Omnetpy: Integración del Lenguaje Python en el Entorno de Simulación OMNeT++

Marcos Modenesi

Trabajo Especial de la Licenciatura en Ciencias de la Computación

Facultad de Matemática, Física y Astronomía

Directores: Dr. Juan A. Fraire, Lic. Pablo G. Ventura

Marzo 2022

Omnetpy: Integración del Lenguaje Python en el Entorno de Simulaciónn OMNeT++
por Marcos Modenesi se distribuye bajo una
Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional.

Resumen

En este trabajo elaboramos un paquete que permite el desarrollo de módulos Python para la herramienta de simulación de redes OMNeT++. También analizamos el estado del arte en el uso de Python para la simulación de redes. Durante el desarrollo exploramos formas tanto de extender el intérprete de Python como de embeber el intérprete en OMNeT++, donde se hace necesario incluir un análisis de la arquitectura interna de OMNeT++ y cPython. Además, se detalla el uso de la herramienta utilizando los ejemplos que ofrece OMNeT++ desarrollados en Python. Por último, se realiza una comparación de rendimiento entre el uso de Python y C++ en OMNeT++.

Abstract

In this work we provide a package that enables the devolopment of Python modules to be used in the network simulation tool OMNeT++. We also present the state of the art regarding the usage of Python for network simulation. During the process we explore ways of extending the Python interpreter, as well as embedding it in OMNeT++, for which an analisis of the internal architecture of OMNeT++ and cPython is needed. We offer a detailed description on how to use the package, by reimplementing in Python the sample simulations found in OMNeT++. Finally, the performance of several OMNeT++ models implemented both in Python and C++ is compared.

Índice general

1 Introducción **1**

1.1 Objetivo . . . 1
1.2 Hipótesis . . . 1
1.3 Metodología . . . 1
1.4 Impacto . . . 1
1.5 Estructura . . . 2

2 Estado del arte **3**

2.1 Simuladores de eventos discretos . . . 3
2.2 Simulador OMNeT++ . . . 5
2.2.1 Introducción a OMNeT++ . . . 6
2.2.2 Ecosistema y casos de uso . . . 9
2.3 Otras herramientas similares . . . 10
2.3.1 ns3 . . . 10
2.3.2 NetSim . . . 11
2.3.3 SimPy . . . 11
2.4 C++ vs. Python . . . 12
2.4.1 Generalidades . . . 12
2.4.2 Sintaxis . . . 13
2.4.3 Forma de uso . . . 13
2.4.4 Distancia al hardware . . . 14
2.4.5 Paradigmas soportados . . . 14
2.4.6 Sistema de tipos . . . 15
2.4.7 Ecosistema . . . 15
2.4.8 Manejo de memoria . . . 16
2.4.9 C++ y Python como lenguajes de enseñanza . . . 17

3 Metodología **18**

3.0.1 Integrando C++ y Python . . . 18
3.0.2 Herramientas de más alto nivel para lograr la integración . . . 24
3.0.3 Entorno de trabajo . . . 27
3.0.4 Caso de estudio . . . 28

4 Desarrollo **30**

4.1 Arquitectura de OMNeT++ . . . 30
4.1.1 El ciclo de vida de una simulación . . . 30
4.1.2 Code fragments: ejecución de código genérico . . . 31

4.1.3 El registro global de clases . . . 32
4.1.4 El macro Define_Module . . . 33
4.2 Registrando módulos Python . . . 35
4.2.1 PySimpleModule: heredando cSimpleModule en Python . . . 36
4.2.2 Creación de módulos definidos en Python desde C++ . . . 38
4.2.3 La función de casteo . . . 40
4.2.4 Ciclo de vida del intérprete . . . 41
4.2.5 Juntando todas las piezas . . . 41
4.2.6 El macro Define_Python_Module . . . 42
4.3 Generalización . . . 42
4.3.1 Mas allá de cSimpleModule y cMessage . . . 43
4.3.2 El macro EV . . . 44
4.3.3 WATCH: inspección del estado desde la GUI . . . 45
4.3.4 Recolección de estadísticas . . . 45
4.3.5 El mecanismo de las señales . . . 46
4.4 Limitaciones y dificultades . . . 47
4.4.1 Ciclo de vida de un mensaje . . . 47
4.4.2 El método deleteModule . . . 49
4.4.3 Automatización de la creación de bindings . . . 49
4.4.4 Mayor cubrimiento de la librería original . . . 49
4.4.5 Depuración . . . 50
4.4.6 La función de casteo . . . 50

5 Omnetpy **51**
5.1 Arquitectura del proyecto . . . 51
5.2 Separando omnetpy de OMNeT++ . . . 51
5.3 El módulo pyopp . . . 52
5.4 Compilación del proyecto . . . 52
5.5 Manual de usuario . . . 53
5.5.1 Prerrequisitos . . . 53
5.5.2 Opción 1: generar imagen docker . . . 53
5.5.3 Opción 2: utilizar una imagen pregenerada . . . 54
5.5.4 Escribiendo una simulación con omnetpy . . . 54
5.5.5 Preparando un nuevo proyecto . . . 55
5.5.6 Agregando un archivo NED . . . 55
5.5.7 Agregando un archivo C++ . . . 56
5.5.8 Agregando un archivo makefrag . . . 56
5.5.9 Agregando un archivo Python . . . 57
5.5.10 Correr la simulación . . . 57

6 Evaluación **59**
6.1 Verificación . . . 59
6.2 Rendimiento . . . 60
6.2.1 Aloha . . . 60

6.2.2 Tictoc 61
6.2.3 Conclusión 62
6.3 Complejidad de código 63

7 Conclusiones **66**
7.1 Desafíos abiertos 67
7.1.1 Generación automática de librerías de extensión 67
7.1.2 Depurador interactivo en Python 67
7.1.3 Formas de distribución y disponibilización 68

Bibliografía **69**

1 Introducción

1.1. Objetivo

El objetivo de esta tesis es proveer una interfaz simple y compacta para especificar el comportamiento de modelos en el simulador de eventos discretos OMNeT++ mediante el uso del lenguaje Python, eliminando toda necesidad de conocer C++ para utilizar dicho simulador.

1.2. Hipótesis

Es posible definir módulos simples de OMNeT++ en el lenguaje de programación Python (en reemplazo de C++) e integrarlos en una simulación de manera que el usuario experimente mínima o nula complejidad en el proceso.

1.3. Metodología

La metodología en la que se basa esta tesis es extender y embeber el intérprete de Python para que pueda recibir y transmitir comportamiento desde y hacia C++. A su vez, se hace un análisis de la arquitectura del proyecto OMNeT++ para encontrar formas de intervenirlo y habilitar la integración con Python.

1.4. Impacto

Este trabajo[1] resulta beneficioso para todas aquellas personas que quieren aprovechar la plataforma OMNeT++ para realizar simulaciones, pero para quienes la programación directa en C++ resulta un obstáculo. En particular, facilita la utilización de OMNeT++ como herramienta pedagógica en cursos de ciencias de computación donde las y los estudiantes suelen contar con un dominio bastante bueno de Python, pero en general desconocen C++. El interés de este proyecto ha sido reconocido por el equipo que desarrolla OMNeT++[2].

[1] Disponible en `https://github.com/mmodenesi/omnetpy/`
[2] Ver `https://omnetpp.org/download-items/omnetpy.html`

1.5. Estructura

El capítulo 2 de esta tesis presenta el estado del arte alrededor de OMNeT++ y la simulación de eventos discretos. Se realiza una comparación entre C++ y Python. El capítulo 3 presenta la metodología. El capítulo 4 discute el detalle del desarrollo. La herramienta omnetpy se presenta en el capítulo 5. El capítulo 6 evalúa cualitativa y cuantitativamente omnetpy. Las conclusiones se resumen en el capítulo 7, así como los desafíos abiertos.

2 Estado del arte

2.1. Simuladores de eventos discretos

Un sistema es una colección de entidades que interactúan bajo ciertas reglas para alcanzar un fin común. Decimos que un sistema es *continuo* cuando las variables que permiten describir su estado cambian de forma contínua con respecto al tiempo (por ejemplo, altura y velocidad de un avión). En oposición, los sistemas *discretos* son aquellos en los que las variables de estado cambian de forma instantánea en determinados momentos en el tiempo (por ejemplo, cantidad de clientes en una estación de servicio).

A menudo es de interés estudiar un sistema para entender los parámetros que gobiernan su funcionamiento, realizar mejoras o comprender de qué manera serán impactados por ciertos cambios en sus entradas.

Para llevar a cabo un estudio de esta naturaleza, una primera opción es experimentar con el sistema real. No obstante, esto no siempre es deseable (podría implicar muchos gastos y disrupciones en el funcionamiento normal del sistema) o siquiera posible (quizás el sistema aún no ha sido construido). Entonces se puede recurrir a experimentar con un *modelo* del sistema.

Un modelo puede ser *físico* (modelo a escala) o *matemático*. Este último consiste en una serie de suposiciones sobre el funcionamiento del sistema que se pueden formalizar en un conjunto de relaciones matemáticas y lógicas entre las entidades que lo componen.

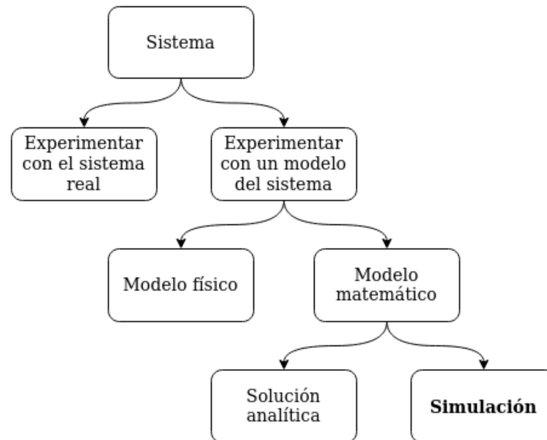
A veces un sistema es lo suficientemente sencillo como para que el modelo matemático del mismo pueda ser resuelto analíticamente, mediante el uso de herramientas tales como el cálculo diferencial, álgebra, teoría de probabilidad. Sin embargo, cualquier modelo más o menos interesante de un sistema no trivial suele ser muy complejo para ser tratado mediante estos métodos. Aparece entonces la simulación como una forma de ganar comprensión sobre el funcionamiento del sistema.

En una simulación, el modelo no es resuelto, sino que es *ejercitado* con el fin de recolectar información a partir de la cual se puedan realizar estimaciones sobre los valores de los parámetros de interés.

La construcción de una simulación ofrece las siguientes ventajas:

- Nuevas políticas, procedimientos de operación, reglas de decisión, flujos de información, procedimientos de organización, etc. pueden ser explorados sin perturbar la operación del sistema real.
- Nuevos diseños pueden ser probados sin asignar recursos a la adquisición de elementos físicos, antes de implementar cambios en el mundo real.
- Permite validar hipótesis y responder a preguntas de tipo "¿qué pasaría si...?".

Figura 2.1: Formas de estudiar un sistema [6]



- El diseño de la simulación permite ganar conocimiento sobre el sistema bajo investigación.
- Permite cambiar los valores de entrada para observar cambios en las salidas, lo cual facilita una mayor comprensión sobre cuáles son las variables más relevantes y cómo interactúan.
- Proporciona una herramienta de gran valor pedagógico.
- Permite verificar soluciones analíticas.
- El tiempo es artificial, y por lo tanto puede ser contraído (simular meses o años de funcionamiento de un sistema en algunos minutos), expandido (observar cambios que de otra forma serían demasiado rápidos) e incluso pausado y avanzado manualmente.
- El concepto básico de una simulación es fácil de interpretar, por lo tanto puede ser una herramienta útil a la hora de justificar decisiones a clientes, inversores, gestores, etc.
- Puede ser más confiable que un modelo matemático resuelto de forma analítica, porque necesita hacer menos simplificaciones.

En los modelos estáticos no es relevante el paso del tiempo. Un ejemplo de simulación de este tipo lo ofrece el método Monte Carlo. En los modelos dinámicos, en cambio, el tiempo es relevante ya que se estudia la evolución del sistema a través de una línea temporal artificial.

La simulación de eventos discretos involucra el modelado de un sistema y su evolución mediante una representación en la cual las variables de estado cambian de forma instantánea en determinados puntos del tiempo. Estos puntos del tiempo son aquellos

en los que un evento ocurre, donde un evento está definido como una ocurrencia instantánea que tiene la potencialidad de cambiar el estado del sistema. Decimos que hay una potencialidad de cambiar el estado del sistema porque un evento puede utilizarse de manera que no cambie el sistema pero que cumpla otros fines. Por ejemplo, se puede programar un evento que dará fin a la simulación en un tiempo particular.

Si bien en principio sería posible llevar a cabo una simulación de eventos discretos a mano, registrando el estado del modelo ante cada nuevo evento, la cantidad de información que habría que manipular para simular cualquier sistema del mundo real hace que esta tarea sólo sea abordada mediante el uso de software especializado.

Un simulador de eventos discretos es un programa destinado a soportar la simulación de sistemas mediante modelos dinámicos y discretos. En dichos programas es común encontrar los siguientes componentes:

- Estado del sistema: colección de variables necesarias para describir el sistema en un punto particular del tiempo.
- Reloj de simulación: una variable especial para llevar cuenta del tiempo.
- Lista de eventos: una lista con los eventos futuros.
- Contadores: variables usadas para almacenar información estadística sobre el desempeño del sistema.
- Fuentes de variables aleatorias para las distribuciones más comunes.
- Herramientas de recolección y manipulación de datos, procesamiento estadístico y representación visual de los mismos.
- Interfaz gráfica que proporciona una representación visual del sistema (en dos o tres dimensiones), mostrando animaciones que ayudan al seguimiento de los cambios de estado.

Las fuentes consultadas en la confección de este capítulo fueron [6], [2], [11].

2.2. Simulador OMNeT++

Objective Modular Network Testbed in C++ (OMNeT++) es un simulador de eventos discretos modular y orientado a objetos, escrito en el lenguaje de propósito general C++. Originado en 1997 por András Varga, es un software de código abierto que puede ser utilizado libremente para usos no comerciales (principalmente investigación y enseñanza). Existe una extensión de OMNeT++ para usos comerciales llamada OMNEST.

Principalmente orientado a la simulación de sistemas de comunicación, multiprocesadores y otros sistemas distribuidos, OMNeT++ es lo suficientemente general como para resultar de utilidad en otros dominios (diseño de protocolos, validación de arquitecturas de hardware, evaluación de desempeño de sistemas de software, y en general, cualquier

sistema donde el enfoque de eventos discretos sea apropiado y pueda ser convenientemente mapeado a un conjunto de entidades que intercambian mensajes).

Su diseño modular hace posible que ciertas piezas centrales de la simulación de redes que no son parte del paquete principal (por ejemplo, librerías de simulación de protocolos TCP) aparezcan espontáneamente desde una comunidad de usuarios muy activa.

OMNeT++ hace mucho énfasis en la facilidad de depuración y trazabilidad de los elementos constituyentes del modelo, para lo cual es posible ejecutar las simulaciones en un ambiente de interfaz gráfica que presenta animaciones para los eventos (mensajes) y brinda facilidad de inspección de los módulos intervinientes.

También es posible ejecutar simulaciones en un ambiente de consola (es decir, sin interfaz gráfica), útil para la ejecución en lote de varios experimentos.

Ofrece soporte para simulaciones de gran escala preparadas para ser paralelizadas en un cluster de computadoras.

Busca apoyarse en interfaces abiertas (archivos de texto plano, formatos comunes) para posibilitar la interacción con otras herramientas.

2.2.1. Introducción a OMNeT++

En esta sección se describen en detalles los diferentes componentes de un modelo de simulación de OMNeT++ desde la perspectiva del usuario.

Modelos y estructura jerárquica En OMNeT++, un modelo consiste de módulos que se comunican mediante el paso de mensajes. Los módulos simples se escriben en el lenguaje de programación C++, utilizando clases de la librería de simulación. Estos módulos simples constituyen el componente atómico (no se pueden subdividir) y activo (es donde el usuario introduce la lógica del modelo) del modelo.

Estableciendo una estructura jerárquica sin límite de niveles, los módulos simples pueden ser agrupados en módulos compuestos, lo cual permite dividir los modelos complejos en partes más pequeñas, o agrupar varias partes en un único módulo. Esta es una decisión de diseño que corre por cuenta del usuario.

Los módulos se comunican mediante el paso de mensajes. Además de una marca de tiempo, los mensajes pueden tener cualquier tipo de atributo. Los módulos presentan compuertas de entrada y de salida que constituyen su interfaz. Los mensajes usualmente se destinan a compuertas, pero también es posible enviarlos directamente a módulos específicos.

Los módulos pueden tomar parámetros, utilizados para pasar configuraciones y definir la topología del sistema. Asimismo, las conexiones entre módulos pueden tener atributos (demora de propagación, tasa de error, tasa de transferencia), permitiendo definir conexiones como tipos (canales) para reutilizarlos.

Los parámetros pueden tomar valores específicos, declarados en archivos de configuración, o pueden ser tomados de una fuente de valores aleatorios.

El lenguaje NED NED es un lenguaje declarativo que fue creado específicamente para OMNeT++, en el cual el usuario define la estructura del modelo (módulos e intercone-

xiones).

Una definición de modelo puede constar de declaraciones de módulos simples, módulos compuestos y definiciones de la red:

- Los módulos simples describen su interfaz: compuertas y parámetros.
- Los módulos compuestos son definidos mediante la declaración de su interfaz externa (compuertas y parámetros) así como la definición de sus submódulos y sus interconexiones.
- La definición de una red típicamente declara al modelo como una instancia de un módulo particular.

Las definiciones NED soportan la partición en diferentes archivos mediante el mecanismo de inclusión.

OMNeT++ incluye un editor gráfico que toma como partida los archivos NED. El usuario puede hacer cambios tanto en la vista gráfica como directamente en el texto del archivo, y cambiar de vista en cualquier momento.

NED es un lenguaje declarativo que contiene constructos parecidos a los condicionales y los bucles de un lenguaje imperativo, lo que posibilita parametrizar las topologías. Esto es una ventaja sobre otros simuladores donde o bien no es posible parametrizar (sólo se pueden expresar topologías fijas) o donde la especificación del modelo se mezcla con la lógica de la simulación, haciendo imposible la edición gráfica.

El lenguaje NED es compatible con el formato XML, lo que posibilita la carga dinámica de modelos que provengan de otras fuentes.

Programación de módulos simples Los módulos simples son los elementos activos del sistema y funcionan como bloques constituyentes de los módulos complejos y, en última instancia, del modelo de simulación.

El usuario implementa la funcionalidad de un módulo simple heredando de la clase `cSimpleModule`, que es parte de las librerías de simulación de OMNeT++ y puede optar por dos paradigmas de programación diferentes:

- basado en corrutinas
- basado en el procesamiento de eventos

En el paradigma basado en corrutinas, el código del modelo corre en su propio hilo, el cual recibe el control del kernel de simulación cada vez que el módulo recibe un mensaje. Típicamente, este código nunca incluye un `return`: sus puntos de entrada son llamadas a `receive` (esperar el siguiente mensaje) y sus puntos de salida son llamadas a `send` (enviar un mensaje)

En el paradigma basado en el procesamiento de eventos, el kernel de simulación llama a una función específica del módulo (`handleMessage`) con el mensaje como argumento. Esta función debe ejecutar un `return` al terminar de procesar el mensaje, efectivamente devolviendo el control al kernel de simulación.

Además de el código destinado al envío y recepción de mensajes, es posible escribir código que se ejecuta durante la inicialización y finalización del módulo (este último, sólo si la simulación terminó exitosamente).

