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

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

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

MANUEL OSVALDO J. OLGUÍN MUÑOZ

PROFESOR GUÍA:
SANDRA CÉSPEDES U.

MIEMBROS DE LA COMISIÓN:
JAVIER BUSTOS
NANCY HITSCHFELD

Este trabajo ha sido parcialmente financiado por NIC Chile Research Labs

SANTIAGO DE CHILE
JULIO 2017

# Capítulo 1

# 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 éstos módulos puede observarse en la figura 1.1, y puede explorarse en línea en el repositorio del proyecto [1].

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

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

**Figura 1.1:** Estructura de archivos del código fuente del framework.

## 1.1. 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 1.2) 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 1.2 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 1.4) 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 de simulación para cada vehículo en la simulación de tráfico de Paramics – `leadSpeed()` se invoca para aquellos vehículos que no tienen otro vehículo delante, y `followSpeed()` es invocada para todos los demás.

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.

## 1.2. 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 instancia en el archivo plugin.c.

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

Estos archivos no fueron desarrollados por el memorista, sino que fueron obtenidos desde el código fuente de SUMO [1], distribuidos bajo una licencia BSD[2].

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

### 1.2.1. Inicio de conexión TraCI

Como se mencionó en la sección 1.1, al iniciarse el *plugin* se crea un nuevo *thread*, en el cual se instancia un objeto de la clase `TraCIServer`, al cual se le invoca su método `waitForConnection()` (código 1.1).

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

[1]Fuente SUMO: `https://github.com/planetsumo/sumo/tree/master/sumo/src/foreign/tcpip`. Debe notarse que, a su vez, los creadores de SUMO originalmente obtuvieron dichos archivos del código fuente del simulador de eventos discretos para redes de sensores *SHAWN* [2]. Su fuente original se encuentra en `https://github.com/itm/shawn/tree/master/src/apps/tcpip`

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

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

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

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

**Código 1.1:** Rutina de inicio de conexión.

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.

### 1.2.2. Recepción e Interpretación de Comandos Entrantes

Como se explicó en la sección **??**, 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.

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 1.2.2), 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, aunque un poco tediosa, y su esqueleto puede observarse en el código 1.2. 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 subsecuente validación y activación de ésta. Por el contrario, en caso

de recibir un comando con identificador fuera de dicho rango, se procede a verificar su tipo mediante un *switch*. Cada caso se relaciona con un comando y método específico a invocar, y en caso de no encontrarse el identificador en cuestión se notifica al cliente que el comando deseado no está implementado.

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

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

**Código 1.2:** Esqueleto de `parseCommand()`

Se definieron una serie de métodos en `TraCIServer` encargados de obtener variables de la simulación o modificar el estado de ésta. El funcionamiento de éstos es idéntico en todos los casos (a excepción de `cmdGetPolygonVar()`), y se limita al siguiente procedimiento (ver ejemplo en el código 1.3):

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.

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

**Código 1.3:** Ejemplo de método de obtención y empaquetado de variables en `TraCIServer`

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

### Evaluación de suscripciones

Como se explicó en la sección 1.2.2, 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()`.

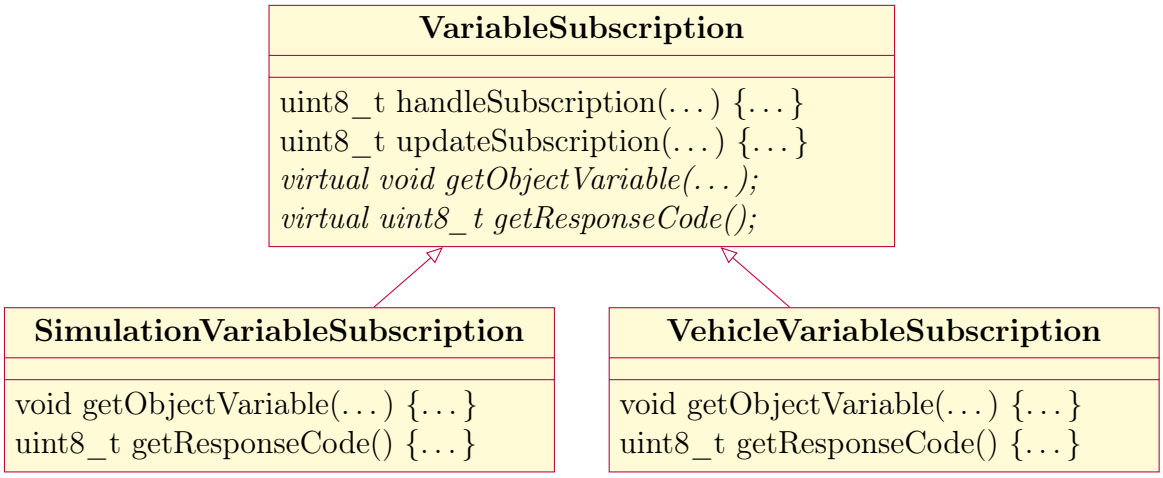



**Figura 1.2:** Diagrama de herencia, `VariableSubscription`

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

La raíz de éste árbol, la clase `VariableSuscription`, implementa la funcionalidad completa de evaluación y actualización de una suscripción en los métodos `handleSubscription()` y `updateSubscription()` (implementación completa de estos métodos en C.4), abstrayendo la obtención de datos específicos a cada tipo en los métodos `getObjectVariable()` y `getResponseCode()`. Estos métodos son *virtuales* en la clase base, y son implementados por las clases derivadas, de manera que `TraCIServer` sólo necesita mantener un vector de variables del tipo base, las cuales se evalúan de manera polimórfica.

