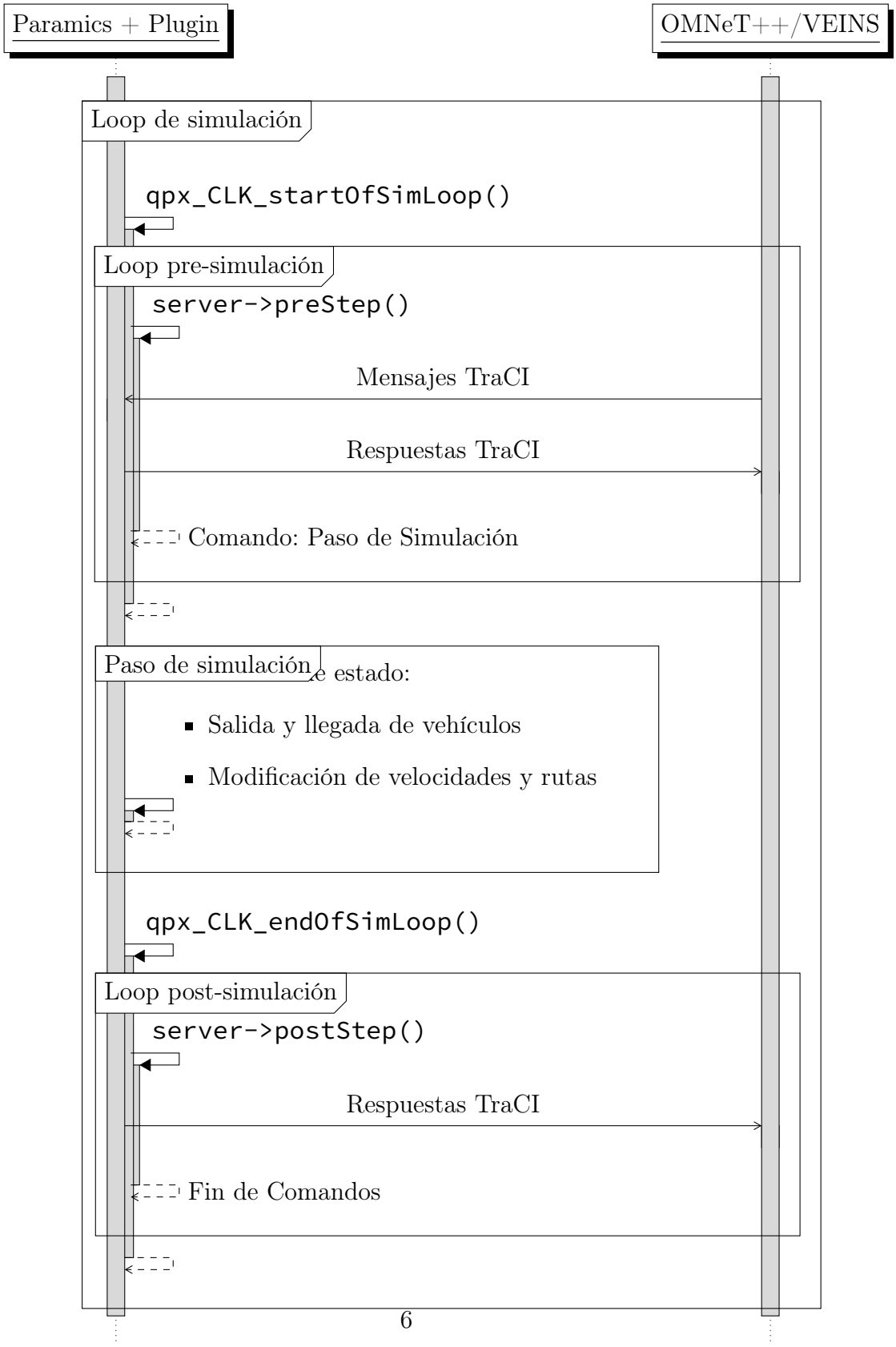


Figura 1.3: Arquitectura final del *framework*

1.2. Funcionalidad Implementada

El protocolo TraCI define más de 30 comandos distintos, cada uno con una gran cantidad de variables y parámetros asociados (ver apéndice A para una descripción más detallada del funcionamiento de este protocolo). Implementar esta gran cantidad de funcionalidades no hubiese sido factible, por lo que se escogió un subconjunto acotado de éstas a implementar, considerando en específico aquellos comandos esenciales para simulaciones de ITS.