Como parte de la simulación, un módulo puede cambiar sus entradas y salidas, o crear dinámicamente nuevos módulos, lo cual hace posible modificar la topología inicial de la red.

Separación del modelo y los experimentos Como se describió anteriormente, el comportamiento del modelo es capturado en archivos C++ mientras que la topología es capturada en archivos NED.

En una simulación cualquiera, uno está generalmente interesado en conocer cómo se comporta el modelo a partir de diferentes datos de entrada. Esta información no pertenece ni a los archivos C++ ni a los archivos NED: la configuración de los experimentos se coloca en otros archivos, con extensión `ini`. En ellos se puede especificar cuántas corridas de una simulación se desea realizar, con qué valores para los parámetros, con qué distribuciones aleatorias, cuánto tiempo puede durar como máximo la simulación, etc.

Entorno de desarrollo integrado La edición de los diferentes archivos que intervienen (archivos de C++, archivos NED, archivos `ini`) puede realizarse cómodamente desde un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) que es parte de la distribución oficial de OMNeT++. Las ventajas de programar desde este IDE son múltiples:

- Introspección de código, navegación a la definición de una variable, navegación a la documentación.
- Asistencia para operaciones comunes de refactorización de código.
- Acceso rápido a funcionalidades comunes como compilación o ejecución.
- Resaltado de sintaxis de los diferentes lenguajes
- Depuración de código.
- Asistente para la generación de nuevos proyectos.
- Edición de archivos NED desde la vista gráfica.

A su vez, este entorno de desarrollo integrado ofrece la posibilidad de visualizar gráficos de diversos tipos a partir de los datos generados durante una simulación.

Ejecución de una simulación El código de los módulos simples se compila enlazando con las librerías de simulación de OMNeT++ creando un binario que constituye el ejecutable de la simulación. Cambios posteriores a los archivos NED (cambios en la topología y en los parámetros) o en los archivos `ini` (cantidad y duración de las simulaciones, distribuciones de las variables aleatorias, valores de los parámetros) no requieren nuevos pasos de compilación.

El binario de la simulación puede ejecutarse de forma que la misma se corra en un entorno gráfico apto para la visualización y depuración del modelo, o sin este entorno gráfico (más apropiado para correr muchas simulaciones en lote).

2.2.2. Ecosistema y casos de uso

Desde sus comienzos, cerca del año 2000, OMNeT++ ha tenido una amplia recepción en entornos de investigación, educativos, así como también en la industria. Diversos proyectos independientes han aportado nuevas librerías reutilizables y modelos de dominios específicos, algunos de los cuales se listan a continuación[1].

- INET Framework (`https://inet.omnetpp.org`) provee soporte para simulaciones de redes de comunicación que utilizan los protocolos de Internet (TCP, UDP, IPv4, IPv6, OSPF, BGP, etc.), así como protocolos cableados e inalámbricos de la capa de enlace (Ethernet, PPP, IEEE 802.11, etc.) y soporte para redes móviles, protocolos MANET (Mobile Ad Hoc Networks), DiffServ, etc. Muchos otros proyectos toman INET como base y la extienden en direcciones específicas, como redes de vehículos, overlay, P2P, LTE.

- Simu5G (`http://simu5g.org/`) producto de un proyecto de investigación conjunta entre Intel Corporation y el Computer Networking Group de la Universidad de Pisa, Italia, provee soporte para redes 5G NewRado [7].

- SimulLTE (`https://simulte.com/`) provee librerías para la simulación de redes LTE y LTA [21].

- Veins (`https://veins.car2x.org`) es un framework de simulación de código abierto de IVC (Inter-Vehicular Communication) que utiliza OMNeT++ para la simulación de redes así como SUMO, un simulador de tráfico vehicular, usando cosimulación [14].

- CoRE4INET (`https://core-researchgroup.de/projects/simulation.html`) es una extensión a la herramienta INET para simulaciones basadas en eventos de real-time Ethernet dentro del simulador OMNeT++.

Desde el año 2008 en forma continuada, la comunidad de OMNeT++ se reúne anualmente en un evento en el que se comparten experiencias, casos de usos y se realizan “hackatones”, maratones de programación en los que se busca atacar un problema concreto de forma creativa en pocas horas. Algunos de los temas presentados:

[1] Para una lista completa, visitar `https://omnetpp.org/download/models-and-tools`

- Development and Testing of Automotive Ethernet-Networks together in one Tool - OMNeT++ (Patrick Wunner, Stefan May, Kristian Trenkel, and Sebastian Dengler)
- Implementation of an Adaptive Energy-efficient MAC Protocol in OMNeT++ / MiXiM (Van Thiep Nguyen, Matthieu Gautier, and Olivier Berder)
- Attack of the Ants: Studying Ant Routing Algorithms in Simulation and Wireless Testbeds (Michael Frey and Mesut Günes)
- Modeling Quantum Optical Components, Pulses and Fiber Channels Using OMNeT++ (Ryan D. L. Engle, Douglas D. Hodson, Michael R. Grimaila, Logan O. Mailloux, Colin V. McLaughlin and Gerald Baumgartner)
- High Frequency Radio Network Simulation Using OMNeT++ (Jeffery Weston and Eric Koski)
- Stacked-VLAN-Based Modeling of Hybrid ISP Traffic Control Schemes and Service Plans Exploiting Excess Bandwidth in Shared Access Networks (Kyeong Soo Kim)
- DoS Protection through Credit Based Metering - Simulation Based Evaluation for Time-Sensitive Networking in Cars (Philipp Meyer, Timo Häckel, Franz Korf and Thomas Schmidt).

Para una lista completa se recomienda visitar la página oficial de los archivos de la conferencia en https://summit.omnetpp.org/archive/.

Las fuentes consultadas en la confección de este capítulo fueron [19], [20], [18].

2.3. Otras herramientas similares

Esta sección presenta brevemente los principales proyectos que cumplen el rol de simulador de eventos discretos, sus características y sus diferencias con OMNeT++. En particular, se toma el caso de SimPy que es, hasta donde alcanza nuestro conocimiento, el único programa de este tipo basado en el lenguaje Python.

2.3.1. ns3

Herramienta orientada a la simulación de redes de comunicación, dirigido principalmente al uso académico y educativo. Es un proyecto de código abierto, gratuito, liberado bajo la licencia GNU GPLv2 mantenido por su comunidad. Está basado en C++ y parte de la API de simulación cuenta con "bindings" (enlaces) para Python que son generados automáticamente (hasta donde este proceso es automatizable) y cuya completitud y correctitud no es el principal objetivo del proyecto [8] [17].

A diferencia de OMNeT++, la simulación y la visualización (depuración, trazabilidad) son temas separados. Por defecto, no presenta interfaz gráfica para mostrar animaciones de los modelos. Las trazas generadas durante una simulación pueden procesarse en

otro software denominado NetAsim que permite la visualización, o incluso en otros programas no relacionados, como WireShark. Existe también una extensión (PyViz) que permite realizar una visualización interactiva (es decir, durante la simulación) pensada con propósitos de depuración.

A pesar de su filosofía de modularización, esta herramienta no separa los múltiples aspectos de una simulación (lógica, configuración, topología) en diferentes archivos, si no que todo esto se realiza directamente mediante el código C++. Por esta razón, el usuario tiene que proveer más código C++ que en una simulación típica de OMNeT++, ya que el programa incluye desde el tratamiento de los argumentos del programa (si los hubiere), la declaración y configuración de los nodos, los canales, las aplicaciones, el nivel de logging, etc, hasta la misma inicialización y finalización de la simulación.

Es una herramienta que supone que el usuario está muy familiarizado con el entorno de línea de comandos y puede configurar y compilar sus fuentes con facilidad para producir el binario adaptado a sus necesidades.

2.3.2. NetSim

NetSim parece ser un nombre bastante común para este tipo de software. En particular, aquí hablamos de la herramienta producida por la compañía india Tetcos (`https://www.tetcos.com/`). NetSim v1 apareció en 2005 y se encuentra actualmente en la versión 13 (mayo 2021).

Esta herramienta, escrita en lenguaje C, corre sobre plataformas Windows y se distribuye bajo diferentes licencias comerciales. Existe incluso una licencia especial para usos académicos y educativos, que también es paga.

Está orientada a la simulación de redes de comunicación, para lo cual provee una extensa librería con los protocolos y elementos necesarios (de acuerdo al tipo de licencia obtenida).

El principal caso de uso es el de establecer un escenario de simulación ubicando componentes (nodos, aplicaciones, enlaces) en una grilla espacial, y caracterizando los canales de comunicación, para luego correr la simulación y recoger métricas que permitan sacar conclusiones.

El usuario sólo puede implementar sus propios algoritmos o recolectores de datos editando el código fuente de NetSim y volviendo a compilar.

2.3.3. SimPy

Se trata de una librería de código abierto para escribir simulaciones de eventos discretos en lenguaje Python, distribuida bajo una licencia MIT que posibilita su uso libre [12].

La simulación se estructura alrededor del concepto de *proceso*, que constituye el elemento activo de la simulación (un cliente, un coche, un nodo de una red), haciendo uso de generadores.

En Python, un generador es un tipo especial de función que incluye la palabra reservada `yield`. Al ejecutar la instrucción `yield`, la función es pausada y el control retorna

al código que la llamó. Posteriormente, la ejecución de la corrutina se puede reanudar, desde el mismo punto donde cedió el control.

Esta característica del lenguaje Python ofrece un recurso que SimPy utiliza para estructurar la simulación, en un modelo de corrutinas (funciones que ceden el control voluntariamente en lo que se denomina concurrencia cooperativa). Los generadores crean eventos y ceden el control al orquestador de la simulación cuando ejecutan un `yield`. Cuando el evento *ocurre* (llega su momento en el tiempo de la simulación) el proceso continúa.

Simpy incluye también funcionalidad para modelar el uso de recursos compartidos y la interacción de procesos.

Es importante notar que SimPy es simplemente una librería que implementa el código básico para estructurar una simulación, pero no provee muchas de las funcionalidades que las otras herramientas tienen incorporadas (interfaz gráfica, animación, recolección de datos, configuración de experimentos, visualización de resultados, etc.).

2.4. C++ vs. Python

Analizando herramientas de simulación de eventos discretos, lo más común es encontrar que se ha utilizado un lenguaje de bajo nivel como C o C++, siendo SimPy, hecho en Python, un caso excepcional. En esta sección se hace una breve comparación entre C++ y Python, para entender qué se gana y qué se pierde al utilizar cada lenguaje. Esto nos permitirá fundamentar mejor por qué este proyecto toma la iniciativa de permitir la utilización de la herramienta OMNeT++ (escrita en C++) desde código escrito enteramente en Python.

2.4.1. Generalidades

Originado por Bjarne Stroustrup en 1979 como una extensión del lenguaje C que añadía soporte para programación orientada a objetos, C++ fue concebido para la programación de sistemas operativos y sistemas embebidos, así como para software con recursos escasos, con metas tales como performance y eficiencia. Ha sido utilizado en muchos otros contextos [15], como ser aplicaciones de escritorio, videojuegos, servidores, y también en aplicaciones de alta criticidad (enlaces telefónicos o sondas espaciales).

Se trata de un lenguaje compilado, por lo que se puede utilizar en cualquier plataforma para la cual exista un compilador de C++.

En el año 1998, el lenguaje fue estandarizado por la International Standarization Organization (ISO). A dicho estándar siguieron ampliaciones y modificaciones en los años 2003, 2011, 2014, 2017, siendo el último estándar publicado en 2020 en lo que se denomina informalmente C++20.

Python fue originado a finales de la década de 1980 por Guido Van Rossum, y conoció su primera edición oficial en 1991. La versión más reciente del lenguaje, al momento de escribir este documento, es la 3.10, liberada en octubre de 2021. Desde sus inicios, el lenguaje ha puesto énfasis en producir un lenguaje fácil de escribir y de leer, poniendo el acento en que el tiempo de las personas es más valioso que el tiempo del CPU.

El desarrollo y crecimiento del lenguaje es guiado por una organización sin fines de lucro llamada Python Software Foundation.

2.4.2. Sintaxis

C++ hereda su sintaxis del lenguaje C, terminando cada sentencia con punto y coma y delimitando bloques por medio de corchetes.

En general, Python tiene una sintaxis más sencilla, con menos palabras clave y utiliza sangrías para definir bloques de código. Leer un programa en Python muchas veces se puede asemejar a leer un texto en inglés.

2.4.3. Forma de uso

C++ es un lenguaje compilado. El código fuente es finalmente convertido a un formato binario que puede ser ejecutado instrucción por instrucción por el procesador de la computadora. Un paso intermedio realizado por una herramienta conocida como "preprocesador" puede además realizar cambios en el conjunto de caracteres escrito directamente por el programador, insertando otros archivos (directiva `#include`), incluyendo o excluyendo ciertos bloques (directiva `#ifdef`) o también expandiendo lo que comúnmente se conoce como "macros". Esto agrega un nivel de dificultad más a la sintaxis de C++, ya que lo que el programador escribe, no suele ser exactamente igual a lo que el compilador recibe para convertir a binario.

Python no es un lenguaje compilado, como C++ o C, si no que es un lenguaje interpretado: el código fuente no es convertido directamente a código máquina para su ejecución en un procesador. El código escrito en Python debe ser proporcionado a un programa especial que se conoce como "intérprete". Tras una primera etapa de preprocesado (parseo, elaboración del AST, generación de bytecode propio de Python), el intérprete se encarga de llevar adelante la ejecución instrucción por instrucción. Este intérprete es en sí mismo un programa que se llama comúnmente "Python" (o también Python3, Python35, etc., dependiendo de la versión).

No existe un único intérprete de Python. El más popular y difundido, conocido como "cPython", es el intérprete de referencia de Guido Van Rossum (creador del lenguaje) escrito en C. Puede considerarse la implementación oficial: se lo referencia como "Python" y se usa el término "cPython" (como en este párrafo) cuando se busca distinguirlo de otras implementaciones [10].

Notar que el término "Python" tiene (al menos) dos significados:

- Es un lenguaje de programación definido mediante ciertas reglas sintácticas que delimitan las cadenas de caracteres que constituyen un programa válido y las separan de aquellas que, contrariamente, no son programas válidos (ver `https://docs.python.org/3/reference/grammar.html`).
- Es un binario que sirve de intérprete (ejecutor) de ciertos programas escritos usando tales reglas sintácticas.

A lo largo de este trabajo, cuando hablamos de “Python” nos referimos al lenguaje. Para referirnos al binario que ejecuta archivos fuente en este lenguaje, hablamos del “intérprete” o “intérprete de Python”. Más específicamente, nos referimos a la implementación oficial en C (cPython).

2.4.4. Distancia al hardware

El factor que hace que un lenguaje sea considerado de alto o bajo nivel es su distancia a los detalles físicos de la computadora que ejecuta el código. En el nivel más bajo posible, podríamos intentar escribir un programa cuidadosamente disponiendo ceros y unos de forma de componer un archivo binario que llegara a ser ejecutable por el procesador. Un poco por encima de este nivel, podríamos utilizar lenguaje assembly. Si bien aquí no se utilizan ceros y unos directamente, sí se manipulan los registros del microprocesador, se direcciona la memoria manualmente, se opera al nivel de bytes.

En general, en lenguajes de tan bajo nivel, se gana un control extremo sobre el uso del hardware, pero se invierte un tiempo enorme en realizar tareas triviales, y se hace en extremo difícil evitar errores.

Si bien C++ presenta muchas características de lenguaje de alto nivel, en general no hay una gran abstracción sobre los detalles de una computadora. Por el contrario, el lenguaje brinda herramientas para aprovechar esos detalles. Como ejemplo, pensemos que quien programa tiene a su disposición diversos tipos de datos para representar un entero (`char`, `int`, `short int`, `long int`, `long long int`, en sus versiones `signed` o `unsigned`) lo cual permite elegir el más apropiado para cada aplicación (según el rango de valores que se necesite representar, y la cantidad de memoria que se pueda invertir en ello), pero también agrega un nivel de dificultad extra para quien no domina los detalles, o simplemente no necesita concentrarse en ellos. Python es considerado un lenguaje de alto nivel. No busca exponer los detalles internos de la máquina virtual ni de la máquina física subyacente. Esto, por supuesto, puede venir con una penalidad en la performance y una ganancia en la usabilidad del lenguaje.

2.4.5. Paradigmas soportados

C++ es un lenguaje imperativo que puede utilizarse principalmente para escribir programas de paradigma procedimental u orientado a objetos. Asimismo, una característica del lenguaje que son los llamados templates o “plantillas” habilita a formas de programación genérica (código genérico cuya información de tipo será definida en tiempo de compilación).

Python también es un lenguaje imperativo y soporta fuertemente los paradigmas procedimental y orientado a objetos. Si bien no es un lenguaje funcional, incluye características que permiten la adopción de un estilo de programación funcional. Soporta también la programación orientada a aspectos y metaprogramación.

2.4.6. Sistema de tipos

C++ emplea un sistema de tipos estático (toda variable tiene un tipo), explícito (el tipo de la variable debe ser declarado, es parte del código fuente). Como el tipo de cada variable se conoce en tiempo de compilación, muchos errores de tipado son detectados y prevenidos por el compilador.

Python, en cambio, emplea un sistema de tipos dinámico, implícito (son los valores de un programa -y no las variables- los que tienen tipo determinado, pero estos tipos no son expresados como parte del programa). En general, esto conduce a programas más fáciles de escribir y de leer, aunque no permite detectar errores de tipos si no hasta que el código es ejecutado. Esto último, sin embargo, no es visto como un problema en Python, por el contrario, se dice que Python utiliza duck typing [9] (tipado del "pato"), explicándolo con la frase "si camina como un pato y hace el sonido de un pato, entonces es un pato". En la práctica, esto quiere decir que Python favorece el paradigma en el cual un objeto sirve en cualquier circunstancia donde pueda hacer lo que se espera de él, independientemente de si tiene un tipo u otro. No se chequean los tipos de los objetos (de hecho, herramientas del lenguaje que permiten hacerlo como la función "isinstance", generan fuertes opiniones en contra de su uso [13]) si no que se espera que un objeto pueda proveer el comportamiento necesario (interfaz).

2.4.7. Ecosistema

C++, además del núcleo del lenguaje, provee la STL (Standard Template Library), un conjunto de plantillas de clases que implementa muchos de los tipos de datos más populares (listas, vectores, colas, stacks), así como algunos algoritmos.

Python es considerado un lenguaje con "baterías incluidas" por la vastedad de funcionalidad que se encuentra en su librería estándar. En ella encontramos módulos para realizar operaciones comunes (manejo de fechas, establecer comunicaciones http, compresión de archivos, envío de email, operaciones comunes con texto y cadenas de caracteres, manejo de matrices, etc.) de forma simple y sin escribir código especial.

Otros módulos que no son parte de la distribución oficial del lenguaje pueden ser distribuidos e instalados mediante el uso de herramientas tales como `pip`, `pipenv`, `poetry` (manejadores de paquetes) a través de repositorios de paquetes, siendo pypi.org el servidor oficial. Algunos paquetes dentro de esta categoría han llegado a estandarizarse como las librerías por defecto para realizar ciertas funciones:

- visualización de gráficos: maptlotlib, pyplt
- procesamiento numérico: NumPy
- ciencia de datos: Pandas, SciKit-learn, PyTorch, TensorFlow, Keras

2.4.8. Manejo de memoria

En C++, las variables pueden ser alojadas dos tipos de almacenamiento[2]:

- Memoria automática (stack)
- Memoria manual (heap)

Las variables locales de cada función (especialmente cuando son valores que caben en pocos bytes, como algunos enteros, caracteres, etc.) son almacenadas en la pila o stack, y liberadas cuando la función hace `return` (cuando el activation record se remueve del stack). Esta memoria es manejada por el compilador y el programador no necesita preocuparse por su manejo explícito.