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

De esta manera, la evaluación en `TraCIServer` se simplifica, ya que la instancia sólo necesita mantener un vector con punteros a objetos de la clase base, dado que independiente de la implementación específica de los métodos `getObjectVariable()` y `getResponseCode()` de cada subclase, `handleSubscription()` es exactamente igual para todas (ver línea 7 en el código 1.4). Además, esto facilita la extensión futura del software.

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

**Código 1.4:** Rutina de evaluación de suscripciones en `TraCIServer`. `subs` es una variable de instancia de `TraCIServer` correspondiente a un vector de punteros `VariableSuscription*`, poblado de elementos de clases derivadas de `VariableSuscription`.

### Creación de Nuevas Suscripciones

Por otro lado, para crear nuevas suscripciones, `TraCIServer` debe considerar la categoría de objetos a la cual se está suscribiendo, e insertar un puntero a un objeto con el tipo correspondiente en la variable de instancia `subs`. Esto sucede al recibir un comando con un identificador correspondiente a una suscripción; los parámetros de la suscripción son extraídos y delegados al método `addSubscription()` (código 1.2,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 dealocar la memoria asignada al puntero.

3. `STATUS_ERROR`: Los parámetros corresponden a una actualización de esta suscripción (categoría e identificador de objeto son los mismos), pero sucedió un error en la actualización. `addSubscription()` escribe un mensaje de notificación al cliente y retorna.

4. `STATUS_OK`: Los parámetros corresponden a una actualización de esta suscripción (categoría e identificador de objeto son los mismos), y la actualización fue exitosa. `addSubscription()` escribe un mensaje de notificación al cliente y retorna.

```
1   for (auto it = subs.begin(); it != subs.end(); ++it)
2   {
3       uint8_t result = (*it)->updateSubscription(sub_type,
            object_id, start_time, end_time, variables, temp, errors);
4
5       switch (result)
6       {
7       case VariableSubscription::STATUS_OK:
8           // update ok, return now
9           debugPrint("Updated subscription");
10          writeStatusResponse(sub_type, STATUS_OK, "");
11          writeToOutputWithSize(temp, true);
12          return;
13      case VariableSubscription::STATUS_UNSUB:
14          // unsubscribe command, remove the subscription
15          debugPrint("Unsubscribing...");
16          delete *it;
17          it = subs.erase(it);
18          // we don't care about the deleted iterator, since we
                return from the loop here
19          writeStatusResponse(sub_type, STATUS_OK, "");
20          return;
21      case VariableSubscription::STATUS_ERROR:
```

```
22            // error when updating
23            debugPrint("Error updating subscription.");
24            writeStatusResponse(sub_type, STATUS_ERROR, errors);
25            break;
26         case VariableSubscription::STATUS_NOUPD:
27            // no update, try next subscription
28            continue;
29         default:
30            throw std::runtime_error("Received unexpected result " +
                 std::to_string(result) + " when trying to update
                 subscription.");
31         }
32      }
```

**Código 1.5:** Verificación de actualización en `addSubscription()`.

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

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

**Código 1.6:** Creación de una nueva suscripción. Notar la instanciación polimórfica.

### 1.2.3. Envío de resultados al cliente

`TraCIServer` mantiene una variable de instancia `tcpip::Storage outgoing`, en la cual se almacenan los mensajes de estado y resultados de comandos TraCI. El envío de

estos al cliente se efectúa al final de cada iteración del *loop* en `preStep()` o, en el caso de recibir un comando de avance de simulación, en `postStep()`, enviando así conjuntamente las respuestas a todos los comandos obtenidos desde el cliente en el último paso de tiempo. Gracias a las clases `tcpip::Storage` y `tcpip::Socket` utilizadas, la operación de enviar los datos almacenados se reduce a una invocación del método `sendExact()` del objeto `tcpip::Socket`, la cual recibe un objeto `tcpip::Storage`, le adjunta una cabecera con su tamaño total y lo envía a través del *socket* al cliente.

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

```
void traci_api::TraCIServer::writeStatusResponse(uint8_t cmdId,
   uint8_t cmdStatus, std::string description)
{
   debugPrint("Writing status response " + std::to_string(cmdStatus) +
      " for command " + std::to_string(cmdId));
   outgoing.writeUnsignedByte(1 + 1 + 1 + 4 +
      static_cast<int>(description.length())); // command length
   outgoing.writeUnsignedByte(cmdId); // command type
   outgoing.writeUnsignedByte(cmdStatus); // status
   outgoing.writeString(description); // description
}

void traci_api::TraCIServer::writeToOutputWithSize(tcpip::Storage&
   storage, bool force_extended)
{
   this->writeToStorageWithSize(storage, outgoing, force_extended);
}

void traci_api::TraCIServer::writeToStorageWithSize(tcpip::Storage&
   src, tcpip::Storage& dest, bool force_extended)
{
   uint32_t size = 1 + src.size();
   if (size > 255 || force_extended)
   {
      // extended-length message
      dest.writeUnsignedByte(0);
      dest.writeInt(size + 4);
   }
   else
      dest.writeUnsignedByte(size);
   dest.writeStorage(src);
}
```

**Código 1.7:** Escritura de datos en almacenamiento saliente.

## 1.3. 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 instanciación de un único objeto de este tipo en la ejecución del programa. Esto ya que, por razones lógicas, cada ejecución del *plugin* está asociada a una única simulación en Paramics, y por ende no tiene sentido que pueda existir más de un objeto de acceso a ésta. Este patrón de diseño tiene además la ventaja que simplifica el acceso a la instancia global de la clase en el sistema, desde cualquier otro objeto u función.

