UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

PyDistSim: entorno de simulación de algoritmos distribuidos en redes

Informe de Proyecto de Grado presentado por

Agustín Recoba

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Javier Baliosian
Eduardo Grampin

Montevideo, 10 de diciembre de 2024

Agradecimientos

En cuanto al proyecto, principalmente a mis supervisores, Javier Baliosian y Eduardo Grampin y a Sebastián Pizard por su ayuda al definir el plan de pruebas. También a los alfa y beta *testers* que ayudaron a validar el desarrollo y a Damir Arbula y el equipo de desarrolladores del proyecto original Pymote.

Personalmente, por el apoyo a lo largo de la carrera: a mi novia, mi familia y a mis compañeros de estudio.

Resumen

En este proyecto se presenta PyDistSim, un *framework* de simulación de algoritmos distribuidos. Permite implementar, simular, visualizar y comparar algoritmos distribuidos. El nombre proviene de la contracción de *"Python Distributed Algorithms Simulator"* y apunta a ser una herramienta de referencia para la enseñanza de algoritmos distribuidos.

El diseño y la implementación tienen como eje principal su uso en el curso Análisis y Diseño de Algoritmos Distribuidos en Redes de la Facultad de Ingeniería de la UDELAR, con el propósito de reemplazar a DisJ, software que se usaba hasta el momento; mejorando en funcionalidades, usabilidad y facilidad de instalación.

Para validar las capacidades de la herramienta, se realizaron dos fases de pruebas alfa y beta con estudiantes de facultad seleccionados de forma idónea; concluyendo de forma muy satisfactoria con la utilización del simulador para la entrega de trabajos prácticos de la asignatura en su edición 2024.

PyDistSim está implementado en Python 3.11 y utiliza NetworkX para modelar las redes, y Jupyter Notebook y Matplotlib para la interacción y visualización.

PyDistSim es un *fork* de otra herramienta que intenta solventar la misma necesidad, Pymote, un simulador de código abierto basado en Python 2, pero con funcionalidades limitadas y sin mantenimiento. El trabajo de desarrollo se enfocó en actualizar la herramienta, agregar las funcionalidades faltantes y mejorar la usabilidad y la documentación.

Palabras clave: Algoritmos distribuidos, Simulación, DisJ, Python, Pymote

Índice general

Capítulo 1

Introducción

Como se mencionó en el resumen, este proyecto se centra en la construcción de una herramienta de simulación de algoritmos distribuidos, con el objetivo de ser utilizada en el curso ADADR de la Facultad de Ingeniería de la Universidad de la República. El proyecto también abarca: los planes de prueba realizados para validar la herramienta, el soporte dado durante su primer uso en el curso y la recolección de opiniones de los estudiantes. Dejamos fuera del alcance del proyecto el estudio de los aspectos didácticos en los que podría influir el uso de la herramienta. La base de esta decisión es la poca experiencia del autor en el área de la enseñanza y la necesidad de un análisis más profundo para poder realizar afirmaciones concretas.

1.1. Motivación y problema

Este proyecto surge de la necesidad de disponer de una mejor herramienta para la implementación de algoritmos distribuidos en el contexto del curso ADADR, necesidad evidenciada por comentarios de estudiantes de la edición 2023 del mismo, entre los cuales se incluye el autor del proyecto.

La herramienta usada hasta el momento era DisJ, un simulador que funcionaba correctamente y que brindaba la mayoría de las funcionalidades necesarias para el curso. Sin embargo, presentaba problemas de usabilidad, de instalación y de adecuación al contenido del curso.

Frente a esto, se propuso a los docentes de la asignatura (supervisores del proyecto) la realización de un proyecto de grado enfocado en resolver de alguna forma todas estas dificultades. Las opciones que se barajaron fueron: mejorar DisJ, reemplazarlo por una herramienta existente o construir una herramienta desde cero.

1.1.1. ¿Por qué se necesita un simulador de algoritmos distribuidos?

El estudio de algoritmos distribuidos constituye una rama fundamental de la algoritmia computacional, centrada en el diseño y análisis de algoritmos que operan de forma distribuida en redes de computadoras. Su importancia se refleja en su uso extensivo en aplicaciones críticas como:

- Sistemas de comunicación y redes
- Control industrial y automatización
- Infraestructura de transporte inteligente
- Redes de distribución energética
- Sistemas de seguridad y monitoreo

La naturaleza distribuida de estos algoritmos introduce una complejidad significativa respecto a los algoritmos tradicionales. Esta complejidad surge principalmente de:

- La necesidad de sincronización entre componentes
- La posibilidad de fallos en la comunicación
- La consistencia de estado entre nodos
- Las latencias variables en la red

Estas características hacen que el estudio teórico, aunque necesario, no sea suficiente: se requiere validar los planteamientos en condiciones que emulen escenarios reales. Por ello, resulta indispensable contar con herramientas de simulación que permitan implementar, simular y visualizar estos algoritmos en un entorno controlado antes de su despliegue en sistemas reales.

1.2. Objetivos

1.2.1. Objetivo general

El objetivo general del proyecto es disponibilizar una herramienta de simulación de algoritmos distribuidos, que permita la implementación, simulación y visualización de los mismos, con el propósito de ser utilizada en el curso ADADR. La misma debe reemplazar satisfactoriamente a la herramienta actual, de forma de que el cambio represente una mejoría sustancial en la experiencia por parte de los estudiantes.

1.2.2. Objetivos específicos

El objetivo general se verá satisfecho con el cumplimiento de los siguientes objetivos específicos:

- Implementación de un modelo de especificación de algoritmos distribuidos que sea similar (en las estructuras, las semánticas utilizadas e invocación de primitivas) al modelo teórico del libro (Santoro, 2006).
- Implementación de un entorno de simulación que permita la experimentación en redes con topologías arbitrarias y distintas condiciones de comunicación.
- Implementación de mecanismos que permitan la observación de la simulación y su evolución.
- Validación funcional de la herramienta en todos sus módulos.
- Validación de la herramienta en la posibilidad de utilizar sus funcionalidades no solo en los algoritmos que demanda ADADR, sino también en otros más complejos.
- Validación de la mejoría con respecto a DisJ, la herramienta anterior, en todas las facetas relevantes.
- Comprobación de la adecuabilidad de la herramienta a las necesidades de los estudiantes de ADADR.

1.3. Resultados esperados y alcanzados

El resultado esperado en las etapas iniciales del proyecto era contar con una herramienta completa en cuanto a funcionalidades indispensables y con algunas de las opcionales, sin dejar de cumplir con un estándar alto de usabilidad, facilidad de instalación y adecuación al contenido del curso.

Para las validaciones de corrección, mejoría y adecuabilidad, se esperaba contar con *testers* seleccionados específicamente para cada punto; mientras que para la validación de la completitud se esperaba realizar demostraciones técnicas implementadas por el autor del proyecto, que incluyeran indispensablemente la implementación de algoritmos más complejos que los que se piden en ADADR.

El resultado obtenido fue el esperado al inicio del proyecto, pero con algunos adicionales que resultan claves en cuanto a la validación de la calidad de la herramienta: se puso el software en manos de todos los estudiantes de la edición 2024 de ADADR, quienes lo utilizaron para la entrega de trabajos prácticos y luego participaron en dos instancias de *feedback*, una encuesta anónima y una entrevista grupal estilo *Focus Group*. Los resultados de ambas instancias fueron muy satisfactorios en todos los aspectos prioritarios para el proyecto y satisfactorios en general para los otros. Como conclusión principal, se comprobó que PyDistSim está listo para ser utilizado en el curso ADADR.

1.4. Estructura del documento

Antes de comenzar con el marco teórico, resumiremos la organización del documento para facilitar al lector la lectura y rápida consulta de aspectos particulares. El documento se organiza en capítulos con contenidos claramente delimitados: el Capítulo 2, de marco teórico, presenta los conceptos teóricos necesarios para entender el trabajo; entre ellos, el modelo de computación distribuida del libro (Santoro, 2006); en el Capítulo 3 se hace una revisión de antecedentes y se presentan herramientas de simulación que se encuentran disponibles y que son relevantes para el proyecto (se incluyen herramientas de simulación de eventos discretos y de algoritmos distribuidos en particular).

Terminada la parte de "introducción", tenemos el Capítulo 4, centrado en la construcción del *framework* en sí, se detallan los requisitos funcionales implementados, las decisiones tomadas y un resumen del diseño e implementación; en el Capítulo 5 pasamos a presentar el plan de pruebas, los resultados obtenidos y las conclusiones de los mismos. Aquí se incluye la validación de la herramienta con estudiantes de ADADR, las comparaciones con DisJ y la implementación de Mega-Merger. Se finaliza con el Capítulo 6, de conclusiones generales del trabajo, los aspectos mejorables y las posibles líneas de trabajo futuro.

Por último, se agregan los anexos A, B, C, D y E, que incluyen las encuestas realizadas, estadísticas de desarrollo con relación al esfuerzo y la contribución de código, manuales de usuario, manuales de desarrollo y el glosario.

Capítulo 2

Marco teórico

En el libro "Design and Analysis of Distributed Algorithms" (Santoro, 2006), DADA a partir de ahora, se presenta un modelo de computación distribuida diseñado para la didáctica de esta materia. En este capítulo se presentan los conceptos clave de este modelo, que son la base de la herramienta PyDistSim.

El universo que será contemplado por el modelo es una abstracción de las redes de computadoras, donde los nodos son las entidades autónomas y las aristas son los canales de comunicación entre ellas. Cada nodo tiene un estado interno y puede comunicarse con sus vecinos únicamente a través de mensajes. Estas entidades usarán sus capacidades de comunicación para realizar tareas en conjunto. Aunque cada nodo tenga capacidad de cómputo, no será especialmente relevante en el modelo, ya que el foco estará en la comunicación y coordinación entre las distintas entidades.

Computación distribuida con memoria compartida (*Shared Memory*)

El modelo de memoria compartida presupone la existencia de una memoria global a todos los nodos de una red, con características deseables como podrían ser la coherencia global, resiliencia frente a fallos, etc.

Hacemos énfasis en que el modelo no incorpora comunicación alguna que no sea a través de mensajes. Con esto aclaramos que no se contempla el modelo basado en memoria compartida, usado también extensivamente en otras literaturas e implementaciones productivas.

Sería incorrecto querer utilizar este marco para el estudio de algoritmos que funcionen con memoria compartida, ya que esta es una abstracción a un nivel superior que la de la comunicación por mensajería. Sin embargo, invitamos a los interesados en esta arquitectura a estudiar este modelo, más primitivo, con el objetivo de comprender cómo los sistemas de memoria compartida pueden basarse en la mensajería para lograr la abstracción de la memoria compartida y sus características.

2.1. Algoritmos y protocolos

En este universo, para resolver un problema se utilizarán algoritmos distribuidos o protocolos, conjuntos de reglas que especifican cómo cada entidad debe comportarse. Estos protocolos, en la mayoría de los casos, se ejecutan sin ningún tipo de sincronización o supervisión centralizada, en configuraciones de red arbitrarias. Dada esta naturaleza, es crucial poder demostrar que un protocolo es correcto, es decir, que cumple con su objetivo sin importar el estado inicial de la red (a menos de restricciones específicas) y que lo hace con un uso eficiente de los recursos que tiene disponibles.

Se observa que la acepción de *"protocolo"* que se usa en DADA no coincide con el concepto comúnmente usado en el ámbito de estudio de redes de computadoras, que generalmente usa la palabra para referirse al conjunto de reglas de formato, estructura de mensajes y comportamiento externo durante la comunicación entre entidades.

La principal particularidad de este modelo con respecto a otros modelos es que los protocolos serán planteados como reactivos, es decir, que las entidades reaccionan a eventos que ocurren en la red. A opinión de Santoro, esta forma es muy didáctica y permite una mejor comprensión de los problemas y soluciones. Al estudiar un protocolo, uno puede centrarse en estudiar las reacciones individuales, interpretándolas como "pasos", y luego en la coordinación de estos pasos para lograr el objetivo final.

2.2. Entidades

La entidad es la unidad computacional mínima del modelo. Dependiendo del sistema que está siendo modelado, esta entidad puede interpretarse como un proceso, un procesador, un *router*, un *switch*, etc. Se referirá como V al conjunto de todas las entidades en la red.

Cada entidad $x \in V$ posee acceso a una memoria local privada M_x. Las entidades pueden acceder a dicha memoria para almacenar o recuperar información. Además, las entidades tienen la capacidad de realizar cómputo local y de comunicarse (enviar y recibir mensajes) con otras entidades.

La memoria de una entidad incluye un conjunto de registros predefinidos cuyos valores iniciales también son predefinidos. Entre ellos, se encuentra el registro de estado, $status(x)$, y el registro de entrada, $value(x)$. El registro de estado toma valores de un conjunto predefinido y finito S, ejemplos de valores pueden ser: "$dormido$", "$esperando$", "$finalizado$", etc. El registro de entrada es un registro que se utiliza para almacenar información que llega a la entidad a través de mensajes.

Adicionalmente, cada entidad tiene un reloj interno que le permite establecer *timers*.

2.2.1. Eventos

Como mencionamos antes, el comportamiento de una entidad es reactivo, es decir, reacciona a eventos externos que ocurren en la red. Por consecuencia, una entidad que no registra eventos no realiza ninguna acción.

Para el modelo se considerarán tres tipos de eventos: impulsos espontáneos, mensajes recibidos y alarmas autoconfiguradas.

La llegada de mensajes y el disparo de alarmas son eventos externos a la entidad, pero que se originan dentro del sistema. Por otro lado, los impulsos espontáneos son eventos que son externos al sistema y que escapan de la "percepción" del universo de la entidad. Ejemplos de estos impulsos espontáneos pueden ser el pedido de un usuario, la llegada de información de un sistema externo, etc.

2.2.2. Acciones y comportamiento

Al detectar un evento, una entidad realizará una "acción": una secuencia de operaciones que debe ser atómica, finita y debe terminar.

Se contempla que una acción pueda ser no hacer nada: la acción *nil*.

Nos referiremos al comportamiento de una entidad como una función que toma un evento y un estado y devuelve una acción. Formalmente, el comportamiento de una entidad x es una función $B_x : E \times S \to A$, donde E es el conjunto de eventos, S es el conjunto de estados y A es el conjunto de acciones. Esta relación es llamada "regla".

La especificación de comportamiento de una entidad, será entonces el conjunto de reglas que definen cómo la entidad reacciona a los eventos que le llegan. Este conjunto debe ser completo, es decir, debe haber una regla para cada evento posible; y debe ser no ambiguo, es decir, para un evento y un estado, debe haber una única acción posible.

Frecuentemente, a este conjunto de reglas se le es llamado "protocolo" o "algoritmo distribuido".

Por último, Santoro se centra en utilizar protocolos *homogéneos*, lo que significa que todas las entidades tienen el mismo conjunto de reglas, pero cuyo rol en la ejecución puede ser distinto según las condiciones iniciales o aspectos externos al comportamiento.

2.3. Comunicación

En un sistema distribuido, la comunicación entre entidades es un aspecto fundamental. En el modelo en DADA, la comunicación se realiza a través de mensajes. El mensaje es la unidad de comunicación mínima y, en su definición más abstracta, es una secuencia finita de bits.

En general, una entidad no puede enviar mensajes directamente a cualquier otra entidad de un sistema. Denotamos como $N_{out}(x) \subseteq V$ al conjunto de entidades a las que x puede enviar mensajes, y $N_{in}(x) \subseteq V$ al conjunto de entidades

que pueden enviar mensajes a x. En general, $N_{out}(x) \neq N_{in}(x)$. Llamaremos a estos conjuntos los vecinos de salida y entrada de x, respectivamente.

La relación obtenida a partir de N_{out} y N_{in} nos permite definir un grafo dirigido $G = (V, E)$, donde $E = \{(x, y) \in V \times V : y \in N_{out}(x)\}$. Este grafo, entonces, modela la red de comunicación, donde cada nodo es una entidad y cada arista denota que una entidad puede enviar mensajes a la otra.

Los mensajes serán procesados en el orden en el que llegan a la entidad. En el caso de que dos mensajes lleguen al mismo tiempo, no se especifica un orden de procesamiento. Las entidades y la comunicación pueden fallar de formas que veremos más adelante.

2.4. Axiomas y restricciones

2.4.1. Axiomas

Todo el contenido de DADA está basado en dos hipótesis básicas, referidas como axiomas fundamentales:

- **Axioma de retraso en las comunicaciones:** los mensajes enviados por una entidad serán entregados en tiempo finito, a excepción de fallas en la comunicación.
- **Axioma de orientación local:** una entidad puede distinguir e identificar a sus vecinos. Llevado a los conjuntos N_{out} y N_{in}, esto significa que una entidad puede distinguir a sus vecinos de salida y puede distinguir a sus vecinos de entrada; pero no necesariamente podrá establecer una correspondencia entre los vecinos de salida y los de entrada.

En otras palabras, el primer axioma establece que, a menos que un mensaje se pierda, el mismo llegará eventualmente.

El segundo axioma establece que una entidad podrá saber de qué vecino procede un mensaje o a cuáles vecinos quiere enviar un mensaje. Esto denota la existencia de dos funciones locales $\lambda_x^{in}(y)$ y $\lambda_x^{out}(y)$ que asocian etiquetas o números de puerto a los vecinos de entrada y salida, respectivamente. Esta función es inyectiva. Para reforzar que esta función es local, aclaramos que cada arista (a, b) en la red tendrá dos etiquetas: $\lambda_a^{out}(b)$ y $\lambda_b^{in}(a)$ las cuales no necesariamente son iguales.

2.4.2. Restricciones

En general, los sistemas distribuidos son muy distintos en cuanto a las condiciones de comunicación y a las capacidades de las entidades. Para poder tener en cuenta estas diferencias, se establece una serie de restricciones que limitan el poder de las entidades y de la comunicación. Estas restricciones serán explicitadas en cada protocolo y serán parte de la especificación del mismo. En algunos casos, el protocolo se diseñará para que funcione correctamente a pesar

de estas restricciones; en otros casos, el protocolo se diseñará para aprovecharlas y funcionar mejor en ese contexto.

Didácticamente, las restricciones también sirven para guiar la implementación, de forma que inicialmente se implemente una versión muy restringida, pero simple, y luego se vayan agregando complejidades para liberar las restricciones.

En el libro se especifican muchas restricciones, agrupadas en categorías más generales:

- **Restricciones de comunicación.** Hablan, por ejemplo, del ordenamiento que siguen los mensajes dentro de un enlace y de la asignación de etiquetas λ_x.
- **Restricciones de fiabilidad.** Restringen temas de fiabilidad o propensidad a fallos de las comunicaciones, entidades y su eventual detección.
- **Restricciones topológicas.** Son las que impone la topología de una red, como la conexidad, el "tipo morfológico" del grafo asociado (anillo, completo, estrella), etc.
- **Restricciones temporales.** Refieren a aquellas relacionadas con el tiempo, como sincronización de relojes, retrasos en la entrega de mensajes, etc.

2.5. Ejemplo de un protocolo: *Broadcast* mediante *Flooding*

Al igual que hace Santoro en DADA, presentaremos un ejemplo de protocolo para ilustrar cómo se relacionan los conceptos presentados hasta ahora.

El *broadcast* es un problema clásico en los sistemas distribuidos. Consiste en que una entidad, llamada emisora, haga llegar cierta información a todas las entidades de la red. Para simplificar el problema, se asume que se cumplen cuatro restricciones básicas: un único iniciador[1], la red es conexa, enlaces bidireccionales[2] y fiabilidad total[3].

El protocolo de *"Flooding"* (inundación), es un protocolo simple que resuelve este problema. Consiste en que el emisor envía un mensaje a todos sus vecinos, y cada entidad que recibe el mensaje por primera vez, lo reenvía a todos sus vecinos, excepto al que le envió el mensaje. Este proceso se repite hasta que todos los nodos de la red han recibido el mensaje.

La adaptación de este protocolo al modelo de DADA es sencilla. Cada entidad tiene un estado que puede ser "*iniciador*", "*dormido*" o "$finalizado$". Todas las entidades ejecutan el mismo protocolo, que consiste en:

[1] Un solo emisor desea enviar información.

[2] Para todos los nodos, el conjunto de vecinos de entrada es el mismo que el de salida, y los números de puerto de salida son iguales a los de entrada.

[3] No ocurrirá ningún tipo de falla.

- Si el estado es "*iniciador*" y se detecta un impulso espontáneo, se envía la información a todos los vecinos y se cambia a "$finalizado$".
- Si el estado es "*dormido*" y se recibe un mensaje, se envía el mensaje a todos los vecinos y se cambia a "$finalizado$".
- Para cualquier otra combinación de estado y evento, no se realiza ninguna acción.

Además de las restricciones de las cuales depende el algoritmo, el conjunto de reglas que deberá ejecutar cada nodo y el conjunto de estados posibles, una especificación completa deberá incluir además los estados iniciales y finales que se admiten. Recordamos que, como DADA se centra en protocolos homogéneos, no es necesario dar un conjunto de reglas para cada entidad.

Dicho esto, presentamos cómo se vería el pseudocódigo de este protocolo en el modelo de DADA en la Figura 2.1.

```
PROTOCOL Flooding

- Status Values: S = {INICIADOR, DORMIDO, FINALIZADO}
- Initial Status = {INICIADOR, DORMIDO}
- Final Status = {FINALIZADO}

- Restricciones: Enlaces bidireccionales, fiabilidad total,
                 único iniciador, conexidad.

INICIADOR
    Spontaneously
    begin
        send(I) to N(x);
        become FINALIZADO;
    end

DORMIDO
    Receiving(I)
    begin
        send(I) to N(x) - {sender};
        become FINALIZADO;
    end
```

Figura 2.1: Pseudocódigo de *Flooding*, adaptado de (Santoro, 2006)

Capítulo 3

Revisión de antecedentes

En este trabajo se presenta una herramienta de simulación de algoritmos distribuidos, que se basa en el modelo teórico presentado en el Capítulo 2. Un aspecto importante de este tipo de herramientas es que caen dentro de la categoría de simuladores de eventos discretos, que son ampliamente utilizados en la investigación de sistemas complejos.

Nos centraremos en librerías y herramientas para Python, porque creemos que es la mejor opción en cuanto a flexibilidad y facilidad de instalación, además de que es el lenguaje de programación que tiene una proyección de uso mayor en los próximos años (TIOBE organization, s.f.). Se harán algunas excepciones en la investigación cuando se encuentren opciones que lo ameriten.

Basándonos en esto, cuando comenzamos la investigación de antecedentes, la dividimos en dos partes: simulación de eventos discretos y simulación de algoritmos distribuidos.

La motivación de la primera categoría es que, si bien la simulación de eventos discretos es un campo muy amplio y con muchas aplicaciones, la simulación de algoritmos distribuidos es un subcampo más específico y con menos herramientas disponibles. A la hora de planificar el desarrollo de PyDistSim, se tuvo que tomar la decisión entre implementar sobre un simulador de eventos discretos como "motor", o implementar desde cero los sistemas de colas. Para este relevamiento nos limitaremos firmemente a simuladores de código abierto para Python.

La investigación de la segunda categoría es más directa, ya que se busca entender si el trabajo propuesto ya ha sido realizado, y si es así, qué aspectos se pueden mejorar o qué aspectos se pueden tomar como referencia. También se considera la posibilidad de extender el trabajo existente, si es que existe y es posible.

3.1. Simulación de eventos discretos

La simulación de eventos discretos es una técnica de simulación en la que el sistema modelado evoluciona en instantes discretos de tiempo, en los cuales

ocurren eventos que cambian el estado del sistema. Cada evento es procesado en orden de ocurrencia y puede generar nuevos eventos futuros.

Es una técnica muy utilizada en la investigación de sistemas complejos, ya que permite modelar sistemas con comportamientos no lineales, no deterministas y con múltiples interacciones. Además, es una técnica muy flexible, ya que permite modelar sistemas con diferentes niveles de detalle y complejidad.

En general, un simulador de eventos discretos consta de tres componentes principales: el modelo del sistema, el motor de simulación y la visualización de resultados. El modelo del sistema define las entidades, su comportamiento y los eventos que forman parte del sistema; el motor de simulación ejecuta los eventos en orden de ocurrencia, y la visualización de resultados muestra los resultados de la simulación.

3.1.1. SimPy

SimPy (Scherfke y cols., 2020) es un *framework* de simulación de eventos discretos basado en procesos para Python. El principal aspecto de la versión actual, SimPy 3, es que implementa una interfaz sencilla para la definición de procesos basada en generadores de Python. Es un *framework* muy conocido y utilizado en la comunidad de Python, con una documentación muy completa y una comunidad activa, además de que está en constante desarrollo/mantenimiento.

Otro punto a favor es que es código abierto y tiene licencia MIT.

Por otro lado, el uso de los mecanismos más complejos de la herramienta, como los recursos compartidos o la interacción entre procesos, son más difíciles de entender y de implementar.

No dispone de módulos, funciones o instrucciones acerca de cómo visualizar de alguna manera las simulaciones. La comunidad ha tendido a utilizar pygame (McGugan, 2007), una librería de videojuegos en Python, para crear visualizaciones.

3.1.2. Salabim

Salabim (van der Ham, 2018) es otro *framework* de simulación de eventos discretos desarrollado para Python 3. Tiene incluido un módulo para crear visualizaciones de la simulación rápidamente. Es código abierto y tiene licencia MIT. El módulo de visualizaciones parece lo suficientemente potente (ver Figura 3.1) para lo que se necesita en este proyecto[1].

Como principal competidor de SimPy, Salabim tiene una documentación igual de completa pero menos organizada y de menor calidad. La comunidad es mucho más pequeña y menos activa, lo que puede ser un problema a la hora de buscar ayuda o soluciones a problemas específicos.

[1] Varias demostraciones más en `https://www.salabim.org/`.

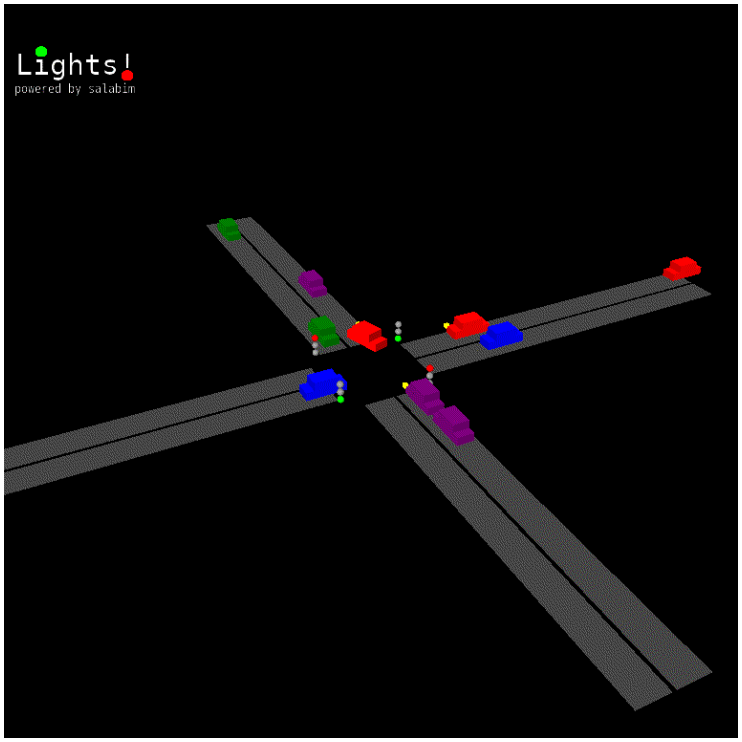


Figura 3.1: Captura de una animación de Salabim. Simula un sistema de semáforos y el cruce de autos.

3.2. Simulación para algoritmos distribuidos

3.2.1. DisJ

DisJ (Piyasin, 2013) es un simulador de algoritmos distribuidos que implementa 4 modelos de computación distintos, incluyendo uno muy similar al que se da en la bibliografía de ADADR.

El simulador se debe usar integrado con Eclipse, el conocido IDE de Java. Dicha integración no es trivial: los algoritmos deben ser desarrollados dentro del IDE y las redes sobre las cuales se va a simular deben ser creadas usando un complemento externo a DisJ.

DisJ lleva más de 10 años sin ser actualizada. Lo mismo con el complemento mencionado. Esto provoca que la configuración inicial del simulador sea problemática: precisa de una versión no actual de Eclipse, y configuración manual de las dependencias.

Además, aunque el simulador provee 4 interfaces distintas (los 4 modelos de computación) para la implementación de los algoritmos, resulta muy cerrado a implementaciones que no estén 100 % alineadas con las intenciones originales. Creemos que esto demuestra poca confianza en las capacidades del desarrollador o usuario. No permite modificar o extender la lógica de simulación interna y se recurre a prácticas como implementar muchas veces la misma operación (hay 56 implementaciones distintas de la operación `sendTo`), interpretamos esto como que intenta "llevar de la mano" al desarrollador principiante.

Por último, la interfaz correspondiente al modelo de pasaje de mensajes no es "simétrica" al que se usa en DADA (si es equivalente y, por lo tanto, suficiente como para usarlo en el curso). La diferencia radica en que en vez de implementar las acciones por estado y evento, lo hace solamente por eventos, lo que lleva a implementaciones más complejas, menos parecidas a la referencia teórica y, por lo tanto, más propensas a errores.

Es flexible en cuanto a las condiciones de simulación que se pueden recrear. Permite definir las distribuciones de probabilidad de los retrasos en los mensajes, la política de ordenamiento del *inbox* de cada entidad y la posibilidad de fallas en la comunicación, entre otras no relevantes para el modelo del DADA. Para la definición de redes, cuenta con un conjunto de facilidades que se llama "Topology Library" que permite dibujar varias topologías conocidas. En el aspecto que se ve más limitado es en la definición de los algoritmos: no se cuenta con interfaces para desactivar o modificar el tiempo de alarmas previamente establecidas, tampoco existe un mecanismo para definir o restringir la información que es capaz de acceder cada entidad, ni cualquier otro tipo de restricción a nivel del algoritmo.

Estas cuestiones fueron las que presentó el autor para evidenciar que era necesario un cambio.

Detalles de implementación del *core* de la simulación

De manera genérica, se define una interfaz de administrador de algoritmos única: `IProcessor`. Define las operaciones para administrar una corrida (setear grafo, pausar, seguir, obtener estados, etc.).

El simulador del modelo de computación usado en ADADR se implementa en `MsgPassingProcessor`, que usa una cola de eventos (como pueden ser inicializar, ring de alarma, llegada de mensaje, etc.) única para todas las entidades, lo que permite ir ejecutando las acciones asociadas una a una.

El orden de ejecución de cada evento es según su timestamp (seteado al momento de creación del mismo y en función del tipo de evento que representa). En caso de empates en timestamp, se atienden por la prioridad del tipo de evento (fijo y predefinido). La prioridad de los eventos es: inicialización, ring de alarma, llegada de mensaje.

Naturalmente, la detección global de la finalización de una simulación es equivalente a obtener una cola de eventos vacía.

Interfaz gráfica

Permite crear topologías, asignar propiedades a los enlaces y nodos de la red y seleccionar de forma independiente qué algoritmo va a correr. Permite pausar la corrida del mismo, además de bajar/subir la velocidad o reiniciarla.

Para la implementación, usa el patrón arquitectónico MVC clásico y con los componentes bien separados. La clase `SimulatorEngine` es el controlador, que provee los métodos para controlar la simulación y la información para visualizarla. Para la parte gráfica, usa SWT y Eclipse JDT UI.