1.2.1. 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 comienzaron y terminaron de teletransportarse en el último paso de simulación, así como las variables `0x6c`, `0x6d`, `0x6e` y `0x6f`, las cuales corresponden a números y listas

de vehículos que comenzaron y terminaron de estar estacionados, fueron implementadas "parcialmente". En estricto rigor, los mecanismos subyacentes no se implementaron porque no se consideraron críticos, pero se implementó una respuesta *dummy* de 0 vehículos para asegurar su funcionamiento con VEINS.

1.2.2. Comandos de modificación de estado

`0xc4` Variables de vehículo

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

Apéndice A

TraCI

TraCI (**Tra**ffic **C**ontrol **I**nterface) es una arquitectura para la interacción con simuladores de redes de transporte, cuyo principal propósito es facilitar el diseño y la implementación de simulaciones de Sistemas de Transporte Inteligente [2]. 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.

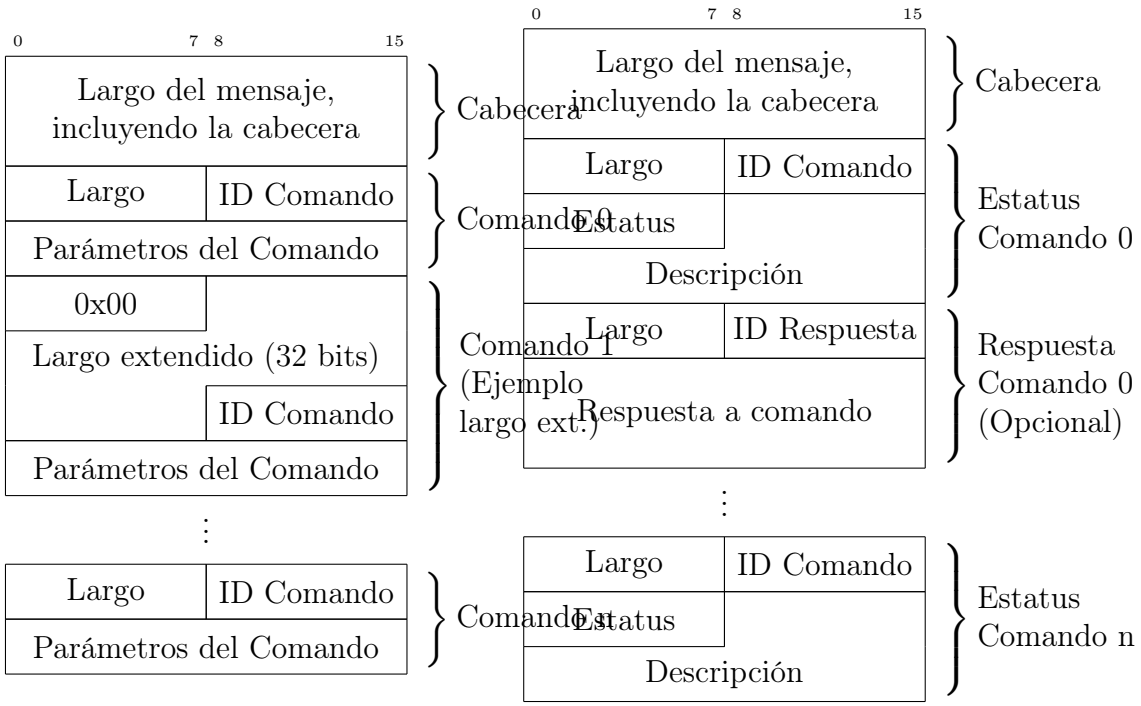
Hoy en día, dicha arquitectura se encuentra integrada en SUMO, y se utiliza en conjunto con simuladores de redes de comunicación inalámbrica como OMNeT++ y NS2 para la simulación y estudio de Sistemas de Transporte Inteligente.

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,



(a) Comandos

(b) Respuesta

Figura A.1: Formatos de mensajes TraCI

en el caso de que se trate de un comando que asigne algún valor a una variable de la simulación. En caso de que el valor del largo exceda 255, se agrega un campo de 32 bits para almacenar dicho valor y el campo original se fija en `0x00`.