### 1.3.1. Obtención de variables

Las variables obtenibles desde este módulo son todas aquellas que se relacionan con la simulación como ente abstracto, enumeradas en el ítem ?? de la sección ??. 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.7 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`, 1.4, para más detalles.

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

En segundo lugar se hablará del comando de obtención de las coordenadas de los límites de la simulación. Este es de extrema importancia para VEINS, ya que con estos valores se crea el escenario de comunicación inalámbrica en OMNeT++; de ser erróneos, tarde o temprano la posición de un vehículo (representado por un nodo de comunicación en OMNeT++) quedará fuera del escenario, gatillando un error fatal en la simulación. Desafortunadamente, si bien la API de Paramics cuenta con un comando para, supuestamente, obtener estas coordenadas, por razones que no se lograron dilucidar, este comando retorna valores altamente erróneos (esto se verificó con múltiples redes de transporte). Se debió entonces implementar el cálculo correcto de éstos límites en el módulo mismo, en el método, apropiadamente nombrado, `getRealNetworkBounds()` (expuesto en el código C.8 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.

## 1.4. 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 **??**). 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.

### 1.4.1. Estado interno

Para simplificar muchas de las operaciones de obtención de variables y modificación de estados, el módulo mantiene un estado interno congruente con el estado de la simulación en Paramics. Para este fin se ocupan los llamados de la API de Paramics mencionados en la sección 1.1.

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 1.1). 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 1.4.3), para efectuar cambios de velocidad dictados por el cliente TraCI.

**`vhc_routes`** Mapa para el manejo de cambios de ruta desde TraCI (ver sección 1.4.3).

**`lane_set_triggers`** *Hashmap* utilizado para relacionar vehículos con eventuales coman-

dos de cambio de pista (ver sección 1.4.3).

### 1.4.2. Obtención de variables

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

`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 alguna 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 1.3.1, 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.7 para un acercamiento a la implementación real de éstos).

### 1.4.3. 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 **??** 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 tiene debe aplicarse por un periodo mayor a un sólo 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 velo-

cidad. Como se comentó anteriormente, en la sección 1.4.1, `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 1.1), 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.

```
bool traci_api::VehicleManager::speedControlOverride(VEHICLE* vhc,
   float& speed)
{
   BaseSpeedController* controller;
   try
   {
      controller = speed_controllers.at(vhc);
      speed = controller->nextTimeStep();

      if (!controller->repeat())
      {
         speed_controllers.erase(vhc);
         delete controller;
      }

      return true;
   }
   catch (std::out_of_range& e)
   {
      return false;
   }
}
```

**Código 1.8:** 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`.

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.9.

### 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 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.

```
int traci_api::VehicleManager::rerouteVehicle(VEHICLE* vhc, LINK*
    lnk)
{
    if (0 == qpg_VHC_uniqueID(vhc)) // dummy vhc
        return 0;

    // check if the vehicle has a special route
    std::unordered_map<LINK*, int>* exit_map;
    try
    {
        exit_map = vhc_routes.at(vhc);
    }
    catch (std::out_of_range& e)
    {
        // no special route, return default
        return 0;
    }

    int next_exit = 0;
    try
    {
        next_exit = exit_map->at(lnk);
    }
    catch (std::out_of_range& e)
    {
        // outside route, clear
        exit_map->clear();
        delete exit_map;
        vhc_routes.erase(vhc);
    }

    return next_exit;
}
```

**Código 1.9:** Método de reruteo en VehicleManager, para vehículos con rutas dictadas por un cliente TraCI.

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 1.1).

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. Desafortunadamente, 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.

```
void traci_api::LaneSetTrigger::handleTrigger()
{
   int t_lane = target_lane;
   // make sure we stay within maximum number of lanes
   int maxlanes = qpg_LNK_lanes(qpg_VHC_link(vehicle));
   if (t_lane > maxlanes)
      t_lane = maxlanes;
   else if (t_lane < 1)
      t_lane = 1;

   int current_lane = qpg_VHC_lane(vehicle);
   if (current_lane > t_lane) // move outwards
      qps_VHC_laneChange(vehicle, -1);
   else if (current_lane < t_lane)
      qps_VHC_laneChange(vehicle, +1); // move inwards
   else
      qps_VHC_laneChange(vehicle, 0); // stay in this lane
}
```

**Código 1.10:** Cambio de pista, implementado en `LaneSetTrigger`

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 `BaseTrig-`

ger, 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.

```
void traci_api::VehicleManager::handleDelayedTriggers()
{
   // handle lane set triggers
   debugPrint("Handling vehicle triggers: lane set triggers");
   for (auto kv = lane_set_triggers.begin(); kv !=
      lane_set_triggers.end();)
   {
      kv->second->handleTrigger();

      /* check if need repeating */
      if (!kv->second->repeat())
      {
         delete kv->second;
         kv = lane_set_triggers.erase(kv);
      }
      else
         ++kv;
   }
   debugPrint("Handling vehicle triggers: done");
}
```