Como se puede ver en la Figura 3.2, para el estándar actual, es una interfaz gráfica pobre y ligeramente fea. Tiene *bugs* molestos como la necesidad de cerrar y abrir la ventana de simulación para que se tomen los cambios que se realizan al algoritmo que se está corriendo, lo que enlentece y genera cierto hastío en el ciclo de desarrollo.

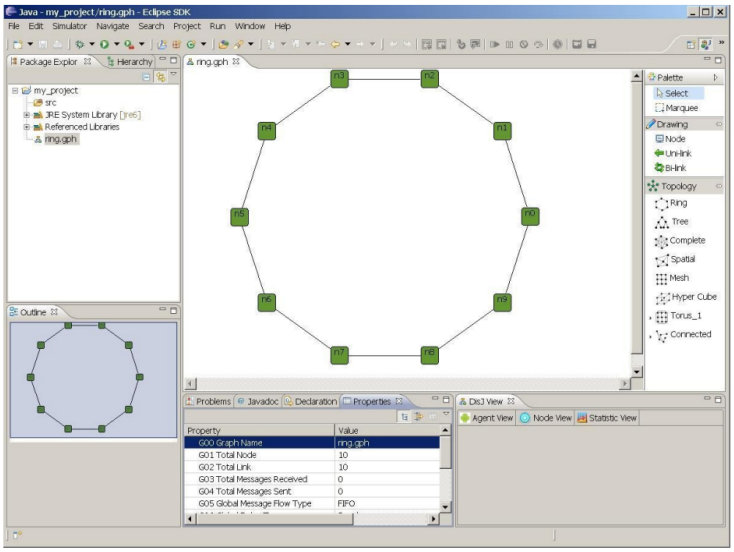


Figura 3.2: Captura de la interfaz gráfica de DisJ, tomada de su manual de usuario.

Documentación

Como comentario final, la documentación de DisJ es muy completa y detallada en lo que refiere a manuales de uso; explica muy bien cómo usar cada una de las funcionalidades de la herramienta. Sin embargo, no se pudo encontrar documentación de desarrollo, lo que dificultó la comprensión de la lógica interna de la herramienta.

3.2.2. Pymote

Pymote (Arbula y Lenac, 2013) es un simulador en Python 2 basado en DADA. Implementa el modelo de computación distribuida por estados y eventos. No está integrado en un IDE específico, aunque no tiene tantas funcionalidades como DisJ.

Entre sus puntos fuertes se encuentra que es un proyecto de código abierto con licencia BSD, aunque es un proyecto muerto, con 6 años sin contribuciones.

Otro punto importante es que usa la biblioteca NetworkX como *backend* de los grafos/redes y tiene una considerable integración con la terminal interactiva de Python y los cuadernos de Jupyter.

Analizando la herramienta, se concluyó que la documentación creada y el exhaustivo uso de comentarios en el código aportan mucho a la calidad de Pymote en sí. En ningún momento se tuvo que "deducir" que hacía cierto módulo o clase.

Entre las principales deficiencias de la herramienta tenemos que su parte

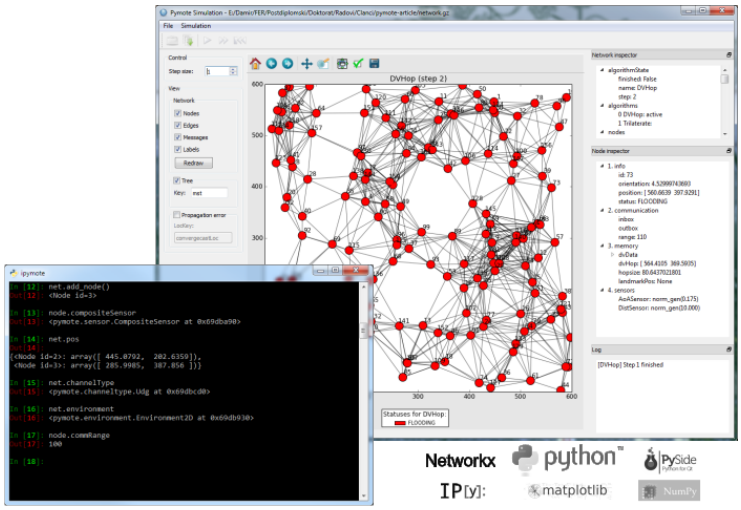


Figura 3.3: Captura demostrando el uso de Pymote, una ventana con IPython para configurar el ambiente y otra hecha con Qt para visualizar el resultado.

gráfica no permite modificar las redes, ni cambiar el algoritmo a usar con las mismas. Tampoco incorpora el mecanismo de *timers* ni la posibilidad de introducir retrasos o pérdidas en la comunicación.

Detalles de implementación del *core* de simulación

A diferencia de DisJ, no hay manejo de eventos de manera explícita. La simulación es un bucle que realiza las mismas operaciones en secuencia. Permite avanzar de a pasos.

Las operaciones que realiza son:

1. Inicializar algoritmo (solo se hace en la primera iteración). Coloca el mensaje INI en el *inbox* del nodo inicializador.
2. Para cada nodo, ejecuta un solo "paso". Toma un solo mensaje del *inbox* y lo procesa (ejecuta la acción asociada a la TAG recibida con el mensaje).
3. Cuando los nodos se quedan sin mensajes para recibir, se termina la simulación.

Los mensajes del *inbox* de cada nodo no son solamente mensajes de otros nodos, sino también mensajes producto de alarmas e de impulsos espontáneos.

Interfaz gráfica

La parte gráfica está hecha con `Qt` (Molkentin, 2007), a través del *binding* de Python: `pyside` (Musayev, 2023). A través de esta interfaz gráfica se pueden cargar redes, algoritmos y correr simulaciones. La ventana solo sirve para

visualizar el resultado de la simulación, no para modificar la red o el algoritmo. Contiene un panel a la derecha que permite ver el estado interno de cada nodo. La simulación solo se puede parar, correr de a pasos o reiniciarse.

La configuración de la red y del algoritmo que correrá se hace a partir de la creación de un solo archivo *pickle* de Python, el cual debe crearse a través de la terminal interactiva de Python o de un cuaderno de Jupyter. Esto quiere decir que para cada par (red, algoritmo) se debe crear un archivo distinto. No es posible generar un archivo de algoritmo y usarlo con varias redes o viceversa. Estos archivos se cargan en la ventana de visualización, ver Figura 3.3.

El redibujado de la red se da a partir de señales que ejecuta el simulador a cada paso que realiza. No reconozco el nombre de este patrón de arquitectura (si lo tiene), pero tiene el obvio problema de que acopla muchísimo el módulo de la UI (y la tecnología usada) con el *core* de simulación.

3.2.3. Sinalgo

Sinalgo (EDC-Group y cols., 2008) es un *framework* de simulación, con el propósito de probar y validar algoritmos para redes. Usa Java 5. No se basa en la literatura de Santoro. Fue desarrollado por el grupo de computación distribuida de ETH Zürich ("ETH Zurich", s.f.). Es altamente configurable. Permite re-implementar todo (y está diseñado para eso) según las necesidades del desarrollador. Es código abierto con licencia BSD-3. Tiene integrada una interfaz gráfica que permite visualizar la simulación en tiempo real, ver Figura 3.4.

No ha tenido actualizaciones desde hace 5 años ni tiene ningún *fork* activo.

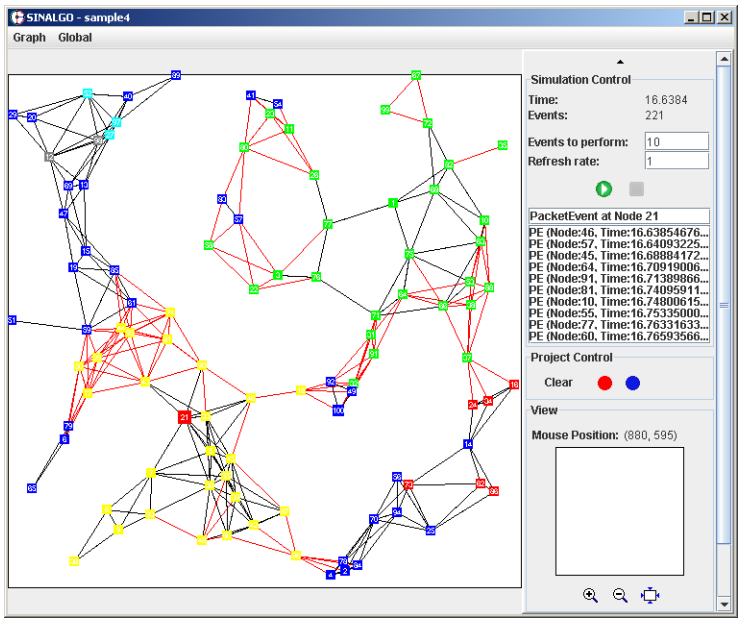


Figura 3.4: Captura extraía de la documentación de Sinalgo. Muestra la interfaz gráfica de la herramienta.

3.2.4. The JBotSim Library

JBotSim (Casteigts, 2015) es una librería de simulación para algoritmos distribuidos en redes dinámicas. El modelo de programación es orientado a eventos. El dinamismo de la red puede ser programado o hecho en tiempo real a través de la interfaz gráfica. Los algoritmos se programan en Java 8 o superior. Es código abierto con licencia GNU Lesser GPL 3.

La forma de usarlo es similar a DisJ, pero sin la dependencia de Eclipse. Las interfaces que tiene para la definición de protocolos son tan parecidas a DisJ, que da la impresión de que uno debía ser derivado del otro, pero no.

Al igual que DisJ, no sigue el modelo de DADA y además está muy enfocado al dinamismo en las redes: nodos móviles, cambios en la topología, etc. No es lo que se necesita para el curso.

3.2.5. Resumen y decisión del punto de inicio

Por último, antes de continuar el capítulo de construcción, concluiremos con una comparación de las herramientas investigadas.

En cuanto a simulación de eventos discretos, se considera que ninguna de las herramientas presentadas provee lo que necesitamos: SimPy es muy flexible, pero no tiene una interfaz gráfica y el aprendizaje necesario para implementaciones complejas no parece justificar su uso; Salabim tiene una interfaz gráfica muy

buena, pero no tiene una comunidad activa y la documentación es pobre, lo que casi asegura problemas a futuro. Basándonos en esto, descartamos el uso de un simulador de eventos discretos como motor para nuestro desarrollo.

Por otro lado, las herramientas de simulación de algoritmos distribuidos presentadas tienen sus pros y contras. DisJ es muy completo y flexible, pero su alta integración con Eclipse condiciona el tiempo que podrá dedicarse a la implementación de nuevas funcionalidades. Pymote es más simple y mucho menos completo, pero su simplicidad y la facilidad de uso de Python lo hacen una buena opción para el desarrollo de PyDistSim. Luego, JBotSim ya mencionamos que se especializa en redes dinámicas, algo que no es prioritario en DADA; y Sinalgo es muy configurable y flexible, pero su implementación en Java y la falta de actualizaciones recientes lo hacen una opción mucho menos atractiva.

Dado el análisis de antecedentes realizados, la decisión final se redujo a dos opciones: mejorar DisJ y solucionar los problemas detectados o actualizar Pymote y agregar las funcionalidades faltantes.

Si se trabajase sobre DisJ, habría que enfocarse en extraerlo del ecosistema de Eclipse y llevarlo a algo más independiente; lo que implica tener que armar la GUI de 0. Obviamente, también habría que trabajar sobre todos los problemas que se detallaron en la Sección 3.2.1.

Si se optara por continuar el trabajo de Pymote, hay que trabajar en llevar la implementación de Python 2 a Python 3, agregar todas las funcionalidades básicas faltantes y expandir sobre ideas nuevas.

Como se adelantó en el resumen del comienzo del trabajo, se decidió que PyDistSim sea un *fork* de Pymote. La decisión estuvo altamente influenciada por la experiencia con los dos lenguajes involucrados. Se cuenta con mucha más experiencia en Python que en Java, usado solamente en contextos relacionados con la facultad como los talleres de programación. Por otro lado, la documentación de Pymote es significativamente superior a la de DisJ (apenas tiene) y, al ser un proyecto más chico, se podrían emprender cambios grandes sin tanto costo.

Capítulo 4

Construcción del *framework*

En este capítulo se detallarán las decisiones tomadas en la construcción de PyDistSim. Se comenzará con la definición de los requisitos, luego se describirá el diseño de la solución y finalmente se presentará la implementación realizada.

En el Anexo C se agregan detalles relacionados con el esfuerzo que requirió la construcción, además de estadísticas que ayudan a comprender qué proporción del código generado es heredado de Pymote.

4.1. Funcionalidades y atributos de calidad

En esta sección se detallarán los requisitos implementados durante el proyecto, dejando para la Sección 6.2.1 los puntos no conseguidos y posibles desarrollos a futuro. Se categorizan los mismos en función de a qué punto del proceso del desarrollo y simulación de un algoritmo corresponde.

También se detallarán los requisitos no funcionales más importantes y las características de la documentación y manuales que se generaron.

4.1.1. Implementación de algoritmos distribuidos

1. **Definición de comportamiento de entidades.** La herramienta establece mecanismos específicos para la definición precisa del comportamiento de una entidad frente a una acción de tipo impulso espontáneo, recibimiento de mensajes y alerta de alarmas autoconfiguradas.
   Además de ser una especificación completa bajo el modelo teórico, el mecanismo de definición es tal que el código generado para dicha especificación es muy similar a las especificaciones teóricas del DADA.

2. **Interfaces básicas.** La herramienta provee métodos y funciones para la invocación de los procedimientos primitivos definidos en el modelo teórico:

enviar mensajes, definir *timers*, conocer a los vecinos de una entidad dada, etc.

3. **Interfaces extras.** La herramienta provee métodos y funciones para la realización de operaciones que pueden resultar repetitivas de implementar, como, por ejemplo, enviar un mensaje a todos los vecinos.

4. **Manejo avanzado del *timer*.** La herramienta provee métodos y funciones para un manejo completo de los *timer*. Dado un *timer* ya creado, se puede modificar el tiempo (acortar o extender), desactivarlo y chequear si ya fue ejecutado. También se admite la creación de más de un *timer* de forma concurrente.

5. **Explicitación del acceso a la información.** En la definición de cada comportamiento, es explícita la fuente de toda la información usada. Si el acceso a cierta información es una restricción inherente al algoritmo, es posible marcarlo con el mecanismo de restricciones.

6. **Acceso a la información.** En la misma línea que el punto anterior, el acceso a información "privilegiada" (todo aquello que se obtiene por medios que no sean la mensajería o inicialización de la red), es desaconsejado (de forma programática y en documentación) pero no prohibido, dando al programador la decisión final en cada caso.

4.1.2. Generación de redes

1. **Definición de redes como grafos.** La herramienta permite la definición de una red de nodos como un grafo dirigido o no dirigido, sin limitación a las topologías posibles, a menos de restricciones de memoria. Todas las instancias son utilizables para la simulación de los algoritmos.

2. **Características de simulación personalizables.** La herramienta permite la especificación de atributos de comportamiento de cada red. Incluye características a nivel canal o enlace: retraso, pérdida de mensajes y ordenamiento de los mismos; y características de los nodos: información inicial de la que dispone o si es capaz de recibir impulsos espontáneos.

3. **Creación de topologías comunes.** La herramienta implementa interfaces para la creación de redes con características topológicas comunes, como el anillo, *mesh*, completo e hipercubo.

4. **Creación de redes aleatorias sujetas a condiciones.** La herramienta implementa interfaces para la creación de redes aleatorias, pero con restricciones idóneas para la simulación como: nivel de conexidad, cotas para la cantidad de nodos y grado promedio.

4.1.3. Simulación

1. **Simulación de los algoritmos.** La herramienta permite la simulación (por pasos o de forma completa) de los algoritmos implementados por los usuarios siguiendo la especificación mencionada (ver Ítem 1, de la Sección 4.1.1), en las redes elegidas por los mismos. La simulación lleva un concepto de tiempo global y avanza monótonamente sobre el mismo. La misma respeta las condiciones establecidas tanto por la red como por el algoritmo, ejecutando las rutinas de acciones de forma sincrónica, manteniendo la causalidad de los eventos y respetando las colas de mensajería de cada entidad.
   Al finalizar la simulación, se pueden comprobar los estados de los nodos, así como la información recavada por los mismos, de forma que el usuario pueda entender si el algoritmo cumplió su propósito.

2. **Inspección parcial de una simulación.** En cualquier paso o momento de la simulación, el usuario puede parar la misma e inspeccionar el estado interno de cada nodo, cola de mensajes o cola de eventos.

3. **Detección del fin de la simulación.** La simulación finaliza automáticamente cuando no quedan eventos pendientes de ser procesados.

4. **Reiniciar la simulación.** La herramienta permite reiniciar una simulación al conjunto de condiciones iniciales de la misma.

5. **Inspección detallada de una corrida.** La herramienta brinda mecanismos, orientados a los usuarios avanzados, para inspeccionar el comportamiento interno de una simulación, como envío y recibimiento de mensajes, cambio de estado de cada nodo, pasos de simulación realizados, etc.

6. **Especificación de restricciones.** La herramienta provee interfaces específicas para la especificación de restricciones de los algoritmos (ver Sección 2.4.2).

7. **Uso de las restricciones.** Las restricciones especificadas en el Ítem 6 pueden ser asignadas a la definición de comportamiento de las entidades, de forma que una simulación que no cumpla con las restricciones sea abortada.
   Este mecanismo no es obligatorio, pero sí es el comportamiento por defecto de la herramienta.

8. **Pre-especificación de restricciones usadas en DADA.** La herramienta predefine las restricciones más usadas en el libro de referencia de ADADR y otras también útiles. Las mismas son:

   - *"Message Ordering"*: Los mensajes llegan en el orden en que fueron enviados.

- *"Reciprocal Communication"*: Para cada nodo, el conjunto de nodos que pueden enviarle mensajes es igual al conjunto de nodos a los que puede enviar mensajes.
- *"Bidirectional Links"*: Se cumple *"Reciprocal Communication"* y además los nodos son capaces de distinguir qué arista de entrada corresponde a qué arista de salida.
- *"Initial Distinct Values"*: Cada nodo tiene un valor único al inicio de la simulación.
- *"Network Size"*: Todos los nodos conocen la cantidad de nodos en la red.
- *"Connectivity"*: La red es conexa.
- *"Unique Initiator"*: Un único nodo recibirá un impulso espontáneo al inicio de la simulación.
- *"Complete Graph"*: La red es un grafo completo.
- *"Cycle Graph"* (o *"Ring"*): La red es un anillo.
- *"Tree Graph"*: La red es un árbol.
- *"Star Graph"*: La red es un grafo estrella.
- *"Total Reliability"*: El sistema, en su totalidad, no tiene fallos de enlaces ni de nodos.
- *"Partial Reliability"*: No ocurrirán más fallos en el futuro (pero pueden haber ocurrido en el pasado).
- *"Guaranteed Delivery"*: Todos los mensajes enviados serán recibidos con su contenido sin corromperse.
- *"Bounded Communication Delays"*: Los mensajes no tardarán más de un tiempo T, finito, en llegar a su destino.
- *"Unitary Communication Delays"*: Los mensajes tardarán exactamente una unidad de tiempo en llegar a su destino.
- *"Synchronized Clocks"*: Todos los nodos tienen relojes sincronizados.
- *"Simultaneous Start"*: Todos los nodos comienzan la simulación al mismo tiempo. No garantiza que sus relojes se mantengan sincronizados a partir de ese instante.

4.1.4. Visualización gráfica

1. **Visualización estática de una red.** Se puede visualizar la red en cualquier momento, independientemente de la simulación y el algoritmo. Se tiene la posibilidad de cambiar los tamaños, colores, etiquetas y posición de los nodos. También se puede excluir tanto nodos como aristas de la visualización.

2. **Visualización animada de las ejecuciones.** Se cuenta con un módulo que permite la creación de animaciones de las simulaciones, mostrando el estado de los nodos y las comunicaciones entre ellos en cada paso de la simulación.
   En particular, se usan colores para representar los estados de los nodos y de los mensajes según la cola en la que se encuentren (*inbox*, *outbox*, atrasados y perdidos).
   A la hora de configurar la animación, se tendrán las mismas opciones que en la visualización estática de la red (1); y se podrá elegir la velocidad de la animación, la cantidad de pasos a mostrar por segundo y la cantidad de pasos a mostrar en total.

3. **Exportar animaciones a formatos de video.** La herramienta permite exportar las animaciones generadas a formatos de video comunes, como MP4 o GIF.

4.1.5. *Benchmarking* de algoritmos

1. ***Benchmarking* automatizado.** La herramienta provee mecanismos para hacer *benchmark* de los algoritmos implementados, permitiendo la comparación de los mismos en distintas redes y condiciones de simulación.
   Se podrán estudiar métricas como tiempo de finalización, cantidad de mensajes enviados, cantidad de mensajes recibidos, cantidad de mensajes perdidos, etc.

2. **Gráficas de resultados.** Se facilita la generación de gráficas idóneas con los resultados obtenidos en el *benchmarking*, de forma de que sea sencillo extraer conclusiones de las métricas en función de las condiciones de simulación.

4.1.6. Requisitos no funcionales: atributos de calidad

Además de la correctitud funcional, PyDistSim apunta a cumplir con atributos de calidad igualmente de deseables que las funcionalidades principales. De ellos, tres se destacan como fundamentales:

1. Usabilidad

2. Facilidad de instalación y configuración

3. Independencia de entornos de ejecución específicos

Rememorando la Sección 3.2.1, observamos que estos puntos son las principales desventajas encontradas en el software que se intenta reemplazar (DisJ).

Usabilidad

Para esta sección nos basaremos en la norma ISO 25000 (comúnmente conocida como *SQuaRE*) (ISO 25000:2014, 2014).

SQuaRe define cuatro formas de descomponer la usabilidad: *"Effectivenes in use"*, *"Efficiency in use"*, *"Satisfaction in use"* y *"Usability in use compliance"*.

La primera refiere a la capacidad de los usuarios de cumplir sus objetivos mediante el uso del software, ponderando la completitud de contextos y la precisión del resultado.

La segunda, la eficiencia en uso, apunta a un buen balance entre los recursos que debe invertir el usuario sobre el retorno que obtiene. PyDistSim, al ser un software de simulación con aplicación directa en la educación, tiene solo dos insumos de uso: el tiempo y carga cognitiva dedicados a programar las simulaciones y la extracción de conclusiones de las mismas.

La tercera, sobre la satisfacción, es exactamente eso: el grado de satisfacción que experimentan los usuarios con el uso del software. Se suele descomponer esta categoría en dos aspectos: cognitiva (el software cumple la funcionalidad que el usuario necesitaba) y emocional (el usuario se siente atraído a usar más la aplicación).

PyDistSim se propone alcanzar la suficiencia en estas tres verticales. Ir al Capítulo 5 para entender cómo se medirá esto en las pruebas de usuario y ver los resultados obtenidos.

Facilidad de instalación y configuración

Para este punto no tendremos una definición estricta de qué es lo correcto, pero sí es un punto que se medirá en las pruebas de usuario. Se tomaron decisiones de diseño para que la instalación y configuración de PyDistSim sea sencilla y no requiera de conocimientos avanzados de programación o de sistemas operativos. También se tuvo en cuenta que la cantidad de pasos en el proceso de instalación sea la mínima posible.

Independencia de entornos de ejecución específicos

PyDistSim se desarrolló en Python 3.11, lo que significa que es compatible con cualquier sistema operativo que soporte dicha versión de Python. Además, se evitó el uso de librerías específicas de un sistema operativo o de un entorno de ejecución específico. La herramienta se puede ejecutar en cualquier entorno que soporte Python 3.11 o superior.

Adicionalmente, dado a las últimas modas de programación científica, existen muchas herramientas que permiten la ejecución de Python y cuadernos de Jupyter en la nube, lo que facilita la ejecución de PyDistSim en cualquier entorno, directo desde el navegador, sin necesidad de instalar ningún software.

4.1.7. Documentación y manuales

Para la documentación y manuales de uso se siguió lo que hoy en día es el estándar en la industria de software: generar una web de documentación con Sphinx. Incluye una guía de instalación, una guía de uso, una guía de referencia de la API y una guía de contribución. Además, se generaron muchos ejemplos de uso y tutoriales para facilitar la adopción de la herramienta por parte de los usuarios. Puede consultar el Anexo E para acceder a la web publicada.

4.2. Diseño e implementación

En esta sección expondremos los aspectos más importantes del diseño e implementación de PyDistSim. Se detallará la estructura de los módulos, la arquitectura de la aplicación y las decisiones de diseño más importantes.

4.2.1. Lineamientos de diseño y arquitectura

A lo largo de la implementación se tuvieron en cuenta los siguientes lineamientos de diseño y arquitectura:

- Mantener el diseño orientado a objetos de Pymote, aprovechando las características de programación funcional de Python cuando sea oportuno.
- Seguir los principios de diseño SOLID: se buscó que las clases y módulos sean simples, cohesivos y con responsabilidades bien definidas. En particular, se utilizó ampliamente de la inyección de dependencias para desacoplar los módulos y facilitar la reutilización y extensión del código.
- Desarrollar pensando que el *framework* será usado por desarrolladores y, por lo tanto, se priorizó la claridad de las interfaces, la simplicidad de uso y la consistencia en la nomenclatura y la estructura de los módulos. Siguiendo este pensamiento, y en virtud de respetar la inteligencia del usuario, se evitó ocultar o encapsular en exceso la lógica interna del *framework*, más bien apuntando a la fácil comprensión y extensión del código.
- Conservar la esencia del proyecto: no realizar implementaciones que no estén relacionadas con la simulación de algoritmos distribuidos en el contexto específico del marco teórico de DADA. Esto implica no agregar modelos de computación alternativos, no agregar funcionalidades que no sean útiles para la simulación de algoritmos distribuidos y no agregar restricciones que no estén contempladas en el libro de referencia.
- Uso de tipado en interfaces y documentación: se utilizó el tipado opcional de Python para especificar los tipos de los parámetros y retornos de las funciones y métodos, así como para documentar las interfaces y clases. Esto facilita la comprensión del código y la detección de errores en tiempo de desarrollo, así como la generación de documentación automática y la usabilidad de la API.

4.2.2. Estructura de módulos

PyDistSim está organizado en módulos que representan las diferentes partes del famework. Son: `network`, `algorithm`, `simulation`, `restrictions`, `logging`, `message`, `observers`, `metrics`, `benchmark` y `gui`. A continuación se detallará el propósito de cada uno de ellos.

- `network`: Contiene las clases y funciones necesarias para la definición y manipulación de redes. Incluye las subclases de `networkx.Graph`: `DirectedNetwork` y `BidirectionalNetwork`, que representan redes direccionadas y bidireccionales, con capacidad de comunicación entre sus nodos. Toda la lógica de comunicación se encuentra aquí.

- `algorithm`: Contiene las clases y funciones necesarias para la definición y ejecución de algoritmos distribuidos. Incluye la definición de la clase abstracta `BaseAlgorithm` y su subclase especializada en computación distribuida, `NodeAlgorithm`. Por la alta cohesión con `NodeAlgorithm`, también se incluyen en este módulo las clases `_NodeWrapper` y `WrapperManager`, principales responsables de administrar el accseso a la información a la que disponen los nodos según las restricciones de la simulación.

- `simulation`: Contiene la clase `Simulation`, que es la encargada de orquestar la simulación de uno o varios algoritmos en una red dada. Se encarga de avanzar la simulación, ejecutando las acciones de los nodos y manteniendo la consistencia de la simulación.

- `restrictions`: Contiene las clases que definen cada restricción implementada, así como la interfaz que todas implementan: `Restriction`. Se incluye una capa más de abstracción: una diferenciación que se tuvo que hacer para distinguir aquellas restricciones que son posibles de chequear contra una red en particular vs. aquellas que se deberán aplicar para que la restricción sea cumplida.

- `logging`: Módulo que configura y gestiona los *logs* que se usarán de forma interna en el *framework*. También expone funciones para que el usuario pueda configurar los *logs* a su gusto.

- `message`: Contiene la clase `Message` para la definición y manipulación de mensajes.

- `observers`: Contiene las clases necesarias para la definición de clases observables y sus observadores. Incluye la definición del *mixin* `ObserverManagerMixin` (para los observables) y la interfaz `Observer` (observadores).

- `metrics`: Implementación puntual de un *observer*, `MetricCollector`, que es capaz de observar todas las clases observables y recopilar métricas de las mismas.

- `benchmark`: Contiene las clases y funciones necesarias para la realización de *benchmarks* de algoritmos. Incluye la definición de la clase `AlgorithmBenchmark`, responsable de la corrida, recopilación de datos y generación de gráficos de los *benchmarks*.

- `gui`: Contiene el submódulo **drawing**, que incluye la lógica de dibujado de las redes y animaciones de la simulación para Jupyter e IPython. Además, este módulo contiene parte del código deprecado de la interfaz gráfica de Pymote. Está parcialmente funcional, pero no se recomienda su uso.

4.2.3. Simulación de algoritmos distribuidos

Para esta parte describiremos a grandes rasgos cómo funciona la definición y ejecución de un algoritmo distribuido en PyDistSim.

Protocolo

Como vimos en el modelo teórico, en el Capítulo 2, la ejecución de un algoritmo distribuido se basa en la reacción de los nodos a los eventos, en función del estado. Con base en eso es que se definía un protocolo como el conjunto de asociaciones $((\text{estado}, \text{evento}) \to \text{acción})$.

Intentando replicar esto de la forma más fiel posible, en PyDistSim, un protocolo se implementa al hacer una subclase de `NodeAlgorithm` y definir los métodos que representan las acciones que se pueden realizar en cada evento, además del enumerado de estados posibles y el conjunto de restricciones que aplican al protocolo.

El enumerado de estados, que obviamente define cuáles son los mismos, también almacena la asociación $((\text{estado}, \text{evento}) \to \text{acción})$. Para esto, se requiere que el mismo sea una subclase de `StatusValues`. El mecanismo que brindamos es el siguiente:

1. Como los estados serán instancias de `StatusValues`, tendrán sobrecargado el operador de llamada (prácticamente, se pueden llamar como funciones).

2. El objetivo de esta función es que recibirá un método de la clase con el nombre de la acción a la que se quiere asociar (*"receiving"*, *"spontaneously"* y *"alarm"*).

3. Al ser llamado, se guardará la asociación en un diccionario interno de la clase, que luego se usará para ejecutar la acción correspondiente en el momento adecuado.

4. Finalmente, ya que tenemos una función que recibe métodos, se puede usar como un decorador de métodos de la clase y se evita tener que hacer una llamada explícita a la función de asociación[1].

[1] Recordamos que el uso de `@deco` para decorar una función es equivalente a `meth = deco(meth)` cuando la función ya está definida.

Es complicado de explicar todo esto en conjunto, para eso se deja de ejemplo cómo se traduce el protocolo de la Figura 2.1 a PyDistSim.

```
class Flooding(NodeAlgorithm):

    class Status(StatusValues):
        # Se define cada estado posible
        INICIADOR = "INICIADOR"
        DORMIDO = "DORMIDO"
        FINALIZADO = "FINALIZADO"

    # Se define la lógica de cada acción en cada estado

    @Status.INICIADOR
    def spontaneously(self, node, message):
        self.send(
            node,
            data=INFO,
            destination=node.neighbors()
        )
        node.status = self.Status.FINALIZADO

    @Status.DORMIDO
    def receiving(self, node, message):
        self.send(
            node,
            data=message.data,
            destination=node.neighbors() - {message.source}
        )
        node.status = self.Status.FINALIZADO
```

De esta manera, queda definida la asociación de acciones de forma inequívoca a nivel funcional, pero también al leer el código. Se sugiere volver a la Figura 2.1 para apreciar cómo con este mecanismo conseguimos acercar todavía más la definición de una clase Python al pseudocódigo usado en DADA.