Muchas veces, sin embargo, el programador debe recurrir a crear variables en el heap (ya sea porque necesita alojar muchos objetos, o se trata de objetos "grandes", de varios bytes, que no caben en el stack). Este "heap" constituye una fuente de memoria que debe ser primeramente pedida al sistema operativo y luego, cuando ya no es utilizada, debe ser devuelta. Fallar en manejar correctamente este proceso implicaría la terminación inmediata del programa (si trata de acceder a memoria no pedida) o la pérdida de recursos que forman parte del ordenador pero que ya no pueden ser utilizados por ningún otro proceso (al menos, hasta que termine la ejecución del programa). El programador tiene que ser muy cuidadoso en este aspecto y suele ser uno de los elementos más difíciles de la programación en C++.

En Python, el proceso de pedir, utilizar y devolver memoria es manejado enteramente por un subsistema del intérprete o Python Runtime Environment. Al momento de introducir variables o crear objetos, el programador puede confiar que la memoria necesaria será pedida al sistema operativo, y esta será devuelta a su momento. Todo es manejado automáticamente. Un subproceso informalmente conocido como Garbage Collector (recolector de basura) se encarga de gestionar qué porciones de memoria ya no están siendo referenciadas por ninguna variable del programa y pueden, por lo tanto, ser recicladas o sencillamente devueltas al sistema operativo.

En términos de practicidad, es claro que es más fácil programar en Python, donde uno no necesita ni siquiera pensar en qué tipo de memoria están los objetos, en pedir o devolver memoria, en pensar cuántos bytes ocupan, cómo están alineados los objetos en relación a las direcciones de memoria, chequear que la memoria está correctamente utilizada, etc. Todo esto está resuelto por el intérprete. Pero, a su vez, tiene costos:

- El algoritmo del Garbage Collector necesita tiempo para correr, y este tiempo es en detrimento de la ejecución de nuestra aplicación.
- El diseño de la gestión de la memoria busca minimizar el número de llamadas al sistema operativo y suele pedir recursos en bloques grandes, para gestión interna, conduciendo a aplicaciones que generalmente reservan más memoria de la estrictamente necesaria.

[2] Omitimos las variables en memoria estática, aquellas que se declaran e inicializan en tiempo de compilación, por no ser relevantes para esta discusión

Otra razón por lo que los programas hechos en Python suelen usar más memoria que programas equivalentes hechos en C++ es que en Python, todo es un objeto, incluso valores de tipos básicos como puede ser un entero, internamente son representados mediante objeto Python, lo cual ocupa mucho más que 4 bytes.

2.4.9. C++ y Python como lenguajes de enseñanza

Numerosos estudios comparativos entre Python y C++ [22][1] muestran que el primero es menos eficiente que el segundo (en términos de tiempo y espacio), en general por varios órdenes de magnitud. Sin embargo, se señala que los programas hechos en Python son más fáciles de entender y modificar. En general, las mismas características de C++ que lo hacen un lenguaje poderoso para administrar los recursos disponibles, son aquellas que lo convierten en un lenguaje más difícil, propenso a errores en manos no expertas. Se ha señalado que Python es un lenguaje más amigable para novatos y más apropiado para la enseñanza de conceptos en ciencias de la computación [4].

3 Metodología

3.0.1. Integrando C++ y Python

Existen diversas formas documentadas en las que Python y C (o C++) pueden interactuar y utilizarse conjuntamente. A continuación se exploran dichos mecanismos.

Siguiendo la documentación oficial [3], es común clasificar las integraciones entre C/C++ y Python en dos grupos:

- **Extender el intérprete**: invocar código escrito en C/C++ desde un programa escrito en Python (fig 3.1).
- **Embeber el intérprete**: Invocar código originalmente escrito en Python desde un programa escrito en C/C++ (fig 3.2).

Figura 3.1: Extender el intérprete

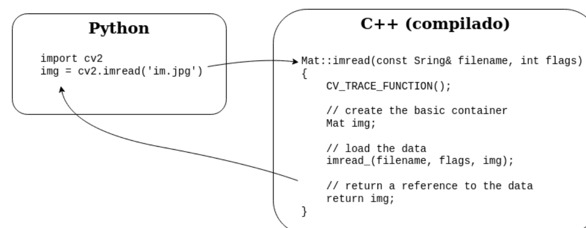
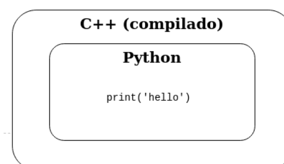


Figura 3.2: Embeber el intérprete



Cuando extendemos el intérprete, el programa principal es el intérprete de Python que adquiere, a través de este mecanismo, la posibilidad de realizar llamadas a módulos escritos en C/C++. Esto brinda principalmente dos beneficios: por un lado, delegar parte de la computación en código de mejor performance (procesamiento numérico, imágenes) y, por otro, la posibilidad de realizar llamadas a las APIs del sistema en forma directa. Esta es una de las formas en que muchos módulos propios de Python están escritos. [1]

[1] Ver por ejemplo https://github.com/python/cpython/tree/master/Modules)

Cuando embebemos al intérprete, en cambio, el programa principal es otra aplicación que usa al intérprete de Python como una librería más. Esta librería tiene la capacidad de leer y ejecutar código fuente escrito en lenguaje Python. Si bien se trata de un enfoque menos común, se puede utilizar como una manera de agregar funcionalidad de scripting a sistemas escritos en C/C++ (por ejemplo, configuración dinámica o elaboración de plugins). Notar que esto tiene cierta afinidad con nuestro objetivo, que es lograr que una aplicación hecha en C++ invoque código escrito en Python.

Para que el lector pueda tomar dimensión del nivel de dificultad que conlleva la tarea de hacer que Python y C/C++ trabajen en conjunto (en cualquiera de las dos direcciones) se brindan a continuación ejemplos de ambos tipos de integración.

Extendiendo Python con C/C++ Supongamos que queremos escribir en C una función que compute el cuadrado de un número, y queremos que dicha función sea accesible al intérprete de Python. Una implementación simple en C podría lucir así:

```
1  int square(int x)
2  {
3      return x * x;
4  }
```

Nuestro objetivo sería poder escribir código Python como el siguiente:

```
1  import example
2  print(38 + example.square(2))   # 42
```

Cuando el código Python invoca la función square tienen que ocurrir tres cosas:

1. El parámetro 2, que es un objeto Python debe ser convertido al tipo `int`.
2. El parámetro debe ser pasado a la función y el CPU debe ejecutar el código binario generado a partir del código fuente C.
3. El valor de retorno (un `int` con el valor 4) debe ser convertido a un objeto Python.

Para comprender por qué esto es así, debemos recordar en primer lugar que Python es un lenguaje interpretado. El código fuente debe ser convertido en una serie de instrucciones para la máquina virtual de Python. En nuestro caso, dichas instrucciones son:

```
1        0 LOAD_CONST               0 (0)
         2 LOAD_CONST               1 (None)
         4 IMPORT_NAME              0 (example)
         6 STORE_NAME               0 (example)

2        8 LOAD_NAME                1 (print)
        10 LOAD_CONST               2 (38)
        12 LOAD_NAME                0 (example)
```

```
         14 LOAD_METHOD              2 (square)
         16 LOAD_CONST               3 (2)
         18 CALL_METHOD              1
         20 BINARY_ADD
         22 CALL_FUNCTION            1
         24 POP_TOP
         26 LOAD_CONST               1 (None)
         28 RETURN_VALUE
```

A continuación, esta serie de instrucciones son ejecutadas una a una. Esto es muy diferente al código binario que genera una compilación de un archivo C.

Otra gran diferencia es que el valor 2 en Python es un objeto bastante más complejo que en el mundo de C. En cPython, toda variable es una versión especializada del tipo base que corresponde a `PyObject`.

Entonces, la tarea de extender el intérprete con código hecho en C/C++ consiste no sólo en escribir funcionalidad nueva en estos lenguajes, si no de realizar todo el trabajo necesario para exponerlo al intérprete de Python.

El siguiente es un ejemplo completo de cómo sería la implementación de nuestro módulo.

```c
/* Inclusión de las librearías de Python,
 * para usar PyObject, PyArg_ParseTuple, etc... **/
#define PY_SSIZE_T_CLEAN
#include <Python.h>

/* Definimos la función que computa el cuadrado de un entero */
static PyObject *square(PyObject *self, PyObject *args) {
  int input;
  if (!PyArg_ParseTuple(args, "i", &input)) {
    return NULL;
  }

  return PyLong_FromLong((long)input * (long)input);
}

/* Debemos incluir el nombre y la dirección de la función
 * en una tabla de métodos */
static PyMethodDef ExampleMethods[] = {
    {
        "square",
        square,
        METH_VARARGS,
        "Returns a square of an integer"
    },
/* Valor "sentinela", señala el final del arreglo */
    {NULL, NULL, 0, NULL},
};


/* Definición del módulo, para que podamos
 * importarlo desde el intérprete de Python */
static struct PyModuleDef examplemodule = {
    PyModuleDef_HEAD_INIT,
    "example",
    "example module containing square() function",
    -1,
    ExampleMethods,
```

```
38  };
39
40  /* Esta función será llamada cuando se inizialice el módulo
41   * en el intérprete de Python */
42  PyMODINIT_FUNC PyInit_example(void) {
43    PyObject *m = PyModule_Create(&examplemodule);
44    return m;
45  }
```

Notar que la función `square` ahora tiene los tipos que exporta el archivo de encabezados `Python.h`. Notar también que, además de la definición de la función, se escribe código extra que representa la definición del módulo donde esta función vive.

Si bien los detalles de compilación varían de acuerdo a las características del sistema (lo cual no es relevante en este momento), así podríamos producir un archivo `example.so` que puede ser importado como librería desde el intérprete:

```
# g++ -O3 -Wall -fPIC -I/usr/local/include -I/usr/include/Python3.6m -c example.cpp -o out.o
# g++ -shared out.o -L/usr/local/lib -o example.so

# Python3
>>> import example
>>> example.square(9)
81
```

Intérprete embebido En este escenario el programa principal es una aplicación hecha en C/C++. Una de las tareas de esta aplicación será la de inicializar (y finalizar) un intérprete a través del cual pueda acceder a la funcionalidad desarrollada en Python. El siguiente ejemplo, tomado de la documentación oficial, da muestra de este enfoque. Supongamos que contamos con la siguiente definición de función hecha en Python y almacenada en el archivo `mul.py`:

```
1  def multiply(a, b):
2      print('Will compute', a, 'times', b)
3      c = 0
4      for i in range(0, a):
5          c = c + b
6      return c
```

El siguiente programa en C permitirá, una vez compilado, ejecutar dicho código:

```
1   #define PY_SSIZE_T_CLEAN
2   #include <Python.h>
3
4   int
5   main(int argc, char *argv[])
6   {
7       PyObject *pName, *pModule, *pFunc;
8       PyObject *pArgs, *pValue;
9       int i;
10
```

```c
    /* chequeo de argumentos */
    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    /* inizialización del intérprete */
    Py_Initialize();

    /* Se omite el chequeo de errores de pName */
    pName = PyUnicode_DecodeFSDefault(argv[1]);

    /* se importa el módulo */
    pModule = PyImport_Import(pName);

    /* la variable pName ya no es necesaria */
    Py_DECREF(pName);

    if (pModule != NULL) {
        /* obtenemos una referencia a la función */
        pFunc = PyObject_GetAttrString(pModule, argv[2]);

        if (pFunc && PyCallable_Check(pFunc)) {
            /* construimos los argumentos que le pasaremos a la función */
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                PyTuple_SetItem(pArgs, i, pValue);
            }
            /* llamada a la función */
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n", PyLong_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
        }
        else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();

```

```
72            fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
73            return 1;
74        }
75        if (Py_FinalizeEx() < 0) {
76            return 120;
77        }
78        return 0;
79    }
```

Para la compilación, los detalles varían de acuerdo a las características del sistema, pero algo parecido a

```
# g++ -I/usr/include/Python3.6m -c cal.c
# g++ cal.o -lPython3.6m -o cal
```

genera un binario llamado `cal`, que permite hacer lo siguiente:

```
# ./cal mul multiply 12 4
Will compute 12 times 4
Result of call: 48
```

Notar que la primera línea de salida proviene del archivo Python, mientras que la segunda, del archivo C.

Extensiones e intérprete embebido en nuestro trabajo Ahora que contamos con marco de referencia para pensar las interacciones entre C/C++ y Python, podemos plantearnos la pregunta: ¿cuál de los dos enfoques es el apropiado para nuestro objetivo de extender OMNeT++ usando Python?

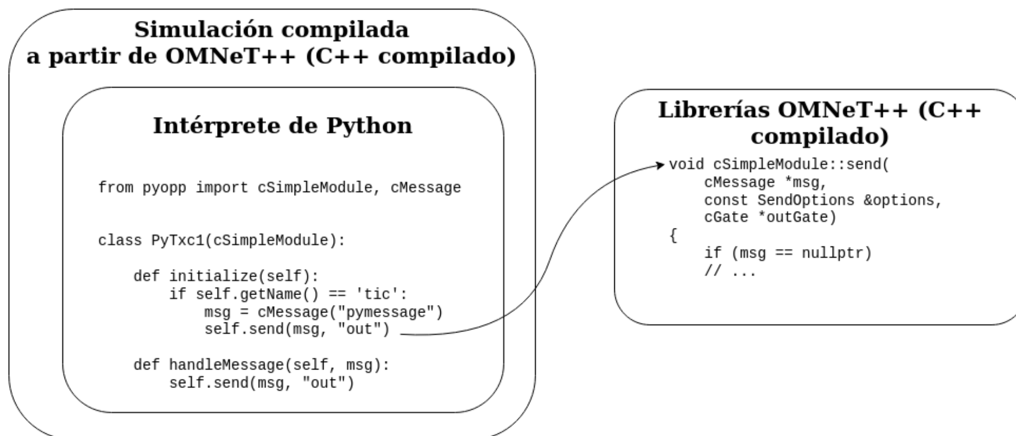
Al extender (enfoque del tipo 1) el programa principal es el intérprete de Python, que hace uso de código escrito en otro lenguaje. Al embeber el intérprete (enfoque del tipo 2), el programa principal es escrito en C/C++ y se encarga de inicializar (y también finalizar) un intérprete de Python para acceder a funcionalidad escrita en este lenguaje. Sin duda, este último parece más alineado con lo que queremos realizar. No obstante, ambos enfoques son necesarios:

Debemos extender el intérprete de Python para que usuarios puedan heredar de `cSimpleModule` al especificar el funcionamiento de sus módulos. El sistema sólo acepta módulos que son subclases de aquella. Si esta tarea de especificación ha de realizarse en Python, como nos proponemos, entonces es menester habilitar la escritura de algo como `from omnetpp import cSimpleModule` para comenzar a definir los módulos a partir de dicha clase base. Se sigue que al menos parte del código de OMNeT++ debe ser expuesto al mundo de Python.

Debemos embeber el intérprete de Python en OMNeT++ para que la aplicación principal siga siendo el binario de simulación que se obtiene al compilar un proyecto tradicional de OMNeT++. Si esto no fuera así, deberíamos reescribir, entre otras cosas, el código de la interfaz de usuario de las simulaciones. Para conservar

toda esa funcionalidad, queremos que el binario que genera OMNeT++ cambie lo menos posible, pero que entre esos cambios se encuentre la capacidad de cargar módulos y ejecutar clases escritas en Python. Esto impone la necesidad de inicializar y finalizar un intérprete de Python (a su debido tiempo).

Figura 3.3: Integrando OMNeT++ con Python



Esto plantea dos desafíos, que son el núcleo del trabajo realizado:

1. ¿Cómo exponer al intérprete de Python toda la funcionalidad preexistente en las librerías de simulación, para que el usuario pueda escribir módulos simples en Python?

2. ¿Cómo intervenir el proyecto OMNeT++ de forma que instancie y ejecute un intérprete de Python embebido en él?

3.0.2. Herramientas de más alto nivel para lograr la integración

Como se aprecia en los ejemplos de código provistos hasta el momento (extensión e intérprete embebido), la tarea dista de ser trivial, a pesar de tratarse apenas de funciones pequeñas con argumentos de tipos básicos. Es de esperar grados de complejidad mucho mayores cuando el código involucre estructuras más sofisticadas del lenguaje (funciones con número variable de argumentos, argumentos con nombre, argumentos con valor por defecto, manejo de errores y excepciones, clases, etc.).

Por otro lado, a diferencia de lo hecho en el primer ejemplo, donde escribimos una función ad-hoc con el sólo objetivo de exponerla en Python, muchas veces se busca exponer en Python varios miles de líneas de código que nunca fueron hechos con este objetivo en mente, donde cada función, cada clase, debe ser envuelta con una capa que traduzca los argumentos con los tipos que envía el “mundo” de Python en argumentos de C++ y viceversa (con los valores de retorno).

Un ejemplo de esta problemática puede encontrarse en la generación de los bindings para Python de OpenCV (librería hecha en C++, especializada en algoritmos de procesamiento de imágenes). En How OpenCV-Python bindings are generated? [2] leemos:

> In OpenCV, all algorithms are implemented in C++. But these algorithms can be used from different languages like Python, Java etc. This is made possible by the bindings generators. These generators create a bridge between C++ and Python which enables users to call C++ functions from Python. (...) So extending all functions in OpenCV to Python by writing their wrapper functions manually is a time-consuming task. So OpenCV does it in a more intelligent way. OpenCV generates these wrapper functions automatically from the C++ headers using some Python scripts (...)

En particular, el proyecto OpenCV resuelve el problema mediante scripts propios que a partir de los archivos de encabezado de C++ pueden producir el código necesario para exponer las clases y funciones al intérprete.

Existen en el ecosistema de la creación de bindings y wrappers varios proyectos que buscan resolver este problema de forma más genérica. Algunos de ellos son:

- swig (`http://swig.org`): herramienta para la generación de código mediante el cual se puede conectar una aplicación hecha en C o C++ con una variedad de lenguajes de programación. Típicamente se utiliza para parsear el código en C o C++ y proveer el código necesario para exponerlo en el lenguaje de destino.

- boost (`https://www.boost.org/doc/libs/1_77_0/libs/python/doc`): librería hecha en C++ con el objetivo de proveer integración entre C++ y Python. Facilita la tarea de exponer proyectos hechos en C++ como módulos de Python disminuyendo la cantidad de código que el usuario debe escribir para generar los bindings. Hay que mencionar que es una pequeña parte de un proyecto mucho más abarcativo formado por numerosas librerías para C++.

- pybind11 (`https://pybind11.readthedocs.io/en/stable/`): Comparte el objetivo (y mucha de la sintaxis) con boost.Python. Intenta diferenciarse de aquella en que es una librería formada enteramente por encabezados (archivos .h). Su código es más escueto y en ocasiones permite simplificar la sintaxis en relación a boost.Python.

- binder (`https://cppbinder.readthedocs.io/en/latest/about.html`): Intenta solucionar la generación automática del código necesario para exponer C++ a Python utilizando la librería pybind11.

- cpppy (`https://cppyy.readthedocs.io/en/latest/`): Haciendo uso de las librerías del proyecto LLVM permite ejecutar código C++ desde Python sin la necesidad de generación previa de bindings ni del uso de un lenguaje intermedio.