**Código 1.11:** Manejo de *triggers* para operaciones diferidas en VehicleManager

## 1.5. Otros módulos

### 1.5.1. 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.7), 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.

```
traci_api::Network::Network()
{
   int routes = qpg_NET_busroutes();
   for (int i = 1; i <= routes; i++)
   {
      BUSROUTE* route = qpg_NET_busrouteByIndex(i);
      std::string name = qpg_BSR_name(route);

      route_name_map[name] = route;

      int link_n = qpg_BSR_links(route);
      std::vector<std::string> link_names;

      LINK* current_link = qpg_BSR_firstLink(route);
      link_names.push_back(qpg_LNK_name(current_link));

      for (int link_i = 0; link_i < link_n - 1; link_i++)
      {
         current_link = qpg_BSR_nextLink(route, current_link);
         link_names.push_back(qpg_LNK_name(current_link));
      }

      route_links_map[route] = link_names;
   }
}
```

**Código 1.12:** Constructor del módulo `Network`

### 1.5.2. 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.

### 1.5.3. 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*.

```
#define DUMMY_CONST 0x42
```

```
static const
DUMMY_CONST = 0x42;
```

**Figura 1.3:** 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 lenguage es negligible.

### 1.5.4. 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, alterado 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. Este archivo define parámetros de simulación como la red vehicular a utilizar y la *semilla* deseada para la generación de valores pseudoaleatorios.

```
<?xml version="1.0"?>
<launch>
   <basedir path="X:\PVEINS\pveins" />
   <network name="example8_network" />
   <seed value="1234" />
</launch>
```

**Código 1.13:** Ejemplo de archivo XML de inicialización de la simulación.

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. Finalmente, 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*).

## 1.6. 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 **??**), 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. En tercer lugar 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. Cuarto, 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 **??**, 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* [3], dada su popularidad, extenso soporte y documentación y la familiaridad del memorista con este sistema. El servicio de *hosting* específico escogido para sistema fue *GitHub* [4]; el código fuente del proyecto puede encontrarse en el perfil personal del autor [5], en el repositorio [1].

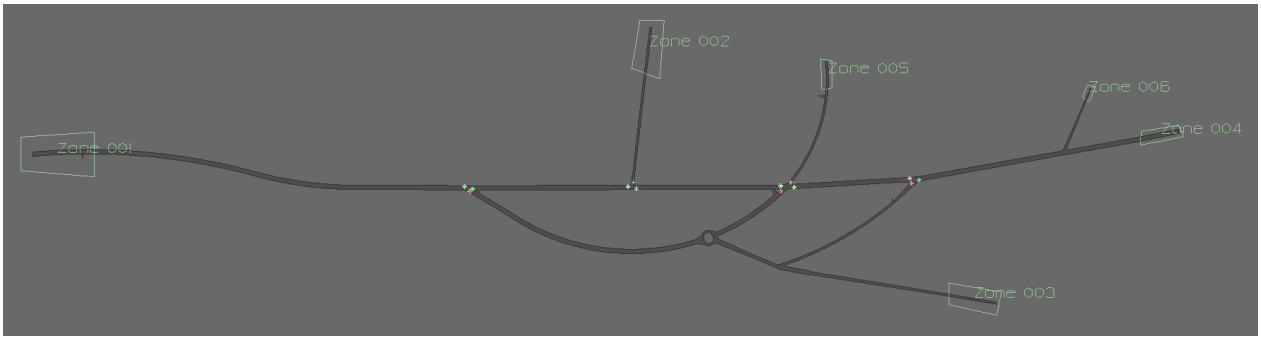



**Figura 1.4:** Red de transporte utilizada para las pruebas preliminares.

## 1.7. Pruebas preliminares

La validación preliminar del *framework* se realizó utilizando la implementación de TraCI en Python incluída 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 ([6], [7]), 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 1.4). 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 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 **??**.

### 1.7.1. Ejemplo de script de prueba: cambio de ruta

El código 1.14 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).

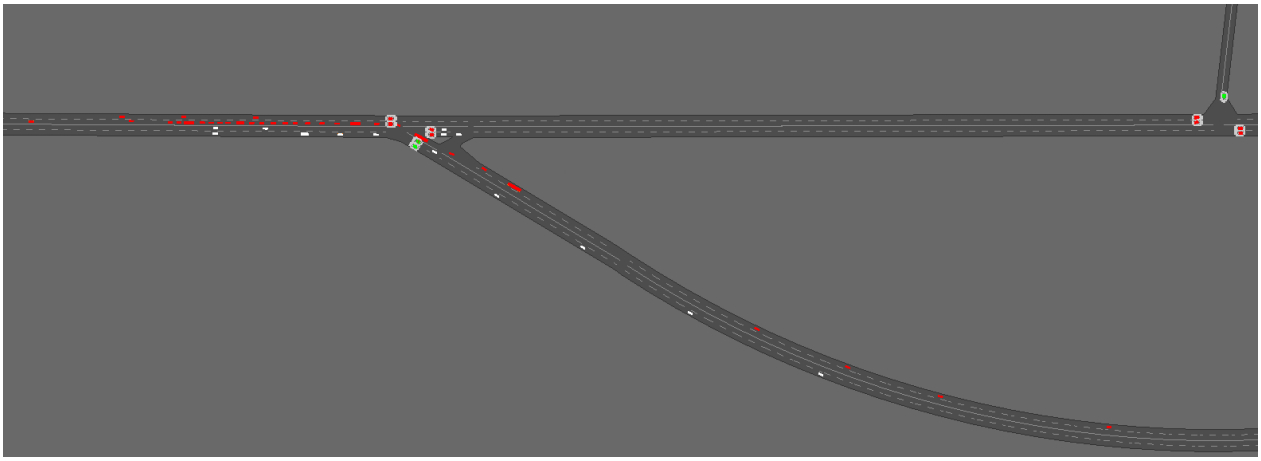


**Figura 1.5:** Visualización del *test* de cambio de ruta en curso. Los vehículos pintados de rojo son aquellos afectados por el cambio.