Colecciones de mensajes y eventos

Para la implementación de PyDistSim se mantuvo la característica de Pymote de que no hubiera una única cola para toda la simulación. Específicamente, hay 3 tipos de cola que se utilizarán:

1. **Cola de entrada:** Cada nodo mantiene una cola con los mensajes que tiene pendientes de procesar. Esta cola cumple siempre la política FIFO.

2. **Cola de salida:** Cada nodo mantiene una cola con los mensajes que generó durante el último paso de simulación. Esta cola también siempre cumple la política FIFO.

3. **Cola de tránsito:** Para cada arista y para cada dirección, se mantiene una cola con los mensajes que fueron enviados, pero que aún no llegaron a su destino. Esta cola no está ordenada a menos que la red tenga especificado el ordenamiento de mensajes como característica de comportamiento. Además de cada mensaje, mantiene el *delay*, o sea, la cantidad de pasos que faltan para que el mensaje sea entregado. Notar que si el ordenamiento está activado, el *delay* de un mensaje será igual al máximo *delay* de los mensajes que lo preceden y él mismo.

Además de las colas, hay 2 colecciones que se utilizan para información relacionada con eventos:

1. **Mensajes perdidos:** Para cada arista y para cada dirección, se mantiene una colección con los mensajes que fueron enviados, pero que se perdieron por fallos de comunicación. Se mantiene ordenada por fines de análisis de la simulación, pero no existen mecanismos para volver a incorporar estos mensajes en la comunicación.

2. **Alarmas:** Para cada algoritmo instanciado, se mantiene una única colección de alarmas, asociada cada una a un nodo. No está asociado directamente a la red como las otras colecciones porque escapa a la responsabilidad de la misma.

Flujo base de la simulación

La simulación se realiza de a pasos que se componen en: una fase de comunicación, una fase de procesamiento de alarmas y una fase de computación local de cada nodo.

Para la comunicación, a grandes rasgos, lo que se hace es tomar todos los mensajes en cola de salida y colocarlos en la cola de tránsito con un *delay* que será asignado según la política configurada para la red. Este *delay* es un entero mayor a uno. Una vez que se colocaron todos los mensajes en la cola de tránsito: se procesa la misma, disminuyendo en uno el *delay* de cada mensaje. Si el *delay* llega a cero, el mensaje se coloca en la cola de entrada del nodo destino.

La fase de alarmas consiste en chequear si la alarma debe ser activada en este paso y, si es así, formar un mensaje de alarma y colocarla en el *inbox* del nodo asociado a la misma.

Para la computación local, se toma el mensaje más antiguo de la cola de entrada y se ejecuta la acción correspondiente según el estado del nodo. Una vez que se ejecutó la acción, se elimina el mensaje.

Adicionales

Ahora que tenemos definido el esquema base de la simulación, podemos empezar a agregar los comportamientos opcionales.

El primer punto notable es que todas las simulaciones comienzan con un paso de inicialización, definido también por el protocolo para establecer las condiciones de las que depende para funcionar correctamente. Este paso suele incluir la asignación de identificadores a cada nodo, la elección de cuáles nodos comenzarán con un impulso espontáneo, entre otros.

Adicionalmente, la comunicación de cada nodo está condicionada a la configuración de comportamiento de la red. Esta configuración establece:

- El ordenamiento de mensajes (ya explicado en la Sección 4.2.3).
- La probabilidad de que un mensaje se pierda. Se modela como una función de probabilidad que recibe el mensaje y la red y devuelve un booleano. Cómo se computa ese booleano es completamente flexible y puede depender de cualquier característica de la red o del mensaje.
- El *delay* que está asociado a cada mensaje. Igual al caso anterior, se modela como una función de probabilidad que recibe el mensaje y la red y devuelve un entero mayor a uno.
- La velocidad de incremento de los relojes internos de cada nodo. Se modela como una función de probabilidad que recibe el nodo y devuelve un entero mayor a uno, representando la cantidad de tics que debe dar el reloj en cada paso de simulación.
- La velocidad de procesamiento de cada nodo. Se modela como una función de probabilidad que recibe el nodo y devuelve un entero mayor a uno, representando cada cuántos pasos de simulación se procesará un mensaje.

Todas estas configuraciones se realizan a partir de la creación de instancias de la clase `NetworkBehaviorModel` y se asigna al atributo `behavioral_properties` de la red sobre la que se va a simular.

4.2.4. Modelado de redes

Para la definición de redes, se decidió utilizar la biblioteca NetworkX (Hagberg y Conway, s.f.), que es una biblioteca de Python para la creación, manipulación y estudio de estructuras de redes complejas. NetworkX proporciona una interfaz simple y flexible para la definición de grafos y la manipulación de sus propiedades. Como estructura de datos base, provee la clase `Graph`, que es una generalización de los grafos no dirigidos; y `DiGraph`, que es una generalización de los grafos dirigidos.

Esta dualidad de grafos dirigidos y no dirigidos nos fuerza a recrear la misma estructura en PyDistSim. Se implementaron dos clases: `DirectedNetwork` y `BidirectionalNetwork`. La primera modela las redes dirigidas, mientras que la segunda modela las redes bidireccionales. Ambas clases heredan de `networkx.DiGraph` y `networkx.Graph`, respectivamente, y agregan funcionalidades específicas que necesitábamos para la simulación de algoritmos distribuidos. Para lograr esto, se utilizó la técnica de herencia múltiple, que es una

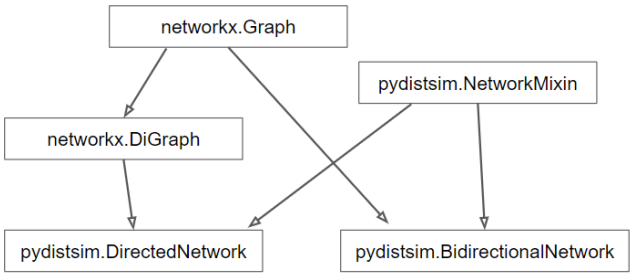


Figura 4.1: Diagrama de herencias de NetworkMixin.

característica de Python que permite a una clase heredar de más de una clase base. Se generó la clase `NetworkMixin`, que no es más que un *mixin* diseñado para agregar funcionalidades a las clases de `networkx` y de esa manera poder reutilizar el código.

Se muestra el diagrama de herencia en la Figura 4.1.

Las funcionalidades/responsabilidades que agrega `NetworkMixin` a las clases se pueden resumir en: posicionamiento de los nodos en un espacio bi o tridimensional, aplicación de propiedades de comportamiento (ver 4.2.3), capacidad de creación de árboles recubridores en función de la información distribuida en los nodos y, principalmente, comunicación por mensajería con retrasos y pérdida.

4.2.5. Ayuda para la creación de redes

Como se mencionó en el punto 3, de la sección de requisitos, PyDistSim provee una serie de funciones para la generación de redes comunes en el contexto de algoritmos distribuidos. Estas funciones se encuentran en la clase `NetworkGenerator` y se pueden importar directamente desde el paquete principal de PyDistSim.

Si bien esta funcionalidad no aparenta ser estrictamente necesaria, definimos que era importante incluirla para reducir la fricción en el uso. Se trata de que los usuarios no tengan que dedicarle tiempo a especificar qué topología tendrá cada red que quieran usar; ayuda a que puedan concentrarse en la parte importante: la implementación y prueba de los algoritmos.

Las topologías que se decidieron incorporar en esta etapa son, sencillamente, todas aquellas que se mencionan de forma explícita en DADA. Estas son: el anillo, estrella, *mesh*, completo e hipercubo. Se muestran las 5 variantes en la Figura 4.2.

Por otro lado, se implementó también la posibilidad de generar fácilmente redes aleatorias (Ítem 4 de la Sección 4.1.2). La generación de redes aleatorias es útil para efectuar pruebas de algoritmos en redes de gran tamaño y con topologías no estándar. Los parámetros que permite definir son: cantidad de nodos, conectividad, grado promedio, tolerancia sobre el grado promedio y la cantidad de nodos, si la red será dirigida o no, y la distancia máxima para que

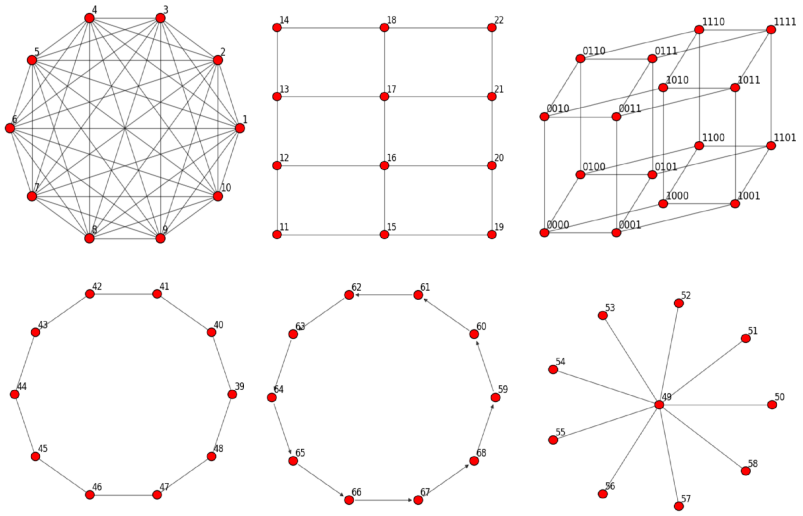


Figura 4.2: Redes generadas por topología. Notar que el segundo anillo es dirigido.

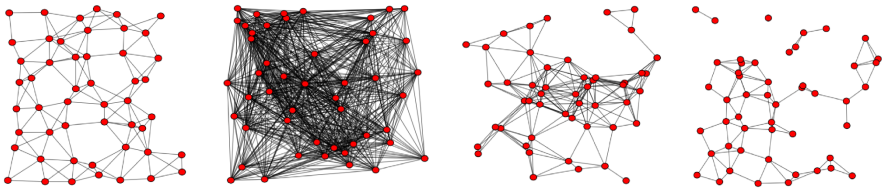


Figura 4.3: Redes generadas aleatoriamente, varios parámetros.

dos nodos estén conectados. Se muestran algunos ejemplos en la Figura 4.3.

4.2.6. *Observers* multipropósito, *benchmarking* e integración con la GUI

Se decidió usar el patrón de diseño `Obersver` (Gamma, 1995) para implementar dos aspectos que necesitaban de la suscripción a eventos: la recolección de métricas para el *benchmarking* de algoritmos y la señal de redibujado de la interfaz gráfica.

Gracias a este punto se logra solucionar en gran medida un aspecto que criticamos cuando se hizo el análisis de Pymote en la Sección 3.2.2: lo mucho que estaba acoplado el módulo de interfaz gráfica con el resto del código. En PyDistSim, el *core* es completamente independiente de la implementación puntual gráfica que se esté usando, y la comunicación entre ambos se realiza a través del *observer* `QThreadObserver` y la interfaz de la clase `Simulation`.

Aprovechando el mismo recurso, se definió la clase `MetricCollector`. Es un *observer* que se suscribe a todos los eventos posibles: de simulación, de comunicación en la red y de acciones en los algoritmos. De esta forma, se puede recolectar métricas de cualquier aspecto de la simulación y se pueden generar gráficos de los resultados obtenidos. Por defecto, se usa para medir los aspectos clave de la algoritmia de DADA: cantidad de mensajes enviados, cantidad de mensajes recibidos y cantidad de pasos de simulación.

La definición de estos mecanismos se logra mediante dos clases: `Observer` y `ObserverManagerMixin`. La primera es una interfaz que define los métodos que deben implementar los observadores, y la segunda es una clase *mixin* que se encarga de agregar la funcionalidad de observable a cualquier otra clase. De esta forma, cualquier clase que herede de `ObserverManagerMixin` podrá ser observada por cualquier clase que implemente la interfaz `Observer`. Se puede ver el diagrama de herencias en las figuras 4.4 y 4.5.

`ObserverManagerMixin` se encarga de mantener la lista de observadores y de notificar a cada uno de ellos cuando se produce un evento. Los observadores, a su vez, implementan los métodos `on_{EVENT_NAME}`, que se ejecutan cuando se produce un evento.

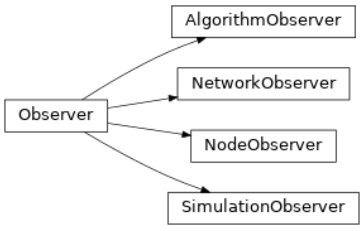


Figura 4.4: Diagrama de herencias de `Observer`.

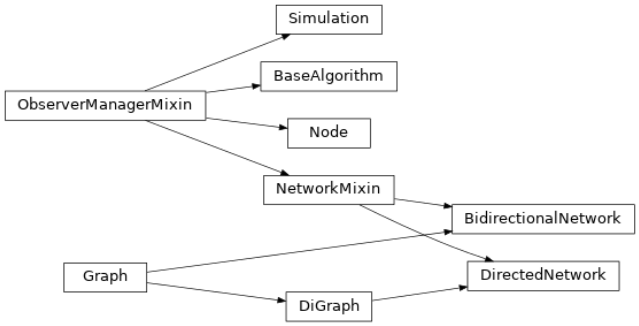


Figura 4.5: Diagrama de herencias de `ObserverManagerMixin`.

4.3. Tecnologías y dependencias

Se concluirá el capítulo con un resumen de las tecnologías y herramientas usadas en el desarrollo de PyDistSim, aunque muchas de ellas ya se mencionaron en la sección anterior.

4.3.1. Dependencias de instalación

PyDistSim es un paquete de Python 3, lo que significa que **Python es el lenguaje de programación principal** utilizado para interactuar con él. El código fuente de PyDistSim también está escrito en Python.

Estas son las tecnologías, herramientas y lenguajes que utiliza PyDistSim:

- **Python:** Como lenguaje principal, Python se utiliza para escribir algoritmos distribuidos, definir redes, configurar simulaciones y analizar resultados. PyDistSim requiere específicamente Python 3.11 o superior.
- **NetworkX:** PyDistSim utiliza la biblioteca NetworkX (Hagberg y Conway, s.f.) para representar grafos, que son fundamentales para modelar la topología de la red en sistemas distribuidos. NetworkX proporciona herramientas para crear, manipular y analizar grafos, lo que facilita el manejo de las redes en PyDistSim.
- **IPython y Jupyter Notebook:** PyDistSim admite IPython Console (Pérez y Granger, 2007) y Jupyter Notebooks (Randles, Pasquetto, Golshan, y Borgman, 2017) como interfaces principales para ejecutar simulaciones e interactuar con el *framework*. Estas herramientas proporcionan un entorno interactivo para experimentar con los algoritmos distribuidos, visualizar resultados y analizar datos.
- ***Virtual Environments* (venv):** Aunque no es obligatorio, se recomienda utilizar entornos virtuales para gestionar las dependencias de PyDistSim y evitar conflictos con otros paquetes de Python instalados en el sistema.

- **Herramientas de prueba:** PyDistSim utiliza pytest para las pruebas unitarias y de integración, pytest-cov y coverage para el análisis de cobertura de código, y memray (Wozniski, s.f.) para la evaluación del uso de memoria.
- **Documentación con Sphinx:** La web de documentación de PyDistSim se genera utilizando Sphinx (*Sphinx*, s.f.), una herramienta popular para crear documentación de software en Python. Se utilizan extensiones como intersphinx y seed_intersphinx_mapping para enlazar automáticamente con documentación externa.
- **Generación de gráficas:** En el módulo de *benchmarking* (ítem 2 de la Sección 4.1.5), se usa Maplotlib (Tosi, 2009) en conjunto con SeaBorn (Waskom, 2021) para la generación de gráficas personalizables a partir de tablas de datos. También se usa Matplotlib como *backend* para la visualización de las redes y animaciones de la simulación.
- **Computación científica con SciPy y Numpy:** SciPy (Bressert, 2012) se utiliza en el contexto de los sensores, específicamente para definir funciones de probabilidad que modelan la incertidumbre de las mediciones. NumPy (Van Der Walt, Colbert, y Varoquaux, 2011) se usa a lo largo de todo el *framework* como proveedor de funciones matemáticas elementales que no están incluidas en la librería estándar de Python 3.
- ***Logging:*** Para esta tarea se utiliza Loguru (Delgan, s.f.), librería que provee una interfaz de *logging* muy sencilla pero también muy personalizable.

4.3.2. Dependencias débiles

Con dependencia débil, referimos a que se utilizaron servicios o paquetes externos para cumplir con algunas necesidades rutinarias. Son muy fáciles de reemplazar por otra herramienta similar y la elección de cada una se dio principalmente por consideraciones de facilidad de uso y costo. A continuación se detalla cada uno.

- **Calidad de código y estandarización de formato:** Para gestionar el uso de varias herramientas de este estilo, se usan los *"hooks"* de Git a través de la aplicación PreCommit ("PreCommit: A framework for managing and maintaining multi-language pre-commit hooks.", s.f.). En la misma se configuraron herramientas de chequeo automático de sintaxis, formato y fijación de dependencias, entre otros.
- **GitHub:** Para el control de versiones y publicación del código de PyDistSim, se utiliza GitHub. GitHub proporciona un entorno de colaboración para el desarrollo de software, incluyendo control de versiones, seguimiento de problemas, solicitudes de mejoras, entre otros.
- **Travis CI:** En integración continua, se utiliza Travis CI para ejecutar pruebas automatizadas en cada *commit* de código. Travis CI se integra con

GitHub y permite ejecutar pruebas en un entorno de construcción aislado. También permite definir una matriz de pruebas para ejecutar pruebas en diferentes versiones de Python y sistemas operativos. Luego de cada prueba, se comunica con GitHub para reportar el resultado de las pruebas y con Coveralls para reportar la cobertura de código.

- **Coveralls:** Coveralls se utiliza para medir la cobertura del código de PyDistSim. Coveralls se integra con GitHub y Travis CI para recopilar y mostrar estadísticas de cobertura de código en cada *commit* de código. Además, adjunta reportes de cobertura de código a cada *merge request*, lo que permite a los desarrolladores revisar la cobertura de código antes de fusionar cambios.

- **ReadTheDocs:** Para la generación y publicación de la documentación de PyDistSim, se utiliza ReadTheDocs (Read the Docs, s.f.). ReadTheDocs es un servicio de alojamiento de documentación que se integra con GitHub y Sphinx para generar y publicar documentación automáticamente a partir del código fuente. Se configuró para compilar nuevas versiones de la configuración para cada *merge request* y poder ver la documentación de la rama en cuestión.

- **PyPi:** A través de GitHub Actions, se usa PyPi para la distribución de PyDistSim. PyPi es el principal repositorio de paquetes de Python y se utiliza para distribuir paquetes de Python a través de `pip`. PyDistSim se publica en PyPi para que los usuarios puedan instalarlo fácilmente.

Capítulo 5

Experimentación

El software a desarrollar en este proyecto de grado es un simulador de algoritmos distribuidos en redes, para poder ser utilizado en el curso ADADR. En resumen, el objetivo del mismo es que supere en funcionalidades y usabilidad al simulador que se usa actualmente (DisJ), y que sea, en cierta medida, resistente al paso del tiempo.

Las secciones 5.1 y 5.2 refieren a las pruebas de usuario que se realizaron. En esta parte del trabajo destacamos la ayuda especializada de Sebastián Pizard, docente de la FING del Grupo de Ingeniería de Software (GRIS, s.f.). En la Sección 5.3 mostramos otro tipo de prueba, interna, en la que se intenta explorar los límites del simulador implementando un algoritmo mucho más complejo de los que se suelen pedir como ejercicio para ADADR.

Además de lo mencionado recién, se agrega que el desarrollo fue acompañado de la creación de pruebas unitarias y de integración entre los módulos. Se crearon 90 casos de prueba, cubriendo un 88 %[1] del código total.

5.1. Alfa *Testing*

Llamamos *"Alfa Testing"*, o simplemente fase alfa, a la primera evaluación externa que tuvo el proyecto. Se realizó apenas se terminó con el desarrollo del *core* de la herramienta, pero previo al desarrollo de las funcionalidades de visualización enumeradas en la Sección 4.1.4.

El objetivo de esta fase es evaluar principalmente la usabilidad desde la perspectiva del usuario y encontrar fallos que no hayan sido encontrados durante el desarrollo o las experimentaciones propias. Otro objetivo clave de esta etapa es la comparación con el software que se está reemplazando: DisJ.

A excepción de ese último punto, la evaluación tendrá un enfoque para identificar lo más macro y busca responder la pregunta "¿PyDistSim está listo para ser usado en ADADR?". Poder responder esto de la manera más realista posi-

[1] Más detalles en `https://coveralls.io/github/agustin-recoba/PyDistSim`

ble, nos habilitará a realizar la siguiente fase de pruebas en la edición 2024 del curso.

5.1.1. Plan

Para esta prueba, se esperaba poder conseguir aproximadamente tres *testers*, idealmente estudiantes de ediciones anteriores de ADADR para aportar en cuanto a las comparaciones con DisJ. Finalmente, se consiguió solo un *tester* con esta característica, a través de un módulo de taller para estudiantes de la FING. No se establecieron otros criterios específicos para la selección de los *testers*.

La prueba se llevó a cabo en un escenario ya conocido del curso, donde el *tester* utilizó su propia computadora personal. De esta manera, se simularán las condiciones reales de uso que los estudiantes enfrentarán al utilizar PyDistSim en el aula o en sus estudios. El problema a resolver fue el 2.9.4 del DADA, extraído directamente del EVA del curso ("Análisis y Diseño de Algoritmos Distribuidos en Redes", s.f.).

Letra del problema 2.9.4: Considere un grafo genérico donde cada entidad x tiene un valor inicial $v(x)$; estos valores no son necesariamente distintos. El *rank* de una entidad x será el *rank* de su valor, esto es, $rank(x) = 1 + |\{y \in V : v(y) < v(x)\}|$ (la entidad que tenga el menor valor tendrá *rank* 1). Diseñe un protocolo eficiente para determinar el *rank* de todas las entidades, pruebe su correctitud y analice su complejidad.

Se asume:

- Enlaces bidireccionales
- Confiabilidad total
- Mensajes en orden
- Iniciador único

Además de la letra, se entregó una guía de resolución que es parte del material docente de uso de apoyo al libro, con el fin de desviar la atención de la resolución teórica y apuntar exclusivamente a la resolución práctica.

Se instó al *tester* a explorar y utilizar todas las características principales del simulador, asegurando que estas funcionen correctamente y que la experiencia de usuario sea sólida y sin interrupciones. Se buscó identificar cualquier problema significativo que pueda afectar el uso general del software y recoger impresiones iniciales sobre su efectividad en un entorno educativo.

La recolección de *feedback* se llevó a cabo principalmente a través de formularios de respuesta estructurada (respuestas 1 al 5 y opciones NS/NC y "No entendí el enunciado.") y muy pocas preguntas de respuesta libre.

Para la confección del cuestionario nos basamos en (Ralph, 2020) y las normas ISO de usabilidad: ISO 9241, ISO 9126, ISO 14598, e ISO 25000.

La comunicación con el *tester* fue principalmente por correo electrónico, con consultas menores sobre funcionalidades e instrucciones claras sobre lo que se esperaba de él.

Consideraciones adicionales

Notamos a continuación algunos puntos que fueron tenidos en cuenta en la realización del cuestionario. Nuevamente, agradecemos el apoyo de Sebastián Pizard en la guía para enfocar correctamente las preguntas.

- El software a probar es un reemplazo de otro, por lo que, además de las preguntas directas de comparación *PyDistSim vs. DisJ*, se debe poder deducir indirectamente a partir de las otras respuestas si el software es mejor o peor que el actual.

- Se usarán preguntas "demográficas" para categorizar al entrevistado según su experiencia con sistemas distribuidos y con simuladores de eventos discretos. Dicha clasificación se podría usar para ponderar los resultados de las otras preguntas realizadas y poder comparar (indirectamente) nuestro simulador con las experiencias previas de los encuestados.

- Se usarán preguntas que no aportan en la respuesta, pero sirven para situar al encuestado en un estado mental consciente de lo que está evaluando. Por ejemplo: "¿Estás familiarizado con los desafíos técnicos asociados a la implementación de un simulador de sistemas distribuidos?".

Metodología para la evaluación de resultados

El motivo de usar encuestas con valor 1-5 es poder extraer conclusiones rápidamente. Gracias a eso, se pudo analizar los resultados apenas vencido el plazo de la encuesta.

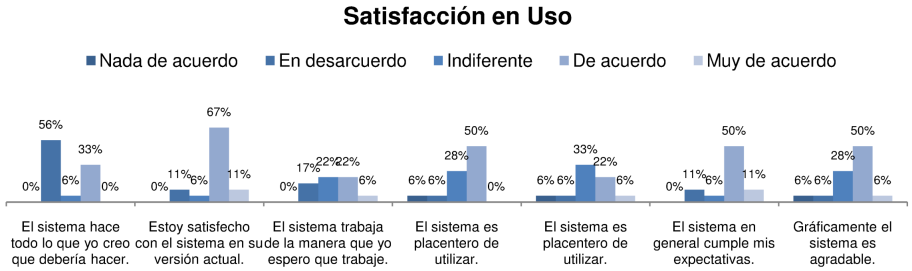


Figura 5.1: Ejemplo de resultados de una encuesta de satisfacción

Criterio de interpretación de resultados. El resultado será ambiguo o inconcluso si:

- el valor 3 acumula más del 30 % de los votos***, o***
- si la suma de votos de los valores 1 y 2 tienen una cantidad de votos comparable con la suma de 4 y 5***, o***
- si hay más de un 20 % de respuestas ***"NS/NC"*** y ***"No comprendí la pregunta"***.

Si un resultado no es ambiguo: es **negativo** si acumula más del 45 % de los votos en los puntajes 1 a 3. De forma análoga, el **positivo** debe acumular más del 55 % en los puntajes 4 y 5.

Observar que si una pregunta se hace por la negativa, la interpretación debe invertirse.

Tomando como ejemplo la imagen 5.1, los cuadros 1, 3 y 5 son "ambiguos". El 2, 4, 6 y 7 son positivos.

Guía de decisiones en función de los resultados. Para la fase alfa, se esperaba que los resultados sirvieran para guiar las preguntas de la segunda oleada y para enfocar los esfuerzos de *fix*/ajuste.

En particular, el resultado de la primera oleada afectaría de la siguiente forma:

- Resultados ambiguos (o inconclusos) para cierto punto X motivarán la creación de preguntas más específicas en relación con el punto X, intentando aclarar la ambigüedad.
- Resultados negativos para X motivarán la creación de preguntas más específicas en relación con el punto X, pero con un perfil que apunte a entender cuál sería un plan de acción adecuado. En caso de ser oportuno, motivarán modificaciones a la herramienta que serán puestas a prueba en la siguiente oleada.

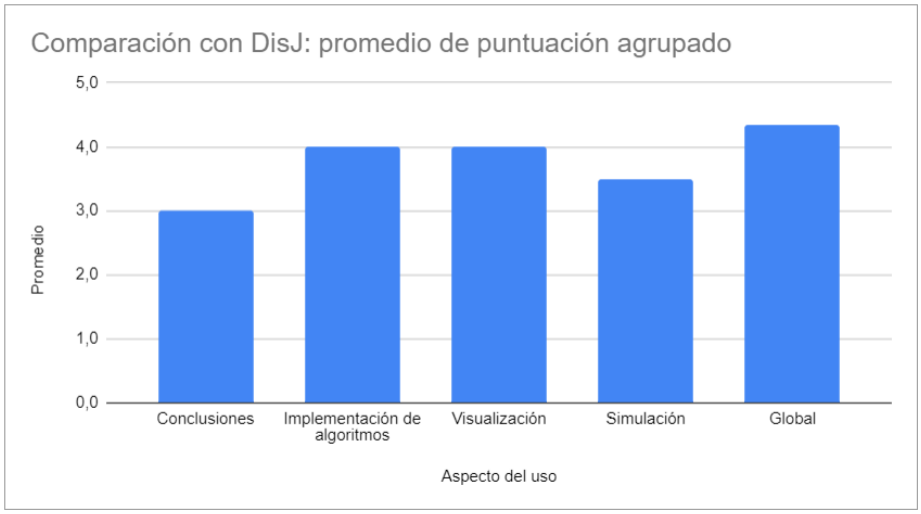


Figura 5.2: Puntajes promedio de los ítems de comparación con DisJ, agrupados por aspecto del uso.

Se hará seguimiento de estos efectos en la siguiente prueba para tener trazabilidad de los puntos inconclusos o la mejoría de los puntos negativos.

Esta metodología será replicada en la siguiente fase de pruebas, ver 5.2.

Cuestionario: Se incluye en el Anexo A el cuestionario preparado. Para facilitar la experiencia se usó Google Forms (Vasantha Raju y Harinarayana, 2016). El mismo se validó con los supervisores y con Sebastián Pizard. En la Sección A.2.2 se agregan referencias a qué característica de la usabilidad se está evaluando con cada pregunta.

5.1.2. Resultados

Al contrario de lo esperado y como ya fue adelantado, la primera fase solamente tuvo un *tester*. Igualmente, se intentó mantener la esencia del análisis propuesto, con la particularidad de que no podremos enfocarnos en cada pregunta de forma individual, pues se violaría el acuerdo de participación. Se podrá hacer este análisis en la siguiente fase, ya que se contó con más encuestados.

La primera observación es que no hubo **ningún resultado negativo**. Este resultado en sí es muy positivo, ya que marca que la herramienta no es peor que DisJ (en ningún aspecto evaluado).

Resultados neutros/inconclusos:

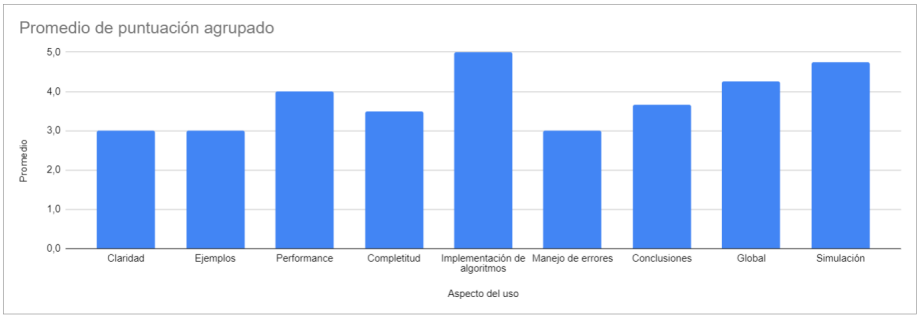


Figura 5.3: Puntajes promedio de los ítems sobre PyDistSim, agrupados por aspecto del uso.

En la comparación con DisJ:

- Los mecanismos para la extracción de conclusiones.
- Los mecanismos para observar la simulación a medida que ejecuta.

Por lo que no somos capaces de concluir si PyDistSim es mejor o peor que DisJ en estos aspectos.

En las preguntas globales:

- Todos los puntos sobre documentación.
- Facilidad de uso en el módulo de *benchmarks*.
- Manejo de errores y mensajes de error.