[2] `https://docs.opencv.org/master/da/d49/tutorial_py_bindings_basics.html`

El código en C++ se provee directamente como strings a funciones que producen LLVM IR (LLVM Intermediate Representation, una especie de assembly que usa LLVM antes de pasar a código máquina de una arquitectura específica) y se compila en tiempo de ejecución [5].

Para dar idea al lector del tipo de beneficio que se obtiene al emplear estas herramientas, se provee a continuación una reimplementación del segundo ejemplo (embeber el intérprete) utilizando pybind11.

Recordemos que tenemos la siguiente definición de función en Python, en el archivo mul.py

```
1  def multiply(a, b):
2      print('Will compute', a, 'times', b)
3      c = 0
4      for i in range(0, a):
5          c = c + b
6      return c
```

El siguiente código en C++, en el archivo cal.cpp

```
1  #include <iostream>
2  #include <pybind11/embed.h>
3
4  namespace py = pybind11;
5
6  int main() {
7      // start interpreter and keep it alive
8      py::scoped_interpreter guard{};
9
10     // import mul.py
11     py::module m = py::module::import("mul");
12
13     // call function multiply
14     py::object result = m.attr("multiply")(12, 3);
15
16     std::cout << "Result is " << result.cast<int>() << std::endl;
17     return 0;
18 }
```

puede ser compilado con

```
# g++ -O3 -Wall -std=c++11 -fPIC `Python3 -m pybind11 --includes` cal.cpp -o cal -lPython3.6m
```

y ejecutado

```
# ./cal
Will compute 12 times 3
Result is 36
```

obteniendo los mismos resultados que antes, pero con un código muchísimo más escueto y sencillo.

Se eligió la herramienta pybind11 para lograr la interacción entre Python y C++ por resultar la más sencilla de utilizar y por ser su documentación muy completa. Por otro lado, como se puede apreciar en el repositorio oficial [3], resulta ser un proyecto activamente mantenido y con mucho uso en la comunidad.

Si bien resulta muy atractivo el proyecto binder y su promesa de generar código para exponer el proyecto usando pybind11 de forma automática, el objetivo de este trabajo no fue nunca la automatización del proceso de escritura de las librerías, si no encontrar una manera de extender OMNeT++ utilizando Python.

El proyecto cpppy resulta muy atractivo por el hecho de no tener que escribir código intermedio alguno, pero al momento de este trabajo parece tener poca actividad y su documentación no es tan clara y extensa.

3.0.3. Entorno de trabajo

La primera etapa del trabajo consistió en familiarizarse con el código de OMNeT++. Para ello se procedió a descargar su código fuente del repositorio público que se encuentra en github. Al momento de comenzar el trabajo, la última versión estable era la 5.5.1.

Una vez obtenido el código fuente, se procedió a la compilación del mismo. Para ello se siguieron las instrucciones provistas en la documentación del proyecto y en numerosas fuentes web.

Para facilitar la repetición de esta experiencia (así como el uso de la versión modificada de OMNeT++ que permitiera una integración con Python) se utilizó desde un principio el desarrollo dentro de contenedores Docker. Se procedió a la definición de una imagen base que tuviera todas las dependencias de OMNeT++ preinstaladas y que permitiera fijar la versión de todos y cada uno de los componentes involucrados (compilador, librerías, Python, etc.). Esta fue una decisión muy acertada que permitió no sólo la repetibilidad de los resultados, si no también minimizar posibles fuentes de error o incertidumbre dado el control casi total del entorno de desarrollo.

Herramientas y versiones El siguiente es un listado de los principales componentes de software que intervinieron en el desarrollo de este trabajo:

- OMNeT++ 5.5.1
- Linux 1902cc2cffcf 5.3.0-29-generic
- Ubuntu 18-10 (Cosmic)
- g++ (Ubuntu 8.3.0-6ubuntu1 18.10.1) 8.3.0
- GNU Make 4.2.1
- Python 3.6.8
- pybind11 2.4.3

[3] https://github.com/pybind/pybind11

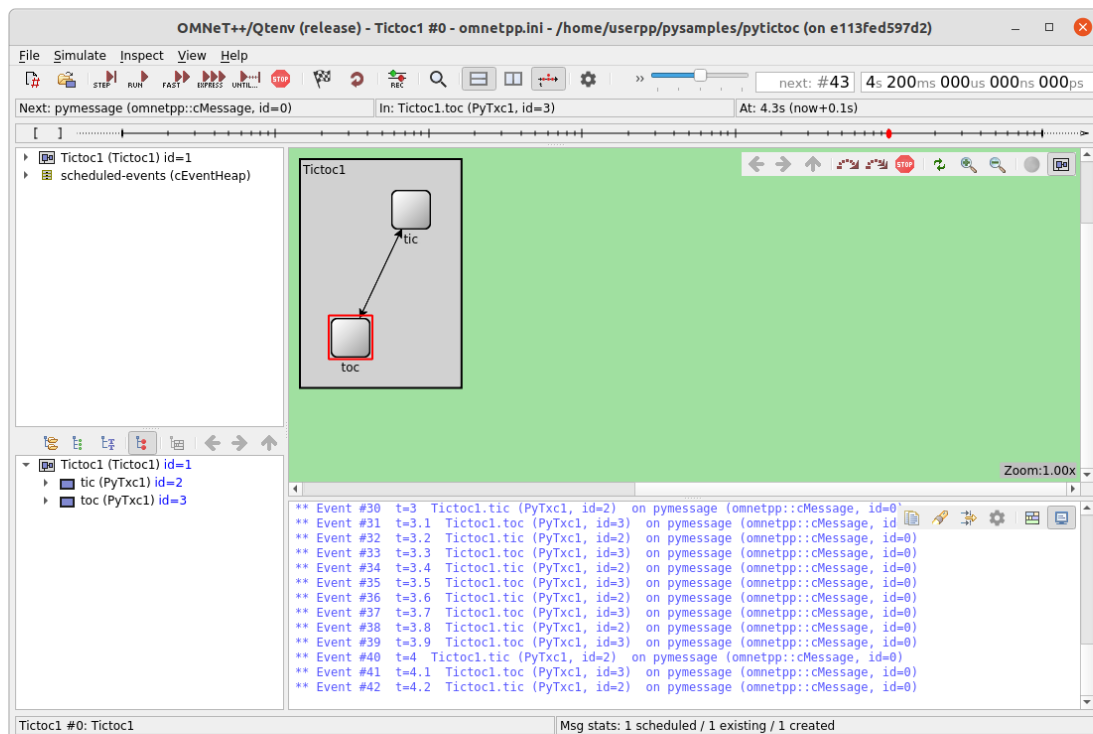
En una etapa final del proyecto se diseñaron mecanismos para probar nuestro código contra distintas versiones de OMNeT++ y se comenzó a utilizar Ubuntu 19.10 con Python 3.7.5, sin dificultades observables.

3.0.4. Caso de estudio

La documentación oficial de OMNeT++ provee un tutorial en siete partes que va complejizando gradualmente una simulación llamada "tictoc" [4]. De esta forma se busca que los nuevos usuarios puedan familiarizarse con las funcionalidades principales que ofrece la herramienta.

La simulación "tictoc" (si bien conoce variaciones a lo largo del tutorial) consiste esencialmente en una red con dos módulos (llamados tic y toc) que intercambian un mismo mensaje ad infinitum (ver fig. 3.4).

Figura 3.4: Ejecución gráfica de la simulación "tictoc".



Por la sencillez inicial y la complejidad incremental de esta simulación, se usó como guía de todas las pruebas. Se estableció como primer objetivo poder escribir los módulos del tutorial tictoc en Python. Esto permitió concentrarse en las necesidades generales que permitieran la integración funcional de OMNeT++ con las clases escritas en Python.

[4] https://docs.omnetpp.org/tutorials/tictoc/

Sólo una vez que se consiguió el objetivo primario, se prosiguió ampliando la integración para permitir la implementación de módulos de mayor complejidad. En una segunda etapa, la mayoría de las muestras en el directorio samples fueron portadas a Python. Asimismo, se implementaron en Python algunos trabajos prácticos de la cátedra Redes y Sistemas Distribuidos de FaMAF.

4 Desarrollo

4.1. Arquitectura de OMNeT++

Tomemos como punto de partida la siguiente definición de módulo simple en C++ (obviando la implementación de los métodos que no son relevantes en este momento).

```
#include <omnetpp.h>

using namespace omnetpp;

class Txc1 : public cSimpleModule
{
    protected:
        virtual void initialize() override;
        virtual void handleMessage(cMessage *msg) override;
};

// The module class needs to be registered with OMNeT++
Define_Module(Txc1);
```

Luego de compilar de dicho modelo junto al kernel de simulación, el binario logra crear instancias del módulo `Txc1` (tantas como los archivos NED indiquen en la descripción de la topología de la red) e invocar a sus métodos. Nuestro próximo objetivo es entender los mecanismos puestos en juego por OMNeT++ para lograr esto. Se exponen a continuación los principales elementos que contribuyen a entender la solución encontrada.

4.1.1. El ciclo de vida de una simulación

Cuando el usuario ejecuta el binario de una simulación el inicio de la ejecución ocurre en la función main donde únicamente se llama a `omnetpp::envir::evMain` con los argumentos recibidos del sistema operativo. En esta última función se llama a `setupUserInterface`, nuevamente pasando los argumentos de invocación del programa.

Esta función presenta tres etapas claramente señaladas mediante comentarios en el código:

1. SETUP (preparación): procesamiento de la configuración, instanciación de una interfaz de usuario (gráfica o de consola).

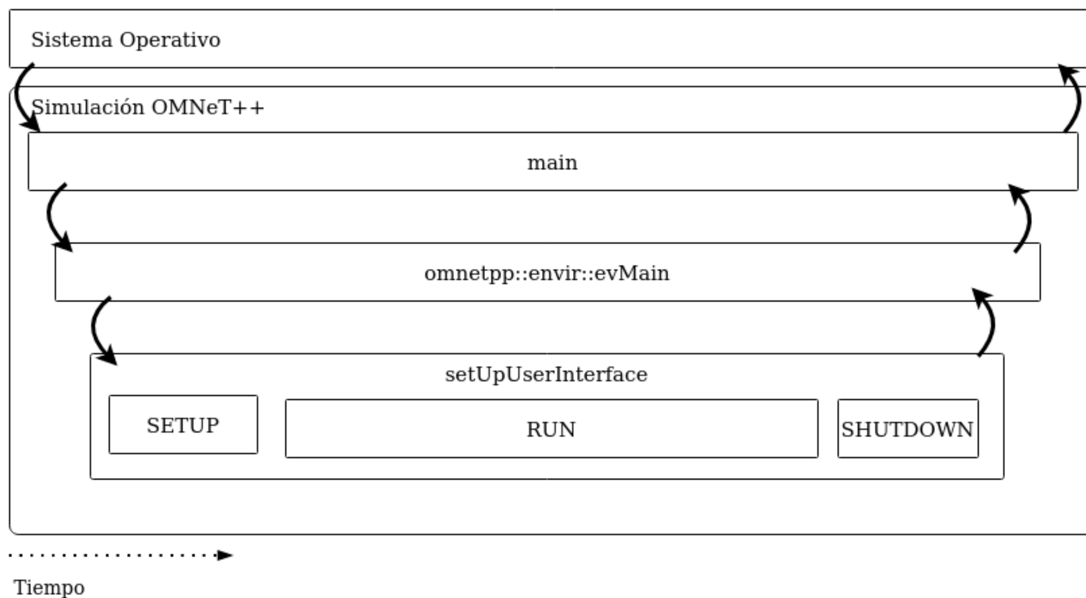
2. RUN (ejecución): el entorno creado (qtenv, o cmdenv o tkenv) es efectivamente puesto a correr. Los tres tipos de ambientes comparten mucho código y terminan llamando a código común en una clase base para correr la simulación. Aquí se

consumen los eventos (mensajes) que los módulos van generando y enviando. Si se acaban los eventos o se acaba el tiempo designado para la simulación, esta finaliza.

3. SHUTDOWN (finalización): limpieza final, antes de devolver el valor de retorno de la simulación.

Nos interesa particularmente lo que ocurre en las etapas de SETUP y SHUTDOWN, por lo que podemos obviar los llamados que ocurren en RUN. La figura 4.1 permite entender mejor lo expuesto y dónde estamos poniendo el foco de atención.

Figura 4.1: Ciclo de vida de una simulación



4.1.2. Code fragments: ejecución de código genérico

Existe en OMNeT++ una clase denominada `CodeFragments`, la cual analizamos a continuación.

Una instancia de `CodeFragments` se inicializa con dos argumentos:

1. un puntero a una función de tipo `void -> void`.
2. una instancia de un tipo enumerado que representa cuándo debe ser ejecutada tal función:
   - `CodeFragments::STARTUP` para designar a las funciones a correr durante la etapa de preparación
   - `CodeFragments::SHUTDOWN` para designar a las funciones a correr durante la etapa de finalización

A su vez, `CodeFragments` tiene atributos de clase (`head` y `next`) que sirven para implementar una lista enlazada global. Cada vez que se inicializa una variable de tipo `CodeFragments` esta se coloca como cabeza de la lista y almacena un puntero al siguiente elemento (la última de las instancias que se crearon antes que ella), es decir que se inserta por delante y hace crecer la lista. Esta estructura de datos se conoce como “stack” o pila.

Existe también la función de clase `CodeFragments::executeAll`, que toma un tipo (ya sea `CodeFragments::STARTUP` o `CodeFragments::SHUTDOWN`) y recorre toda la lista, ejecutando las funciones de aquellos `CodeFragments` que tienen el tipo especificado.

Cerca del inicio de la simulación, durante la estapa de STARTUP se ejecuta

```
CodeFragments::executeAll(CodeFragments::STARTUP);
```

mientras que cerca del final de la simulación, durante la etapa de SHUTDOWN, se ejecuta:

```
CodeFragments::executeAll(CodeFragments::SHUTDOWN);
```

Este es un dato muy importante, ya que nos proveerá un mecanismo para manejar el ciclo de vida del intérprete de Python embebido, lo cual será abordado en la sección 4.2.4

4.1.3. El registro global de clases

A la hora de construir la red para lanzar la simulación, OMNeT++ va creando recursivamente los módulos y sus submódulos a partir de su nombre (proveniente de los archivos NED). Es necesario pasar de un simple string como `"Txc1"` o `"Tictoc"` a un objeto de la clase apropiada.

Para tal fin, OMNeT++ cuenta con la clase `cRegistrationList`, la cual implementa un mapeo de tipo `std::string -> omnetpp::cOwnedObject*`. Para lograr que tal mapeo sea único y que sea accesible desde cualquier punto del programa, la clase `cGlobalRegistrationList` implementa una variación del patrón de diseño conocido como “singleton”. En efecto, no hay una única instancia de `cRegistrationList`, si no que hay una especializada para cada tipo de objeto. En particular, existe una de estas `cGlobalRegistrationLists` llamada `classes` donde se mapea de nombres de tipos (por ej, `"Txc1"`, `"Tictoc"`, etc.) a instancias de `omnetpp::cObjectFactory`.

Cada instancia de `cObjectFactory` sabe cómo crear objetos de un tipo determinado. Almacena una función de creación, una función de casteo y una descripción en forma de string.

Poniendo todas estas piezas juntas, lo que OMNeT++ hace para obtener una instancia de un módulo, en la etapa de construcción de la red (se omite el manejo de errores):

- llama al método de clase `cObjectFactory::createOne(className)`
- el método de clase busca en el `cGlobalRegistrationList` llamado `classes` al `cObjectFactory` apropiado para tal nombre de clase

- una vez identificado el `cObjectFactory` adecuado, se llama a la función de creación que este contiene.

Todo este andamiaje es transparente para el usuario. Su única responsabilidad es definir un módulo que herede de `cSimpleModule`, y presentarlo al sistema mediante `Define_Module`. Todos estos detalles sobre cómo hace OMNeT++ para instanciar su clase están ocultos.

Es importante descubrir, entonces, quién provee el `cObjectFactory` que sabe construir y castear instancias del módulo definido por el usuario, y además, quién registra ese `cObjectFactory` en la instancia global de `cGlobalRegistrationList` llamada `classes`.

4.1.4. El macro `Define_Module`

Luego de la definición de la clase `Txc1`, para su uso en una simulación es mandatorio escribir `Define_Module(Txc1);`. Sin esta instrucción, OMNeT++ simplemente no toma conocimiento del módulo agregado por el usuario.

`Define_Module` parece a simple vista una función, sin embargo es un macro, cuya definición luce así:

```
1  #define Define_Module(CLASSNAME)  __REGISTER_CLASS(CLASSNAME, \
2    omnetpp::cModule, "module")
```

Esta definición involucra al macro `__REGISTER_CLASS`, que a su vez está construido sobre otro macro:

```
1  #define __REGISTER_CLASS(CLASSNAME, BASECLASS, DESC) \
2    __REGISTER_CLASS_X(CLASSNAME, BASECLASS, DESC, /*nothing*/ )
```

Expandiendo recursivamente todos los macros involucrados (por ejemplo, utilizando la opción `-E` del compilador) y formateando el espaciado para mejorar la legibilidad se logra ver que `Define_Module(Txc1);` termina convirtiéndose en:

```
1  static omnetpp::cObject *__factoryfunc_13()
2  {
3      omnetpp::cModule *ret = new Txc1;
4      return ret;
5  }
6
7  static void *__castfunc_13(omnetpp::cObject *obj)
8  {
9      return (void*)dynamic_cast<Txc1*>(obj);
10 }
11
12 namespace
13 {
14     void __onstartup_func_13()
15     {
16         omnetpp::classes.getInstance()->add(
17             new omnetpp::cObjectFactory(
```

```
                omnetpp::opp_typename(typeid(Txc1)),
                __factoryfunc_13,
                __castfunc_13,
                "module"));
    }

    static omnetpp::CodeFragments __onstartup_obj_13(
        __onstartup_func_13,
        omnetpp::CodeFragments::STARTUP);
}
```

Analicemos un poco el resultado.

`static omnetpp::cObject *__factoryfunc_13()`: define una "factory function", encargada de la creación de nuevas instancias de `Txc1`.

`static void *__castfunc_13(omnetpp::cObject *obj)`: define una función de casteo, que a partir de un puntero a una instancia de `omnetpp::cObject` devuelve el resultado de reinterpretar ese puntero como un puntero a instancia de `Txc1`.

`void __onstartup_func_13()`: define una función que registra ante OMNeT++ la existencia de un nuevo tipo de módulo simple llamado `Txc1`, junto con las funciones de creación y casteo definidas ad hoc.

Se hace notar que los nombres incluyen el número "13" porque en el archivo original, `Define_Module(Txc1);` estaba escrito en la línea 13. Con esto se busca diferenciar funciones en el potencial caso de tener diversos usos del macro `Define_Module` en el mismo archivo. Efectivamente encontramos que, de ejecutarse la función `__onstartup_func_13`, OMNeT++ pasaría a tener conocimiento de un módulo llamado `"Txc1"` y contaría con un mecanismo para crear instancias del mismo.