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 esta nueva, al mismo tiempo pintándolos de un color rojo para poder distinguirlos del resto. El resultado puede observarse en la figura 1.5.

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 1.6 – 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(...)`).

```
PORT = 8245
new_route = ["2:6c", "6c:25", "25:15"]
affected_cars = []

def run():
    """execute the TraCI control loop"""
    traci.init(PORT)
    print("Server version: " + str(traci.getVersion()))
    traci.vehicle.subscribe("x",[0]) # sub to list of vehicles in sim
    for i in range(0, 10000):
        traci.simulationStep()
        car_list = traci.vehicle.getSubscriptionResults("x")[0]
        for car in car_list:
            current_road = traci.vehicle.getRoadID(car) # get road
            if (current_road == new_route[0]) and (car not in
                affected_cars):
                print("route change for " + str(car))
                traci.vehicle.setColor(car, (255, 0, 0, 0))
                traci.vehicle.setRoute(car, new_route)
                affected_cars.append(car)
    traci.close()
```

**Código 1.14:** *Script* para la prueba de cambio de ruta.

# Bibliografía


[1] (Mayo de 2017). Repositorio PVeins en GitHub, dirección: https://github.com/molguin92/paramics_traci.

[2] 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.

[3] (Mayo de 2017). Git, a FOSS distributed version control system, dirección: https://git-scm.com/.

[4] (Mayo de 2017). GitHub development platform, dirección: https://github.com/.

[5] (Mayo de 2017). Perfil personal del autor en GitHub, dirección: https://github.com/molguin92.

[6] (Mayo de 2017). Python TraCI Library Documentation, dirección: http://www.sumo.dlr.de/pydoc/traci.html.

[7] (Mayo de 2017). Código fuente cliente TraCI Python, GitHub, dirección: https://github.com/planetsumo/sumo/tree/master/sumo/tools/traci.

# 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 [**traci**]. Proporciona una interfaz unificada que permite no sólo la obtención de datos desde la simulación de transporte, sino que también permite el control directo sobre la ejecución de ésta y provee métodos para la modificación del comportamiento de sus componentes. Así, TraCI permite a un agente externo (como, por ejemplo, un simulador de redes) comunicarse de manera bidireccional con la simulación de la red de transporte, posibilitando un desarrollo dinámico de dicha simulación en reacción a estímulos externos.

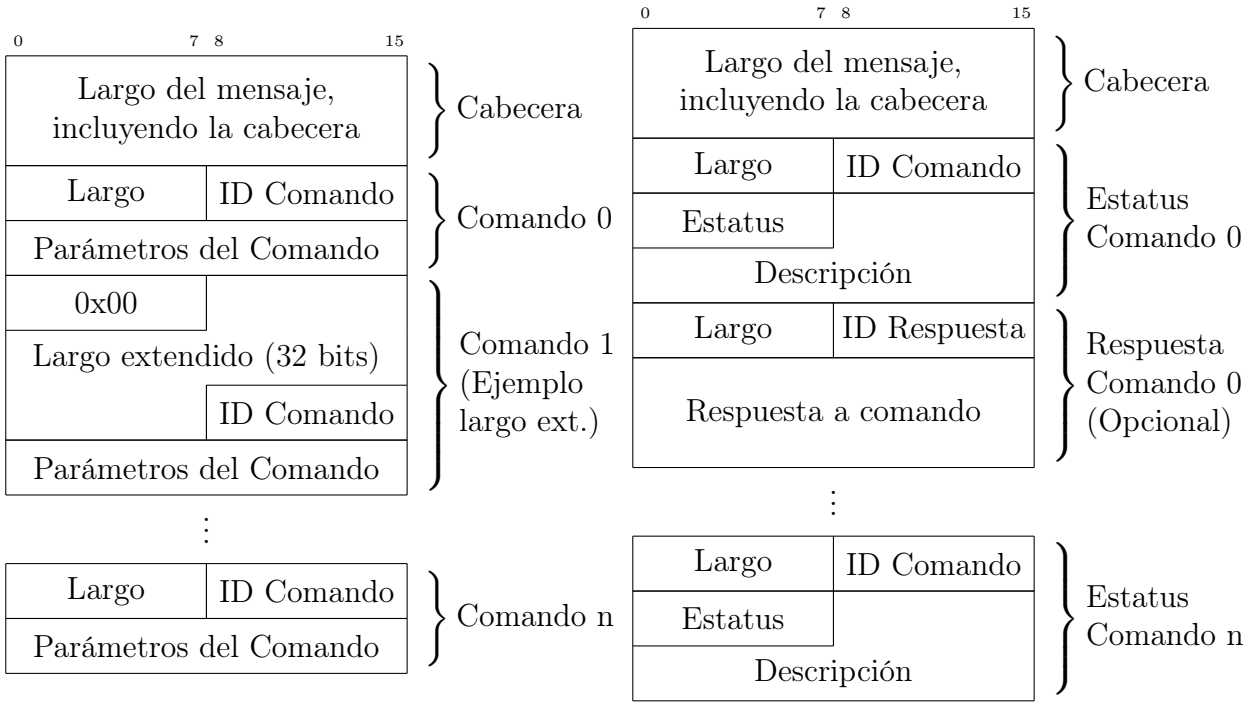
Hoy en día, dicha arquitectura se encuentra integrada en SUMO, y se utiliza en conjunto con simuladores de redes de comunicación inalámbrica como OMNeT++ y NS2 para la simulación y estudio de Sistemas de Transporte Inteligente.