Como fue adelantado en el plan, estas cuestiones serán abordadas en la siguiente fase de pruebas.

El resto fueron resultados positivos. Los más positivos fueron:

En la comparación con DisJ:

- Experiencia general.

En las preguntas globales:

- Usabilidad en la implementación de algoritmos.
- Usabilidad de la generación de redes para pruebas.
- Correctitud de la simulación.

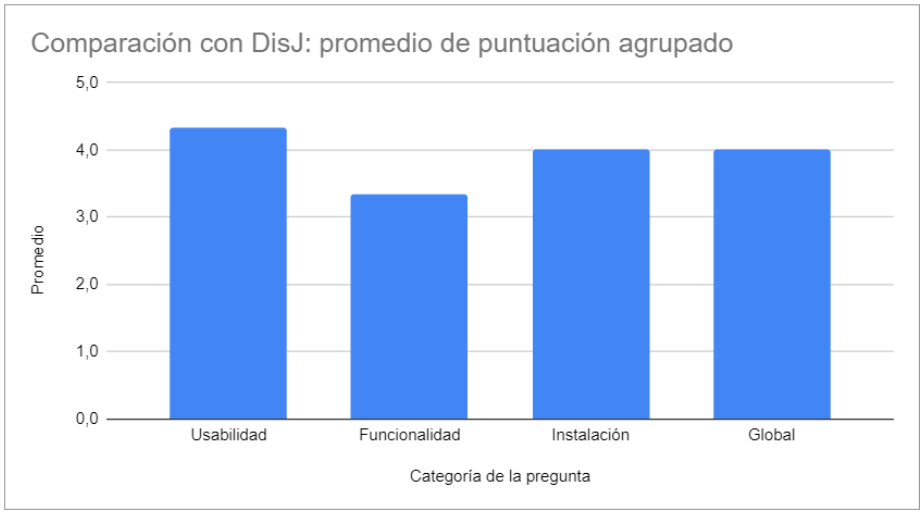


Figura 5.4: Puntajes promedio de los ítems de comparación con DisJ, agrupados por atributo de calidad.

En cuanto a las respuestas abiertas, se destacó la usabilidad de la herramienta, así como la facilidad al resolver los ejercicios del curso.

Además, se comunicaron algunos problemas en la comprensión de ciertos aspectos secundarios de la simulación, como por ejemplo el acceso a información externa a la entidad simulada. Todos los puntos mencionados fueron usados para poner énfasis tanto en la documentación de referencia de cada módulo como en los tutoriales guiados y en la inclusión de mensajes de *warning* al consultar información intencionalmente ofuscada.

Se generaron gráficas para observar los resultados agrupados por aspecto o categoría. Ver en las figuras 5.2, 5.3, 5.4 y 5.5.

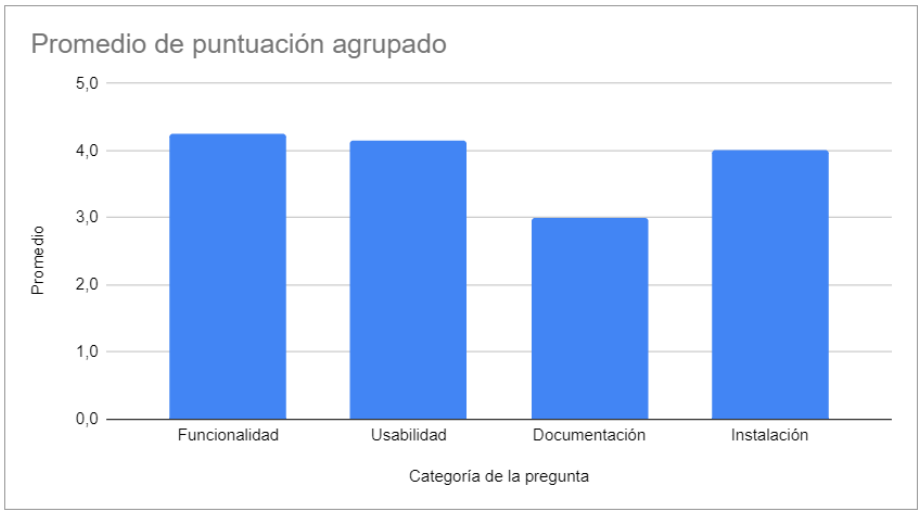


Figura 5.5: Puntajes promedio de los ítems sobre PyDistSim, agrupados por atributo de calidad.

5.2. Beta: uso en ADADR

Dado el resultado positivo de las pruebas Alfa, se tomó la decisión en conjunto de los docentes de ADADR de avanzar con el plan de usar PyDistSim en la edición 2024 del curso.

Esto en sí es una decisión que no estaba prevista en el plan original del proyecto, por lo que requiere una reestructuración parcial del esfuerzo del trabajo durante las semanas en las que se desarrollará el curso. Además, el uso de una herramienta experimental en un curso de grado implica una serie de consideraciones éticas que desarrollaremos a continuación.

5.2.1. Consideraciones éticas

Como mencionamos, el uso de una herramienta experimental en un curso de grado implica tener en cuenta algunos aspectos éticos.

Primero, se debe priorizar en todo momento el aprendizaje de los estudiantes y no perjudicarlo con la utilización de una herramienta que no esté a la altura de las circunstancias. Esto creemos que ya fue abordado en la fase Alfa, pero se debe tener en cuenta que la herramienta no está completamente validada y que puede haber problemas no detectados.

Como consecuencia, se debería ofrecer un servicio de soporte a los estudiantes que utilicen la herramienta. Esto implica tener tiempos reservados para atender consultas y problemas que puedan surgir durante el uso de la herramienta. Esto incluiría la resolución de problemas técnicos, arreglo de *bugs*, dudas sobre el uso

de la herramienta y cualquier otro tipo de consulta que pueda surgir. Decidimos que para tener suficiente disponibilidad, se reservarían 10 hs semanales a este fin, con el establecimiento de un plazo de 48 hs para arreglar posibles *bugs* y de 24 hs para responder emails, mensajes en el foro del curso o consultas en el GitHub del proyecto.

Por último, se debería confirmar el consentimiento de los estudiantes en toda recopilación de datos. Esto implica que se debería informar a los estudiantes sobre qué datos se van a recopilar y con qué fin se van a utilizar. Obviamente, se debería obtener el consentimiento explícito de los estudiantes para recopilar y utilizar sus datos y se les garantizará que los mismos no serán usados por los docentes para la evaluación en el curso.

5.2.2. Objetivo

Los resultados de la segunda oleada se usarán para obtener una respuesta acerca de si el software cumple con los objetivos. Para ello, agruparemos los ítems preguntados por disciplina o tipo de requisito y contaremos la cantidad de positivos, negativos o ambiguos para cada agrupación. Se dará como aprobada una agrupación si la mayoría de los ítems de la misma son positivos.

5.2.3. Plan

Se esperaba que la cantidad de *testers* sea mayor que en la primera oleada. Específicamente, a la fecha de entrega de este trabajo, 9 alumnos siguen participando de forma activa en la entrega de ejercicios de ADADR y, por lo tanto, fueron *testers* de la herramienta.

Los problemas a resolver fueron *"Ranking"* (mismo que en la fase alfa, ver 5.1.1), *"Stages"* y *"UniStages"*. A continuación, se adjunta la letra de estos problemas (adaptado de DADA).

Letra del problema 3.10.29: Muestre una ejecución paso a paso de los algoritmos *Stages* (especificado en la figura 3.14, página 124) y *UniStages* (especificado en la figura 3.23, página 137) en un anillo de 14 nodos. Indique, para cada paso, los valores conocidos para los candidatos.

Nota para el curso: Implemente en el simulador PyDistSim.

Como se puede ver, el problema pide implementar y mostrar los valores internos de los nodos a cada paso de una corrida. Se espera que los estudiantes puedan utilizar el módulo de animaciones para lograr lo primero y alguna de las estrategias de *logging* o monitoreo implementadas en PyDistSim para lo segundo.

Se grabó la presentación del software por parte del autor. Esto es para que el *tester* reciba el software y la documentación de igual forma que la recibirán

los estudiantes de las próximas iteraciones del curso, pues la grabación quedó disponible en EVA.

Cuestionario: Se usará una versión adaptada del cuestionario de la fase alfa, apoyándonos en el resultado obtenido para la generación de nuevas preguntas. En el apartado A.3 del anexo se explicita cuáles son estas nuevas preguntas.

***Focus Group*:** Se llevará a cabo una reunión de 40 minutos al final de una clase virtual de ADADR. Se invitará a aquellos alumnos que quieran participar de forma totalmente voluntaria. La esencia de la reunión será de "charla" guiada por un moderador (el autor del trabajo). Se espera obtener un *feedback* que muestre los puntos a mejorar del software, asuntos que hayan hecho "ruido" durante el uso o cuestiones en general que no sería fácil capturar con un cuestionario de múltiple opción como el que se hizo.

5.2.4. Resultados

Para repasar lo que se obtuvo de esta fase de pruebas, separaremos el análisis de los puntajes numéricos de las preguntas abiertas y lo recopilado durante la reunión de *Focus Group*.

Resultados cuantitativos

Interpretación de resultados. Se seguirá lo definido en la Sección 5.1.1 y, dado que esta es la evaluación final, se usarán estas conclusiones como base para la redacción de las conclusiones del trabajo.

A la fecha de realización del análisis de resultados, se tienen tres respuestas al cuestionario solamente. Basándose en esto último, también se muestra en las gráficas la respuesta del *tester* alfa (para tener mayor cantidad de datos). Asimismo, en el caso de las preguntas que no se hicieron en la fase alfa, se subirá el porcentaje mínimo para considerar un resultado ambiguo a un 34 % (en lugar del 30 % de la fase alfa), para evitar que un solo voto cambie la interpretación de los resultados.

Comenzando con la pregunta de valoración global, se observa en la Figura 5.6 que todas las respuestas fueron positivas y la mayoría ampliamente positiva. Se puede concluir con confianza que a los estudiantes les gustó la herramienta y se da el ítem como aprobado.

Pasando a los ítems de usabilidad, se observa en la Figura 5.7 que la facilidad de uso fue bien valorada (y aprobada) en los aspectos: simulación *per se*, creación de animaciones y creación de redes. El punto restante, sobre el módulo de *benchmarks*, tuvo un 50 % de respuestas neutras, por lo que el resultado se considera ambiguo. Se destaca que para esta pregunta únicamente hubo dos respuestas numéricas (el resto fue *"NS/NC"*) y tiene sentido dado que ninguna de las consignas para las pruebas incluía análisis de performance de los algoritmos.

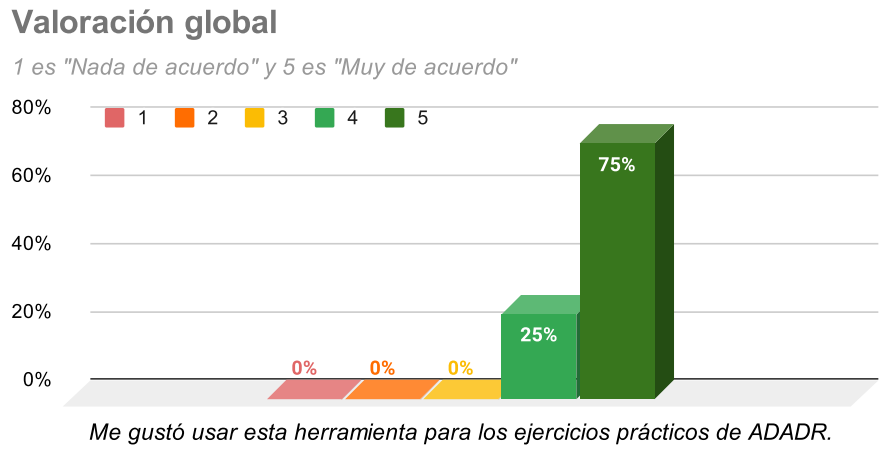


Figura 5.6: Gráfica de las respuestas a la pregunta de valoración global.

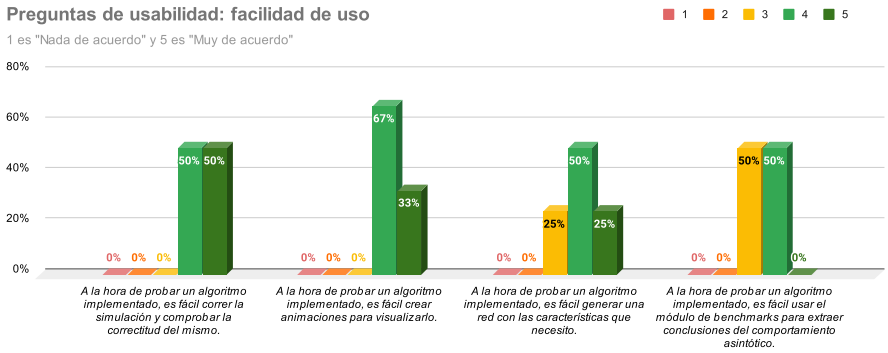


Figura 5.7: Gráfica de las respuestas a los ítems de usabilidad, centrados en la facilidad de uso.

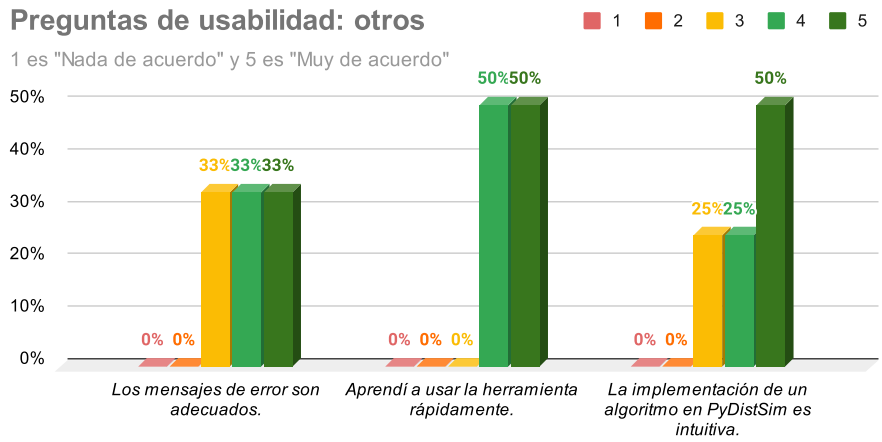


Figura 5.8: Gráfica de las otras respuestas a las preguntas de usabilidad.

Siguiendo con usabilidad, en la Figura 5.8 se ven las preguntas relacionadas con mensajes de error, intuitividad y facilidad de aprendizaje. Los tres ítems están dentro de los límites para marcarse como aprobados. Comparado con los resultados de la fase alfa (ver en la Sección 5.1.2), se logra apreciar una mejoría en los mensajes de error, punto que antes quedaba inconcluso.

La sección de documentación fue la que peor resultado obtuvo, como se puede ver en la Figura 5.9. Los únicos puntos que tuvo como aprobados fueron la facilidad de entenderla y que los ejemplos son de ayuda para la comprensión de la herramienta. Luego, la pregunta sobre la completitud tuvo resultados ambiguos y, por último, la pregunta sobre la completitud de los escenarios cubiertos por los ejemplos fue negativa, con 2 votos negativos. Estos puntos ambiguos y negativos pueden explicarse por los comentarios recibidos durante las etapas de *feedback* abierto, comentados en la Sección 5.2.4. No se alcanza el criterio de aprobación para este apartado.

Es importante también mencionar que con este *feedback* se logra aclarar el resultado inconcluso de la fase alfa, comentado en la Sección 5.1.2.

Para las preguntas de configuración e instalación, se observa en la Figura 5.10 que todas las preguntas se pueden clasificar como aprobadas. Se destaca el buen desempeño en el ítem de instalación, uno de los requisitos no funcionales claves del proyecto.

En cuanto a las funcionalidades, se observa en la Figura 5.11 que las preguntas centradas en la obtención de resultados fueron todas aprobadas, todas con respuestas puramente positivas. En la Figura 5.12 se ven las preguntas sobre correctitud, completitud y performance[2]. Excepto la pregunta sobre errores

[2]Hay una negación en las primeras dos preguntas, por lo que el análisis se invierte en esos

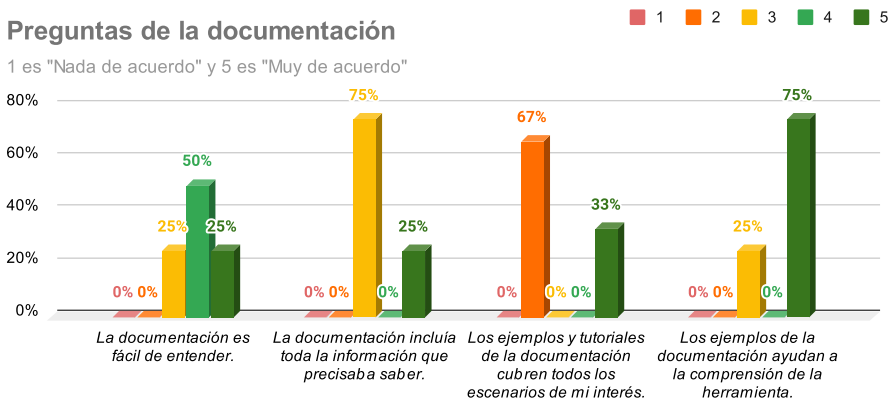


Figura 5.9: Gráfica de las respuestas a las preguntas de documentación.

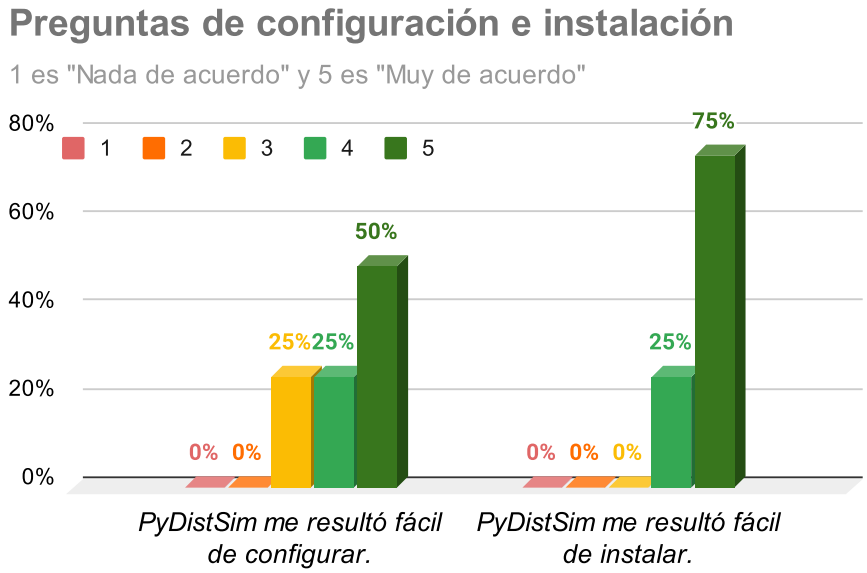


Figura 5.10: Gráfica de las respuestas a las preguntas sobre la instalación.

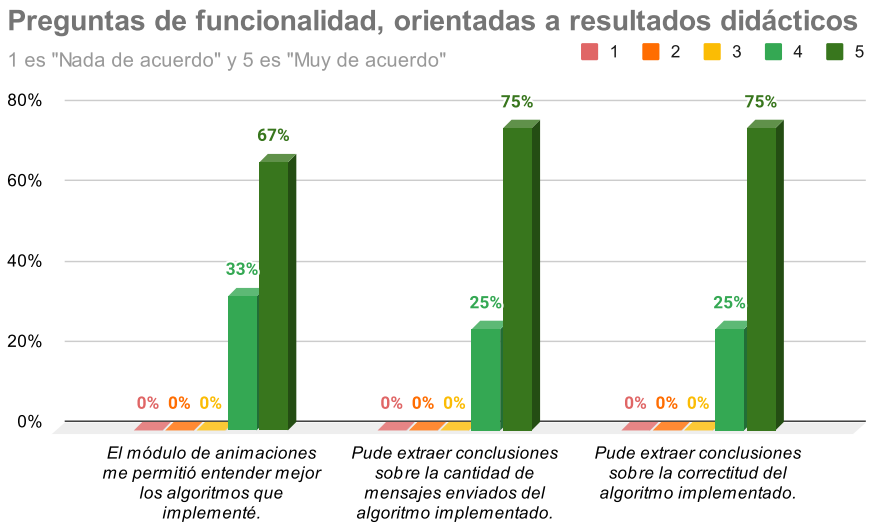


Figura 5.11: Gráfica de las respuestas a los puntos sobre las funcionalidades, centrados en obtención de resultados.

inesperados, todas fueron aprobadas con respuestas igual de positivas. Creemos que el resultado negativo en la pregunta de errores puede deberse a un *bug* detectado durante las pruebas, relacionado con la generación de animaciones sobre anillos dirigidos[3].

Por último, cabe mencionar que los puntos *"El módulo de benchmarks funcionó según lo que esperaba."* y *"El módulo de benchmarks me permitió entender mejor los algoritmos que implementé."* tuvieron únicamente respuestas *"NS/NC"*. Se le atribuye a esto lo mencionado antes, que los estudiantes no probaron la funcionalidad porque no estaba dentro de la consigna de los ejercicios. De esta forma, quedan también estos puntos como ambiguos y, por lo tanto, inconclusos.

casos.

[3]El *bug* se corrigió en menos de 24 hs desde que fue reportado, por lo que se puede asegurar que el impacto en el aprendizaje fue mínimo.

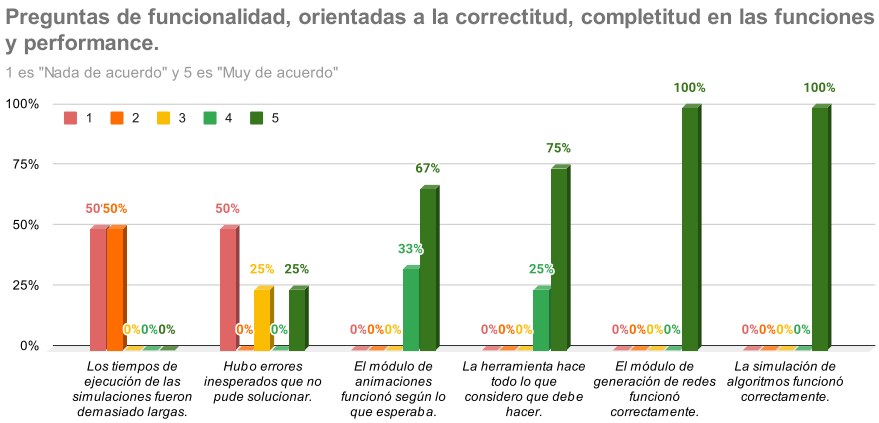


Figura 5.12: Gráfica de las respuestas a las preguntas sobre las funcionalidades, correctitud, completitud y performance.

Resultados cualitativos

Pasamos ahora a mencionar brevemente lo recopilado en las preguntas abiertas y en el *Focus Group*.

Comenzando con los aspectos positivos, se mencionó la usabilidad y la intuitividad de la herramienta, así como la completitud en las funcionalidades disponibles, que resultaron muy útiles para la resolución de los ejercicios del curso.

En cuanto a aspectos negativos, surgió que la navegación de la documentación no facilitaba encontrar cierta información, que sí existía. También en lo que se refiere a la documentación, se mencionó que sería de utilidad agregar ejemplos sobre la asignación de identificadores únicos aleatorios en tiempo de inicialización y más ejemplos de generación de redes dirigidas.

En cuanto a las funcionalidades, se dieron varias sugerencias para mejorar[4]:

1. En las animaciones, sería de utilidad poder ver más información de la memoria interna de cada nodo, para entender qué pasa a medida que avanza la simulación y por qué pasa cada cosa.

2. Cambiar el nombre de la clase que representa las redes direccionadas (`Network`), ya que es muy ambiguo en comparación con el de las no direccionadas (`BidirectionalNetwork`). [5]

3. Simplificar la generación de redes, hay partes un poco confusas y que costó

[4]Se abordaron los puntos 2, 3 y 4 en la última versión de PyDistSim.
[5]Se cambió el nombre de la clase a `DirectedNetwork`.

encontrar cómo hacer que la generación del anillo sea direccionado. [6]

4. Incluir la documentación de las clases y funciones directamente en el código, ya que muchos usan la función del IDE "ctrl + click" para ir a la implementación de cada función directamente. [7]

5. En algunos casos, cuando un algoritmo hacía uso de `print()` y el mismo se animaba, se generaban *prints* duplicados. Parece un *bug* menor a corregir.

Por último, comentar que en el *Focus Group*, al final de la reunión, nadie consideró que haya funcionalidades faltantes que fueran a necesitar para las implementaciones de algoritmos más complejos.

5.3. Ejemplificación con un algoritmo complejo: Mega-Merger

De forma de dar una validación interna a PyDistSim, se decidió experimentar con la implementación de un algoritmo más complejo que los que se suelen pedir en ADADR. Para ello, se eligió el algoritmo Mega-Merger (Gallager, Humblet, y Spira, 1983), un algoritmo que se da a alto nivel en el curso, pero que no se pide implementar ni se mencionan los detalles de implementación.

De esta forma aprovecharemos para usar esta experiencia como una manera de tender un puente entre la teoría y la práctica, y de paso, explorar los límites de PyDistSim. Adicionalmente, se espera que la implementación también sirva de referencia didáctica para la enseñanza de este algoritmo.

5.3.1. El algoritmo

Mega-Merger es un algoritmo que soluciona el problema de elección distribuido en topologías arbitrarias, sin conocimiento previo de la red por parte de los nodos. El problema de elección consiste en que cada nodo de la red debe elegir un líder, de forma que todos los nodos elijan al mismo líder. La elección de este líder es clave para la ejecución de otros algoritmos distribuidos, ya que permite que todos los nodos de la red tengan un punto de referencia común, capaz de dirigir ejecuciones que de otro modo serían imposibles o muy costosas de coordinar.

El algoritmo Mega-Merger soluciona este problema de forma eficiente, a la vez que genera un árbol recubridor de toda la red, con raíz en el líder elegido. La idea de la ejecución del algoritmo es que cada nodo comienza conceptualizado como una ciudad y, a lo largo de la ejecución, se van fusionando ciudades hasta que todos los nodos pertenecen a una misma metrópolis.

[6] Se agregó un ejemplo de generación de un anillo dirigido en la documentación: https://pydistsim.readthedocs.io/notebooks/network_generators.html

[7] Algunas funcionalidades ya contaban con esto, se llevó a otras más que lo necesitaban, pero no se cubrió el 100 % de las clases y funciones.

La base del algoritmo es en que cada fusión se puede realizar de forma determinista si elegimos siempre la ciudad más cercana para hacer la fusión. Observar que la definición de cercanía se puede tomar al asignarle pesos únicos a cada enlace, lo cual no es una restricción en sí, ya que estos pesos pueden asignarse con base en los valores de los nodos, que son únicos.

Dadas estas condiciones, la vida de una metrópolis (inicialmente con una sola ciudad) va transcurriendo de a fases:

1. Determinar qué carreteras de la metrópolis llevan a una metrópolis distinta o a ella misma (carreteras externas o internas, respectivamente).

2. Determinar la ciudad más cercana, tomando la mínima distancia de las carreteras externas.

3. Fusionar la metrópolis con la ciudad más cercana.

4. Repetir hasta que no queden carreteras externas.

La clave para realizar todas estas fases de forma eficiente en cuanto a cantidad de mensajes es que cada metrópolis siempre cuente con una capital: un nodo que es el representante de la metrópolis y que es el encargado de tomar las decisiones de fusión. La capital de una metrópolis cambia solamente cuando se fusiona con otra metrópolis y se decide cuál de las dos capitales anteriores es la nueva capital. Además, cada metrópolis va llevando la cuenta de cuántas fusiones ha hecho hasta el momento, de forma de que se pueda usar este número como una especie de "jerarquía" que permita hacer fusiones rápidas, sin "mediar" entre las capitales.

5.3.2. Artefactos generados

Además de la implementación de Mega-Merger[8], se generaron tres archivos Jupyter Notebook que ejemplifican cómo se pueden usar los diversos módulos de PyDistSim para experimentar con un algoritmo distribuido. Estos archivos configuran la sección de ejemplos de uso avanzados de la documentación y se espera que sean de utilidad para demostrar cómo aprovechar al máximo las funcionalidades de PyDistSim.

Pruebas de la implementación

Se mostró cómo se puede usar la generación de muchísimas redes aleatorias y no aleatorias para asegurar que un algoritmo cumple con su cometido correctamente, independientemente de la topología de la red[9].

[8] https://pydistsim.readthedocs.io/_modules/pydistsim/demo_algorithms/santoro2007/mega_merger/algorithm.html#MegaMergerAlgorithm

[9] https://pydistsim.readthedocs.io/notebooks/mega_merger/tests.html

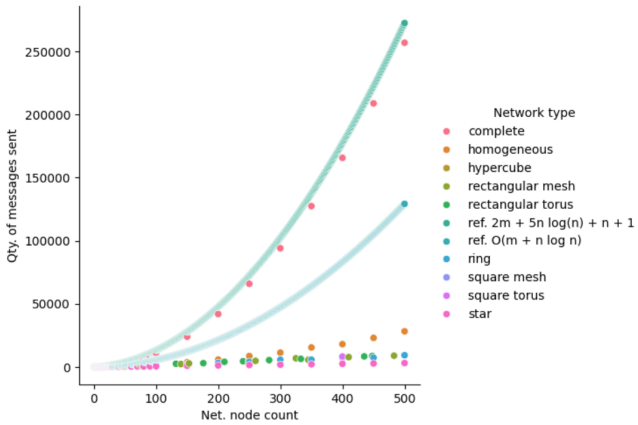


Figura 5.13: Gráfica de la cantidad de mensajes enviados por Mega-Merger en función de la cantidad de nodos de la red. Se usan colores para segmentar el tipo de red de cada punto y se agregan las curvas teóricas como referencia.

Generación de animaciones

Se mostró cómo se puede usar el módulo de animación para visualizar la simulación[10]. También se ejemplifica cómo se pueden utilizar los distintos parámetros de configuración para aprovechar nuestro conocimiento del algoritmo y de esta forma generar animaciones mucho más informativas e interesantes.

Alguna de las cosas que se ejemplifican son: cómo mostrar el árbol recubridor a medida que se construye, cómo usar el color de los nodos para simbolizar las metrópolis y así ver cómo se fusionan, cómo usar el tamaño de los nodos para diferenciar a las capitales, cómo usar la etiqueta de los nodos para mostrar alguna variable interna de los mismos, entre otros.

Benchmarking y análisis de performance asintótico

Finalmente, se usa el módulo de *benchmarks* para medir el rendimiento de un algoritmo y relacionar el mismo con las cotas teóricas de complejidad[11]. Se ejemplifica cómo se puede usar el módulo de *benchmarks* para medir la cantidad de mensajes enviados y la cantidad de tiempo que tarda en ejecutarse el algoritmo en función de la cantidad de nodos y aristas de la red.

Permitió corroborar la cota de $\mathcal{O}(2m + 5n \times log(n) + n + 1)$ (Gallager y cols., 1983) en la cantidad de mensajes enviados por el algoritmo, donde n es la cantidad de nodos y m la cantidad de aristas de la red. Se incluye en la Figura 5.13 la gráfica generada.