Notar que hasta el momento se trata de definiciones de funciones, pero ninguna de ellas ha sido invocada. Lo que sigue es diferente:

```
static omnetpp::CodeFragments __onstartup_obj_13(
        __onstartup_func_13,
        omnetpp::CodeFragments::STARTUP);
```

Esto es una declaración e inicialización de una variable estática. Como tal, su inicialización ocurre al comenzar la ejecución del programa. Es decir que antes de que la función `main` sea invocada, esta llamada a constructor ya se ejecutó y por lo tanto la instancia de `CodeFragments` que se acaba de crear ya se encuentra formando parte de la lista global de `CodeFragments`. Por lo tanto, el resultado final de la instanciación de esta variable es la colocación de `__onstartup_func_13` de para ser ejecutada durante la etapa de STARTUP de la función `setupUserInterface`. Más aún, cuando esa función se ejecute, el resultado es que se registra ante OMNeT++ un tipo de módulo simple nuevo (`Txc1`) así como las funciones necesarias para su creación y casteo. Esto permitirá a OMNeT++ crear y manipular instancias de `Txc1` cuando sea necesario (según lo dicten los archivos de descripción de la topología de la red).

En resumidas palabras, el macro `Define_Module(T)` (donde `T` es la clase definida por el usuario) sirve para:

1. definir una función (F1) de creación de objetos del tipo `T`
2. definir una función (F2) de casteo a objetos del tipo `T`
3. definir una función (F3) que registra ante OMNeT++ el tipo `T` junto con las funciones F1 y F2.
4. definir una variable de tipo `CodeFragments` que al ser estática se inicializa antes de llamar a la función principal del programa. Esto último garantiza que F3 se ejecuta durante la etapa de STARTUP de `setupUserInterface` y, como consecuencia, cuando llega el momento de instanciar a la clase `Txc1`, OMNeT++ ya puede hacerlo.

4.2. Registrando módulos Python

Luego de conocer el ciclo de vida de la simulación en OMNeT++, los mecanismos que se activan a la hora de declarar un módulo escrito por el usuario (puestos en juego por el macro `Define_Module`) así como el andamiaje provisto por `CodeFragments` y las llamadas

```
CodeFragments::executeAll(CodeFragments::STARTUP);

CodeFragments::executeAll(CodeFragments::SHUTDOWN);
```

que realiza la función `setupUserInterface`, se empieza a definir una estrategia para la implementación de módulos directamente en Python.

En una primera etapa, buscamos definir una serie de funciones y definiciones que hagan en su conjunto lo mismo que el macro anteriormente estudiado. Convertir todo ese código en un macro lo más sencillo posible sería bueno (para la comodidad del usuario), pero por el momento se posterga.

Buscamos:

- Una función que al ser invocada (sin argumentos) devuelva un puntero a una instancia de `PySimpleModule` convertido a puntero a instancia de `cModule`.
- Una función de casteo a objetos del tipo `PySimpleModule`.
- Una función que sea registrada (como un `CodeFragment`) para ser ejecutada durante la etapa de SETUP de `setupUserInterface` para registrar este nuevo tipo ante OMNeT++.

A su vez, es necesario inyectar en el ciclo de vida de OMNeT++ la inicialización y finalización del intérprete de Python.

Por último, es necesario que el usuario pueda especificar clases en Python heredando de una clase base que sea la versión Python de `cSimpleModule`, de forma de poder realizar los casteos a los tipos de C++ que espera OMNeT++.

4.2.1. `PySimpleModule`: heredando `cSimpleModule` en Python

Comenzamos a trabajar en extender el intérprete de Python con las clases de OMNeT++. En una primera etapa se trabajó en exponer sólo aquello que era estrictamente necesario para lograr la implementación de la primera parte del tutorial tictoc: las clases `cSimpleModule` con los métodos `getName` y `send`, y la clase `cMessage`.

También fue en esta etapa en que se trabajó efectivamente con pybind11 por lo que fue también un momento de exploración de esta herramienta (cuya elección probó ser acertada).

Se incluye aquí la primera versión exitosa del código que se utilizó para exponer estas clases, con el objetivo de que el lector tenga noción del tipo de trabajo que se estaba realizando.

```
1  #include <pybind11/pybind11.h>
2  #include "omnetpp/csimplemodule.h"
3  #include "omnetpp/cmessage.h"
4
5  using namespace omnetpp;
6  namespace py = pybind11;
7
8  /* una clase que hereda de cSimpleModule (OMNeT++)
9   * para exponer algunos métodos virtuales */
10 class cSimpleModulePublicist : public cSimpleModule {
11 public:
12     using cSimpleModule::handleMessage;
13     using cSimpleModule::initialize;
14     using cSimpleModule::send;
15     using cSimpleModule::getName;
16 };
17
18 /* la clase que va a ver el intŕeprete de Python */
19 class PycSimpleModule : public cSimpleModule {
20 public:
21     using cSimpleModule::cSimpleModule;
22
23     void initialize() override {
24         PYBIND11_OVERLOAD(void, cSimpleModule, initialize, );
25     }
26
27     void handleMessage(cMessage *msg) override {
28         PYBIND11_OVERLOAD(void, cSimpleModule, handleMessage, msg);
29     }
30     ~PycSimpleModule() {}
31 };
32
33 /* definición del módulo */
34 PYBIND11_MODULE(pyopp, m) {
35
36     m.doc() = "The Python binding for OMNeT++";
37
38     // cMessage
39     py::class_<cMessage> py_cMessage(m, "cMessage");
40     py_cMessage.def(
41         py::init<const char*, short>(),
42         py::arg("name") = nullptr, y::arg("kind") = 0);
43
44     // cSimpleModule
```

```
45      py::class_<
46          cSimpleModule,
47          PycSimpleModule> py_cSimpleModule(m, "cSimpleModule");
48      py_cSimpleModule.def(
49          py::init<unsigned>(),
50          py::arg("stacksize") = 0);
51      py_cSimpleModule.def(
52          "handleMessage",
53           &cSimpleModulePublicist::handleMessage);
54      py_cSimpleModule.def(
55          "initialize",
56          (void (cSimpleModule::*)()) &cSimpleModulePublicist::initialize);
57      py_cSimpleModule.def(
58      "getName", &cSimpleModulePublicist::getName);
59      py_cSimpleModule.def(
60          "send",
61          py::overload_cast<cMessage*,
62                            const char*,
63                            int>(&cSimpleModule::send),
64          py::arg("msg"), py::arg("gatename"), py::arg("gateindex") = -1);
65  }
```

Luego de la compilación de ese archivo en C++ se obtiene un módulo en Python que expone las clases `cSimpleModule`, `cMessage`, así como algunos de sus métodos. Es decir, estamos en condiciones de escribir en Python algo como:

```
1   from pyopp import cSimpleModule, cMessage
2
3   class PyTxc1(cSimpleModule):
4
5       def initialize(self):
6           if self.getName() == 'tic':
7               msg = cMessage('pymessage')
8               self.send(msg, 'out')
9
10      def handleMessage(self, msg):
11          self.send(msg, 'out')
```

Como puede verse en el siguiente uso interactivo, estas clases no son útiles fuera del contexto de una simulación:

```
# python3
>>> from pyopp import cSimpleModule, cMessage
>>> sm = cSimpleModule()
>>> sm.handleMessage(cMessage('hello'))
<!> Error during startup/shutdown: Global simtime_t variable found, with value 0. Global
↪  simtime_t variables are forbidden, because scale exponent is not yet known at the time they
↪  are initialized. Please use double or const_simtime_t instead. Aborting.
```

No obstante, ya contamos con la extensión necesaria para que el código escrito por el usuario herede en Python de la clase `cSimpleModule` (escrita en C++). A continuación exploramos cómo integrar estas clases en una simulación.

4.2.2. Creación de módulos definidos en Python desde C++

Para lograr que OMNeT++ sepa cómo instanciar esta clase definida en Python el paso siguiente fue definir la factory function necesaria. Una primera versión de la misma luce así:

```
static omnetpp::cObject *F1()  // factory function
{
    // importamos el módulo donde está definida la clase
    auto py_module = py::module::import("txc1");

    // Instanciamos la clase, obteniendo un py::object
    py::object obj = py_module.attr("PyTxc1")();

    // conversión a cSimpleModule (la clase base)
    cSimpleModule *tmp = obj.cast<cSimpleModule *>();

    // conversión a la clase base de cSimpleModule
    cObject *ret = dynamic_cast<cObject*>(tmp);
    return ret;
}
```

Esta factory function es esencialmente lo que buscamos, excepto por dos motivos. El primero de ellos es la conversión intermedia a `cSimpleModule`, necesaria porque C++ y pybind11 no conocen la relación entre nuestra clase definida en Python y `cObject`. En una futura versión, en la que el módulo expuesto a Python incluye toda la jerarquía de clases base de `cSimpleModule` (la cual tiene a `cObject` en la raíz), este paso intermedio puede eliminarse.

El segundo motivo, mucho más complejo y difícil de resolver es que al terminar el scope de la función, el objeto Python que acaba de ser creado es eliminado de la memoria por el intérprete de Python. Para entender por qué esto es así, necesitamos hablar de cómo Python maneja la memoria y el ciclo de vida de sus variables.

Recordemos que en C++, los principales lugares donde las variables pueden existir son:

el stack (manejo automático): variables declaradas sin manejo explícito de la memoria y son liberadas tan pronto como salen de scope (el bloque de código que las contiene). Para alojar una variable en el stack, el compilador debe conocer su tamaño. Además, el espacio del stack es bastante limitado.

el heap (manejo manual): una fuente mucho mayor de memoria, administrada por el sistema operativo. Las variables cuya memoria se pide y se devuelve explícitamente tienen un ciclo de vida independiente del stack y el compilador no necesita conocer su tamaño.

Naturalmente, Python aloja todos los objetos creados por el usuario en el heap. A su vez, con el propósito de liberar la memoria de aquellas variables que ya no se utilizan, cada objeto Python contiene información de cuántas referencias hay a él en un programa (variables que sirven para alcanzarlo). Digamos que en el ejemplo:

```
a = 'hola'
b = [a]
```

El objeto apuntado por la variable a es referenciado en dos lugares (por la variable `a` y como primer elemento de la lista `b`). Tan pronto como hacemos:

```
b[0] = 'chau'
a = 1
```

el objeto de tipo string `'hola'` que el intérprete de Python alojó en memoria dinámica ya no es necesario, puesto que no existen referencias a él. No hay manera de que el programa vuelva a utilizarlo. Es susceptible de ser eliminado para devolver la memoria al sistema operativo.

El algoritmo que el intérprete utiliza para reconocer este tipo de situaciones (cuándo es momento de destruir un objeto y devolver su memoria) se conoce como "reference counting". Básicamente consiste en incrementar la cuenta cada vez que se crea una referencia y decrementarla cuando se destruye o sobreescribe una referencia. Cuando la cuenta se vuelve cero el objeto se puede borrar. No existen garantías de que el objeto sea borrado inmediatamente, pero el garbage collector puede hacerlo a partir de ese momento.

Con esta información, no es difícil ver que existe una única referencia al objeto creado en la línea

```
py::object obj = py_module.attr("PyTxc1")();
```

Tan pronto como la variable `obj` (que apunta a un objeto Python en el heap pero está en el stack) sale de scope y es destruida, el intérprete reconoce que ya no hay variables apuntando a dicho objeto (desconoce el casteo a `cSimpleModule*` o `cObject*`) y lo borra (devuelve la memoria al sistema operativo). Por su lado, el programa OMNeT++ continúa utilizando la referencia `ret` que ahora apunta a memoria que es del sistema operativo. Tan pronto como trate de dereferenciarla, tendremos un `Segmentation Fault` y el sistema operativo terminará el programa.

La solución implementada fue sencillamente incrementar el contador de referencias de forma manual antes de que termine la función.

```
    // Instanciamos la clase, obteniendo un py::object
    py::object obj = py_module.attr("PyTxc1")();
    obj.inc_ref();
```

Esto evita que el intérprete devuelva la memoria al sistema operativo y permite que OMNeT++ utilice la referencia devuelta por la función sin incurrir en un comportamiento ilegal.

Sin embargo, esto genera un problema simétrico: ahora el objeto Python no será liberado nunca. Incluso cuando C++ destruya la instancia de cObject que le devolvimos, el objeto Python seguirá existiendo, ya sin referencias asociadas a él. Para evitar esto se procedió a decrementar el contador de referencias en el destructor de la clase base que expusimos a Python:

```
~PycSimpleModule() {
    py::object obj = py::cast(this);
    obj.dec_ref();
}
```

Irónicamente, el código del módulo generado por pybind11 se da cuenta de el objeto C++ que se está borrando es un wrapper a un objeto Python. Al llegar a cero la cuenta de referencias de dicho objeto, el wrapper debe ser eliminado también. Este sería el comportamiento correcto si el programa principal fuera Python y se están invocando librerías escritas en C++. En nuestro caso, es un "double delete" o un segundo intento de borrar el objeto.

Borrar un objeto que ya ha sido borrado y cuya memoria ya ha sido devuelta al sistema operativo es lo mismo que intentar borrar (liberar) una porción de memoria que ya no nos pertenece. Es otra falta que se penaliza con la terminación inmediata del programa.

Afortunadamente para nosotros, pybind11 provee un mecanismo para generar código que no invoca al destructor del objeto C++ una vez que el objeto Python es liberado.

En resumidas cuentas, el problema se solucionó aplicando las siguientes reglas:

1. La cuenta de referencias del objeto Python creado en la factory function que será llamada por OMNeT++ debe ser incrementada manualmente, para evitar que el intérprete decida liberar su memoria tan pronto como la función retorna.

2. Cuando OMNeT++ libera el objeto C++ que en realidad es el wrapper de un objeto Python, la cuenta de referencias del objeto Python debe ser decrementada para permitir que su memoria sea liberada.

3. Pybind11 no debe eliminar el objeto C++ que envuelve al objeto Python cuando este último sea liberado.

Este problema del ciclo de vida de los objetos volverá a aparecer más adelante con los mensajes que se crean en un módulo con destino a otro módulo. El escenario es ligeramente diferente y la solución encontrada, también. Se lo describe más adelante.

4.2.3. La función de casteo

El único uso que se hace de la función de casteo es en el método de `cObjectFactory`

```
/**
* Returns true if the given object can be cast
* (via dynamic_cast) to the class represented by
* this factory object, and false otherwise.
*/
virtual bool isInstance(cObject *obj) const  {
    return castFunc(obj)!=nullptr;
}
```

el cual utiliza la posibilidad o imposibilidad de castear el objeto a la clase declarada como signo de que el objeto en cuestión es o no es instancia de la misma. Nuestra primera versión de la función de casteo es:

```
static void *F2(omnetpp::cObject *obj) {
    return (void*)dynamic_cast<PycSimpleModule*>(obj);
}
```

Notar que esta función devolverá `true` para cualquier objeto que haya sido implementado en Python heredando de `cSimpleModule`. No sirve para identificar las clases derivadas en Python.

No obstante, no seguimos trabajando en una función de casteo más elaborada por considerarlo innecesario. Inspeccionando el código de OMNeT++ se puede constatar que la función `isInstance` se llama sólamente para validar el tipo del tercer argumento de la función cComponent::emit. Dicho tercer argumento es de tipo `cObject*` y tiene como valor por defecto nullptr. No encontramos ejemplo de su uso en la documentación ni en ninguno de los samples.

4.2.4. Ciclo de vida del intérprete

Resta proveer un mecanismo que asegure que un intérprete de Python está vivo durante el tiempo que esto es estrictamente necesario. Es decir, desde antes de la primera instanciación de nuestros módulos hasta después de la destrucción del último de ellos.

Se logró instanciando un `pybind11::scoped_interpreter` (sólo uno) que se mantiene vivo a lo largo de toda la aplicación.

```
class InterpreterManager {
  private:
    static void* interpreter;
  public:
    static void ensureInterpreter();
};
```

La implementación de `ensureInterpreter` simplemente valida que `interpreter` sea un puntero nulo (lo cual sirve para identificar la primera vez que llaman a la función).

Una solución alternativa podría ser agregar la línea

```
pybind11::scoped_interpreter guard{};
```

al comienzo de la función `main` de OMNeT++. Esta solución no nos seduce porque queremos dejar el código original de OMNeT++ lo más intacto posible.

4.2.5. Juntando todas las piezas

En resumidas cuentas, este es el código que hay que escribir para que OMNeT++ levante nuestra implementación del módulo desde un archivo Python:

```
1  #include <omnetpp/omnetpy.h>
2  #include <pybind11/pybind11.h>
3
4  static omnetpp::cObject *F1() {
5      auto py_module = pybind11::module::import("txc");
6      pybind11::object obj = py_module.attr("PyTxc")();
```

```
    obj.inc_ref();
    return obj.cast<omnetpp::cObject *>();
}

static void *F2(omnetpp::cObject *obj) {
    return (void*)dynamic_cast<PycSimpleModule*>(obj);
}

void F3() {
    InterpreterManager::ensureInterpreter();
    omnetpp::classes.getInstance()->add(
    new omnetpp::cObjectFactory("PyTxc", F1, F2, "module"));
}

static omnetpp::CodeFragments cf(F3, omnetpp::CodeFragments::STARTUP);
```

4.2.6. El macro Define_Python_Module

No es difícil ver que el código anterior, necesario para registrar módulos Python, es esencialmente siempre el mismo, con excepción del nombre del módulo así como el nombre del archivo donde este vive.

Con el objetivo de facilitar el registro de módulos hechos en Python, en paralelo al macro `Define_Module`, se creó un macro `Define_Python_Module`. En esencia, esto es todo el código C++ que un usuario debe escribir para lograr el mismo resultado que en el ejemplo anterior:

```
#include <omnetpp/omnetpy.h>

Define_Python_Module("txc", "PyTxc");
```

Donde la clase `PyTxc` está definida en el archivo `txc.py`, y se asume que el mismo puede ser importado sin errores (esto es, `import txc` no produce errores).

4.3. Generalización

Con el desarrollo presentado hasta aquí en esta sección, queda logrado el objetivo de hacer que OMNeT++ instancie clases definidas en Python y las acepte como implementación de módulos de una simulación. El trabajo hecho para extender el intérprete (exponer las clases de C++ como módulos Python) incluyó lo mínimo necesario para poder correr las versiones más simples del tutorial oficial tictoc.

A continuación, se prosiguió escribiendo en Python el resto del tutorial tictoc. Este tutorial tiene unas 17 simulaciones progresivamente más complejas (aún sobre la idea de dos módulos que se intercambian el mismo mensaje).

En cada iteración se fueron agregando nuevas funcionalidades que debían ser expuestas a Python (es decir, los bindings debían ser extendidos para incluir más código de OMNeT++ y hacerlo accesible desde Python).

Algunas nuevas funcionalidades se resolvieron simplemente agregando una definición en el módulo de los bindings, otras presentaron desafíos mayores (borrado de mensajes, logging).

Poco a poco, los principales features de OMNeT++ fueron soportados en un módulo Python:

- recolección de parámetros desde los archivos de configuración
- logging (el macro `EV`)
- emisión de señales
- manipulación de gráficos en la UI
- `WATCH`

Luego de concluir las simulaciones propuestas en el tutorial tictoc se comenzó a implementar en Python las simulaciones provistas en el directorio samples (contiene diversos ejemplos diseñados para mostrar el potencial de la herramienta). Esto permitió seguir ampliando la superficie de código expuesta a Python.

Asimismo, se encontraron algunas limitaciones para las cuales no se encontró una solución, las cuales se encuentran detalladas en la sección 4.4.

4.3.1. Mas allá de `cSimpleModule` y `cMessage`

Las clases de OMNeT++ más elementales para una simulación son las dos mencionadas en el título. `cSimpleModule` es la clase que el usuario debe extender para construir sus agentes de simulación, y `cMessage` es la clase utilizada para producir eventos que empujen el desarrollo de la simulación. Estas son las únicas clases empleadas en las primeras versiones del tutorial tictoc y, por lo tanto, fueron las primeras que se expusieron a Python en una primera etapa exploratoria. Una vez que se vio que el enfoque adoptado fue exitoso, se procedió a exponer a Python una mayor superficie de la librería de simulación de OMNeT++. Qué exponer y en qué orden fue organizado en torno a desarrollar en Python los mismos ejemplos que se proveen en el directorio `samples` del proyecto original. Esto permitió también ir comparando los resultados de las simulaciones hechas en C++ con las simulaciones hechas en Python.