### A.0.1. Diseño

#### Mensajes

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

La figura A.1a ilustra la estructura básica de un mensaje TraCI enviado desde un cliente al servidor. Consiste en una cadena de comandos TraCI consecutivos que deben ser ejecutados por este último; cada comando tiene un largo y un identificador, y puede incluir información adicional – por ejemplo, en el caso de que se trate de un comando que asigne algún valor a una variable de la simulación. En caso de que el valor del largo exceda 255, se agrega un campo de 32 bits para almacenar dicho valor y el campo original se fija en `0x00`.




**(a)** Comandos

**(b)** Respuesta

**Figura A.1:** Formatos de mensajes TraCI

Por otro lado, la figura A.1b ilustra un ejemplo de respuesta del servidor, el cual debe responder a cada uno de éstos mensajes con una notificación del estado de la solicitud ("OK", "ERROR" o "NO IMPLEMENTADO") y, en caso de que corresponda, con información adicional de acuerdo a parámetros específicos definidos para cada comando. Finalmente, la figura A.2 ilustra el flujo de mensajes para la solicitud de una variable de la simulación.

### Comandos

El protocolo define tres categorías de comandos disponibles:

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

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

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

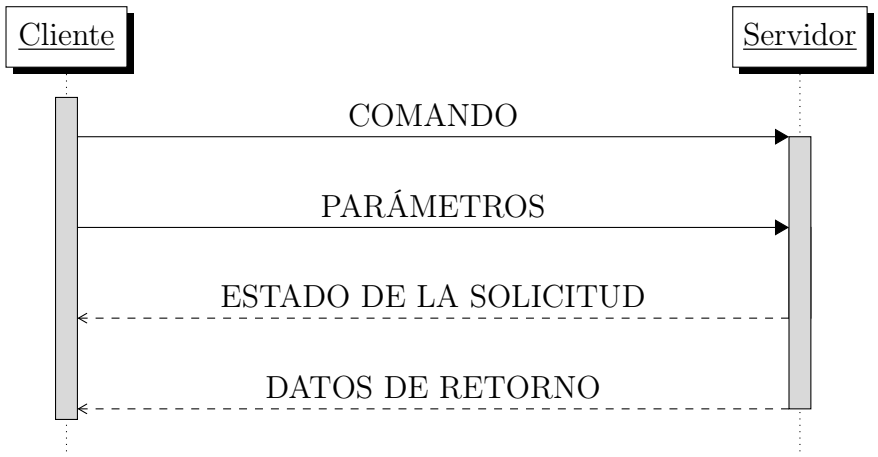



**Figura A.2:** Ejemplo solicitud de variable TraCI.

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

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

- **Obtención de Valores:** Esta categoría abarca una gran cantidad de comandos asociados a variables internas de la simulación vehicular o de sus componentes (vehículos, cruces, etc.). Cada comando representa a un conjunto de variables específicas; por ejemplo, `0xa2 GET TRAFFIC LIGHTS VARIABLE` agrupa y obtiene los valores asociados a las variables propias de los semáforos en la red simulada, mientras que `0xa4 GET VEHICLE VARIABLE` está relacionado exclusivamente con los valores de los vehículos presentes en la red.

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

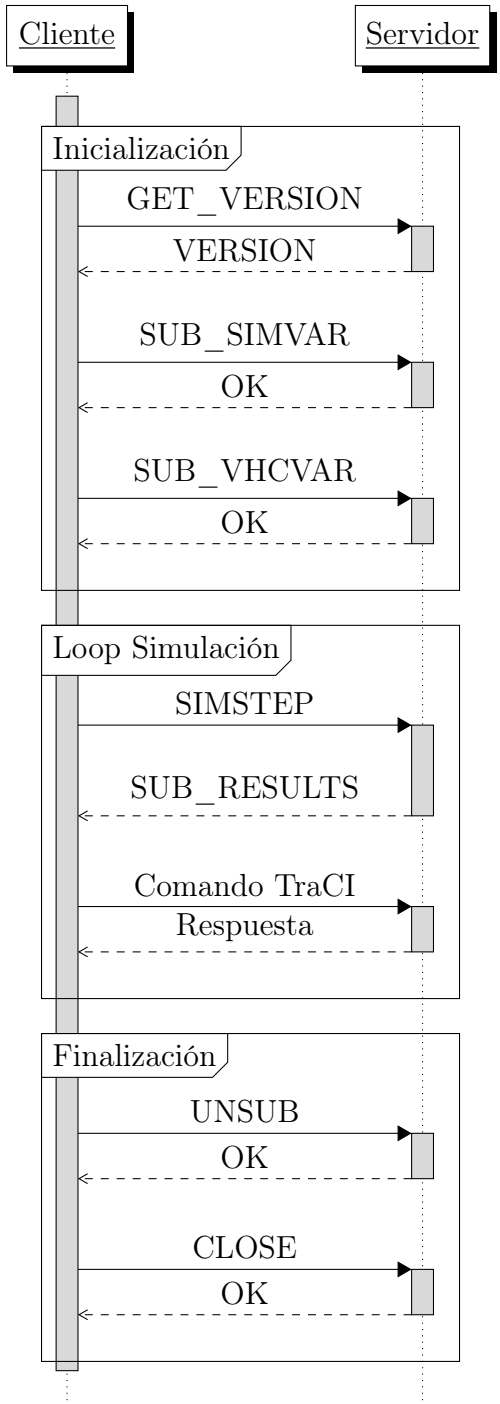



**Figura A.3:** Flujo de comunicación TraCI.

# Apéndice B

# Paramics API

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

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

```
CATEGORIA_DOMINIO_nombre_de_funcion()
```