A la vez, se logró establecer una cota a la cantidad de pasos que tarda el algoritmo en ejecutarse, que no estaba estudiada en el libro de Santoro. Se

[10] https://pydistsim.readthedocs.io/notebooks/mega_merger/animations.html
[11] https://pydistsim.readthedocs.io/notebooks/mega_merger/benchmarking.html

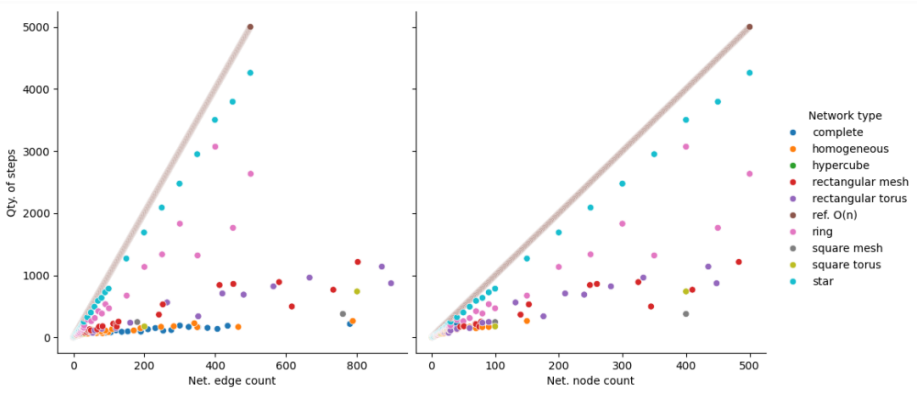


Figura 5.14: Gráfica de la cantidad de pasos que tarda Mega-Merger en ejecutarse en función de la cantidad de nodos y aristas de la red. Se usan colores para segmentar el tipo de red de cada punto y se agrega la recta inferida durante la experimentación.

encontró que el algoritmo tarda $\mathcal{O}(n)$ pasos en ejecutarse. Se incluye en la Figura 5.14 la gráfica generada.

5.3.3. Aprendizajes

Se aprendió que la implementación de un algoritmo complejo como Mega-Merger es posible en PyDistSim y que la herramienta es lo suficientemente flexible como para permitir la implementación de estos algoritmos sin necesidad de modificar la herramienta en sí.

Sin embargo, hubo algunos aspectos que se detectaron que podrían mejorar en pos de una mayor usabilidad y eficiencia en la implementación de algoritmos complejos. Algunos de estos aspectos son:

- Al implementar un algoritmo mediano o grande, resultaba engorroso que fuera necesario usar `node.memory[VARIABLE]` para acceder a cada variable. Frente a esto se implementó una subclase de `NodeAccess` llamada `DMANodeAccess`[12] que permite acceder a las variables de la memoria de un nodo como si fueran atributos de la clase. Esto facilita la escritura del código y lo hace más legible, sin perder la característica de que está explícito que se está accediendo a la memoria de un nodo.

- El módulo de visualizaciones tenía un *bug* que, según qué configuraciones se hacían, a veces el cuadro de leyenda se sobreponía a la red. Se corrigió este *bug* y se mejoró la visualización de la leyenda, permitiendo dar

[12]DMA, sigla de Direct Memory Access, quizá confundible con el conocido modo de IO de una CPU, pero que en este contexto describe exactamente el mecanismo que introduce.

espacio a la misma según la necesidad del usuario. También se agregaron configuraciones adicionales, como poder modificar el tamaño de los nodos de forma individual.

- En el módulo de *logging*, se modificó el *handler*[13] por defecto para que se puedan modificar algunas configuraciones pero dejar el resto inalterado. Esto se usó para redirigir de `stdout` a un archivo de *log*, pero manteniendo la configuración de formato y nivel.

[13] Objeto que representa la configuración actual de *logging*.

Capítulo 6

Conclusiones y trabajo futuro

Este proyecto tuvo como objetivo principal el desarrollo de una herramienta de simulación de algoritmos distribuidos, PyDistSim, que permitiera a los estudiantes de ADADR experimentar con los algoritmos vistos en clase. Para ello, se planteó un plan de trabajo que incluyó la implementación de una serie de funcionalidades que permitieran la simulación de algoritmos distribuidos en redes arbitrarias, la generación de animaciones para visualizar la ejecución de los algoritmos y la creación de una interfaz gráfica para facilitar la interacción con la herramienta.

El desarrollo se basó en extender el trabajo previamente hecho en Pymote, una herramienta que contaba con varias características similares a lo que se necesitaba. Se aprovechó la estructura de Pymote para implementar las funcionalidades de PyDistSim, lo que permitió acelerar el desarrollo y centrarse en la implementación de las nuevas funcionalidades.

6.1. Pruebas y uso en ADADR

Se realizaron tres instancias de evaluación sobre la herramienta. En la primera fase, se logró contar con un estudiante de ediciones anteriores de ADADR para que pruebe y compare PyDistSim con DisJ. Gracias a que se obtuvieron resultados positivos en cuanto a la usabilidad y funcionalidad de la herramienta, se permitió avanzar con el plan de usar PyDistSim en la edición 2024 de ADADR. En esta segunda fase de pruebas, se obtuvieron resultados también muy positivos, con aspectos a mejorar principalmente en la documentación. Finalmente, se implementó un algoritmo complejo, Mega-Merger, para validar la capacidad de la herramienta y se aprovechó para generar ejemplos de uso avanzados para la documentación.

Las tres instancias de evaluación nos brindaron valioso *feedback* que permitió mejorar la herramienta y establecer una base sólida para futuras versiones.

Principalmente, PyDistSim fue utilizado de forma plena en la edición 2024 de ADADR para la resolución de ejercicios y la implementación de algoritmos distribuidos. Los estudiantes pudieron experimentar con las funcionalidades de la herramienta y, lo que es más importante, se logró un hacerlo sin comprometer la calidad de la enseñanza.

Gracias al *feedback* recopilado tanto con los cuestionarios como por reuniones con los alumnos, se pudo comprobar el alto nivel de calidad de PyDistSim, en comparación con las expectativas de los alumnos.

Aunque esta valoración no parezca suficientemente objetiva, recordamos que el uso de la herramienta anterior, DisJ, daba problemas a lo largo de todo su uso en el curso, lo que generaba una percepción negativa de la misma en diversos aspectos (principalmente en instalación, usabilidad y funcionalidades).

Si bien la comparación directa es casi imposible, ya que no se puede recrear el primer uso de una herramienta a la vez que se cursa por primera vez, sí se pueden comparar las reacciones de los estudiantes frente a las mismas: los comentarios sobre los puntos mejorables de PyDistSim son mucho más débiles que las críticas que se hacían a DisJ por sus diversos problemas.

Combinando estas observaciones con los resultados de la primera fase de pruebas, donde el *tester* sí conocía DisJ, se puede concluir que PyDistSim cumple con los objetivos sobre la mejoría con respecto a su antecesor.

6.2. Continuación del trabajo

Por motivos de tiempo y recursos, no se pudieron implementar todas las funcionalidades que se habían planteado en el plan de trabajo original, en pos de realizar una fase de pruebas completa y exhaustiva, aspecto priorizado de esta forma en conjunto con los supervisores.

Las funcionalidades no implementadas son relacionadas principalmente con el desarrollo de una interfaz gráfica independiente y la creación de redes arbitrarias mediante interfaz drag-and-drop. Estas funcionalidades se consideran como trabajo futuro y se espera que puedan ser implementadas en futuras versiones de la herramienta. Para ver la descripción completa de las funcionalidades no implementadas, ver el apartado 6.2.1.

Por otro lado, las instancias de *feedback* permitieron identificar una serie de mejoras que se pueden realizar en la herramienta para aumentar su usabilidad y funcionalidad. Para ver la descripción completa de las mejoras sugeridas, ver el apartado 5.2.4.

Por último, mencionaremos que creemos que la posibilidad de demostrar el funcionamiento de un algoritmo de forma visual, con animaciones, tiene mucho potencial didáctico, prácticamente desaprovechado por ADADR. Siguiendo este pensamiento, se propone comenzar una librería de implementaciones que complemente a los algoritmos ya implementados en PyDistSim, aprovechando este punto fuerte del *framework*.

6.2.1. Funcionalidades no implementadas

Detallamos a continuación las funcionalidades que formaban parte del *backlog* inicial, pero que no pudieron ser implementadas en la versión final de PyDistSim.

1. **Integración de funcionalidades gráficas en interfaz independiente.** Todas las funcionalidades gráficas mencionadas en el apartado 4.1.4 estarán disponibles en una interfaz gráfica independiente, que permitirá la interacción con la simulación en tiempo real. Además de la visualización de la red y la animación de la simulación, se podrá pausar, continuar y retroceder la simulación, así como inspeccionar el estado de los nodos y las colas de mensajes.

2. **Creación de redes arbitrarias mediante interfaz gráfica drag-and-drop.** En su módulo gráfico, la herramienta implementaría el mecanismo de creación de redes a través de "dibujado" de bolas y palos (o flechas en el caso dirigido) para su interpretación como grafo y posterior utilización en las simulaciones.

3. **Uso de semillas para reproducibilidad de corridas.** La herramienta permitiría controlar la aleatoriedad del modelo a través de semillas que volverían reproducibles los resultados obtenidos en una ejecución.

4. **Falla en nodos y enlaces.** Se permitiría el modelado de las circunstancias de fallo a nivel entidad o enlace.

5. **Restricciones.** No se implementaron las restricciones relacionadas con fallas de nodos y enlaces. Tampoco se implementaron aquellas restricciones asociadas al hipercubo o el anillo orientado. Estas son:

   - *"Edge Failure Detection:"* Para todos los enlaces vecinos x e y, ambos nodos x e y detectan la falla del enlace entre ellos y su posterior recuperación, si la hubiera.
   - *"Entity Failure Detection:"* Para todo nodo x, cualquier vecino de entrada o de salida detecta la falla de x y su posterior recuperación, si la hubiera.
   - *"Total y Partial Reliability:"* No ocurrirá ningún fallo (puede haber ocurrido ya).
   - *"Hyper Cube Graph:"* La red es un hipercubo.
   - *"Oriented Hyper Cube Graph:"* La red es un hipercubo y sus nodos tienen orientación de qué dimensión corresponde cada uno de sus vecinos.
   - *"Oriented Cycle Graph:"* La red es un anillo y sus nodos tienen orientación de un sentido de recorrido común (horario o antihorario).

6.3. Conclusión

Todo lo expuesto previamente en este capítulo nos permite dar como cumplidos los objetivos específicos planteados en la 1.2.2 y, por ende, el objetivo general del proyecto.

Además, gracias a lo expuesto en la 6.2, contamos con una hoja de ruta clara para futuras versiones de PyDistSim. Se espera que PyDistSim pueda seguir evolucionando y adaptándose a las necesidades de los estudiantes y docentes, de ADADR y que a su vez pueda ser utilizada en otros cursos de algoritmos distribuidos, dentro y fuera de la Facultad de Ingeniería.

Referencias

Análisis y diseño de algoritmos distribuidos en redes. (s.f.). *EVA Facultad de Ingeniería*. Descargado de https://eva.fing.edu.uy/course/view.php?id=145 (Accedido: 2024-11-03)

Arbula, D., y Lenac, K. (2013, 01). Pymote: High level python library for event-based simulation and evaluation of distributed algorithms. *International Journal of Distributed Sensor Networks*, *2013*, 1-12. doi: 10.1155/2013/797354

Bressert, E. (2012). Scipy and numpy: an overview for developers.

Casteigts, A. (2015, agosto). JBotSim: a Tool for Fast Prototyping of Distributed Algorithms in Dynamic Networks. En *Eighth EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS)*. Athènes, Greece: ACM. Descargado de https://hal.science/hal-01472926

Delgan. (s.f.). *Loguru: Python logging made (stupidly) simple.* Descargado de https://loguru.readthedocs.io/en/stable/index.html (Accedido: 2024-11-03)

EDC-Group, y cols. (2008). *Sinalgo-simulator for network algorithms.* http://dcg.ethz.ch/projects/sinalgo/.

Eth zurich. (s.f.). *Web del grupo de investigación DISCO, de la universidad ETH Zurich*. Descargado de https://disco.ethz.ch/ (Accedido: 2024-11-03)

Gallager, R. G., Humblet, P. A., y Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, *5*(1), 66–77.

Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. *Person Education Inc*.

GRIS. (s.f.). Grupo de ingeniería de software de la facultad de ingeniería. *Instituto de Computación*. Descargado de https://www.fing.edu.uy/es/node/41099 (Accedido: 2024-11-03)

Hagberg, A., y Conway, D. (s.f.). Networkx: Network analysis with python. *Networkx Documentation*. Descargado de https://networkx.github.io (Accedido: 2024-11-03)

McGugan, W. (2007). *Beginning game development with python and pygame: from novice to professional.* Apress.

Molkentin, D. (2007). *The book of qt 4: The art of building qt applications.* No Starch Press.

Musayev, X. (2023). Convenience of pyqt and pyside modules. En *Conference on digital innovation:"modern problems and solutions"*.

Pérez, F., y Granger, B. E. (2007). Ipython: a system for interactive scientific computing. *Computing in science & engineering*, *9*(3), 21–29.

Piyasin, N. (2013). Disj - reactive distributed simulation in java. *Bibliotheque et Archives Canada, Ottawa*. doi: 10.22215/etd/2012-07408

Precommit: A framework for managing and maintaining multi-language pre-commit hooks. (s.f.). *PreCommit*. Descargado de https://pre-commit.com/ (Accedido: 2024-11-03)

Python 2.7 documentation. (s.f.). *Python 2.7.18 Documentation*. Descargado de https://docs.python.org/2.7/ (Accedido: 2024-11-03)

Python 3.13 documentation. (s.f.). Descargado de https://docs.python.org/3/ (Accedido: 2024-11-03)

Ralph, P. (2020). ACM SIGSOFT empirical standards. *CoRR*, *abs/2010.03525*. Descargado de https://arxiv.org/abs/2010.03525

Randles, B. M., Pasquetto, I. V., Golshan, M. S., y Borgman, C. L. (2017). Using the jupyter notebook as a tool for open science: An empirical study. En *2017 acm/ieee joint conference on digital libraries (jcdl)* (pp. 1–2).

Read the Docs, I. (s.f.). *Full featured documentation deployment platform.* Descargado de https://about.readthedocs.com/ (Accedido: 2024-11-03)

Read the docs sphinx theme. (s.f.). Descargado de https://sphinx-rtd-theme.readthedocs.io/en/stable/ (Accedido: 2024-11-03)

ISO 25000:2014. (2014, marzo). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE* (Vol. 2014; Standard). Geneva, CH: International Organization for Standardization.

TIOBE organization. (s.f.). *Tiobe index.* Descargado de https://www.tiobe.com/tiobe-index/ (Accedido: 2024-11-03)

Santoro, N. (2006). *Design and analysis of distributed algorithms (wiley series on parallel and distributed computing)*. USA: Wiley-Interscience.

Scherfke, S., Lnsdorf, O., Grayson, P., LaFevers, E., Pinckney, T., Klein, C., ... others (2020). Simpy. *Discrete Event Simulation for Python*. Descargado de https://simpy.readthedocs.io/en/latest/ (Accedido: 2024-11-03)

Sphinx. (s.f.). Descargado de https://www.sphinx-doc.org/en/master/ (Accedido: 2024-11-03)

Tosi, S. (2009). *Matplotlib for python developers*. Packt Publishing Ltd.

van der Ham, R. (2018). Salabim: open source discrete event simulation and animation in python. En *Proceedings of the 2018 winter simulation conference* (p. 4186). IEEE Press.

Van Der Walt, S., Colbert, S. C., y Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, *13*(2), 22–30.

Vasantha Raju, N., y Harinarayana, N. (2016). Online survey tools: A case

study of google forms. En *National conference on scientific, computational & information research trends in engineering, gsss-ietw, mysore.*

Waskom, M. L. (2021). Seaborn: statistical data visualization. *Journal of Open Source Software*, *6*(60), 3021.

Wozniski, M. (s.f.). *The endgame python memory profiler.* Descargado de https://bloomberg.github.io/memray/ (Accedido: 2024-11-03)

Anexo A

Encuesta para los *testers*

A.1. Introducción y acuerdo de participación

Estimado/a,

Mi nombre es Agustín Recoba, estudiante de Ing. en Computación y responsable del desarrollo de PyDistSim como parte de mi proyecto de grado.

Su participación en esta encuesta nos ayudará a mejorar la usabilidad y funcionalidad del simulador PyDistSim desarrollado en el contexto de un proyecto de grado. Sus respuestas serán tratadas con total confidencialidad, manejadas únicamente por mí, y utilizadas exclusivamente para este estudio. Los resultados no serán publicados de forma individual ni de ninguna forma que pueda ser asociada a usted directamente.

Su participación es voluntaria, y puede retirarse en cualquier momento.

Agradecemos que se responda con la mayor honestidad posible. Tanto las respuestas positivas como negativas serán valiosas para el desarrollo de este software.

El cuestionario consta de tres secciones (introducción, usabilidad, funcionalidades) y para los alumnos que conocen el simulador usado en años anteriores, una cuarta sección (comparación con DisJ). Al final, también se dejará un espacio para escribir *feedback* abierto.

☐ Acepto participar voluntariamente en este estudio.

He sido informado de que los objetivos de este estudio son estrictamente de investigación y tanto mis datos personales, así como información delicada, no serán publicados o difundidos asociados de ninguna forma a mi identidad.

A.2. Preguntas para Alfa *Tester*

Calificar los siguientes enunciados del 1 al 5, donde 1 es “Nada de acuerdo” y 5 es “Muy de acuerdo”. Además, puede elegir “NS/NC” y “No comprendí la pregunta”.

A.2.1. Preguntas introductorias

1. Tengo experiencia utilizando simuladores de sistemas distribuidos.
2. Tengo experiencia utilizando simuladores de eventos discretos.
3. He usado simuladores para poner en práctica el contenido teórico de un curso.
4. ADADR es el único curso no obligatorio que he cursado con relación a sistemas distribuidos.
5. Estoy familiarizado con los desafíos técnicos asociados a la implementación de un simulador de sistemas distribuidos.
6. Estoy familiarizado con los desafíos de usabilidad asociados a la implementación de un software de estudio para ingeniería/computación.

A.2.2. Sección usabilidad y documentación

Se agregan comentarios acerca de qué aspecto de la usabilidad se está midiendo.

1. PyDistSim me resultó fácil de instalar. *(eficiencia, satisfacción emocional)*
2. PyDistSim me resultó fácil de configurar. *(eficiencia, satisfacción emocional)*
3. La implementación de un algoritmo en PyDistSim es intuitiva. *(eficiencia cognitiva)*
4. La implementación de un algoritmo en PyDistSim es similar a la representación teórica del libro. *(efectividad, eficiencia cognitiva)*
5. A la hora de probar un algoritmo implementado, es fácil generar una red con las características que necesito. *(efectividad, eficiencia cognitiva)*
6. A la hora de probar un algoritmo implementado, es fácil correr la simulación y comprobar la correctitud del mismo. *(efectividad, eficiencia cognitiva)*
7. A la hora de probar un algoritmo implementado, es fácil usar el módulo de *benchmarks* para extraer conclusiones del comportamiento asintótico. *(efectividad, eficiencia cognitiva)*
8. Los mensajes de error son adecuados. *(efectividad, eficiencia, satisfacción cognitiva)*
9. Aprendí a usar la herramienta rápidamente. *(eficiencia cognitiva)*
10. Me gustaría haber usado esta herramienta para todos los ejercicios prácticos de ADADR. *(satisfacción emocional)*

11. La documentación es fácil de entender. *(efectividad (de la documentación), eficiencia cognitiva, satisfacción cognitiva)*

12. Los ejemplos de la documentación ayudan a la comprensión de la herramienta. *(efectividad (de la documentación), eficiencia cognitiva, satisfacción cognitiva)*

13. La documentación incluía toda la información que precisaba saber. *(efectividad (de la documentación), eficiencia cognitiva, satisfacción cognitiva)*

A.2.3. Sección funcionalidades y correctitud

1. La simulación de algoritmos funcionó correctamente.

2. El módulo de generación de redes funcionó correctamente.

3. Pude extraer conclusiones sobre la correctitud del algoritmo implementado.

4. Pude extraer conclusiones sobre la cantidad de mensajes enviados del algoritmo implementado.

5. Los tiempos de ejecución de las simulaciones fueron demasiado largos.

6. Hubo errores inesperados que no pude solucionar.

7. Considero que algún aspecto de PyDistSim debería funcionar de otra forma.

8. La herramienta hace todo lo que considero que debe hacer.

A.2.4. Sección comparación con DisJ

1. Recuerdo cómo se realizaba la instalación de DisJ y todas sus dependencias.

2. Recuerdo cómo se implementaban los algoritmos distribuidos en DisJ.

3. Recuerdo cómo se visualizaban las ejecuciones y cómo yo extraía conclusiones acerca de la misma.

Calificar los siguientes enunciados del 1 al 5, donde 1 es “Mucho mejor en DisJ” y 5 es “Mucho mejor en PyDistSim”. Además, puede elegir “NS/NC” y “No comprendí la pregunta”.

1. Experiencia general de uso.

2. Instalación y configuración de la herramienta y el ambiente.

3. La implementación de algoritmos distribuidos.

4. Las interfaces para manipular las ejecuciones, pruebas, etc.
5. Los mecanismos para extraer conclusiones sobre los algoritmos distribuidos.
6. Los mecanismos para entender cómo funciona mi algoritmo a medida que va ejecutando.
7. Visualización gráfica de la red y el algoritmo que corre sobre ella.

A.2.5. Sección libre

Puede usar las cajas de texto de abajo para dejar cualquier comentario que desee acerca de PyDistSim.

- Usabilidad
- Funcionalidades
- Documentación

A.3. Preguntas para Beta *Testers*

A.3.1. Sección usabilidad y documentación

Siguiendo la misma consigna que en A.2.2, se agrega:

- A la hora de probar un algoritmo implementado, es fácil crear animaciones para visualizarlo.

Y se modifica:

- De "Me gustaría haber usado esta herramienta para todos los ejercicios prácticos de ADADR."
- Para "Me gustó usar esta herramienta para los ejercicios prácticos de ADADR."

A.3.2. Sección funcionalidades y correctitud

También, siguiendo la misma consigna, se agrega:

- Los ejemplos y tutoriales de la documentación cubren todos los escenarios de mi interés.
- El módulo de *benchmarks* funcionó según lo que esperaba.
- El módulo de *benchmarks* funcionó según lo que esperaba.
- El módulo de *benchmarks* me permitió entender mejor los algoritmos que implementé.

- El módulo de animaciones funcionó según lo que esperaba.
- El módulo de animaciones me permitió entender mejor los algoritmos que implementé.
- Encontré fallas/*bugs* en la herramienta.

Anexo B

Referencia para mantenedores

En este apéndice se detallan los procesos establecidos para el mantenimiento del software desarrollado. Se incluyen instrucciones para la publicación en PyPi, generación de documentación y pruebas unitarias.

B.1. Publicación a PyPi

La publicación de versiones en PyPi está automatizada a través de GitHub Actions. Para publicar una nueva versión, se debe crear un *tag* en el repositorio con el siguiente formato: `vX.Y.Z`, donde `X`, `Y` y `Z` son los números de versión mayor, menor y de parche, respectivamente. Al pushear el *tag* al repositorio, se disparará el proceso de publicación.

Luego de realizar un lanzamiento, se recomienda poblar el *changelog* con las novedades de la versión. Para ello, se debe acceder a la sección de `Releases` en GitHub (generalmente accesible a través de la URL `github.com/USUARIO/REPOSITORIO/releases`), y crear y editar la descripción de la versión correspondiente.

El código fuente de la acción mencionada puede ser encontrado en el archivo `.github/workflows/python-publish.yml` del repositorio.

B.2. Control de calidad de código con PreCommit

Desde el comienzo del desarrollo se ha utilizado la herramienta PreCommit para automatizar la ejecución de tareas de control de calidad de código. PreCommit se encarga de ejecutar una serie de herramientas de análisis estático de código y formateo de archivos antes de cada *commit*.

La configuración de la herramienta se encuentra en el archivo `.pre-commit-config.yaml` en la raíz del repositorio. Recomendamos visitar la documentación de PreCommit ("PreCommit: A framework for managing and maintaining multi-language pre-commit hooks.", s.f.) para la instalación local de la herramienta.

B.2.1. Controles configurados

- **check-yaml:** Verifica la validez sintáctica de los archivos YAML.
- **check-toml:** Verifica la validez sintáctica de los archivos TOML.
- **check-json:** Verifica la validez sintáctica de los archivos JSON.
- **check-ast:** Verifica la validez sintáctica de los archivos Python.
- **end-of-file-fixer:** Añade una nueva línea vacía al final de los archivos.
- **trailing-whitespace:** Elimina los espacios en blanco al final de las líneas.
- **mixed-line-ending:** Corrige los saltos de línea de los archivos. Toma el fin de línea más frecuente en el archivo y lo aplica a todas las líneas.
- **debug-statements:** Verifica que no haya sentencias de depuración en el código.
- **pyupgrade:** Actualiza el código Python deprecado a la última versión de la sintaxis.
- **isort:** Ordena las importaciones de los archivos Python.
- **black:** Formatea el código Python según las reglas de estilo de Black.
- **black-jupyter:** Formatea el código Python dentro de archivos Jupyter según las reglas de estilo de Black.

B.3. Documentación

B.3.1. Escritura y configuración de la documentación

La documentación, manuales y tutoriales de PyDistSim están escritos en formato RST y se generan automáticamente con Sphinx. Para generar la documentación, se deben instalar las librerías requeridas en `docs/requirements.txt` y ejecutar el comando `make html` desde la carpeta `docs/` del repositorio. La documentación generada se encontrará en la carpeta `docs/_build/html/`.

La configuración de Sphinx se encuentra en el archivo `docs/conf.py`. Los puntos principales son:

- **extensions:** Se especifican las extensiones de Sphinx que se utilizarán en la documentación. `autodoc` y `autosummary` generan parte de la documentación automáticamente a partir de los *docstrings* de los módulos. `inheritance_diagram` habilita la generación a demanda de diagramas de clases y herencias. `nbsphinx` permite la inclusión y visualización completa de notebooks de Jupyter en la documentación.

- **html_theme:** Se selecciona el tema de la documentación. Actualmente, se utiliza el tema de ReadTheDocs (*Read the docs sphinx theme*, s.f.).

Recomendamos visitar la documentación de Sphinx (*Sphinx*, s.f.) para más información sobre la generación de documentación.

B.3.2. Generación de la web de la documentación

La documentación de PyDistSim se encuentra alojada en `https://pydistsim.readthedocs.io/`. La generación de la documentación se realiza automáticamente cada vez que se realiza un *push* a la rama principal del repositorio. Para configurar la generación automática de la documentación, se debe vincular el repositorio de GitHub con la cuenta de ReadTheDocs y seleccionar la rama principal del repositorio como fuente de la documentación.

El archivo `.readthedocs.yaml` en la raíz del repositorio contiene la configuración de la generación de la documentación en ReadTheDocs. Allí se especifica la ruta del archivo de configuración de Sphinx y las versiones de Python en las que se generará la documentación, además de otros parámetros como la instalación de dependencias y la ejecución de comandos previos a la generación de la documentación.

Es posible consultar el historial de generación de la documentación en ReadTheDocs, así como los *logs* de errores y advertencias que se hayan producido durante la generación. Para ello, visitar `https://app.readthedocs.org/projects/pydistsim/`.

En caso de querer hacer cambios drásticos a la configuración global del proyecto, se recomienda visitar la documentación de ReadTheDocs (Read the Docs, s.f.) para más información sobre la configuración de la generación de documentación en la plataforma.

B.4. Pruebas unitarias, cobertura de código y análisis de memoria

Para mantener la calidad del código y asegurar su correcto funcionamiento, se han implementado pruebas unitarias y análisis de cobertura de código. Estas pruebas se ejecutan automáticamente en cada push a la rama principal del repositorio, aunque obviamente se pueden ejecutar manualmente en cualquier momento.

B.4.1. Instalación de dependencias de desarrollo

Para la ejecución manual de las pruebas, es necesario tener instaladas todas las dependencias de PyDistSim, así como las dependencias de desarrollo. Hacerlo debería ser tan sencillo como correr

```
$ pip install -r requirements.txt
```

y

```
$ pip install -r requirements-dev.txt
```

en la raíz del repositorio. Recomendamos hacer esto en un entorno virtual de Python.

B.4.2. Pruebas unitarias y cobertura de código

Para ejecutar las pruebas, se debe correr el comando "`pytest`". Esto ejecutará todas las pruebas unitarias y mostrará un resumen de los resultados. Además, registrará la cobertura de código en un archivo `.coverage`. Para visualizar la cobertura de código, se debe correr el comando "`$ coverage html`" y abrir el archivo `htmlcov/index.html` en un navegador web.

La configuración del comportamiento por defecto de las pruebas y la cobertura de código se encuentra en el archivo `pyproject.toml`, en la sección `[tool.pytest.ini_options]`.

B.4.3. Análisis de uso de memoria

Para verificar que el consumo de memoria de una ejecución típica de PyDistSim es razonable, se utiliza la herramienta `memray` (Wozniski, s.f.). Esta herramienta permite medir el uso de memoria de un *script* de Python y generar un reporte detallado.

Para ejecutar el análisis de memoria, se debe correr el comando "`$ memray run SCRIPT_NAME`". Sin embargo, lo que más se usó es "`$ memray run --output memray_output.bin -m pytest`", esto ejecutará las pruebas unitarias y generará un reporte de uso de memoria de las mismas en el archivo `memray\_output.bin`.

Para generar un reporte de tipo *flamegraph*, se debe correr el comando:

```
$ memray flamegraph memray_output.bin \
    --temporal \
    --force \
    --output memray_flamegraph.html
```

Esto generará un archivo `memray_flamegraph.html` que puede ser abierto en un navegador web para visualizar el reporte. Se adjunta una captura del reporte en la Figura B.1.

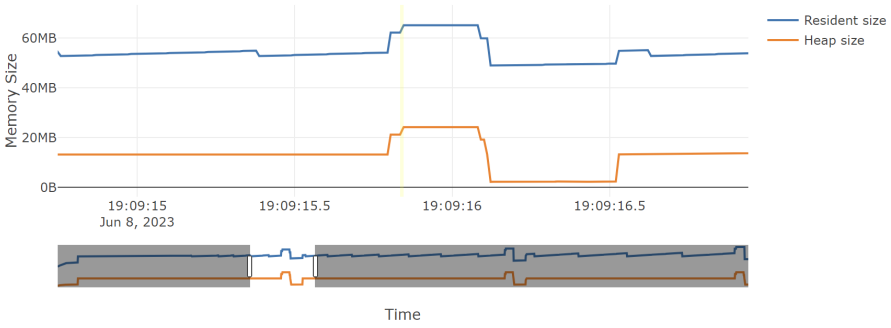


Figura B.1: Ejemplo de gráfico *flamegraph* generado con memray.

Anexo C

Esfuerzo de desarrollo

C.1. Distribución del esfuerzo

El proyecto se planificó en 4 fases. A continuación se detalla el esfuerzo dedicado[1] a cada parte y las actividades planificadas.