Este proceso de migración paulatina de los ejemplos (samples) a Python permitió enriquecer notablemente el rango de posibilidades de la librería `pyopp` (el binding expuesto a Python), añadiendo funcionalidad esencial para un simulador de eventos discretos:

- recolección de métricas (clases `cStatistic`, `cStdDev`, `cAbstractHistogram`, `cHistogram`, `cHistogramStrategy`, `cOutVector`),
- acceso al tiempo de simulación (clase `cSimTime`),
- manipulación de la representación gráfica de la simulación (clases `cCanvas`, `cDisplayString`)

- parametrización de experimentos (clase `cPar`)
- cambios dinámicos en la organización del modelo (clase `cTopology`, `cGate`, `cChannel`, `cDataRateChannel`)

4.3.2. El macro `EV`

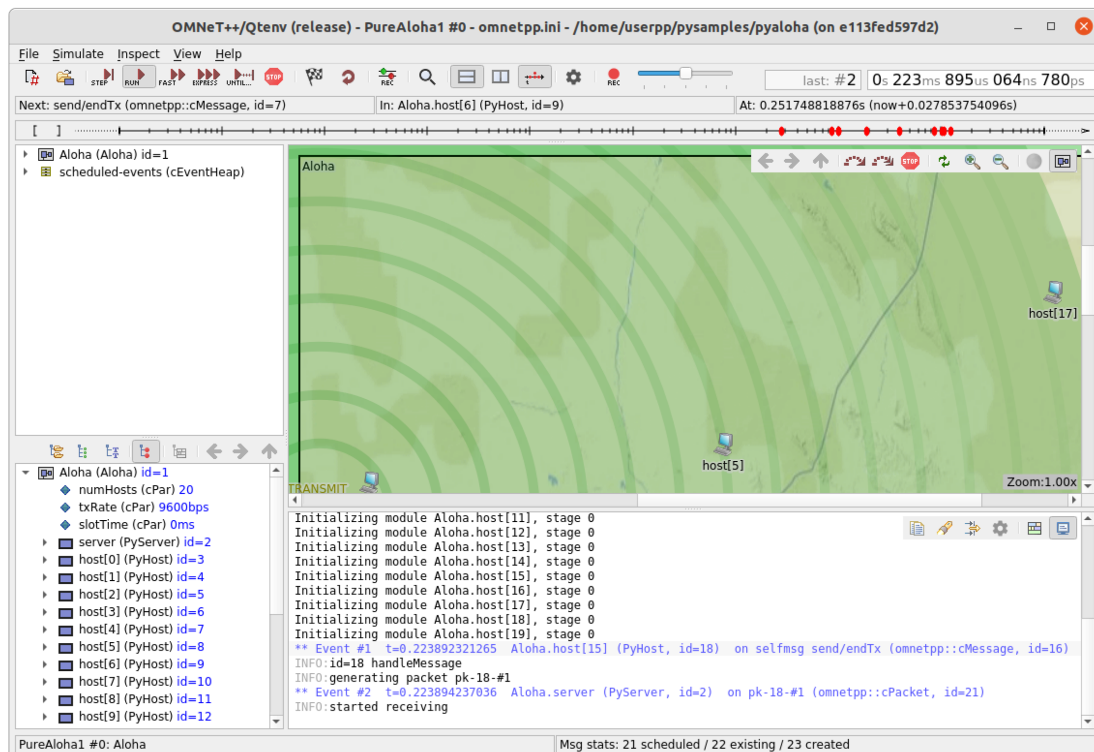
Existe en OMNeT++ algo que luce como un objeto (aunque es en realidad un macro) que sirve para generar mensajes desde el código. Estos mensajes se verán en la interfaz gráfica de la simulación en un tablero especial destinado a tal fin.

Por ejemplo, el siguiente código (samples/aloha/Host.cc):

```
EV << "generating packet " << pkname << endl;
```

produce este efecto en la simulación (ver sección inferior derecha de la fig. 4.2).

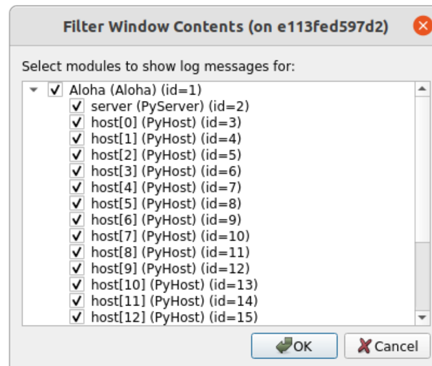
Figura 4.2: Uso de `EV` en simulación aloha



Este macro hace mucho más que simplemente aceptar una cadena de caracteres: captura el contexto de quién envía el mensaje, de forma que en la interfaz gráfica se pueden filtrar mensajes por su origen, como se aprecia en la figura 4.3

A su vez, `EV` es un alias de `EV_INFO`, y existen `EV_WARN`, `EV_DEBUG`, `EV_DETAIL`, etc.

Figura 4.3: Filtrado de mensajes



Al portar esta funcionalidad a Python se buscó que la sintaxis fuera lo más similar posible:

```
from pyopp import EV
...
pkname = "pk-%d-#%d" % (self.getId(), self.pkCounter)
EV << "generating packet " << pkname << '\n'
```

Este objeto EV se hace cargo de investigar el stack de Python para descubrir quién emitió el mensaje y crear así un objeto usando la verdadera API de OMNeT++ (oculta al usuario de `omnetpy`, para simplificar).

4.3.3. `WATCH`: inspección del estado desde la GUI

`WATCH` es otra herramienta de OMNeT++ que apunta a facilitar la trazabilidad de las simulaciones. Se trata de un mecanismo que permite declarar interés en observar permanentemente el estado de cierta variable o cierto atributo de un módulo. Suponiendo que existe la variable `counter`, su uso desde el código sería sencillamente:

```
WATCH(counter)
```

Para acceder a la información de la variable en tiempo real desde el entorno gráfico de la simulación, podemos seleccionar *Inspect* → *Find / Inspect objects*, y utilizar los filtros disponibles hasta dar con lo que nos interesa, como se muestra en la figura 4.4.

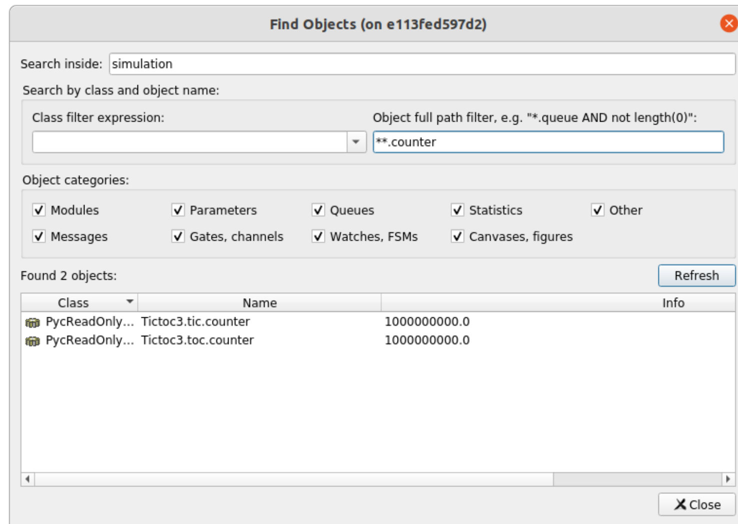
Esta funcionalidad también se expuso a Python, con la sola diferencia de que hay que pasar un string con el nombre de la variable que queremos monitorear:

```
WATCH('counter')
```

4.3.4. Recolección de estadísticas

Los datos generados durante una simulación (los sucesivos tiempos de arribo, de demora, paquetes perdidos, paquetes corruptos, etc.) pueden ser recolectados:

Figura 4.4: Inspección de objetos



- como pares de (tiempo, valor) como una serie temporal
- como valores atemporales, utilizados para calcular media, varianza, etc.

Estos datos son recogidos ya sea en cada invocación del método `handleMessage`, o bien en el método `finish`, el cual se suele aprovechar para escribir los datos a un archivo en disco.

Los datos generados son escritos en archivos en el subdirectorio results dentro del directorio de la simulación, y pueden posteriormente ser analizados utilizando el IDE (entorno de desarrollo integrado).

Las clases necesarias para esta actividad (`cHistogram`, `cOutVector`) también fueron expuestas a Python.

4.3.5. El mecanismo de las señales

OMNeT++ provee un mecanismo más refinado para la recolección de estadísticas dentro de una simulación, que permite remover de los módulos el código específico para registrar los datos en variables o archivos, así como calcular las diversas medidas de interés antes de finalizar la simulación. Esto brinda una mayor separación de responsabilidades, y permite que los módulos se enfoquen en el comportamiento y no en la observación del comportamiento (lo cual muchas veces cambia según el interés de quien realiza la simulación).

El mecanismo alternativo consiste en señales. Estas tienen un nombre y cada vez que se emiten, llevan un valor asociado. Quienes estén interesados en observar el valor de dicha señal se pueden suscribir en los archivos de configuración y de topología.

Declaración de la emisión de una señal de parte del módulo y de cómo debe ser escuchada (NED):

```
simple Txc16
    {
        parameters:
            @signal[arrival](type="long");
            @statistic[hopCount](
                title="hop count";
                source="arrival";
                record=vector,stats;
                interpolationmode=none);
            @display("i=block/routing");
```

Declaración de la variable de tipo `simsignal_t` (C++):

```
class Txc16 : public cSimpleModule
    {
      private:
        simsignal_t arrivalSignal;
                // ...
```

Emisión de un valor para dicha señal (C++):

```
void Txc16::handleMessage(cMessage *msg)
    {
        TicTocMsg16 *ttmsg = check_and_cast<TicTocMsg16 *>(msg);

        if (ttmsg->getDestination() == getIndex()) {
            // Message arrived
            int hopcount = ttmsg->getHopCount();
            // send a signal
            emit(arrivalSignal, hopcount);
```

Sobreescritura de los valores por defecto (qué recolectar y qué ignorar) (archivo ini):

```
[Config Tictoc16]
    network = Tictoc16
    **.tic[1].hopCount.result-recording-modes = +histogram
    **.tic[0..2].hopCount.result-recording-modes = -vector
```

Esta funcionalidad también se expuso en la librería de Python.

4.4. Limitaciones y dificultades

En esta sección se detallan algunos problemas encontrados durante el desarrollo del proyecto. Algunos de los mismos no han sido solucionados o su solución siempre pareció pasajera.

4.4.1. Ciclo de vida de un mensaje

Un mensaje dentro de una simulación OMNeT++ es una instancia de `cMessage` (u otra clase derivada de `cMessage`, como `cPacket` o cualquier clase definida por el usuario que tenga a cMessage entre sus clases base).

El mensaje no sólo es importante en tanto y en cuanto permite “desarrollar la acción” dentro de una simulación, sino que también es el combustible que hace avanzar el código del kernel de OMNeT++. En efecto, cuando un módulo ejecuta su método `send`, el mensaje se encola como un evento a ser procesado por el kernel de la simulación. Cuando no hay más eventos que procesar, la simulación termina.

Existe en OMNeT++ cierto protocolo en relación a quién se hace cargo de destruir los mensajes (liberar su memoria). En cualquier momento, todo mensaje tiene un *dueño* (puede ser el módulo que lo está creando, el propio kernel de la simulación, el módulo que lo recibe). Es el dueño del mensaje el único habilitado para liberar su memoria.

Cualquier intento de salirse de este protocolo es penado por OMNeT++: se paga con la finalización temprana de una simulación. En el otro extremo, si al llegar a la finalización de una simulación resulta que hay mensajes que nadie ha borrado, el sistema emite por pantalla numerosas advertencias de que nadie se está haciendo cargo de borrar los mensajes que ya no se utilizan:

```
undisposed object: (omnetpp::cMessage) HypercubeNetwork.node[6].rte.event -- check module
↪  destructor
```

¿Cómo interactúa este protocolo creación y borrado de mensajes con el manejo automático de la memoria y el conteo de referencias que hace Python? ¿Cómo hacemos que nuestros módulos implementados en Python entren a este escenario sin romper las reglas? ¿Cómo le damos a un módulo implementado en Python la posibilidad de borrar un mensaje definitivamente? Analicemos lo que sucede en nuestro ejemplo habitual:

```python
1  from pyopp import cSimpleModule, cMessage
2
3  class PyTxc1(cSimpleModule):
4
5      def initialize(self):
6          if self.getName() == 'tic':
7              msg = cMessage("pymessage")  # creación de mensaje nuevo
8              self.send(msg, "out")
9
10     def handleMessage(self, msg):
11         self.send(msg, "out")
```

Es claro que la variable local `msg` es la única referencia al mensaje creado. Al finalizar esta función, el algoritmo de conteo de referencias de Python actuará y se eliminará el objeto. Esto ocasionará el borrado del objeto que existe del lado de C++. En este momento, OMNeT++ se quejará de que el mensaje aún no debía ser borrado, dado que este está en el kernel de la simulación. Fin del programa.

Para evitar esto, se recurrió a una solución en Python puro. Se hizo un wrapper de la clase `cMessage` cuyo inicializador guarda una referencia extra al objeto en un objeto externo denominado `RefStore` (almacén de referencias). De esta forma, cada vez que se crea un mensaje existen a él dos referencias:

- la referencia local (como la variable `msg` en el ejemplo anterior),

- la referencia extra almacenada en `RefStore` por el constructor de `cMessage`.

De este modo, al finalizar la función `initialize` la referencia local se pierde, pero el mensaje no es eliminado gracias a que aún existe una referencia (desde el punto de vista de Python).

Simétricamente, para realizar la tarea de limpieza y liberación definitiva del mensaje, agregamos a la clase `cSimpleModule` (nuevamente, en Python) la posibilidad de borrar esta referencia extra mediante el método delete que acepta un mensaje y trata de quitarlo del `RefStore`, eliminando la última referencia existente.

Este escenario (Python crea una variable, se la pasa a OMNeT++, OMNeT++ espera que esa referencia apunte a un objeto durante algún tiempo, Python borra el objeto apenas termina el bloque de código) se repite en algunos otros lugares. Allí donde se descubrió que esto sucedía, la solución fue implementar un wrapper en Python del método original para asegurar la existencia de una referencia extra al objeto en cuestión. Esta no es la solución ideal, no obstante, porque a diferencia de lo que ocurre con los mensajes, otro tipo de objetos no suelen ser explícitamente eliminados en el código Python.

4.4.2. El método `deleteModule`

Existe en OMNeT++ la posibilidad de que un módulo se declare listo para ser eliminado de la red: debe ejecutar su método `deleteModule`. Esto permite la creación de simulaciónes con una topología dinámica. El manejo que hace OMNeT++ de esta situación involucra un uso no trivial de excepciones en C++ como forma de control de flujo. Este método probó ser un verdadero desafío para ser portado y utilizado desde Python. Finalmente se excluyó de la lista de features portadas. Esto implica, por ejemplo, que no pudimos implementar el ejemplo llamado dyna en Python.

4.4.3. Automatización de la creación de bindings

El proceso de creación de bindings fue manual. En general, esta actividad se resistió a una automatización fácil por el hecho de que para cada método de cada clase expuesta a Python hubo que razonar sobre el uso de los parámetros, si se usaba paso por referencia o por valor, si Python debía hacerse cargo de borrar los argumentos o si OMNeT++ esperaba que estos siguieran vivos luego de la llamada.

Este proceso en sí es una limitación del proyecto, ya que es propenso a errores, es lento, y en cada nueva versión de OMNeT++ habría que hacer una revisión de todos las clases que se han agregado o que han sufrido alguna modificación en su interfaz.

Una línea interesante de trabajo futura sería, entonces, el intento de automatizar la creación de los bindings para Python, complementado con alguna suite de tests que hicieran uso del módulo Python generado, apuntando a tener 100 % de cubrimiento.

4.4.4. Mayor cubrimiento de la librería original

Como se mencionó previamente, se eligió exponer a Python aquello que fue siendo necesario para migrar las simulaciones de ejemplo del directorio `samples` del proyecto

original. Esto no es el 100 % de la librería de simulación. Notar que tampoco es deseable exponer cada clase y cada función de OMNeT++ al mundo de Python, sino sólo aquellas que un usuario podría llegar a requerir a la hora de hacer una simulación.

4.4.5. Depuración

El IDE de OMNeT++ habilita la compilación de las simulaciones con símbolos de debugging (depuración). También permite también el seteo de puntos de debugging así como entrar a modo debugging si la aplicación termina inesperadamente. Las funciones de debugging (recorrer código, entrar a funciones, inspeccionar variables, registros) están perfectamente incorporados a la interfaz gráfica del IDE (que como ya se ha dicho, está construido encima de Eclipse).

Al implementar módulos en Python, muchas de estas funcionalidades quedan obsoletas, o se están perdiendo definitivamente. Si la simulación que se está debuggeando tiene módulos implementados en Python, al llegar el momento de llamar a una función implementada en Python, el debugger de C++ no ve un cambio de lenguaje, simplemente nos ofrece entrar al código del intérprete de Python (hecho en C). Esto, asumiendo que la versión de Python con símbolos de debugging está disponible en el sistema y ha sido apropiadamente enlazada a la simulación (durante la compilación).

Nuestro proyecto no se provee compilación con símbolos de debugging de nuestro código ni una versión de las librerías de Python con símbolos de debugging. Aún si estos estuvieran disponibles, sería interesante que la ejecución del debugger permitiera entrar al código Python utilizando pdb (y no gdb).

4.4.6. La función de casteo

Dentro del macro `Define_Python_Module` se provee una función de casteo. Analizando el código de OMNeT++ se puede apreciar que esta se usa en última instancia para poder distinguir si un módulo es una instancia de nuestra clase o no. En el caso de los módulos definidos en Python, la función de casteo está devolviendo `true` para todos. Si bien esto no nos trajo dificultades en la implementación y ejecución de ninguna simulación, sería interesante desarrollar una mejor función de casteo que permitiera reconocer diversas subclases de `PycSimpleModule` implementadas en Python.

5 Omnetpy

5.1. Arquitectura del proyecto

En una primera etapa de trabajo (particularmente durante de investigación de la arquitectura de software y la ejecución de una simulación) se re realizó bastante intervención sobre el código propio de OMNeT++ (con la finalidad de trazar llamadas, imprimir argumentos, etc). Naturalmente, el código propio de nuestro trabajo (nuevas definiciones, bindings) fue siendo añadido a la estructura de directorios propuesta por OMNeT++ (nuevos encabezados en el directorio `include`, nuevas fuentes en el directorio `src`, etc). Con el objetivo de compilar OMNeT++ así como los Python bindings en el mismo proceso, también se realizaron modificaciones sobre el `Makefile` de primer nivel.

Con el tiempo, este enfoque fue dejando ver algunas limitaciones ya que las modificaciones que fueron hechas para OMNeT++ 5.5.1 no necesariamente se dejaban aplicar de forma directa sobre otras versiones. En particular, nos interesó probar nuestros cambios sobre OMNeT++ 6, que incluye soporte para editar archivos Python directamente en el IDE (resaltado de sintaxis, autocompletado, etc.). Si bien no es imposible aplicar los cambios sobre otras versiones de OMNeT++, requiere supervisión manual. En particular, la edición del `Makefile` (para forzar el compilado de nuestro código junto con el resto del proyecto) probó ser un mecanismo difícil de automatizar.