Por otro lado, la figura 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

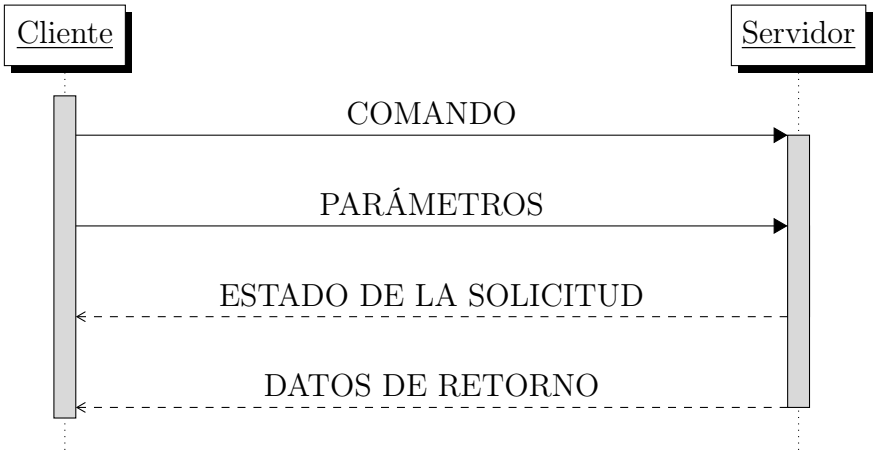


Figura A.2: Ejemplo solicitud de variable TraCI.

simulación:

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

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

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

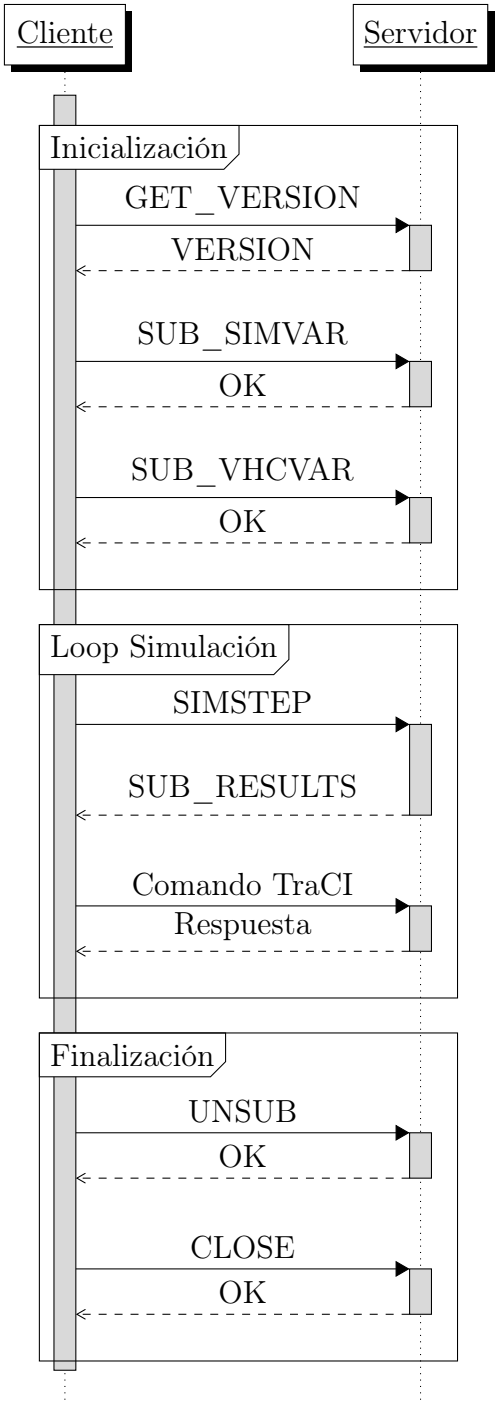


Figura A.3: Flujo de comunicación TraCI.

- **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.

Apéndice B

Paramics API

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

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

```
CATEGORIA_DOMINIO_nombre_de_funcion()
```

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

B.1. Categorías de Funciones

Se definen cuatro categorías de funciones:

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

B.1.1. Funciones QPO

Las funciones de *override*, con prefijo `QPO`, corresponden a funciones que controlan algún comportamiento clave del modelo interno de Paramics y que son sobrescribibles por el usuario. Por ejemplo, la función `float qpo_CFM_leadSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])` se utiliza para modificar las velocidades de cada vehículo que no tiene otro vehículo delante, en cada paso de simulación; esta función es sobrescrita en el *plugin* desarrollado para retornar velocidades distintas a las que retornaría Paramics por defecto, para los casos de vehículos que han recibido comandos de modificación de velocidad desde OMNeT++.

B.1.2. Funciones QPX

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

B.1.3. Funciones QPG

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

B.1.4. Funciones QPS

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

B.2. Dominios

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