- **Conceptualización del proyecto.** Aproximadamente 40 días (86 horas de trabajo) que se usaron para hacer el análisis de antecedentes, presentar la propuesta a los supervisores y hacer una planificación detallada del transcurso del proyecto. Se incluyó en el análisis la prueba de distintas herramientas y el estudio de conceptos relacionados con la simulación de eventos discretos.
- **Revivir Pymote.** Aproximadamente 34 días (73 horas de trabajo) que se usaron para hacer la migración de Pymote a Python 3, la actualización de dependencias y la corrección de errores. Se incluyó en la migración la actualización de la documentación, la implementación de pruebas unitarias, el rediseño al sistema para asociar eventos y estados a implementaciones concretas (lo mencionado en la Sección 4.2.3) y la implementación de las herramientas de ayuda al desarrollo (mencionadas en la Sección 4.3.2).
- **Expandir el *core* y agregar funcionalidades.** Aproximadamente 76 días (163 horas de trabajo) que se usaron para:
  1. Agregar la funcionalidad de *timers* básica y su manejo avanzado.
  2. Refactorizar el modelado de redes para desacoplar la lógica de simulación, la de comunicación y la de generación de redes aleatorias.
  3. Hecha la refactorización, incluir el modelado de redes direccionadas.
  4. Implementar el retraso, pérdida y desordenamiento de mensajes.
  5. Implementar la desincronización de los relojes internos de los nodos.

[1] Se restringe al esfuerzo por parte del autor, sin incluir el apoyo por parte de los supervisores o las recomendaciones de Sebastián Pizard.

Autor	Inserciones	Borrados
Agustín Recoba	919.221	62.415
Damir Arbula	39.970	13.321
Matko Burul	771	90
Diego Sušanj	2	1

Tabla C.1: Conteo de operaciones de Git realizadas al código fuente, desde su implementación original, por autor.

6. Implementar el sistema de restricciones y las restricciones en sí.
7. Implementar el sistema de *observers*, su uso para *benchmarks* y la generación de gráficos.

- **Visualización de las simulaciones y fase de pruebas.** Aproximadamente 45 días (96 horas de trabajo) que se usaron para la implementación de todas las funcionalidades de visualización y animación y para planificar y ejecutar las tres pruebas descritas en el Capítulo 5. Al comienzo de esta fase fue que se decidió no intentar conseguir una interfaz gráfica independiente, sino una sencilla integración con los cuadernos de Jupyter, para poder dedicar el tiempo suficiente a la experimentación y validación.
- **Finalización del proyecto.** Aproximadamente 40 días (96 horas de trabajo) que se usaron en el análisis de los resultados de las pruebas, la redacción de documentación adicional y la redacción de este informe.

C.2. Estadísticas de git

Para ayudar a formar una idea de qué peso tuvo el desarrollo de PyDistSim vs. lo que ya estaba implementado en Pymote, adjuntamos algunas estadísticas obtenidas con `git-quick-stats` [2]

Como se ve en la tabla C.1, el conteo de código nuevo es un orden de magnitud superior a lo previo, mientras que el conteo de eliminaciones es cuatro veces mayor.

[2] https://git-quick-stats.sh/

Anexo D

Lista de símbolos, siglas y glosario

Glosario

benchmark Proceso de obtención y comparación de métricas de rendimiento de sistemas, en este caso, de algoritmos distribuidos. 25,

***binding* de Python** API que permite a Python interactuar con otros lenguajes de programación, bibliotecas y sistemas externos. 17,

changelog Registro de cambios en un software, generalmente organizado por versiones. 75,

commit Conjunto de cambios en el código que se guardan en el historial de versiones de un repositorio. 38, 39, 75,

core Núcleo de una biblioteca o *framework*, que contiene las funcionalidades esenciales. 15, 17, 18, 36, 41, 81,

fix Corrección de un error en el código. 44,

fork Copia de un repositorio de código ajeno en el propio, con el objetivo de modificarlo, mejorarlo o adaptarlo. v, 18, 20,

logging Proceso de registro de eventos en un sistema, con el objetivo de monitorear su funcionamiento. 38, 49,

log Registro de eventos en un sistema. 28, 77,

mesh Topología de grafos cexos en la que los nodos forman una malla regular de cuadrados. 22, 33,

mixin Clase que provee funcionalidades a otras clases, pero no se espera que sea instanciada por sí misma. 28, 33, 36,

pickle Módulo de Python que permite serializar y deserializar objetos de Python. 18,

push Acción de subir cambios locales a un repositorio remoto. 75,

tag Etiqueta que se le asigna a un commit para marcarlo como importante. 75,

ambiente distribuido Conjunto de entidades computacionales que se comunican entre sí.

anillo Topología de grafos en la que cada nodo está conectado a exactamente dos nodos, formando un círculo. 22, 24, 33,

árbol Característica topologica de grafos. Implica conexidad y que no existen ciclos. 24,

Black Herramienta de formateo de código Python, altamente opinionada. 76,

completo Topología de grafos en la que cada nodo está conectado a todos los demás nodos. 22, 24, 33,

comunicación Transmisión de mensaje.

conexidad Propiedad de un grafo de estar completamente conectado, es decir, que exista un camino entre cualquier par de nodos. 22, 24,

eclipse IDE multiplataforma para Java, desarrollado por la Fundación Eclipse. 14, 19,

eclipse JDT UI Paquete Java que implementa la interfaz de usuario de Eclipse para el desarrollo de Java. 15,

estrella Topología de grafos en la que un nodo central está conectado a todos los demás nodos. No hay aristas entre los nodos que no sean el central. 24, 33,

generador Rutina que produce secuencias de elementos, abstracción de un iterador[1]. 12,

grado Propiedad de un nodo de un grafo que indica la cantidad de aristas que lo conectan a otros nodos.

grado promedio Propiedad de un grafo que indica la cantidad promedio de aristas que conectan a un nodo con los demás. Es un indicador de la densidad y nivel de conexidad de la red. 22,

hipercubo Topología de grafos en la que cada nodo y arista está colocado siguiendo la estructura de un hipercubo. 22, 33,

[1] https://en.wikipedia.org/wiki/Generator_(computer_programming)

java Lenguaje de programación de alto nivel, orientado a objetos y multiplataforma, desarrollado por Sun Microsystems. 14,

mensaje Secuencia de bits de largo finito.

PyPi "Python Package Index", es un repositorio de software de Python. IX, 39, 75,

vecinos Los vecinos de entrada de un nodo son aquellos nodos desde los cuales el nodo puede recibir mensajes. Los vecinos de salida de un nodo son aquellos nodos a los cuales el nodo puede enviar mensajes.

Siglas

ADADR Curso Análisis y Diseño de Algoritmos Distribuidos en Redes. VIII, 1–4, 14, 15, 23, 41, 42, 48–50, 56, 61, 62, 64,

API Interfaz de Programación de Aplicaciones.

CLI Interfaz de Línea de Comandos.

EVA Entorno Virtual de Aprendizaje de la Universidad de La República. 42,

FING Facultad de Ingeniería de la Universidad de la República, institución pública de educación superior en Ingeniería del Uruguay. 41, 42,

GIF "Graphics Interchange Format", es un formato de imagen digital. 25,

Git Software de control de versiones distribuido. 38, 82,

GUI Interfaz Gráfica de Usuario. VIII, 20, 36,

IDE Entorno de Desarrollo Integrado. 14, 16, 56,

JSON "JavaScript Object Notation", es un formato de intercambio de datos. 76,

MP4 MPEG-4 Parte 14, es un formato estándar de contenedor multimedia. 25,

MVC Modelo-Vista-Controlador, es un patrón de arquitectura de software para implementar interfaces de usuario. 15,

SOLID Single responsibility, Open-closed, Liskov substitution, Interface segregation y Dependency inversion. 27,

SWT “Standard Widget Toolkit”, es una biblioteca gráfica de Java. 15,

TOML “Tom’s Obvious, Minimal Language”, es un formato de archivo de configuración. 76,

UI Interfaz de Usuario. 18,

YAML Acrónimo recursivo para “YAML Ain’t Markup Language” (en castellano, “YAML no es un lenguaje de marcado“). 76,

Anexo E

Manuales de uso y documentación

La documentación puede accederse en https://pydistsim.readthedocs.io/.

En la página siguiente se agrega la documentación al completo, en caso de que el lector prefiera no acceder o no tenga acceso a la página web.

PyDistSim

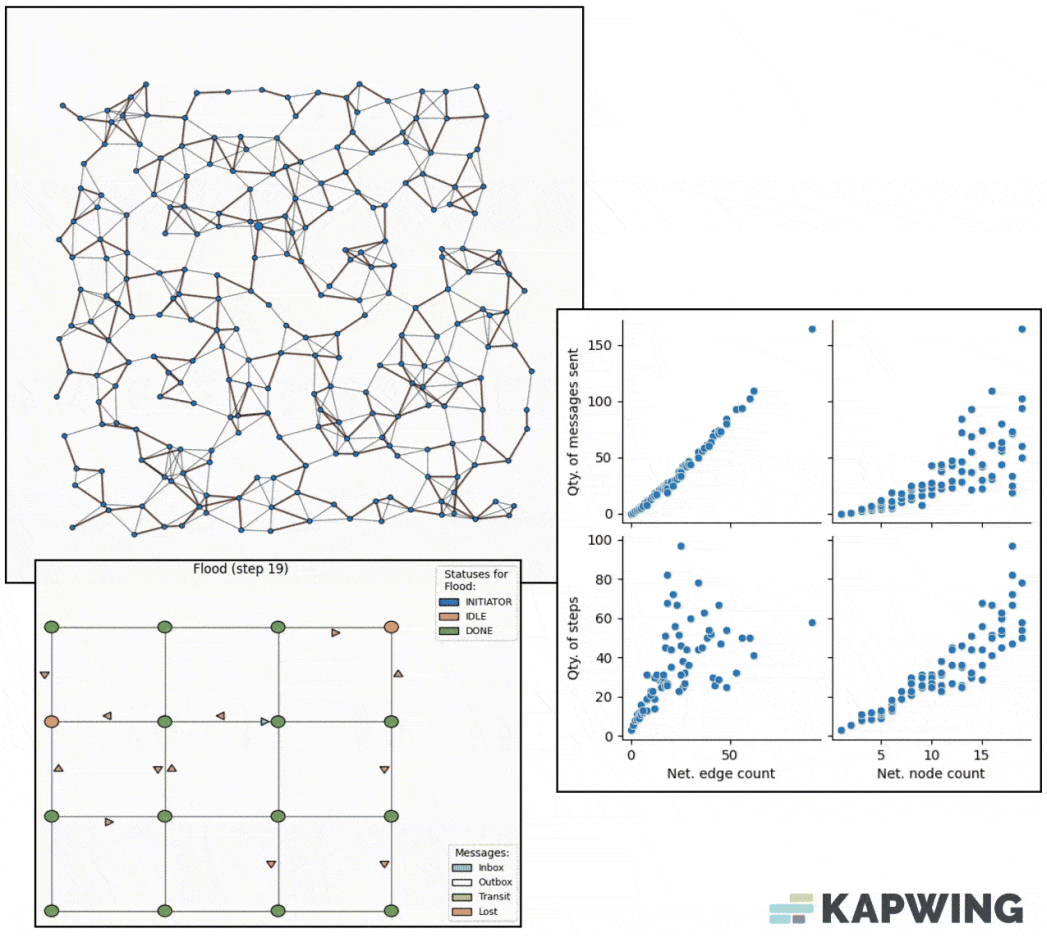
PyDistSim is a Python package for event-based simulation and evaluation of distributed algorithms. It is a fork of the deprecated Pymote.

This fork aims at providing new features, redesigned APIs and better documentation. It is being developed by Agustin Recoba in the context of his grade thesis at Facultad de Ingeniería, Universidad de la República.

Definition of the distributed environment, entities and actions used for making PyDistSim are taken mainly from Design and Analysis of Distributed Algorithms by Nicola Santoro.

PyDistSim's main goal is to provide a framework for fast implementation, easy simulation and data-driven algorithmic analysis of distributed algorithms.

Currently, PyDistSim supports IPython console or Jupyter notebooks. The gui is still in development and is not recommended for any type of use.



PyDistSim is being developed on top of NetworkX and is meant to be used along other scientific packages such as SciPy, NumPy and matplotlib. Currently, gui runs on PySide (Qt bindings) and console is jazzy IPython.

Installation

For installation instructions please visit the documentation.

Literature

Santoro, N.: *Design and Analysis of Distributed Algorithms*, 2006

Arbula, D. and Lenac, K.: *Pymote: High Level Python Library for Event-Based Simulation and Evaluation of Distributed Algorithms*, International Journal of Distributed Sensor Networks, Volume 2013

Recoba, A: *PyDistSim: Framework de simulación de algoritmos distribuidos en redes en Python*, 2024

Explore the documentation

Indices and tables

This document refers to the PyDistSim version 2.1.1.

Build date: Nov 02, 2024

Installation

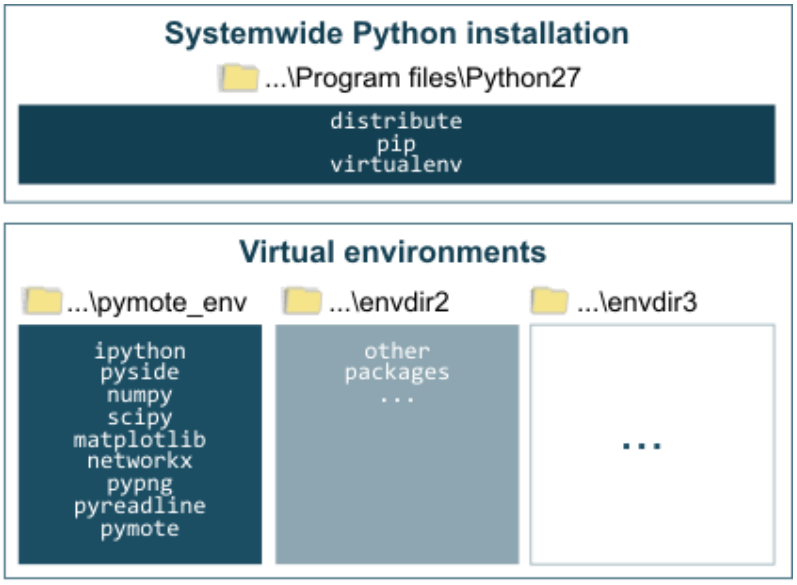
This document assumes you are familiar with using command prompt or shell. It should outline the necessary steps to install software needed for using PyDistSim.

Requirements

PyDistSim requires Python 3.11. Every other dependency will be installed automatically by pip.

Note

Since there can be only one version of any package installed system-wide in some cases this can result in situation where two programs need different versions of the same package. This is resolved by using isolated virtual environments.



Virtual environments live in a separate directories and they are independent form system-wide Python installation.

Alternatively, if none of the above is your concern, although not recommended, PyDistSim and all required packages can be installed system-wide using their respective instructions for appropriate OS, just jump to the end of this document.

Windows

Windows version should be as simple as installing Python 3.11 for windows.

Linux (Debian/Ubuntu based)

To install Python 3.11, run:

```
$ sudo apt install python3.11
```

Depending of the flavour of linux, some packages are required for getting and compiling the source, only install if the above command fails:

```
$ sudo apt-get install libxkbcommon-x11-0 libegl1 opencv-python-headless libgl1-mesa-glx
```

Virtualenv

pip and virtualenv (venv) are included in Python 3.11. To create a new virtual environment run:

```
$ python3.11 -m venv pydistsim_env
```

Activate virtual environment on Linux:

```
$ source pydistsim_env/bin/activate
```

Activate virtual environment on Windows:

```
$ pydistsim_env\Scripts\activate
```

PyDistSim

Finally, in order to download and install PyDistSim and all other required packages, run:

```
(pydistsim_env) $ pip install PyDistSim
```

Tutorials

Tutorials assume that the PyDistSim and all required packages are installed. If not, please refer to the Installation section of this documentation.

Full explanation of topics

- Implementing a distributed algorithm
- Specifying network restrictions for an algorithm
- Network generation intro

Demo notebooks

Hello distributed world

This tutorial demonstrates a distributed version of the classic *Hello world* example. Basic usage of the library is described in this demo, just follow the steps.

Automatic network generation

In this tutorial, we demonstrate how to generate a network automatically. Access the network generation notebook.

Creating an animation of a run

Here we demonstrate how to generate an animation of a simulated algorithm in any given network. Look at the animation notebook.

Visualize the construction of a tree

This tutorial demonstrates how to visualize the construction of a tree in a distributed algorithm. Check out the tree construction notebook.

Visualize the effect of network behavioral properties

This tutorial exemplifies how to visualize the effect of network behavioral properties on a distributed algorithm. Access the network properties notebook.

Algorithm benchmarking

This tutorial demonstrates how to benchmark a distributed algorithm. Access the benchmarking notebook.

Custom algorithm observers

This tutorial covers how to implement a observable algorithm and its corresponding observer. Access the notebook here.

Ultimate example: Mega-Merger implementation and analysis

Check out Mega-Merger implementation and the notebooks:

1. Animation notebooks.
2. Analysis notebook.
3. Testing notebook.

Build date: Nov 02, 2024

Release version: 2.1.1

Implementing a distributed algorithm

In order to implement a distributed algorithm, the class must fulfill these requirements:

1. It must be a subclass of `NodeAlgorithm` class.
2. It should have a class attribute `NodeAlgorithm.Status` which is an enumeration of the possible states of the node. This enumeration must subclass `StatusValues`.
3. Action implementations must be methods of the class, they must be called as the action itself and must be decorated with a member of the Status enumeration. One such implementation should look like this:

```
@Status.IDLE
def receive(self, node: NodeAccess, message: Message):
    if is_my_favorite_neighbor(message.source):
        # Send message
        self.send(
            node,
            data='Hi! Wanna play?',
            destination=message.source,
            header="PLAY INVITATION",
        )
        # Remind myself to send the message again in 10 simulation seconds
        self.set_alarm(
            node,
            time=10,
        )
        # Change my status to WAITING
        self.status = self.Status.WAITING
    else:
        # Ignore the message, I don't like this guy
        node.memory['SPAM_COUNT'] += 1
```

Here, `self` is the instance of the algorithm class, `node` is the node that is executing the action and `message` is the message that triggered the action. You would use the node to access the node's `status` and `memory`, the message to access the message's `data`, `header` and `source`; and `self` to access the algorithm's interfaces for sending messages and setting alarms.

Note

To help the programmer, there is a special action `Actions.default` which will be called if the action was not implemented for a given status.

Warning

Actions without an implementation will only log a warning message, so for no-ops, it is not necessary to implement anything.

Node's own id and neighbor labels

The framework is designed to prevent the programmer from accessing information that should not be available to the nodes. For this reason, we provide two classes that act as proxies for the node and its neighbors, respectively.

NodeAccess class

A `NodeAccess` instance represents a node's own view, encapsulating the knowledge gathered so far (inside the `memory` attribute) and inherent of the node (such as who are it's neighbors).

By default, this class allows read-only access to the node's `clock` attribute and read-and-write access to the `memory` and `status` attributes.

In addition to this attributes, each NodeAccess instance has an integer `id` attribute, by default it's random and won't be unique in the network.

These are the two main ways a programmer comes across a `NodeAccess` instance:

```
@Status.IDLE
def receiving(self, node: NodeAccess, message: Message):
                      ↑

def initializer(self):
    for node in self.nwm.nodes():
          ↑

        node.memory['SPAM_COUNT'] = 0


    for node in self.network.nodes():
          ↑ NOT A NodeAccess INSTANCE
```

NeighborLabel class

A `NeighborLabel` instance represents the knowledge of the neighbor respect to current processing node.

Similar to the `NodeAccess` class, the `NeighborLabel` instances have an integer `id` attribute, by default it's random and won't match any other `id` in the network. It's only used to identify that neighbor among the other neighbors.

These are the two main ways a programmer comes across a `NeighborLabel` instance:

```
@Status.IDLE
def receiving(self, node: NodeAccess, message: Message):
    source = message.source  # The node that sent the message
      ↑

    for neighbor in node.neighbors():
          ↑
```

"The Santoro's interface"

Send a message

The `NodeAlgorithm.send_msg()` method receives a source node and a message. The message will be sent to the destinations specified in the message itself. The destinations must be a list of nodes or a single node. The message will be sent to all the destinations.

The `NodeAlgorithm.send()` method receives a source node, a message content a list of destinations and, optionally, a header. Is equivalent to the `NodeAlgorithm.send_msg()` method, but the message is created automatically.

Set an alarm

The `NodeAlgorithm.set_alarm()` method receives a node and a time. An empty message will be sent to the node after the time has passed. The time is in algorithm steps, which behavior varies according to the algorithm and the simulationparameters.

Check the extended interface methods for advanced use of the alarm mechanism.

Open and close edges/links

The `NodeAlgorithm.close()` and `NodeAlgorithm.open()` methods implement the closing and opening of edges, respectively. The arguments are the blocking node and the blocked node. From the call of `NodeAlgorithm.close()` until the call of `NodeAlgorithm.open()`, the blocking node will not be able to receive any messages from the blocked node. The blocked node will not be notified of the blocking. The blocked node can still send messages to the blocking node, and once the edge is opened, the blocking node will receive all the messages sent during the blocking period. The blocking node can still send messages to the blocked node.

Helper functions and the extended interface

Block and unblock messages - Keep in queue and receive later

With the help of some functional programming, the `NodeAlgorithm.block_inbox()` and `NodeAlgorithm.unblock_inbox()` methods allow a node to block the reception of messages for any given condition. The blocked messages are kept in a queue and will be delivered to the node when the blockade is lifted. The condition is a function that receives a message and returns a boolean. If the function returns `False`, the message will be blocked.

The usage is this simple:

1. Call `NodeAlgorithm.block_inbox()` with the condition function and store the returned object in the node's memory.
2. Call `NodeAlgorithm.unblock_inbox()` with the object returned by the `NodeAlgorithm.block_inbox()` method.

Tip

The condition function can be a lambda function, a function defined in the algorithm class or a function defined outside the algorithm class. One such function could look like this:

```
# Normal function
def ignore_message(message: Message):
    return message.header != 'IGNORE_ME'

# Lambda function
ignore_message = lambda message: message.header != 'IGNORE_ME'
```

Alarm management

The `NodeAlgorithm.set_alarm()` method receives an extra, optional, parameter, the message. This message will be sent to the node when the alarm goes off. The message must be an instance of `Message`. This, however, is not meant to be used to send messages to other nodes, but to the node itself.

Note

To schedule a message to be sent to another node, we recommend using the `NodeAlgorithm.send()` method inside the callback of the alarm.

The `NodeAlgorithm.set_alarm()` method returns an object that represents the alarm. This object can be used to disable the alarm, update the time, check the time left for the alarm to go off and check if the alarm is active.

To disable alarms, the following methods are available:

- `NodeAlgorithm.disable_all_node_alarms()` which disables all alarms of the node.
- `NodeAlgorithm.disable_alarm()` which disables a specific alarm. The argument is the alarm object itself, which is returned by the `NodeAlgorithm.set_alarm()` method.

And finally, to add or subtract time from a pending alarm, `NodeAlgorithm.update_alarm_time()` can be used. The arguments are the alarm object and the time to be added or subtracted. If the time subtracted is greater than the time left for the alarm to go off, the alarm will be triggered the next "alarm triggering event" (usually the next step of the algorithm).

Network generation intro

PyDistSim comes with a battery of different methods for automated network creation. It works in two modes:

1. Instanced generation
2. Static generation

When you use it in instance mode, you instantiate the `NetworkGenerator` with the parameters you want the network to have and on that instance you call `generate_random_network` or `generate_homogeneous_network` to return a network. This is the most configurable way to generate random networks but without any particular structure. Check the `NetworkGenerator` class reference for more information on the parameters you can use.

On the other hand, when you use the class or static mode, you do not need to instantiate NetworkGenerator, just call its class methods:

1. `NetworkGenerator.generate_complete_network()`
2. `NetworkGenerator.generate_ring_network()`
3. `NetworkGenerator.generate_star_network()`
4. `NetworkGenerator.generate_hypercube_network()`
5. `NetworkGenerator.generate_mesh_network()`

These class methods receive a minimal configuration of parameters (in general only the number of nodes and if the network should be directed or not).

Static generation is used for quick network generation without any additional conditions. It is appropriately exemplified in the network generation notebook.

Specifying network restrictions for an algorithm

In order to declare restrictions for a distributed algorithm, the developer must define the class attribute `NodeAlgorithm.algorithm_restrictions` as a list or tuple of restriction types. Such types must be subclasses of `pydistsim.restrictions.base_restriction.Restriction`.

For example, for a broadcast algorithm, the restrictions could be:

```
algorithm_restrictions = (
    BidirectionalLinks,
    TotalReliability,
    Connectivity,
    UniqueInitiator,
)
```

See also

For the simulation to enforce the restrictions during runtime, the attribute `check_restrictions` of `pydistsim.simulation.Simulation` must be set to *True*.

For a full example of a broadcast algorithm with restrictions, please refer to the file `pydistsim/demo_algorithms/broadcast.py` in the source code.

Note

For the full list of available restrictions, please refer to the module reference of `pydistsim.restrictions` in this documentation.

Warning

Some of the implemented restrictions require the algorithm to take action in the algorithm initialization. For example, the `pydistsim.restrictions.topological.UniqueInitiator` restriction requires the algorithm to set the initiator node.

Hello distributed world

The goal of this tutorial is to setup a simulation and run it by using a minimal example.

[2]:
```
%matplotlib inline

from pydistsim import NetworkGenerator, Simulation
from pydistsim.demo_algorithms.santoro2007.yoyo import YoYo


# Create a network with 30 nodes and undirected edges
net_gen = NetworkGenerator(30, directed=False)
net = net_gen.generate_random_network()

# Create a simulation object
sim = Simulation(net)

# Assign a simple algorithm to the network, must be a tuple
sim.algorithms = (YoYo,)

# Create and run the simulation
sim.run()
```

That's it! For more elaborate description please continue reading.

[3]:
```
# For more detailed output, set the log level to INFO
from pydistsim.logging import set_log_level, LogLevels, enable_logger

set_log_level(LogLevels.INFO)
enable_logger()
```

In the example given above, the goal was to simulate distributed algorithm `YoYo` on arbitrary network. Algorithm `YoYo` solves *Election* problem, i.e. the goal of the algorithm is to find the node with the lowest ID in the network.

Generating network

Networks are fundamental objects in the PyDistSim and they can be created by instantiating `Network` class or by using `NetworkGenerator` class. `NetworkGenerator` can receive different parameters such as number of nodes (exact, min, max), average number of neighbors per node etc.

In this example, for simplicity, we use a simpler generator so the only parameter that we want to have in control is number of nodes. which is set to 11.

[4]:
```
net = NetworkGenerator.generate_star_network(11)
```

The result is an instance of the BidirectionalNetwork class:

[5]:
```
net
```

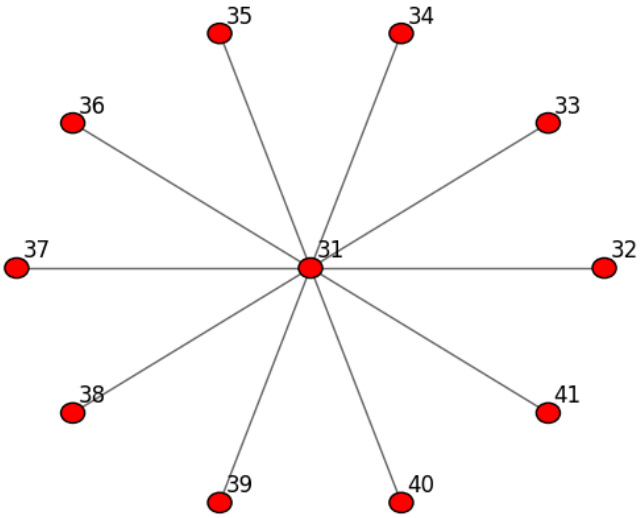
[5]:
```
<pydistsim.network.network.BidirectionalNetwork at 0x7f923c0b0b10>
```

which can be visualized with its `show()` method:

[6]:
```
net.show()
```

```
/mnt/d/Proyectos/pymote/docs/notebooks/../../pydistsim/network/network.py:560: UserWarning:
FigureCanvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```

[6]:



Now, we create the simulation object and set the network to the one we just created:

[7]:
```
sim = Simulation(net)
```

```
2024-10-22 23:40:48.975 | INFO     | pydistsim.simulation:__init__:59 - Simulation 0x7f91cd195b10
created successfully.
```

Algorithm and Simulation

To demonstrate the simulation, we will use a simple `Echo` algorithm. The algorithm is a simple flooding algorithm that sends a message to all neighbors and waits for the response from all of them. Once every neighbor responds, the algorithms terminates.

Implementation of `Echo` algorithm is given below:

[8]:
```
from pydistsim.algorithm.node_algorithm import NodeAlgorithm, StatusValues
from pydistsim.algorithm.node_wrapper import NodeAccess
from pydistsim.message import Message
from pydistsim.restrictions.communication import BidirectionalLinks
from pydistsim.restrictions.reliability import TotalReliability
from pydistsim.restrictions.topological import Connectivity, UniqueInitiator


class Echo(NodeAlgorithm):
    default_params = {
        "echo_message": "Hello world!",
    }

    class Status(StatusValues):
        INITIATOR = "INITIATOR"
        AWAITING_ECHO = "AWAITING_ECHO"
        AWAITING_ECHO_RESPONSE = "AWAITING_ECHO_RESPONSE"
        DONE = "DONE"

    S_init = (Status.INITIATOR, Status.AWAITING_ECHO)
    S_term = (Status.DONE,)

    # Here we define under which restrictions the algorithm is designed to work
    algorithm_restrictions = (
        BidirectionalLinks,  # The algorithm requires bidirectional links, so that the response
can be sent back
        TotalReliability,  # The algorithm requires that all messages are delivered without loss
        Connectivity,  # The algorithm requires that the network is connected
        UniqueInitiator,  # The algorithm requires that only one node is the initiator
    )

    def initializer(self):
        # Set all nodes to AWAITING_ECHO
        for node in self.network.nodes():
            node.status = self.Status.AWAITING_ECHO

        # Choose the initiator
        ini_node = self.network.nodes_sorted()[0]

        # Send the initial message to the initiator
        ini_node.push_to_inbox(Message(meta_header=NodeAlgorithm.INI, destination=ini_node))

        # Set the initiator status to INITIATOR and store the initial information
        ini_node.status = self.Status.INITIATOR
        ini_node.memory["message"] = self.echo_message

    @Status.INITIATOR
    def spontaneously(self, node: NodeAccess, message: Message):
        self.send(
            node,
            data=node.memory["message"],
            destination=list(node.neighbors()),  # send to all neighbors
            header="Echo",
        )
        node.memory["responseCount"] = 0
        node.status = self.Status.AWAITING_ECHO_RESPONSE

    @Status.AWAITING_ECHO_RESPONSE
    def receiving(self, node: NodeAccess, message: Message):
        if message.header == "Echo response":
            # Count the number of responses received, only if the response is the same as the
original message
            node.memory["responseCount"] += 1 if message.data == node.memory["message"] else 0

            # If all responses have been received, set the node status to DONE
            if node.memory["responseCount"] == len(list(node.neighbors())):
                node.status = self.Status.DONE

    @Status.AWAITING_ECHO
    def receiving(self, node: NodeAccess, message: Message):
        if message.header == "Echo":
            node.memory["message"] = message.data

            self.send(
                node,
                data=message.data,
                destination=message.source,
                header="Echo response",
            )
        node.status = self.Status.DONE