Prácticamente estábamos obligados a seguir utilizando los Python bindings con la versión de OMNeT++ inicialmente elegida (5.5.1) o elegir para nuestro proyecto una estructura de versiones que fuera espejo de las de OMNeT++, y realizar adaptaciones particulares para cada una.

Con el fin de simplificar el proceso de integración se comenzó a separar nuestro trabajo en un proyecto independiente que pudiera ser aplicado prácticamente sin cambios a distintas versiones de OMNeT++. Comenzamos a utilizar el término “omnetpy” para referirnos al conjunto de los archivos agregados por nosotros.

5.2. Separando omnetpy de OMNeT++

Dado que nosotros controlamos el ambiente de trabajo de forma prácticamente total (mediante el uso de imágenes docker), aislar nuestros cambios y compilarlos de forma independiente en un proceso automático probó ser una tarea bastante sencilla y con beneficios inmediatos.

En una primera etapa se compila el proyecto OMNeT++ (la versión elegida, no importa cuál sea) y luego se compila omnetpy.

Esta es la estructura de directorios de nuestro proyecto:

- bindings: código para la generación de la librería Python pyopp
- bindings/pyopp: definición Python pura del módulo pyopp
- include: encabezados C++ (definiciones tales como `Define_Python_Module`, `PycSimpleModule`, `InterpreterManager`)
- lib: tras el proceso de compilación, aquí se deja la librería C++ `libomnetpy.so`. Esta debe enlazarse a cualquier binario que desee
- utilizar el macro `Define_Python_Module`.
- src: código C++ (implementación de `InterpreterManager`)

Luego de esta separación, se probó con éxito la integración de nuestro código sobre diversas versiones de OMNeT++ (5.6.1, 6.0.pre6).

Por supuesto, si una versión de OMNeT++ cambia la API de las clases que nosotros deseamos exponer a Python, nuestro código debe también ser diferente. No estamos tomando esas pequeñas diferencias en cuenta. Esto sí obligaría a generar Python bindings ajustados a cada versión de OMNeT++ (ya sea de forma manual o automática).

5.3. El módulo `pyopp`

La compilación de los archivos `bind_<clase de OMNeT++>.cc` y su posterior enlazamiento produce un módulo Python cuyo nombre completo es (dependiendo de la versión de Python) `_pybind.cPython-37m-x86_64-linux-gnu.so`. Si bien este módulo se puede importar desde Python, preferimos agregar una capa hecha completamente en Python que realiza algunas adaptaciones. Así, el módulo que importa el usuario (`pyopp`) es un wrapper que elige qué y cómo exponer del módulo hecho en C++.

5.4. Compilación del proyecto

El proyecto cuenta con algunos archivos escritos en C++ que necesitan ser compilados

para la extensión del intérprete: estos archivos son compilados y enlazados con las librerías de desarrollo de Python, produciendo un módulo que se puede importar desde Python.

para embeber el intérprete: estos archivos no son parte del módulo de extensión, si no que se encargan de que OMNeT++ inicialice un intérprete de Python antes de instanciar la primera clase definida en Python.

Estos archivos son compilados utilizando un `Makefile` que resuelve todos los detalles de enlace, encabezados, flags necesarios, etc. Este paso, además, es realizado como parte de la generación de la imagen Docker que se usó para desarrollar el trabajo.

5.5. Manual de usuario

El uso de OMNeT++ con omnetpy es muy similar a su uso sin omnetpy. Sólo hay que realizar algunos pasos extra, como por ejemplo:

- Escribir las clases derivadas de `cSimpleModule` en Python
- Escribir un archivo C++ utilizando el macro `Define_Python_Module`
- Asegurarse que al compilar el proyecto sean incluidos los encabezados de omnetpy (por ej, donde se define el macro mencionado).

Se incluye a continuación una guía paso a paso que muestra al usuario cómo realizar una simulación usando OMNeT++ con omnetpy, desde el IDE.

5.5.1. Prerrequisitos

Se asume la disponibilidad de:
`git` (para clonar el proyecto)
`make` (para evitar el tipeo de comandos muy largos)
`docker` (para construir imágenes y lanzar contenedores)
X11 socket (para lanzar aplicaciones con entorno gráfico desde docker)

5.5.2. Opción 1: generar imagen docker

Clonar el repositorio del proyecto:

```
git clone https://github.com/mmodenesi/omnetpy.git
```

Generar una imagen docker:

```
make build
```

En este punto se puede especificar la versión de OMNeT++ que se desea utilizar, por ejemplo:

```
OMNETPP_VERSION=5.6 make build
```

Se espera que la versión sea descargable desde la página de releases de OMNeT++, de lo contrario, la creación de la imagen fallará. Consultar la lista de versiones disponibles en https://github.com/omnetpp/omnetpp/releases
Lanzar un contenedor:

```
make run
```

Notar que el comando que se ejecuta en este target de `make` lanza un contenedor efímero en el cual se monta el directorio actual del host en /home/userpp/workspace. Esto quiere decir que al salir del contenedor este será eliminado, pero los archivos creados en el directorio /home/userpp/worspace serán visibles en el host. Alternativamente, si el usuario desea montar otro directorio, o desea que el contenedor no sea efímero, se sugiere ejecutar `make run -n` para ver el comando y ejecutar una versión modificada apropiadamente.

5.5.3. Opción 2: utilizar una imagen pregenerada

Alternativamente, es posible utilizar una de las imágenes de docker pregeneradas y listadas en https://hub.docker.com/r/mmodenesi/omnetpy

Por ejemplo:

```
docker run \
    --rm -ti \
    -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
    mmodenesi/omnetpy:ub18.04_opp5.6.2 bash
```

El nombre de la imagen anuncia qué versión de ubuntu y qué versión de OMNeT++ se utiliza. El ejemplo anterior está utilizando una imagen construida a partir de ubuntu:18.04 y con OMNeT++ 5.6.2.

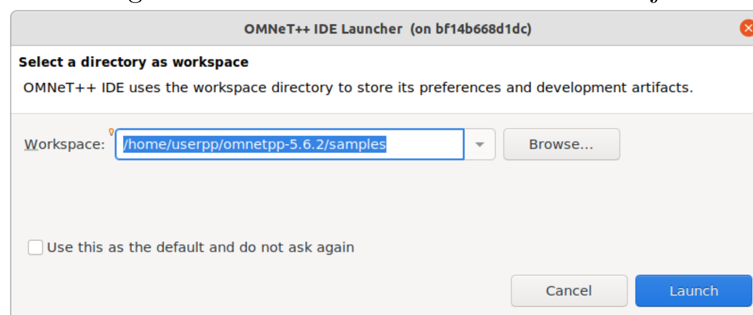
5.5.4. Escribiendo una simulación con omnetpy

Dentro del contenedor, lanzar el IDE

```
omnetpp
```

Cuando levantamos el IDE por primera vez nos pregunta qué directorio utilizar como “workspace” por defecto, como se puede observar en la figura 5.1.

Figura 5.1: Selección de directorio de trabajo

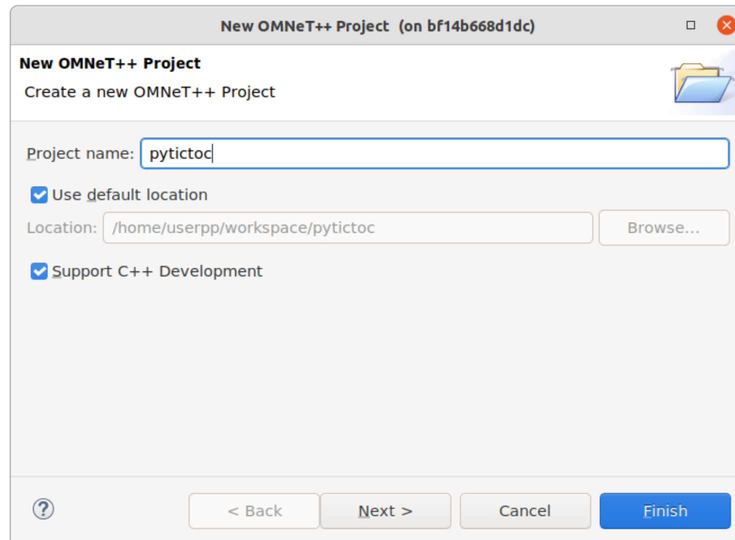


Se recomienda usar la ruta /home/userpp/workspace, ya que en general lanzamos contenedores docker donde esa ruta es un volumen (lo que hacemos desde el contenedor dentro de ese directorio, se persiste en el host, más allá de la vida útil del contenedor).

5.5.5. Preparando un nuevo proyecto

Desde el IDE, seleccionar *File* → *New* → *OMNeT++ Project* (fig. 5.2). Al aparecer el diálogo, ingresar `pytictoc` como nombre del proyecto, seleccionar *Empty project* y luego *Finish*. Un nuevo proyecto aparecerá en el área de proyectos del IDE.

Figura 5.2: Creación de un nuevo proyecto



5.5.6. Agregando un archivo NED

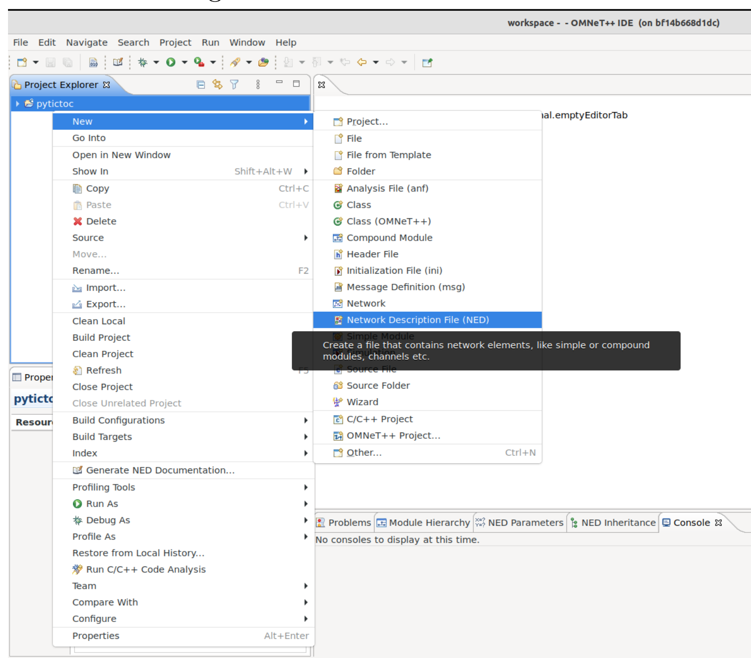
Hacer click derecho en el proyecto y seleccionar *New* → *Network Description File (NED)* (fig. 5.3).

Guardarlo como `pytictoc.ned` con el siguiente contenido:

```
simple PyTxc
{
    gates:
        input in;
        output out;
}

//
// Two instances (tic and toc) of Txc1 connected both ways.
// Tic and toc will pass messages to one another.
//
network PyTictoc
{
    submodules:
        tic: PyTxc;
        toc: PyTxc;
    connections:
        tic.out --> {  delay = 100ms; } --> toc.in;
        tic.in <-- {  delay = 100ms; } <-- toc.out;
}
```

Figura 5.3: Añadir archivo NED



5.5.7. Agregando un archivo C++

Hacer click derecho en el proyecto y seleccionar *New* → *File* y guardarlo como `txc.cc` con el siguiente contenido:

```
1  #include <omnetpy.h>
2
3  Define_Python_Module("txc", "PyTxc");
```

Dado que el archivo `omnetpy.h` no se encuentra en un directorio conocido por el IDE, es esperable que se señalen errores en relación a su inclusión o a `Define_Python_Module`. Para cambiar esto se debe agregar el directorio `/home/userpp/omnetpy/include` a la lista de directorios escaneados en busca de archivos de encabezado. En la versión actual del IDE esto se logra en *Project* → *Properties* → *C/C++ General* → *Path and Symbols* y clickeando el botón *Add*.

5.5.8. Agregando un archivo `makefrag`

Hacer click derecho en el proyecto, seleccionar *New* → *File* y guardarlo como `makefrag` con el siguiente contenido:

```
INCLUDE_PATH += $(shell python3 -m pybind11 --include) -I$(OMNETPY_ROOT)/include
LIBS = -lomnetpy $(shell python3-config --libs | cut -d" " -f1)
LDFLAGS += -L$(OMNETPY_ROOT)/lib
```

Estas líneas van a ser incluidas en el archivo `Makefile` autogenerado por OMNeT++ para compilar la simulación. Es esencial que se llame `makefrag`.

> **Nota importante**: si la versión de Python es 3.8 o mayor,
> `python3-config --libs`
> debe reemplazarse por
> `python3-config --libs --embed`.

5.5.9. Agregando un archivo Python

Hacer click derecho en el proyecto, seleccionar *New* → *File* y guardarlo como `txc.py` con el siguiente contenido:

```
1  from pyopp import cSimpleModule, cMessage
2
3
4  class PyTxc(cSimpleModule):
5      def initialize(self):
6          if self.getName() == 'tic':
7              self.send(cMessage('msg'), 'out')
8
9      def handleMessage(self, msg):
10         self.send(msg, 'out')
```

5.5.10. Correr la simulación

En el panel del editor seleccionar el archivo `txc.cc` (la fuente en C++). Esto es importante para forzar al IDE a ofrecernos la compilación de una simulación C++ en el siguiente paso.

Seleccionar *Run* → *Run* y elegir *OMNeT++ Simulation* (fig. 5.4).

Luego de la compilación, se lanza automáticamente la ejecución de la simulación (fig. 5.5).

Figura 5.4: Correr la simulación

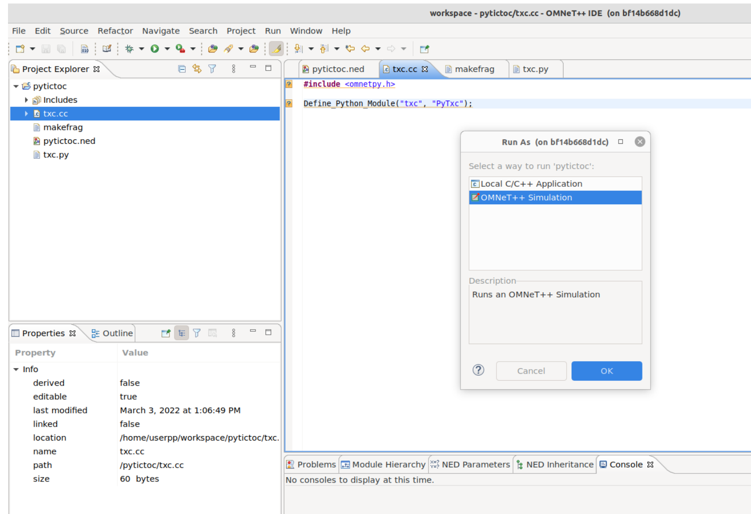
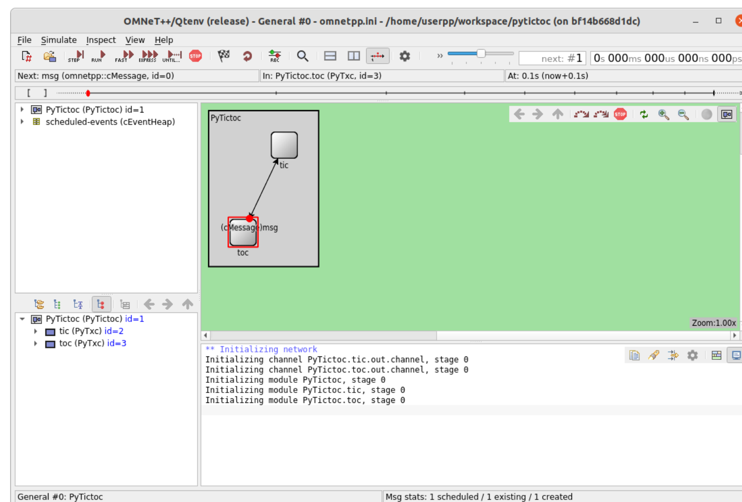


Figura 5.5: Simulación

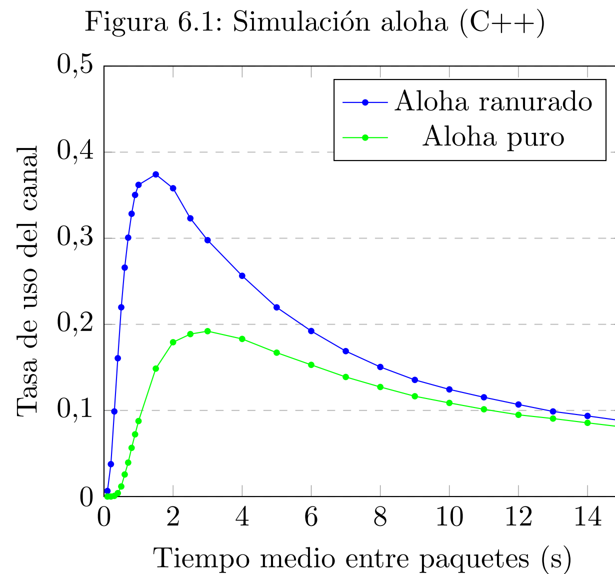


6 Evaluación

Se procedió a verificar el funcionamiento de un modelo de OMNeT++ hecho en Python. Para realizar esta tarea se implementó la simulación de ejemplo aloha (parte de la distribución oficial de OMNeT++) en Python (la llamaremos pyaloha).

Se puede mostrar teóricamente que para el protocolo de transmisión aloha el uso óptimo del canal ($\frac{1}{e}$, aproximadamente 18,4 %) ocurre cuando se emite un paquete cada dos unidades de tiempo, mientras que para el protocolo aloha ranurado, el uso óptimo ($\frac{1}{2e}$, aproximadamente 36, 8 %) ocurre cuando se emite un paquete por unidad de tiempo [16]. Como se puede observar en la figura 6.1, los datos simulados coinciden con los cálculos teóricos.

Figura 6.1: Simulación aloha (C++)



6.1. Verificación

Para la implementación de pyaloha se siguió un patrón de traducción casi línea por línea del código original en C++, para evitar introducir cambios inadvertidos en el modelo.

Se ejecutó la simulación con los siguientes parámetros:

- número de hosts: 15

- tasa de transmisión: 9600kbps
- tamaño de paquete: 952b
- tiempo total de simulación: 60min
- tiempo entre paquetes: variable exponencial con media λ, donde λ se hizo variar entre 0 y 15 (segundos)

Para cada valor de λ se corrió la simulación 4 veces:

- En C++, aloha puro
- En C++, aloha ranurado
- En Python, aloha puro
- En Python, aloha ranurado

y en cada experimento se recogió la tasa de utilización del canal calculada por el programa.

El hallazgo fue que para el modelo pyaloha se obtuvieron los mismos resultados que para el modelo aloha. No es que se obtuvieron resultados similares: se obtuvieron exactamente los mismos resultados. Recordar que la inicialización del generador de números aleatorios utiliza la misma semilla, por lo que los eventos han sido generados en tiempos exactamente iguales, en ambos modelos.

Esto deja claro que la implementación del modelo en Python es tan funcional como la implementación del modelo en C++.