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

## B.1. Categorías de Funciones

Se definen cuatro categorías de funciones:

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

### B.1.1. Funciones `QPO`

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

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

### B.1.2. Funciones `QPX`

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

### B.1.3. Funciones `QPG`

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

### B.1.4. Funciones `QPS`

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

## B.2. Dominios

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

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

# Apéndice C

# Códigos

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

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

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

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

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

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

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

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

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

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

void qpx_NET_complete(void)
{
   close();
}

void qpx_NET_close()
{
   close();
}

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

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

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

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

int qpo_RTM_decision(LINK *linkp, VEHICLE *Vp)
{
   return
      traci_api::VehicleManager::getInstance()->rerouteVehicle(Vp,
      linkp);
}

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

```
129
130 void qpx_VHC_transfer(VEHICLE* vehicle, LINK* link1, LINK* link2)
131 {
132    traci_api::VehicleManager::getInstance()->routeReEval(vehicle);
133 }
134
135 // speed control override
136 float qpo_CFM_followSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])
137 {
138    float speed = 0;
139    if (traci_api::VehicleManager::getInstance()
140             ->speedControlOverride(v, speed))
141       return speed;
142    else
143       return qpg_CFM_followSpeed(link, v, ahead);
144 }
145
146 float qpo_CFM_leadSpeed(LINK* link, VEHICLE* v, VEHICLE* ahead[])
147 {
148    float speed = 0;
149    if (traci_api::VehicleManager::getInstance()
150             ->speedControlOverride(v, speed))
151       return speed;
152    else
153       return qpg_CFM_leadSpeed(link, v, ahead);
154 }
```

## Código C.2: Método `preStep()` en `TraCIServer`

```
void traci_api::TraCIServer::preStep()
{
   std::lock_guard<std::mutex> lock(socket_lock);
   if (multiple_timestep
      && Simulation::getInstance()->getCurrentTimeMilliseconds() <
         target_time)
   {
      VehicleManager::getInstance()->reset();
      return;
   }

   multiple_timestep = false;
   target_time = 0;

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

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

   // receive and parse messages until we get a simulation step
      command
   while (running && ssocket.receiveExact(incoming))
   {
      incoming_size = incoming.size();

      debugPrint("Got message of length " +
         std::to_string(incoming_size));
      //debugPrint("Incoming: " + incoming.hexDump());


      /* Multiple commands may arrive at once in one message,
      * divide them into multiple storages for easy handling */
      while (incoming_size > 0 && incoming.valid_pos())
      {
         uint8_t cmdlen = incoming.readUnsignedByte();
         cmdStore.writeUnsignedByte(cmdlen);

         debugPrint("Got command of length " +
            std::to_string(cmdlen));


         for (uint8_t i = 0; i < cmdlen - 1; i++)
            cmdStore.writeUnsignedByte(incoming
                                       .readUnsignedByte());

         bool simstep = this->parseCommand(cmdStore);
         cmdStore.reset();

```

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

## Código C.3: Método `postStep()` en `TraCIServer`

```
void traci_api::TraCIServer::postStep()
{
   // after each step, have VehicleManager update its internal state
   VehicleManager::getInstance()
               ->handleDelayedTriggers();
   if (multiple_timestep
      && Simulation::getInstance()->getCurrentTimeMilliseconds() <
         target_time)
      return;
   // after a finishing a simulation step command (completely),
      collect subscription results and
   // check if there are commands remaining in the incoming storage
   this->writeStatusResponse(CMD_SIMSTEP, STATUS_OK, "");
   // handle subscriptions after simstep command
   tcpip::Storage subscriptions;
   this->processSubscriptions(subscriptions);
   outgoing.writeStorage(subscriptions);
   // finish parsing the message we got before the simstep command
   tcpip::Storage cmdStore;
   /* Multiple commands may arrive at once in one message,
   * divide them into multiple storages for easy handling */
   while (incoming_size > 0 && incoming.valid_pos())
   {
      uint8_t cmdlen = incoming.readUnsignedByte();
      cmdStore.writeUnsignedByte(cmdlen);
      debugPrint("Got command of length " + std::to_string(cmdlen));

      for (uint8_t i = 0; i < cmdlen - 1; i++)
         cmdStore.writeUnsignedByte(incoming
                                    .readUnsignedByte());

      bool simstep = this->parseCommand(cmdStore);
      cmdStore.reset();

      // weird, two simstep commands in one message?
      if (simstep)
      {
         if(!multiple_timestep)
         {
            multiple_timestep = true;
            Simulation* sim = Simulation::getInstance();
            target_time = sim->getCurrentTimeMilliseconds() +
               sim->getTimeStepSizeMilliseconds();
         }
         VehicleManager::getInstance()->reset();
         return;
      }
   }
```

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

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

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

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

   bool result_errors = false;