    @Status.DONE
    def default(self, *args, **kwargs):
        "Do nothing, for all inputs."
        pass
```

Assigning algorithm to the simulation

Two things should be noted:

- there can be multiple algorithms that are being assigned to the simulation so specific algorithms are elements of `tuple`. Since in this example there is only one algorithm we must append `,` so that Python knows it's a one element `tuple` i.e. `(1)` is `int`, but `(1,)` is `tuple`.
- every algorithm element is `tuple` itself, consisting of two elements. Former is an algorithm class, in this example `Echo`. Later is a `dict` of keyword parameters, i.e. `Echo` must be given 'echo_message' as an optional parameter: the message to be sent.

[9]:
```
sim = Simulation(net)
MESSAGE = "Hello distributed world"
sim.algorithms = ((Echo, {"echo_message": MESSAGE}),)
```

```
2024-10-22 23:40:51.184 | INFO     | pydistsim.simulation:__init__:59 - Simulation 0x7f91cd27f910 created successfully.
```

Running

Finally, run simulation:

[10]:
```
sim.run()
```

```
2024-10-22 23:40:55.964 | INFO     | pydistsim.simulation:_run_algorithm:148 - [Echo] Algorithm finished
```

After the algorithm(s) execution is done we can check if the resullt is as expected, that is every node should have information in their memory:

[11]:
```
for node in net.nodes():
    assert node.memory["message"] == MESSAGE, "Oh..."
    assert node.status == Echo.Status.DONE, "Oh..."

print("Everything is OK!")
```

```
Everything is OK!
```

During simulation execution, network and its nodes are changing their status, memory content etc. To memorize the currently running algorithm and current step of the simulation, network has a special attribute `algorithmsState`:

[12]:
```
sim.algorithmState
```

[12]:
```
{'index': 0, 'step': 17, 'finished': True}
```

If simulation is going to be run again, it must be reset:

[13]:
```
sim.reset()
```

whereby:

- `algorithmState` is returned to the initial value:

[14]:
```
sim.algorithmState
```

[14]:
```
{'index': 0, 'step': 1, 'finished': False}
```

- memory content of all node's is deleted (as long as some other attributes)

[15]:
```
for node in net.nodes():
    assert len(node.memory) == 0
print("All memories are empty!")
```

```
All memories are empty!
```

[16]:
```
sim.run()
```

```
2024-10-22 23:41:11.618 | INFO     | pydistsim.simulation:_run_algorithm:148 - [Echo] Algorithm finished
```

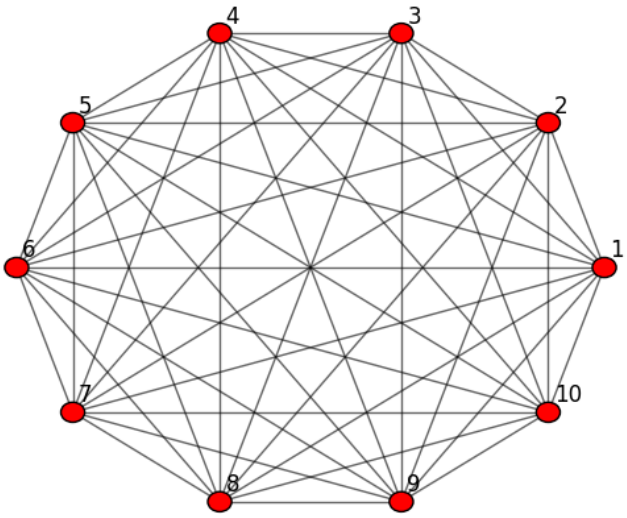
Example usage of the NetworkGenerator class

[3]:
```
# for interactive plots
%matplotlib inline

from pydistsim.logging import set_log_level, enable_logger
from pydistsim.network.generator import NetworkGenerator
```

[4]:
```
set_log_level("INFO")
enable_logger()
```

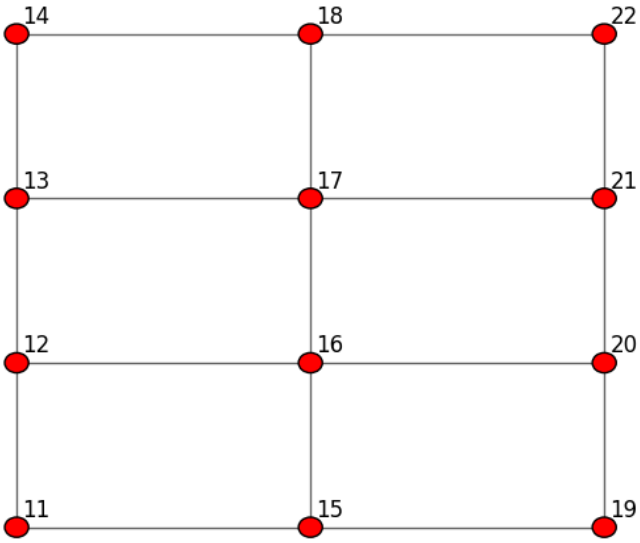
[5]:
```
net = NetworkGenerator.generate_complete_network(10)  # Generate a complete network with 10 nodes
net.show()
```

[5]:



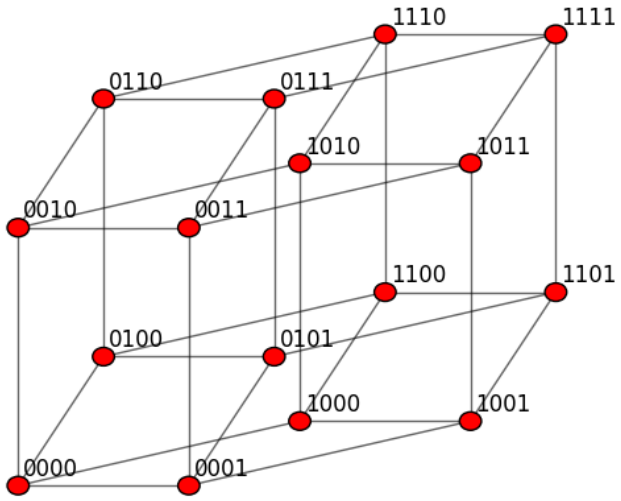
[6]:
```
net = NetworkGenerator.generate_mesh_network(
    12, 3, 4, torus=False
)  # Generate a mesh network with 12 nodes, 4 rows and 3 columns
net.show()
```

[6]:



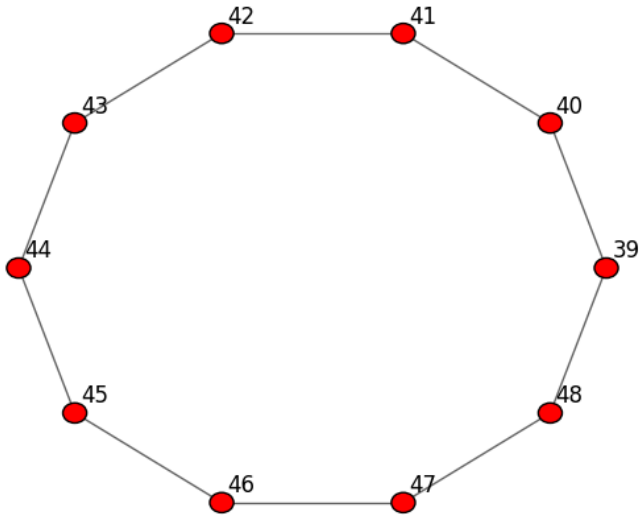
[7]:
```
net = NetworkGenerator.generate_hypercube_network(16)  # Generate a hypercube network with 16
nodes
net.show()
```

[7]:



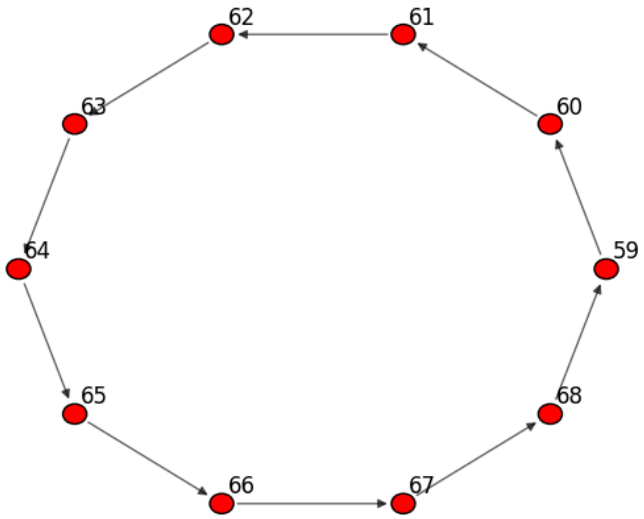
[8]:
```
net = NetworkGenerator.generate_ring_network(10)  # Generate a ring network with 10 nodes
net.show()
```

[8]:



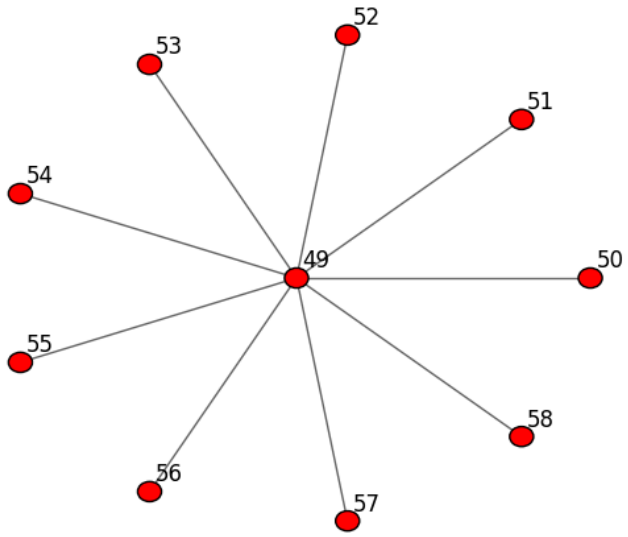
[10]:
```
net = NetworkGenerator.generate_ring_network(10, directed_network=True)  # Now directed
net.show()
```

[10]:



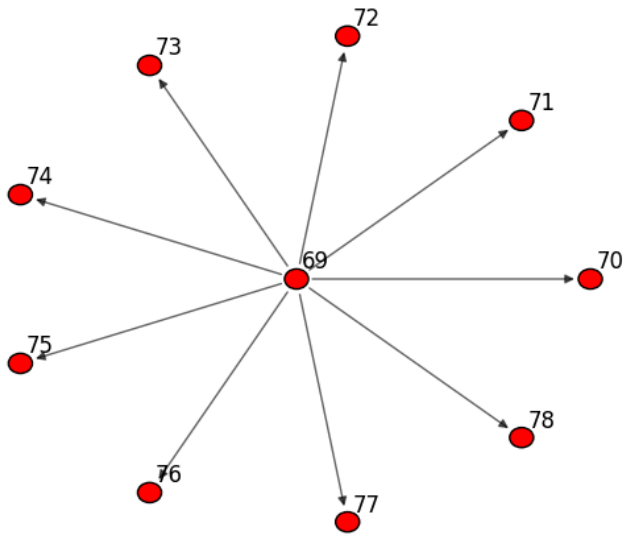
[9]:
```
net = NetworkGenerator.generate_star_network(10)  # Generate a star network with 10 nodes
net.show()
```

[9]:



[11]:
```
net = NetworkGenerator.generate_star_network(10, directed_network=True)  # Now directed
net.show()
```

[11]:



Algorithm run animation creator

[6]:
```
# for interactive plots
%matplotlib notebook

from pydistsim.network import NetworkGenerator
from pydistsim.demo_algorithms.santoro2007.yoyo import YoYo
from pydistsim.simulation import Simulation
from pydistsim.network.behavior import NetworkBehaviorModel
from pydistsim.gui import drawing as draw
```

As always, let's start with network and simulation object creation

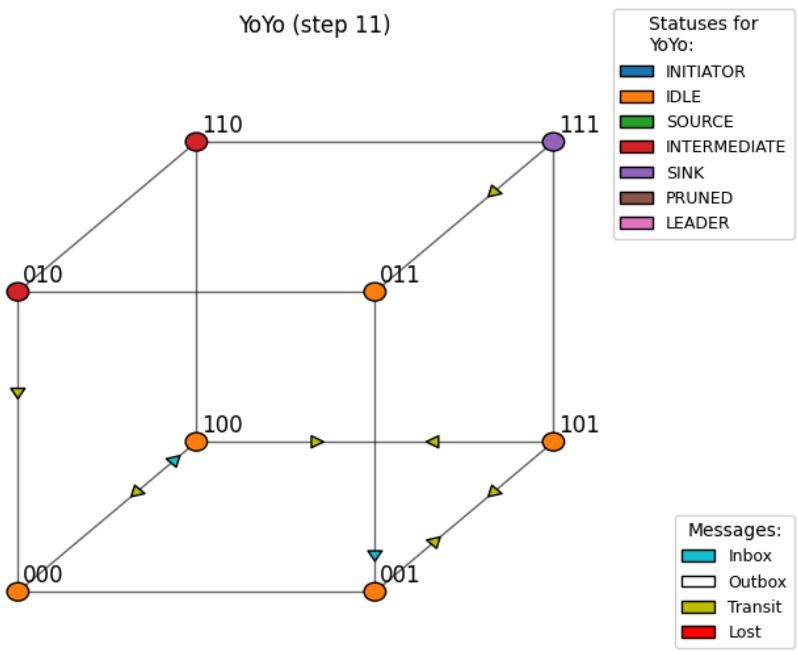
[7]:
```
net = NetworkGenerator.generate_hypercube_network(8)

# Let's add some random delays to the communication, so we can see it in action
net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication


sim = Simulation(net, (YoYo,))
```

Create a frame with the simulation's current state

In the following cell, we show how to create a figure with the current state of the simulation. This will show each node's state and the messages that are currently in transit, lost, delivered, or pending to be sent.

[8]:
```
sim.run(10)
fig = draw.draw_current_state(sim)

fig
```

[8]:

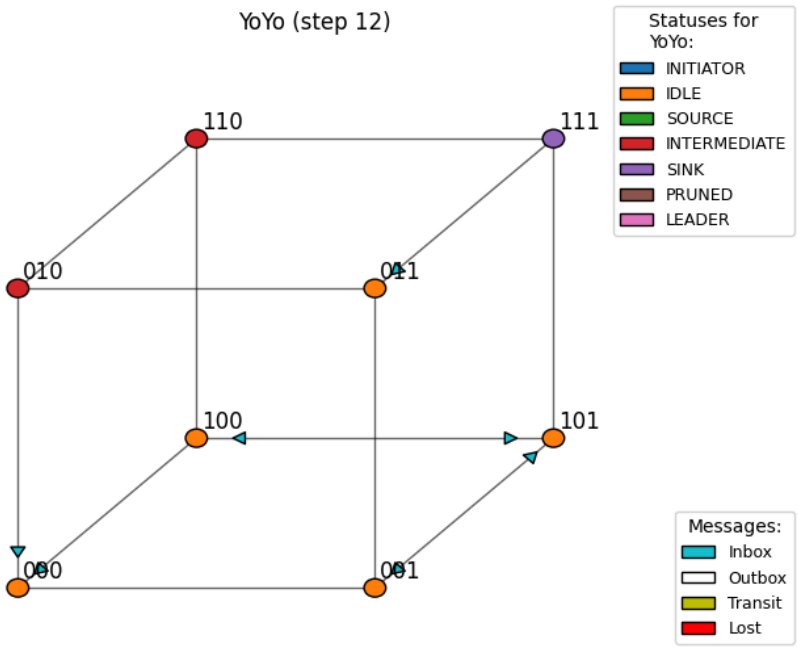


Keep going and watch the simulation as it runs

We can run the simulation and update the figure at each step to see how the algorithm works.

[9]:
```
sim.run(1)
fig = draw.draw_current_state(sim)

fig
```

[9]:



Video creation

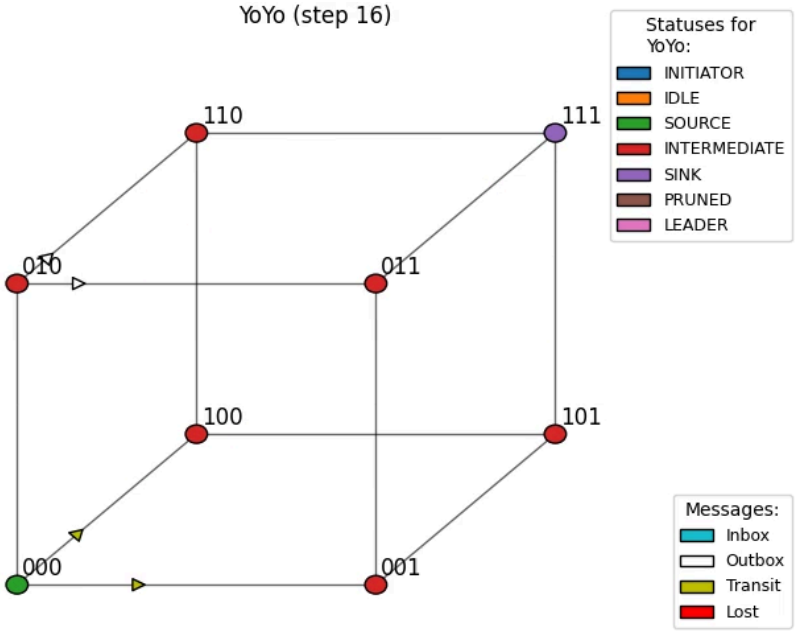
The simplest way to understand how the algorithm works is to watch it in action. We can create something like a stop-motion video of the simulation to see how the algorithm evolves over time.

In the following cell, we show how to create a video with the whole simulation process. You should be able to watch the video from the notebook.

[10]:

```
from IPython.display import HTML

sim.reset()
anim = draw.create_animation(sim)
video = anim.to_html5_video()

HTML(video)  # Careful! This must be the last line of the cell, otherwise the video will not be displayed.
```

[10]:



To save the video, you can use the `save` method of the `Animation` class.

This will raise an exception if FFMpeg is not installed.

Learn more about animation writers at:

https://matplotlib.org/stable/api/animation_api.html#writer-classes

[6]:
```
from matplotlib.animation import FFMpegFileWriter

writer = FFMpegFileWriter()
sim.reset()
anim = draw.create_animation(sim)

anim.save("yoyo.mp4", writer=writer)
```

Visualize the effect of network behavioral properties on distributed algorithms

[6]:
```
# for interactive plots
%matplotlib notebook

from pydistsim.network import NetworkGenerator
from pydistsim.demo_algorithms.broadcast import Flood
from pydistsim.simulation import Simulation
from pydistsim.network.behavior import (
    NetworkBehaviorModel,
    random_loss,
)

from pydistsim.gui import drawing as draw
from IPython.display import HTML
```

Introduction

First, we will define the network and the distributed algorithm that we will use to illustrate the effect of network properties on distributed algorithms. We will use the **Flood** as the distributed algorithm and a 4 by 4 grid as the network.

[7]:
```
net = NetworkGenerator.generate_grid_network(16)
sim = Simulation(net, (Flood,))
```

Now, let's create a helper function to visualize the network and the algorithm, with the behavioral properties as a parameter. We also define whether or not the visualization will display the internal clock of the nodes.

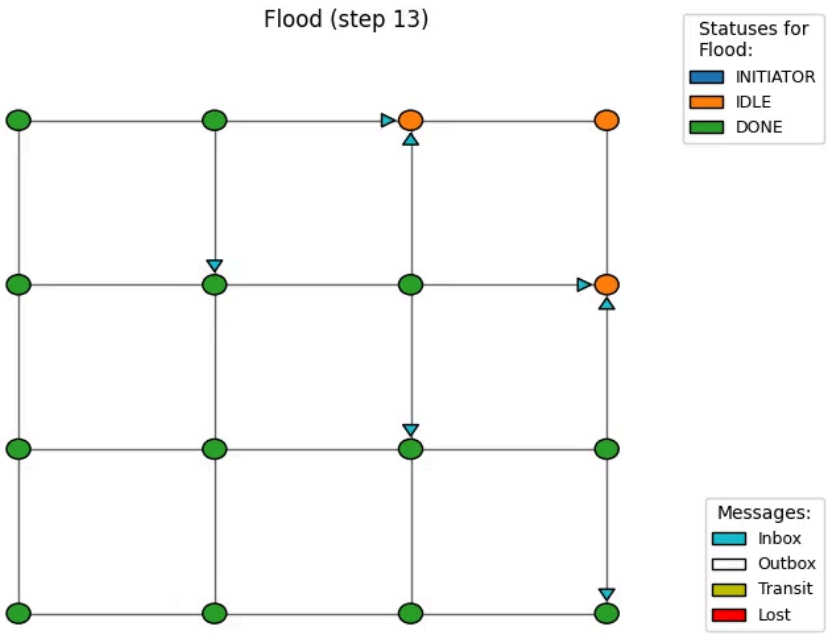
[10]:
```
def make_vid(properties: NetworkBehaviorModel, clock_as_label: bool = False, **kwargs) -> HTML:
    # Set the network behavior properties
    net.behavioral_properties = properties

    # Dont interrupt the simulation when a restriction is violated (Total Reliability in this
case)
    sim.check_restrictions = False
    sim.reset()

    if clock_as_label:
        # Show the clock of each node as a label, for that, we need to pass a lambda function to
the node_labels parameter
        kwargs_new = {"show_labels": True, "node_labels": lambda: {node: node.clock for node in
net.nodes}}
    else:
        kwargs_new = {"show_labels": False}

    kwargs_new.update(kwargs)

    # Create the animation
    anim = draw.create_animation(sim, **kwargs_new)
    video = anim.to_html5_video()

    return HTML(video)
```

Ideal Communication

First, the simplest scenario is when the network has no loss and no delay. In this case, the algorithm will appear to be executing in a synchronous manner.

[13]:
```
make_vid(NetworkBehaviorModel.IdealCommunication)
```

[13]:

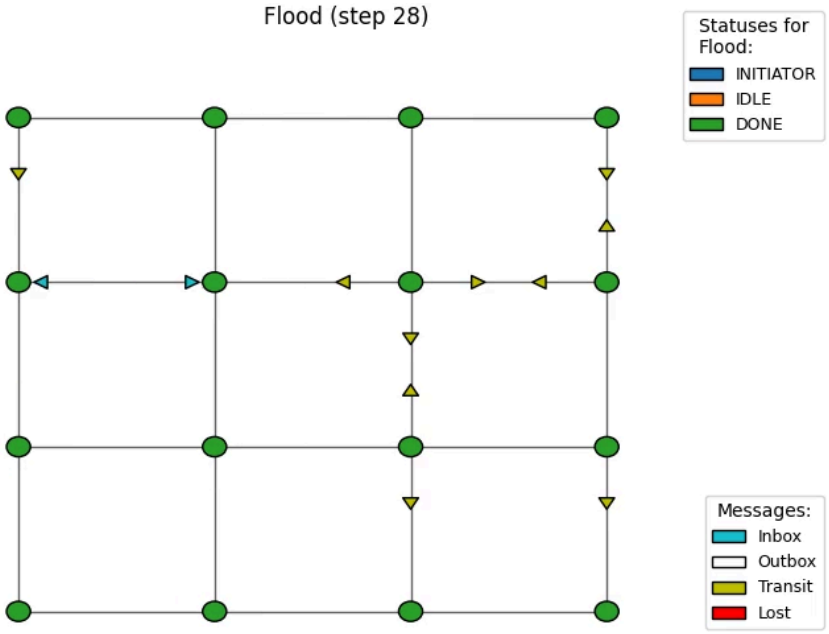


Delay

Now, let's visualize the effect of delay on the distributed algorithm. We will introduce a random delay in every edge of the network. The delay will be between 1 and 16 time units.

[6]:
```
# RandomDelayCommunication introduces random delays in the communication, proportional to the size of the network
make_vid(NetworkBehaviorModel.RandomDelayCommunication)
```

[6]:

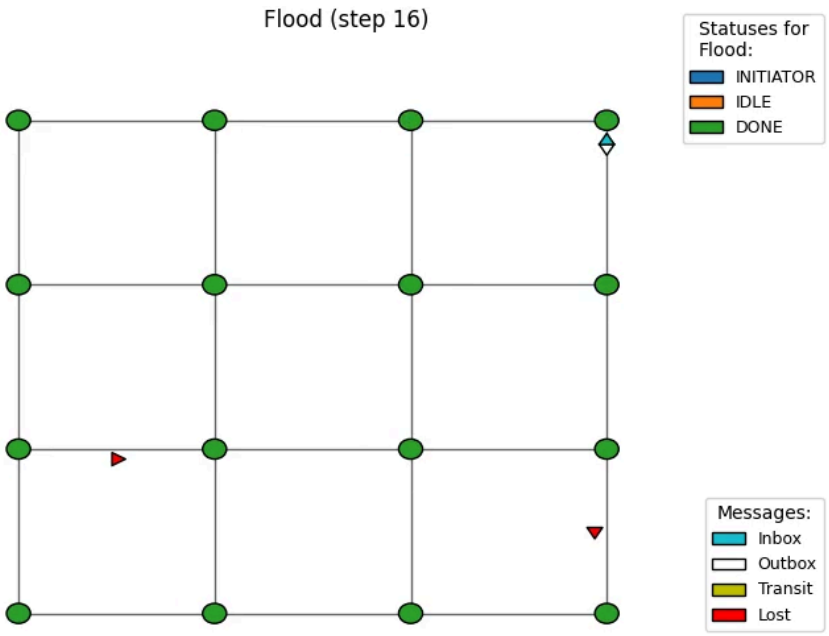


Loss

Here we will visualize the effect of loss on the distributed algorithm. We will introduce a random loss in every edge of the network. The loss will happen with a probability of 0.1.

[7]:
```
# UnlikelyRandomLossCommunication sets the probability of a message being lost to 0.1
make_vid(NetworkBehaviorModel.UnlikelyRandomLossCommunication)
```

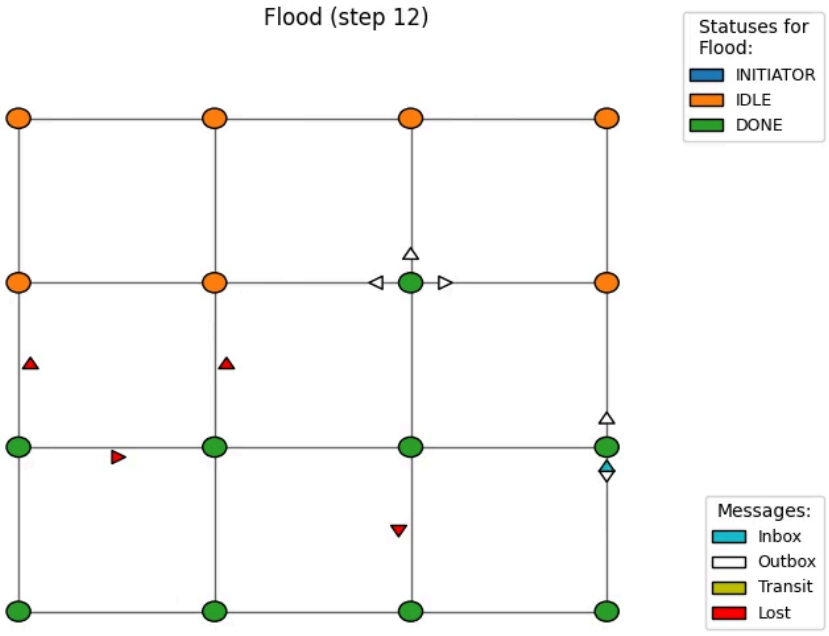
[7]:



We now increase the loss probability to 0.5. It should be low enough to allow the algorithm to terminate on most executions.

[20]:
```
# Create our own NetworkBehaviorModel, with a message loss probability of 0.5
make_vid(NetworkBehaviorModel(message_loss_indicator=random_loss(0.5)))
```

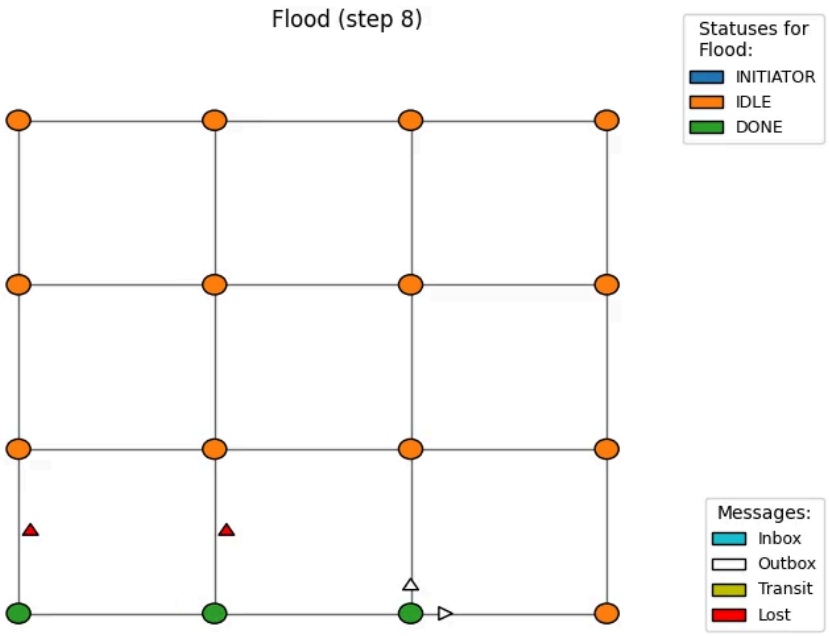
[20]:



Finally, we increase the loss probability to 0.7. The algorithm will surely not terminate.

[38]:
```
# Now with much higher message loss probability
make_vid(NetworkBehaviorModel(message_loss_indicator=random_loss(0.7)))
```

[38]:



Clock skew

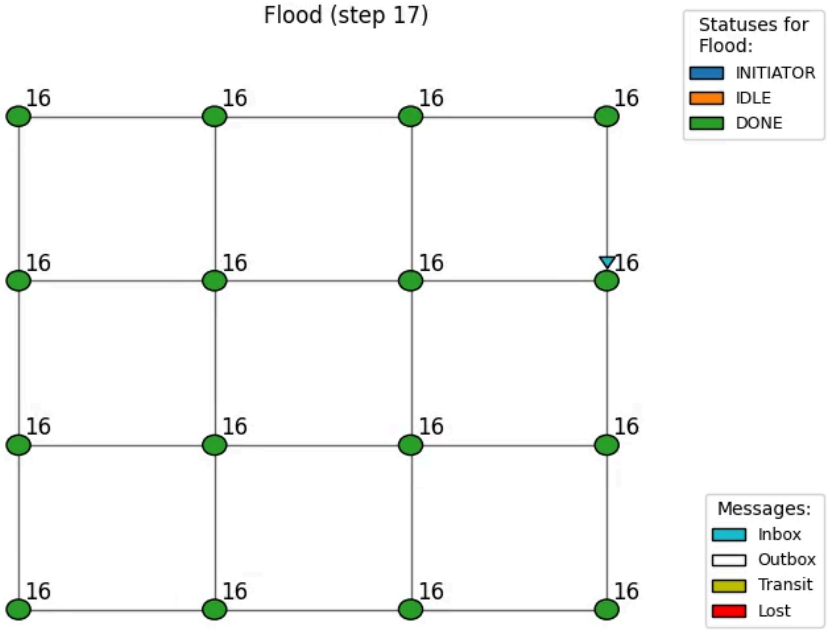
Finally, we will visualize the effect of clock skew on the distributed algorithm. We will start to use the parameter `clock_as_label` to display the internal clock of the nodes in the visualization.

The `clock_increment` parameter of `NetworkBehaviorModel` will be used to simulate the clock skew. It will be added to the internal clock of the nodes in every simulation step. For a fully synchronous network, the clock skew should be a constant value for all nodes.

The next cell does exactly that. We set an increment of 1.

[39]:
```
make_vid(
    NetworkBehaviorModel(clock_increment=lambda node: 1),
    clock_as_label=True,
)
```

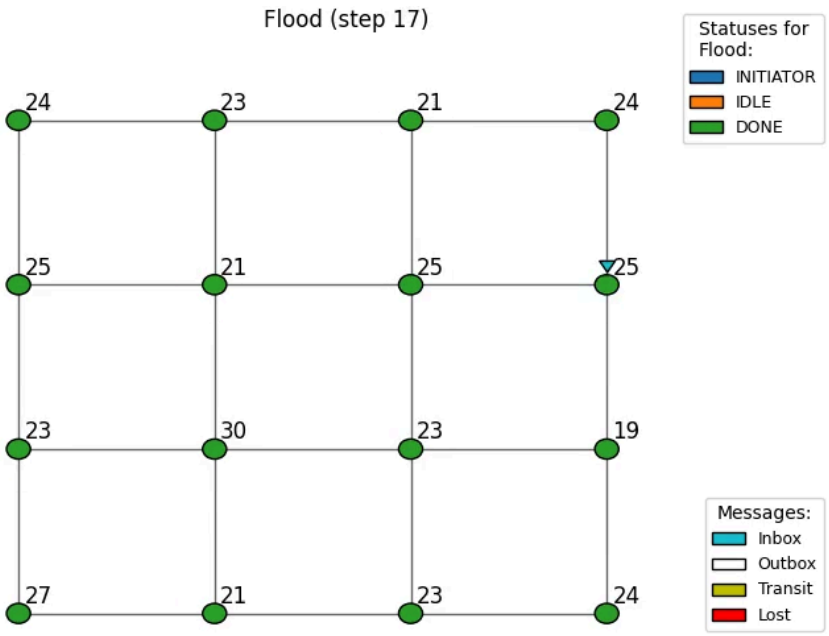
[39]:



The increment now is set to be a random value between 1 and 2. This will make the nodes' internal clocks to drift apart slowly.

[40]:
```
import random