6.2. Rendimiento

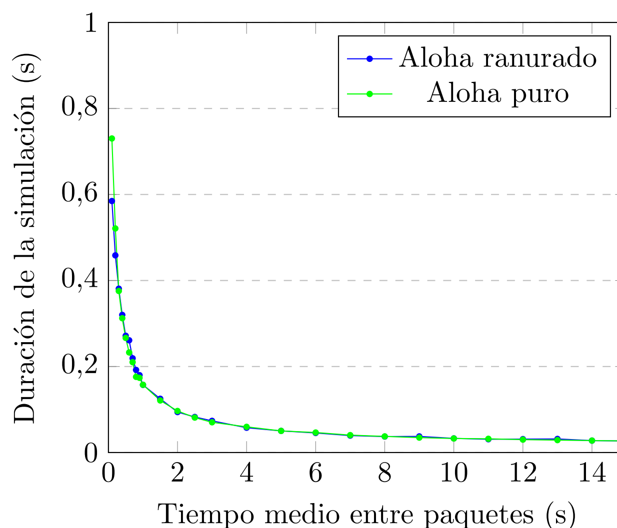
A continuación se presenta una comparación entre los modelos realizados con OMNeT++ (código en C++) y los mismos modelos escritos en Python, en términos de velocidad de procesamiento. Las simulaciones se realizaron en una notebook con procesador Intel i7-1065G7 CPU @ 1.30GHz, con 8GB de memoria RAM. Sin embargo, los resultados se presentan en términos de eventos por segundo (una métrica que OMNeT++ escribe por salida estándar).

6.2.1. Aloha

Mientras menor es el tiempo medio entre paquetes, las simulaciones necesitan más tiempo para finalizar. Esto es debido a que se generan más mensajes (eventos) que deben ser procesados por el kernel de simulación hasta alcanzar el tiempo límite de 60 minutos (simulados).

En la figura 6.2 vemos que la ejecución más lenta del modelo hecho en C++ corre en menos de un segundo:

Figura 6.2: Tiempo de ejecución aloha (C++)



En tanto que para pyaloha, el tiempo alcanzado ronda los 2 minutos, como se aprecia en la figura 6.3.

Independientemente del tiempo medio entre mensajes, del tiempo total de la simulación o de si el protocolo es ranurado o no, se puede observar que el modelo implementado directamente en C++ procesa unos 1200000 eventos por segundo, mientras que el modelo implementado con Python es unas 200 veces más lento, procesando sólo 6000 eventos por segundo.

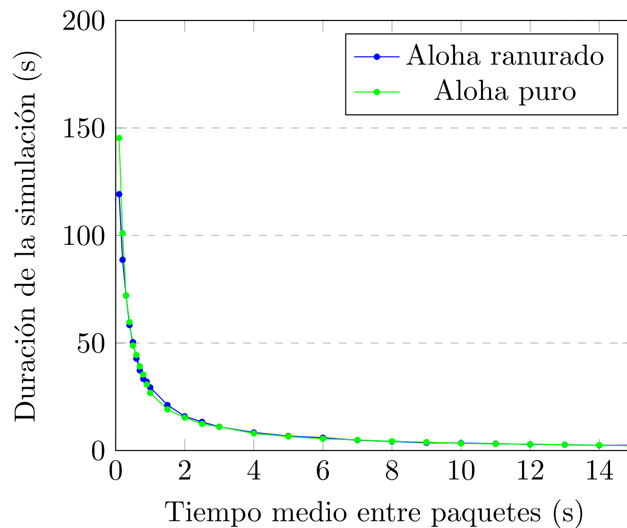
6.2.2. Tictoc

Para Tictoc1, que consiste en un simple pasaje de mensajes entre dos instancias de `Txc`, encontramos que el modelo implementado en C++ procesa unos 5 millones de eventos por segundo, mientras que el que utiliza Python sólo alcanza el medio millón, es decir, es unas 10 veces más lento.

Tictoc2 introduce el uso herramientas para loggear información durante una simulación. En particular, se presenta el constructo `EV`. En este caso, la performance de C++ no se ve afectada, alcanzando también unos 5 millones de eventos por segundo. En cambio, el modelo que utiliza Python sólo alcanza unos 17,000 mensajes por segundos (unas 300 veces más lento). Esto se debe a que en Python, `EV` no está llamando directamente a código C++ de OMNeT++. Como se dijo en la sección 4.3.2, en C++ es un macro y por lo tanto es expandido en tiempo de compilación, mientra sque en Python es un objeto que inspecciona el stack para conocer el contexto y recién entonces llama a una API de C++.

En efecto, si el modelo en C++ dice:

Figura 6.3: Tiempo de ejecución pyaloha (Python)




```
EV << getName() << "'s counter is " << counter << "\n";
```

su traducción directa a Python resulta aproximadamente 1000 veces más lenta

```
EV << self.getName() << "'s counter is " << self.counter << '\n'
```

mientras que la siguiente variante, aparentemente igual, es sólo 300 veces más lenta

```
EV << f"{self.getName()}'s counter is {self.counter}\n"
```

La diferencia viene dada por el hecho de que la primera variante (la que es más lenta) consiste en 4 usos del operador `<<` (left shift) sobre el objeto `EV`, cada uno de los cuales se traduce en una inspección del stack trace para evaluar el contexto de ejecución, mientras que en la segunda variante se utiliza `<<` una única vez.

6.2.3. Conclusión

Como era de esperar, se encontró que los modelos implementados directamente en C++ se terminan de simular más rápidamente. No obstante, qué tanto más rápido no es un valor constante: la complejidad del método `handleMessage` dicta la magnitud de la penalidad en la performance. Esto es consistente con el hecho de que mientras más tiempo pase la simulación ejecutando instrucciones Python dentro del intérprete, más se aleja de la performance ideal alcanzada por un modelo que utilice sólo C++ la totalidad del tiempo.

6.3. Complejidad de código

La siguiente comparación entre modelos de OMNeT++ puro (C++) y OMNeT++ implementando modelos en Python se basa en la reimplementación de los siguientes ejemplos de muestra incluidos en el código fuente de OMNeT++: aloha, canvas, cqn, dyna, fifo, histograms, hypercube, routing, tictoc.

En ciertos fragmentos de código donde predomina la llamada a métodos y funciones de OMNeT++, los modelos en C++ y los modelos en Python son necesariamente casi idénticos, como se observa en el siguiente ejemplo:
C++:

```
void NetBuilder::initialize()
{
    scheduleAt(0, new cMessage());
}
```

Python:

```
    def initialize(self):
        self.scheduleAt(0, cMessage())
```

No obstante, notamos algunos factores que contribuyen a una mejora en la complejidad de código: ausencia de caracteres como corchetes y punto y coma para estructurar el código, ausencia de calificación del nombre de espacios (`Netbuilder::`).

En líneas generales, los modelos de ejemplo que incluye OMNeT++ están diseñados para mostrar cómo trabajar con OMNeT++, lo cual hace que el código consista principalmente en llamadas a las APIs y, por lo tanto, su implementación usando Python sea prácticamente igual en términos de cantidad de líneas.

Donde realmente vemos la diferencia es, por ejemplo, en el siguiente fragmento:

```
    std::map<long, cModule *> nodeid2mod;
    std::string line;

    std::fstream nodesFile(par("nodesFile").stringValue(), std::ios::in);
    while (getline(nodesFile, line, '\n')) {
        if (line.empty() || line[0] == '#')
            continue;

        std::vector<std::string> tokens = cStringTokenizer(line.c_str()).asVector();
        if (tokens.size() != 3)
            throw cRuntimeError("wrong line in module file: 3 items required, line: \"%s\"",
↪   line.c_str());

        // get fields from tokens
        long nodeid = atol(tokens[0].c_str());
        const char *name = tokens[1].c_str();
        const char *modtypename = tokens[2].c_str();
        EV << "NODE id=" << nodeid << " name=" << name << " type=" << modtypename << "\n";

        // create module
        cModuleType *modtype = cModuleType::find(modtypename);
        if (!modtype)
            throw cRuntimeError("module type '%s' for node '%s' not found", modtypename, name);
```

```
        cModule *mod = modtype->create(name, parent);
        nodeid2mod[nodeid] = mod;

        // read params from the ini file, etc
        mod->finalizeParameters();
    }
```

Python:

```
    nodeid2mod = {}

    with open(self.par('nodesFile').stringValue(), 'r') as fh:
        for line in fh.readlines():
            line = line.strip()
            if not line or line.startswith('#'):
                continue

            nodeid, name, modtypename = line.split()
            nodeid = int(nodeid)

            EV << 'NODE id=' << nodeid << ' name=' << name << ' type=' << modtypename << '\n'

            # create module
            modtype = cModuleType.find(modtypename)
            if modtype is None:
                raise RuntimeError(f'module type '{modtypename}' for node '{name}' not found')

            mod = modtype.create(name, parent)
            nodeid2mod[nodeid] = mod

            # read params from the ini file, etc
            mod.finalizeParameters()
```

donde Python es más simple por no declaración de variables, la facilidad de lectura de archivos, la facilidad para manipulación de strings o, como vemos en el siguiente ejemplo, por la facilidad para trabajar con diccionarios:

C++:

```
    std::map<long, cModule *>::iterator it;

    for (it = nodeid2mod.begin(); it != nodeid2mod.end(); ++it) {
            cModule *mod = it->second;
            mod->buildInside();
    }
```

Python:

```
        for mod in nodeid2mod.values():
            mod.buildInside()
```

Como puede apreciarse en la tabla 6.1, la cantidad de líneas necesaria para implementar un modelo en Python siempre fue menor que para el modelo original en C++. En el análisis se tienen en cuenta únicamente archivos `.h`, `.cc`, `.msg` y `.py`.

Además de las características ya mencionadas que hacen de Python un lenguaje más simple de manejar, los modelos implementados en Python tienen menos archivos y menos líneas.

Modelo	C++	Python	Proporción
aloha	511	352	68,88 %
canvas	145	114	78,62 %
cqn	185	100	54,05 %
dyna	401	291	72,57 %
fifo	285	122	42,81 %
histogram	158	139	87,97 %
hypercube	368	263	71,47 %
routing	551	370	67,15 %
tictoc	1755	705	40,17 %

Cuadro 6.1: Cantidad de líneas de código en modelos reimplementados en Python

7 Conclusiones

Se presenta una extensión del simulador de eventos discreto OMNeT++ que permite escribir módulos simples en el lenguaje de programación Python. Esto habilita a la escritura de modelos de simulación sin necesidad de saber programar en el lenguaje oficial de OMNeT++, es decir, C++.

C++ es un lenguaje de programación que provee pocas abstracciones sobre el hardware subyacente y que se compila para obtener un binario ejecutable convierten en herramienta muy poderosa para escribir código de alta performance (administración manual de la memoria, complejo sistema de tipos, posibilidad de incluir código assembly), hacen de este un lenguaje de difícil dominio y con tiempos de desarrollo relativamente largos, si los comparamos con otros lenguajes como Python.

Python es un lenguaje de programación de mucho más alto nivel, donde los detalles del hardware son alejados del usuario final. Un sistema de tipos más flexible que requiere menos especificidad por parte del usuario, manejo automático de la memoria, una sintaxis más cercana al lenguaje natural, hacen de este un lenguaje muy idóneo para principiantes o para enseñanza en carreras relacionadas a la computación. Los programas en Python no se compilan, sino que son consumidos por otro programa (el intérprete de Python) para llevarlos a instrucciones de la máquina virtual de Python y finalmente ejecutar tales instrucciones una a una en el CPU. La implementación más popular de intérprete está hecha en C y se conoce como cPython.

En general, los programas escritos en C++ son varias veces más rápidos que programas con la misma funcionalidad escritos en Python, y los programas escritos en Python son más fáciles de entender, modificar, extender. Es decir que al movernos de C++ a Python, lo que se gana es mayor claridad en el código y tiempos de desarrollo más cortos, mientras que si nos movemos en la otra dirección, lo que se gana es mayor performance computacional.

La posibilidad de escribir simulaciones de OMNeT++ usando Python en lugar de C++ tiene varias ventajas asociadas:

- menor barrera de entrada para usuarios nuevos
- mayor facilidad para incorporar OMNeT++ en cursos de computación
- menor tiempo de prototipado de nuevos modelos
- inmediata disponibilidad de todos los recursos del ecosistema Python
- posibilidad de introducir cambios en el modelo sin pasos extra de compilación

La extensión de la herramienta OMNeT++ fue realizada sin necesidad de modificar el código fuente original, lo cual facilita su aplicación en distintas versiones de OMNeT++.

Las simulaciones ejecutadas con modelos hechos en Python arrojaron idénticos resultados que las versiones originales en C++, aunque con penalidades de performance variable (dependiendo de la complejidad del método `handleMessage`).

7.1. Desafíos abiertos

Como se menciona en la sección 4.4, quedan algunos desafíos abiertos para continuar con la tarea iniciada en este trabajo, los cuales se detallan a continuación.

7.1.1. Generación automática de librerías de extensión

La extensión del intérprete con las librerías de OMNeT++ se realizó manualmente y exponiendo sólo aquellas partes que eran necesarias para portar las simulaciones de ejemplo del proyecto original.

Sería interesante que este proceso fuera automatizado de forma que dada una versión de OMNeT++ se generaran los Python bindings de la totalidad de la librería.

Las ventajas de esto serían:

1. Que cualquier función, clase, método, macro, tipo definido en OMNeT++ (en C++) fuera accesible y extensible desde Python

2. Se podría contar fácilmente con versiones de omnetpy para cualquier versión de OMNeT++ (recordar que el trabajo se basa en la versión 5.6.2 y se probó ligeramente sobre las versiones de perview de OMNeT++ 6).

3. En caso de emprender este esfuerzo, sería interesante dedicar algún tiempo a diseñar una infraestructura de testing. Por ej, realizar simulaciones en C++ y en Python y comparar que las salidas de sus ejecuciones sean idénticas, apuntando a que estas simulaciones utilicen el mayor porcentaje posible de las librerías.

7.1.2. Depurador interactivo en Python

Actualmente, cuando un modelo hecho en Python no se comporta de la forma que se espera y se dificulta encontrar el error, no es posible recurrir a establecer un punto de depuración (debugging) y realizar una inspección interactiva y pausada del estado de las variables. Esto es así porque el intérprete de Python no es el programa principal, sino que está embebido en un programa en C++ (la simulación).

Siendo la trazabilidad y la habilidad de depurar las simulaciones tan central en el proyecto OMNeT++, y siendo que este trabajo apunta a que escribir simulaciones sea más simple, perder esta funcionalidad parece un contrasentido que merece la pena tratar de arreglar.

7.1.3. Formas de distribución y disponibilización

Durante el desarrollo de este trabajo se recurrió a congelar un ambiente controlado dentro de una imagen Docker con versiones conocidas de todos los elementos de software involucrados (compilador, OMNeT++, Python, pybind11, etc). Esto por un lado permitió contar con un ambiente controlado y repetible donde sabíamos que las cosas funcionaban y por otro tuvo la ventaja de hacer que el código pudiera ser aprovechado desde cualquier computadora con muy pocos pasos de preparación.

No obstante, a la larga esto va a convertirse en una limitación: los sistemas operativos dejan de ser mantenidos, las librerías necesarias ya no se consiguen o dejan de tener mantenimiento oficial, Python 3.6 en algún momento será declarado oficialmente sin soporte y ya nadie escribirá librerías para enriquecer su ecosistema. Sería interesante pensar en algún mecanismo más general de compilación y distribución que pudiera fácilmente seguir el ritmo de los nuevos lanzamientos (nuevas versiones de Python, de pybind11, etc).

En cuanto a la tecnología de virtualización elegida (Docker) probó ser muy beneficiosa desde el punto de vista de la facilidad de instalación, pero sería interesante documentar otras formas de usar omnetpy sin recurrir a ella.

Bibliografía

[1] Sabah Abdulkareem y Ali Abboud. "Evaluating Python, C++, JavaScript and Java Programming Languages Based on Software Complexity Calculator (Halstead Metrics)". En: feb. de 2021.

[2] Jerry Banks y col. *Discrete Event System Simulation*. 5.ª ed. Pearson Education Limited, 2014.

[3] *Extending and Embedding the Python Interpreter*. Accessed: 2022-03-04. URL: https://docs.python.org/3/extending/index.html.

[4] Juan Fraire y Juan Duran. "Revising Computer Science Networking Hands-On Courses in the Context of the Future Internet". En: *IEEE Transactions on Education* PP (sep. de 2020), págs. 1-6. DOI: 10.1109/TE.2020.3015673.

[5] Wim Lavrijsen y Aditi Dutta. "High-Performance Python-C++ Bindings with PyPy and Cling". En: nov. de 2016, págs. 27-35. DOI: 10.1109/PyHPC.2016.008.

[6] Averill M. Law y David Kelton. *Simulation Modeling and Analysis*. 3.ª ed. McGraw Hill, 2000.

[7] Giovanni Nardini y col. "Simu5G–An OMNeT++ Library for End-to-End Performance Evaluation of 5G Networks". En: *IEEE Access* 8 (sep. de 2020), págs. 181176-181191. DOI: 10.1109/ACCESS.2020.3028550.

[8] *ns-3 - a discrete-event network simulator for internet systems*. Accessed: 2022-03-04. URL: https://www.nsnam.org/.

[9] *Python Glossary - Python Documentation*. Accessed: 2022-03-04. URL: https://docs.python.org/3/glossary.html#term-duck-typing.

[10] *PythonImplementations - Python Wiki*. Accessed: 2022-03-04. URL: https://wiki.python.org/moin/PythonImplementations.

[11] Robert E. Shannon. "Introduction to the art and science of simulation". En: *Proceedings of the 1998 Winter Simulation Conference*. Institute of Electrical y Electronics Engineers (IEEE), 1998.

[12] *SimPy documentation*. Accessed: 2022-03-04. URL: https://simpy.readthedocs.io/.

[13] Kragen Javier Sitaker. *isinstance() considered harmful*. Accessed: 2022-03-04. URL: http://canonical.org/~kragen/isinstance/.

[14] Christoph Sommer, Reinhard German y Falko Dressler. “Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis”. En: *Mobile Computing, IEEE Transactions on* 10 (feb. de 2011), págs. 3-15. DOI: 10.1109/TMC.2010.133.

[15] Bjarne Stroustrup. *C++ Applications*. Accessed: 2022-03-04. URL: https://www.stroustrup.com/applications.html.

[16] Andrew Tanenbaum. *Redes de computadoras*. Editorial Alhambra S. A. (SP), 2003. ISBN: 9789702601623.

[17] *Using Python to Run ns-3*. Accessed: 2022-03-04. URL: https://www.nsnam.org/docs/manual/html/python.html.

[18] András Varga. *OMNeT++ 5.6.1 User Manual*. Accessed: 2022-03-04. 2020. URL: https://doc.omnetpp.org/omnetpp/manual/.

[19] András Varga. “The OMNET++ discrete event simulation system”. En: *Proc. ESM'2001* 9 (ene. de 2001).

[20] András Varga y Rudolf Hornig. “An overview of the OMNeT++ simulation environment”. En: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops, SimuTools 2008*. Ene. de 2008, pág. 60. DOI: 10.1145/1416222.1416290.

[21] Antonio Virdis, Giovanni Stea y Giovanni Nardini. “Simulating LTE/LTE-Advanced Networks with SimuLTE”. En: ene. de 2016. ISBN: 978-3-319-26470-7. DOI: 10.1007/978-3-319-26470-7.

[22] Farzeen Zehra, Maha Javed y Darakhshan Khan. *Comparative Analysis of C++ and Python in Terms of Memory and Time*. Dic. de 2020. DOI: 10.20944/preprints202012.0516.v1.

Los abajo firmantes, miembros del Tribunal de evaluación de tesis, damos fe que el presente ejemplar impreso se corresponde con el aprobado por este Tribunal.