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

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

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

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

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

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

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

   // if we reach here, it means we need to add a new subscription.
```

```
41      // note: it could also mean it's an unsubscribe command for a car
            that reached its
42      // destination. Check number of variables and do nothing if it's
            0.
43
44      if(variables.size() == 0)
45      {
46          // unsub command that didn't match any of the currently
                running subscriptions, so just
47          // tell the client it's ok, everything's alright
48
49          debugPrint("Unsub from subscription already removed.");
50          writeStatusResponse(sub_type, STATUS_OK, "");
51          return;
52      }
53
54
55      debugPrint("No update. Adding new subscription.");
56      VariableSubscription* sub;
57      switch (sub_type)
58      {
59      case CMD_SUB_VHCVAR:
60          debugPrint("Adding VHC subscription.");
61          sub = new VehicleVariableSubscription(object_id, start_time,
                end_time, variables);
62          break;
63      case CMD_SUB_SIMVAR:
64          debugPrint("Adding SIM subscription.");
65          sub = new SimulationVariableSubscription(object_id,
                start_time, end_time, variables);
66          break;
67      default:
68          writeStatusResponse(sub_type, STATUS_NIMPL, "Subscription type
                not implemented: " + std::to_string(sub_type));
69          return;
70      }
71
72      uint8_t result = sub->handleSubscription(temp, true, errors); //
            validate
73
74      if (result == VariableSubscription::STATUS_EXPIRED)
75      {
76          debugPrint("Expired subscription.");
77
78          writeStatusResponse(sub_type, STATUS_ERROR, "Expired
                subscription.");
79          return;
80      }
81      else if (result != VariableSubscription::STATUS_OK)
82      {
```

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

**Código C.6:** Avance de simulación en el módulo `Simulation`.

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

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

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

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

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

         current_simtime = this->getCurrentTimeSeconds();

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

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

**Código C.7:** 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.

```
bool traci_api::Simulation::packSimulationVariable(uint8_t varID,
    tcpip::Storage& result_store)
{
    debugPrint("Fetching SIMVAR " + std::to_string(varID));

    result_store.writeUnsignedByte(RES_GETSIMVAR);
    result_store.writeUnsignedByte(varID);
    result_store.writeString("");
    try
    {
        getSimulationVariable(varID, result_store);
    }
    catch (...)
    {
        return false;
    }
    return true;
}

void traci_api::Simulation::getSimulationVariable(uint8_t varID,
    tcpip::Storage& result)
{
    VehicleManager* vhcman = traci_api::VehicleManager::getInstance();

    switch (varID)
    {
    case VAR_SIMTIME:
        result.writeUnsignedByte(VTYPE_INT);
        result.writeInt(this->getCurrentTimeMilliseconds());
        break;
    case VAR_DEPARTEDVHC_CNT:
        result.writeUnsignedByte(VTYPE_INT);
        result.writeInt(vhcman->getDepartedVehicleCount());
        break;
    case VAR_DEPARTEDVHC_LST:
        result.writeUnsignedByte(VTYPE_STRLST);
        result.writeStringList(vhcman->getDepartedVehicles());
        break;
    case VAR_ARRIVEDVHC_CNT:
        result.writeUnsignedByte(VTYPE_INT);
        result.writeInt(vhcman->getArrivedVehicleCount());
        break;
    case VAR_ARRIVEDVHC_LST:
        result.writeUnsignedByte(VTYPE_STRLST);
        result.writeStringList(vhcman->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_VHCSTARTTELEPORT_CNT:
64      case VAR_VHCSTARTPARK_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_VHCSTARTTELEPORT_LST:
72      case VAR_VHCSTARTPARK_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.8:** Obtención de los límites del escenario de transporte.

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

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

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

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

    float tempX, tempY, tempZ;

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

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

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

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

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

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

**Código C.9:** Implementación de los controladores de velocidad.

```
// Triggers.h
namespace traci_api
{
   class BaseSpeedController
   {
      public:
      virtual ~BaseSpeedController()
      {
      }
      virtual float nextTimeStep() = 0;
      virtual bool repeat() = 0;
   };

   class HoldSpeedController : public BaseSpeedController
   {
      private:
      VEHICLE* vhc;
      float target_speed;

      public:
      HoldSpeedController(VEHICLE* vhc, float target_speed) :
         vhc(vhc), target_speed(target_speed){}
      ~HoldSpeedController() override {}

      float nextTimeStep() override;
      bool repeat() override { return true; }
   };

   class LinearSpeedChangeController : public BaseSpeedController
   {
      private:
      VEHICLE* vhc;
      int duration;
      bool done;

      float acceleration;

      public:
      LinearSpeedChangeController(VEHICLE* vhc, float target_speed,
         int duration);
      ~LinearSpeedChangeController() override {};

      float nextTimeStep() override;
      bool repeat() override { return !done; }
   };
}

// 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
      duration) : vhc(vhc), duration(0), done(false)
76 {
77    /*
78     * calculate acceleration needed for each timestep. if duration
          is too short, i.e.
79     * it causes the needed acceleration to be greater than the
          maximum allowed, we'll use
80     * the maximum for the duration, but we'll never reach the
          desired speed.
81     */
82
83    float current_speed = qpg_VHC_speed(vhc);
84    float diff = target_speed - current_speed;
85    // first, check if we actually need to change the speed
86    // this will do nothing if we don't
87    if (abs(diff) < NUMERICAL_EPS)
88    {
89       done = true;
90       acceleration = 0;
91       return;
92    }
```

```
93
94      float timestep_sz = qpg_CFG_timeStep();
95      float duration_s = duration / 1000.0f;
96      int d_factor = round(duration_s / timestep_sz);
97      this->duration = d_factor * (timestep_sz * 1000);
98
99      acceleration = diff / (duration / 1000.0f); // acceleration (m/s2)
100     if (diff < 0)
101     {
102        /* decelerate */
103        acceleration = max(qpg_VTP_deceleration(qpg_VHC_type(vhc)),
              acceleration);
104     }
105     else
106     {
107        /* accelerate */
108        acceleration = min(qpg_VTP_acceleration(qpg_VHC_type(vhc)),
              acceleration);
109     }
110 }
111
112 float 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 }
```