make_vid(
    NetworkBehaviorModel(clock_increment=lambda node: random.randint(1, 2)),
    clock_as_label=True,
)
```

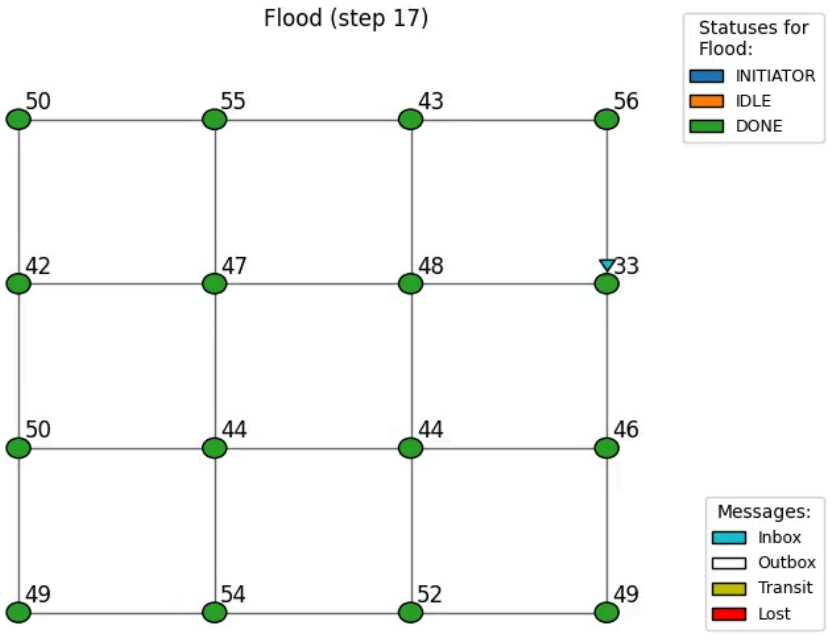
[40]:



To exaggerate the effect of clock skew, we set the increment to be a random value between 1 and 5.

[41]:
```
make_vid(
    NetworkBehaviorModel(clock_increment=lambda node: random.randint(1, 5)),
    clock_as_label=True,
)
```

[41]:

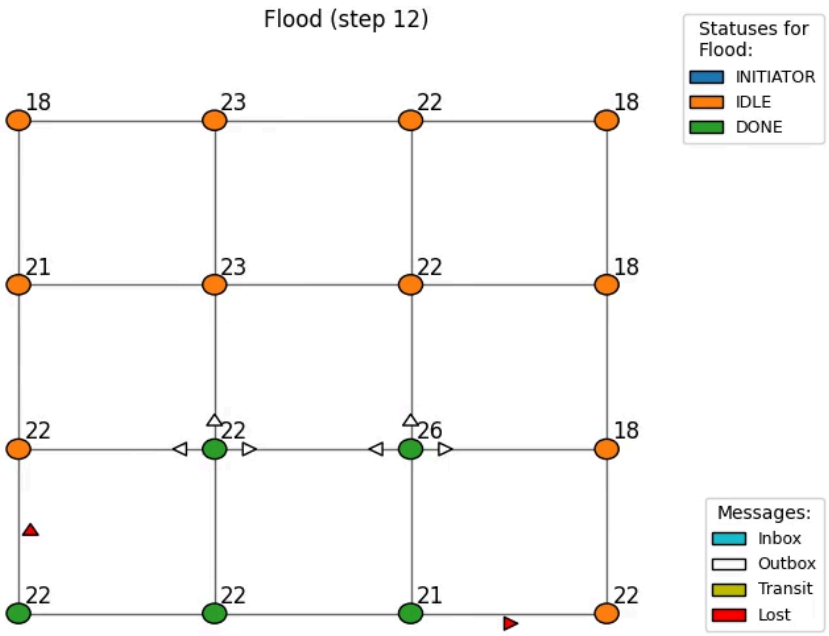


Everything together

Just for fun, let's visualize the effect of all the disruptive network properties together.

[44]:
```
make_vid(
    NetworkBehaviorModel(
        message_loss_indicator=random_loss(0.3),  # 30% packet loss
        message_delay_indicator=lambda network, message: random.randint(1, 5),  # 1 to 5 time
units delay
        clock_increment=lambda node: random.randint(1, 3),  # 1 to 3 time units per clock tick
    ),
    clock_as_label=True,
)
```

[44]:


Flood (step 12)
Statuses for Flood:
INITIATOR
IDLE
DONE
18
23
22
18
21
23
22
18
22
22
26
18
22
22
21
22
Messages:
Inbox
Outbox
Transit
Lost

Tree construction visualization

This notebook shows how to visualize the construction of a tree using what's stored inside the memory of each node. This in-memory information can be obtained by a tree-building algorithm, or by hand.

[2]:
```
# for interactive plots
%matplotlib notebook

from pydistsim.network import NetworkGenerator
```

Manual assignment of tree edges

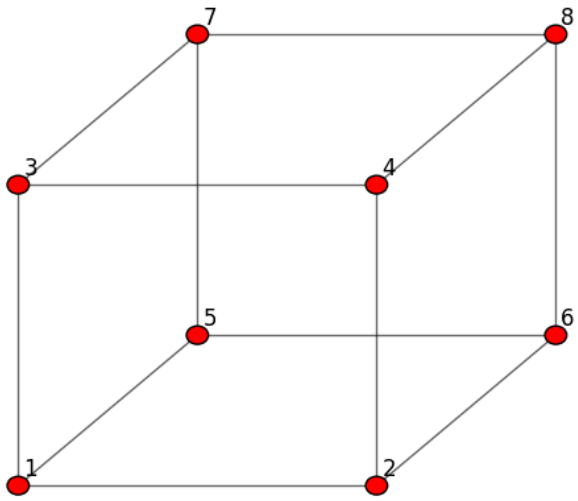
First, let's create a tree by hand, and then visualize it.

We will start with a HyperCube of dimension 3, and then select a few edges to make it a tree.

[3]:
```
net = NetworkGenerator.generate_hypercube_network(dimension=3, use_binary_labels=False)
TREE_KEY = "TREE"
```

[4]:
```
net.show()
```

[4]:



Now we add the tree information to each node. In this cell, we will set the `children` reference for some of the nodes.

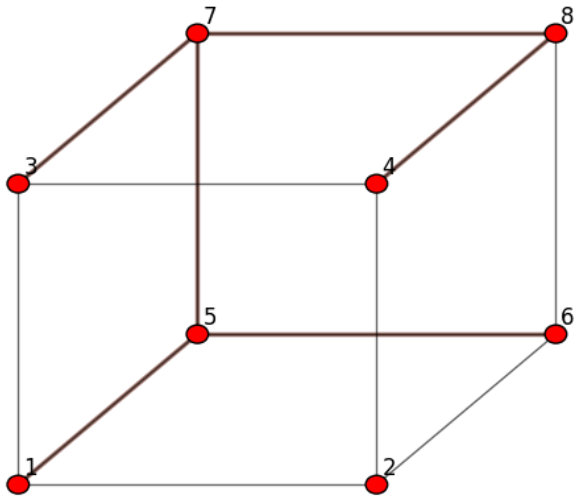
The key idea is that the tree information is distributed among the nodes, and we can reconstruct the tree by following the references. But to find such references, we need to know where in the memory they are stored, this is where the definition of a global `TREE_KEY` is needed.

[5]:
```
node = net.node_by_id(8)
node.memory[TREE_KEY] = {
    "children": [net.node_by_id(7), net.node_by_id(4)],
}

node = net.node_by_id(7)
node.memory[TREE_KEY] = {
    "children": [net.node_by_id(3), net.node_by_id(5)],
}

node = net.node_by_id(5)
node.memory[TREE_KEY] = {
    "children": [net.node_by_id(1), net.node_by_id(6)],
}
```

Now we can visualize the tree by following the references in the memory of each node, but we need the `TREE_KEY` to know where to look for the references.

[6]:
```
net.show(tree_key=TREE_KEY)
```

[6]:

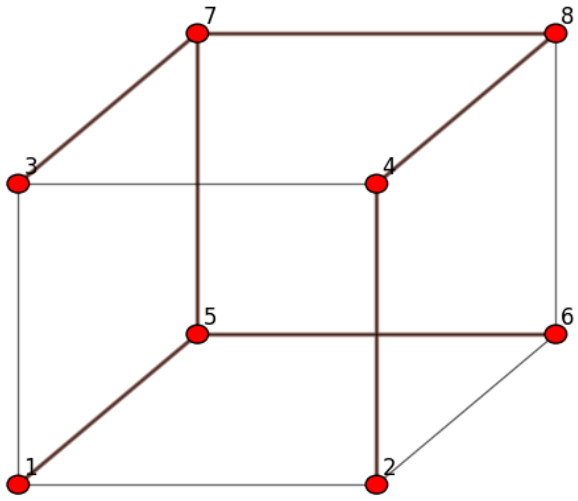


As we can see, node 2 is still missing from the tree. We will add it in the next cell, but using the `parent` reference instead of the `children` reference.

[7]:
```
node = net.node_by_id(2)
node.memory[TREE_KEY] = {
    "parent": net.node_by_id(4),
}


# Alternative tree definition, using children instead of parent:
# node = net.node_by_id(4)
# node.memory[TREE_KEY] = {
#     "children": [net.node_by_id(2)],
# }
```

[8]:
```
net.show(tree_key=TREE_KEY)
```

[8]:

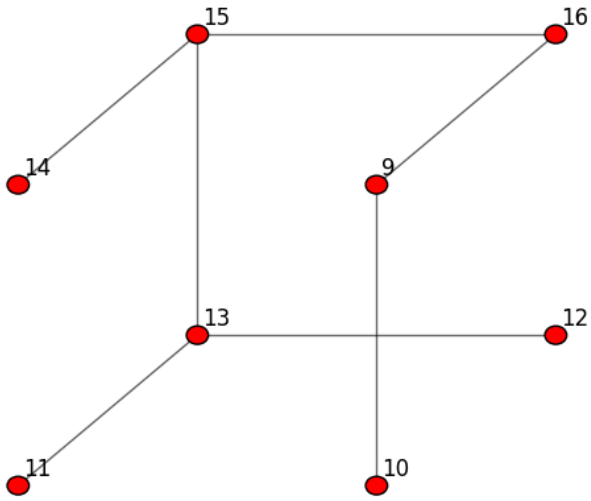


Now the tree is complete.

In the next cell we show how to get a completely new `Network` object from the tree information.

[9]:
```
new_net = net.get_tree_net(TREE_KEY)

new_net.show()
```

[9]:



Tree construction algorithm visualization

Now we show how to visualize the construction of a tree from inside an algorithm. We will use the `FloodTree` algorithm, an extension of the `Flood` algorithm from the `demo_algorithms` module.

Check the module at: https://github.com/agustin-recoba/PyDistSim/tree/main/pydistsim/demo_algorithms

The algorithm is simple: broadcast a message from an initiator and make each node set its parent as the first node that sent the message to it.

[10]:
```
from pydistsim import Simulation
from pydistsim.gui.drawing import create_animation
from pydistsim.network.behavior import NetworkBehaviorModel
from pydistsim.algorithm.node_wrapper import NodeAccess
from pydistsim.message import Message
from pydistsim.demo_algorithms.broadcast import Flood


class FloodTree(Flood):
    """
    Simple extension of the Flood algorithm that uses the first received message to determine the
parent node.
    """

    # It's required to define where the tree information should be stored
    required_params = ["tree_key"]

    def initializer(self):
        # Do the same as Flood, but also initialize the memory for the tree
        super().initializer()

        for node in self.network.nodes():
            node.memory[self.tree_key] = {}

    @Flood.Status.IDLE
    def receiving(self, node: NodeAccess, message: Message):
        # Do the same as Flood, but also store the parent node
        super().receiving_IDLE(node, message)

        node.memory[self.tree_key]["parent"] = message.source
```

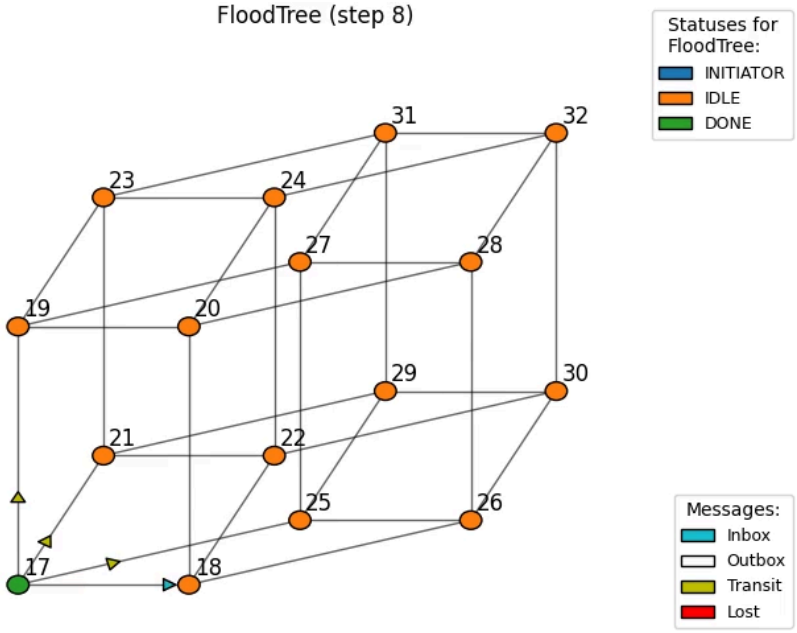
Now, initialize the simulation and create the network.

[11]:
```
net = NetworkGenerator.generate_hypercube_network(16, use_binary_labels=False)


# Let's add some random delays to the communication, so we can see how that affects the tree
building
net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication

# Set the tree_key in which the tree will be stored in the nodes' memory
sim = Simulation(net, ((FloodTree, {"tree_key": TREE_KEY}),))
```

Just as in the animation tutorial, but now we need to pass the key in which the tree information is stored in the nodes.

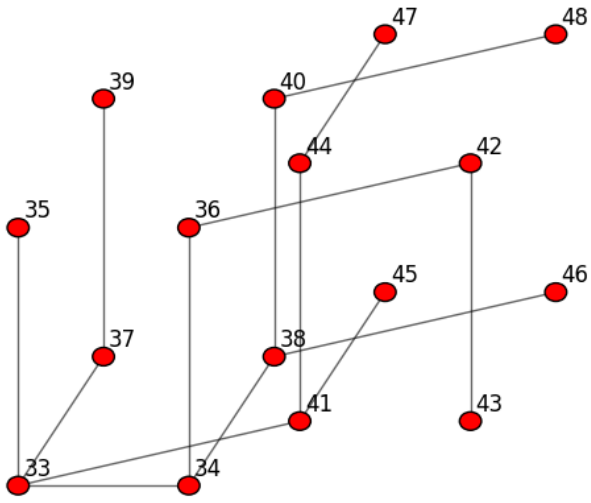
[12]:
```
anim = create_animation(sim, tree_key=TREE_KEY)
video = anim.to_html5_video()

from IPython.display import HTML

HTML(video)
```

[12]:



[13]:
```
final_tree_net = sim.network.get_tree_net(TREE_KEY)
final_tree_net.show()
```

[13]:



Algorithm benchmarking

[2]:
```
from pydistsim.logging import enable_logger
from pydistsim.benchmark import AlgorithmBenchmark
from pydistsim.demo_algorithms.broadcast import Flood
from pydistsim.network.behavior import NetworkBehaviorModel
```

[3]:
```
enable_logger()
```

Set-up the benchmark

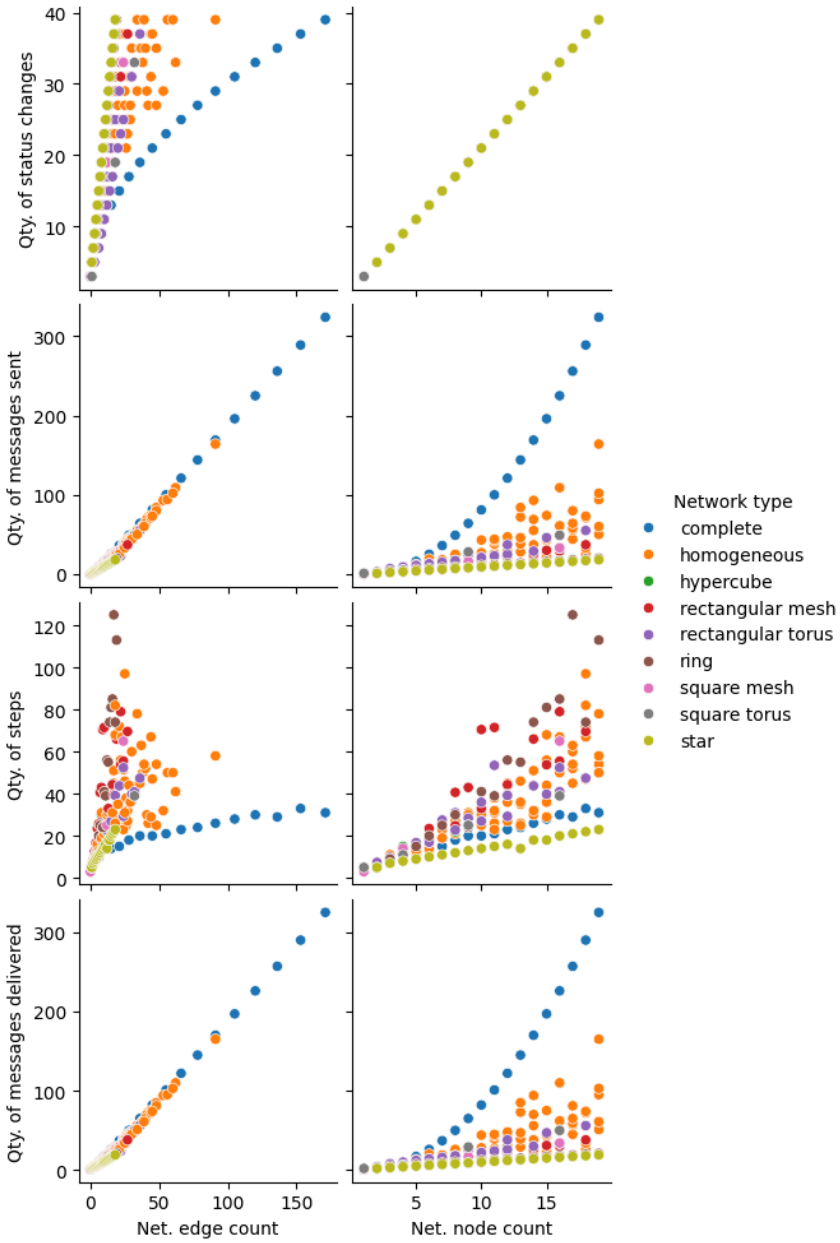
- Pass algorithm, network sizes to test and the network behavior.
- Then run the benchmark.

[4]:
```
benchmark = AlgorithmBenchmark(
    ((Flood, {"initial_information": "Hello Wold!"}),),
    network_sizes=range(1, 20),  # equivalent to [1, 2, 3, ..., 19]
    network_behavior=NetworkBehaviorModel.UnorderedRandomDelayCommunication,  # You can define
their own network behavior, check behavior vizualization notebook
)

benchmark.run()
```

Automatically generated plots with the results

[5]:
```
benchmark.plot_analysis()  # Create a plot
```

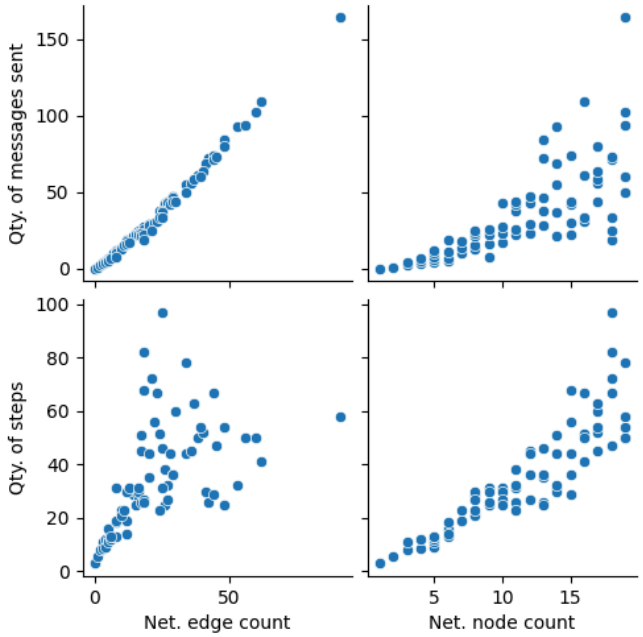
[5]:
```
<seaborn.axisgrid.PairGrid at 0x7fe2df155bd0>
```



Filter the results by network type and only show specific graphs:

[6]:
```
benchmark.plot_analysis(
    y_vars=["Qty. of messages sent", "Qty. of steps"],
    result_filter=lambda result: result["Network type"] == "homogeneous",
)
```

[6]: <seaborn.axisgrid.PairGrid at 0x7fe2df0b13d0>



Get the data as a pandas DataFrame for in-depth analysis

[7]:
```
df = benchmark.get_results_dataframe()  # Get the results as a pandas DataFrame

df  # This will display the results as a table
```

[7]:

	Net. node count	Net. edge count	Network type	Qty. of messages sent	Qty. of messages delivered	Qty. of status changes	Qty. of steps
0	1	0	complete	0	1	3	3
1	1	0	hypercube	0	1	3	3
2	1	0	ring	0	1	3	3
3	1	0	square mesh	0	1	3	3
4	1	1	square torus	1	2	3	5
...	...	...	...	...	...	...	...
349	16	24	rectangular mesh	33	34	33	63
350	18	36	rectangular torus	55	56	37	44
351	19	56	homogeneous	94	95	39	50
352	18	27	rectangular mesh	37	38	37	74
353	15	30	rectangular torus	46	47	31	39

354 rows × 7 columns

Custom observer tutorial

The goal of this tutorial is to setup a custom observer that will be able to monitor certain events of the simulation.

[2]:
```
%matplotlib inline

from pydistsim import NetworkGenerator, Simulation
from pydistsim.logging import set_log_level, LogLevels, enable_logger

set_log_level(LogLevels.INFO)
enable_logger()
```

Algorithm

To demonstrate this, we will re-use the `Echo` algorithm from the "Hello World" example. The algorithm is a simple flooding algorithm that sends a message to all neighbors and waits for the response from all of them. Once every neighbor responds, the algorithms terminates.

The observer will be able to monitor the following events:

- 'echo_response_received': This event is triggered when a response is received from a neighbor.
- 'echo_terminated': This event is triggered when the algorithm terminates.

Implementation of `Echo` algorithm is given below. Notice the `self.notify_observers()` method that is called whenever an event occurs.

[3]:
```python
from enum import StrEnum
from pydistsim.algorithm.node_algorithm import NodeAlgorithm, StatusValues
from pydistsim.algorithm.node_wrapper import NodeAccess
from pydistsim.message import Message
from pydistsim.restrictions.communication import BidirectionalLinks
from pydistsim.restrictions.reliability import TotalReliability
from pydistsim.restrictions.topological import Connectivity, UniqueInitiator


class EchoEvents(StrEnum):
    echo_response_received = "echo_response_received"
    echo_terminated = "echo_terminated"


class Echo(NodeAlgorithm):
    default_params = {
        "echo_message": "Hello world!",
    }

    class Status(StatusValues):
        INITIATOR = "INITIATOR"
        AWAITING_ECHO = "AWAITING_ECHO"
        AWAITING_ECHO_RESPONSE = "AWAITING_ECHO_RESPONSE"
        DONE = "DONE"

    S_init = (Status.INITIATOR, Status.AWAITING_ECHO)
    S_term = (Status.DONE,)

    # Here we define under which restrictions the algorithm is designed to work
    algorithm_restrictions = (
        BidirectionalLinks,  # The algorithm requires bidirectional links, so that the response
can be sent back
        TotalReliability,  # The algorithm requires that all messages are delivered without loss
        Connectivity,  # The algorithm requires that the network is connected
        UniqueInitiator,  # The algorithm requires that only one node is the initiator
    )

    def initializer(self):
        # Set all nodes to AWAITING_ECHO
        for node in self.network.nodes():
            node.status = self.Status.AWAITING_ECHO

        # Choose the initiator
        ini_node = self.network.nodes_sorted()[0]

        # Send the initial message to the initiator
        ini_node.push_to_inbox(Message(meta_header=NodeAlgorithm.INI, destination=ini_node))

        # Set the initiator status to INITIATOR and store the initial information
        ini_node.status = self.Status.INITIATOR
        ini_node.memory["message"] = self.echo_message

    @Status.INITIATOR
    def spontaneously(self, node: NodeAccess, message: Message):
        self.send(
            node,
            data=node.memory["message"],
            destination=list(node.neighbors()),  # send to all neighbors
            header="Echo",
        )
        node.memory["responseCount"] = 0
        node.status = self.Status.AWAITING_ECHO_RESPONSE

    @Status.AWAITING_ECHO_RESPONSE
    def receiving(self, node: NodeAccess, message: Message):
        if message.header == "Echo response":
            self.notify_observers(EchoEvents.echo_response_received, node=node, message=message)

            # Count the number of responses received, only if the response is the same as the
original message
            node.memory["responseCount"] += 1 if message.data == node.memory["message"] else 0

            # If all responses have been received, set the node status to DONE
            if node.memory["responseCount"] == len(list(node.neighbors())):
                self.notify_observers(EchoEvents.echo_terminated, node=node)
                node.status = self.Status.DONE

    @Status.AWAITING_ECHO
    def receiving(self, node: NodeAccess, message: Message):
        if message.header == "Echo":
            node.memory["message"] = message.data

            self.send(
                node,
                data=message.data,
                destination=message.source,
                header="Echo response",
            )
        node.status = self.Status.DONE

    @Status.DONE
    def default(self, *args, **kwargs):
        "Do nothing, for all inputs."
        pass
```

Observer

The observer will be implemented as a class that inherits from both `AlgorithmObserver` and `SimulationObserver` classes.

[4]:
```
from pydistsim.observers import AlgorithmObserver, SimulationObserver
from pydistsim.network import Node


class EchoObserver(AlgorithmObserver, SimulationObserver):
    events = [
        EchoEvents.echo_response_received,
        EchoEvents.echo_terminated,
    ]

    def __init__(self):
        self.list_of_events = []

    def on_echo_response_received(self, node: NodeAccess, message: Message):
        actual_node: Node = node.unbox()
        actual_source: Node = message.source.unbox()
        print(f"Node {actual_node} received echo response from {actual_source}.")
        self.list_of_events.append("received")

    def on_echo_terminated(self, node: NodeAccess):
        actual_node: Node = node.unbox()
        print(f"Node {actual_node} terminated.")
        self.list_of_events.append("terminated")
```

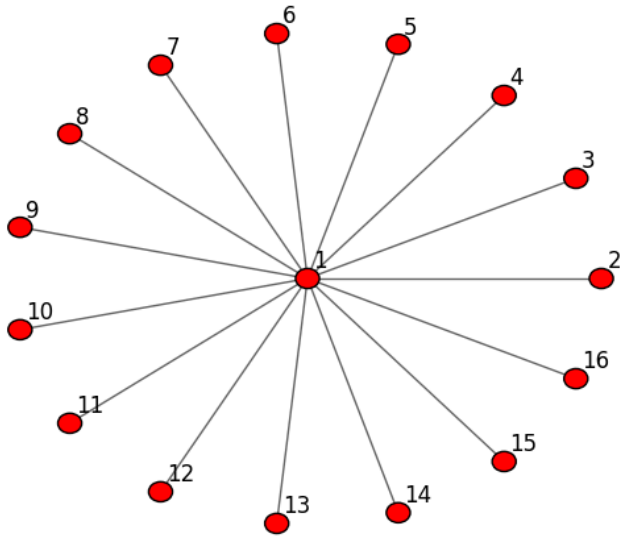
Assigning algorithm and observer to simulation

[5]:
```
MESSAGE = "Hello distributed and observable world"

net = NetworkGenerator.generate_star_network(16)
observer = EchoObserver()
sim = Simulation(net)

sim.algorithms = ((Echo, {"echo_message": MESSAGE}),)
sim.add_observers(observer)
```

```
2024-10-22 23:39:48.901 | INFO     | pydistsim.simulation:__init__:59 - Simulation 0x7f58b45e3190 created successfully.
```

[6]:
```
net.show()
```

[6]:



Running

Finally, run simulation:

[7]:
```
sim.run()
```

```
Node <Node id=1> received echo response from <Node id=2>.
Node <Node id=1> received echo response from <Node id=3>.
Node <Node id=1> received echo response from <Node id=4>.
Node <Node id=1> received echo response from <Node id=5>.
Node <Node id=1> received echo response from <Node id=6>.
Node <Node id=1> received echo response from <Node id=7>.
Node <Node id=1> received echo response from <Node id=8>.
Node <Node id=1> received echo response from <Node id=9>.
Node <Node id=1> received echo response from <Node id=10>.
Node <Node id=1> received echo response from <Node id=11>.
Node <Node id=1> received echo response from <Node id=12>.
Node <Node id=1> received echo response from <Node id=13>.
Node <Node id=1> received echo response from <Node id=14>.
Node <Node id=1> received echo response from <Node id=15>.
Node <Node id=1> received echo response from <Node id=16>.
Node <Node id=1> terminated.
2024-10-22 23:39:52.911 | INFO     | pydistsim.simulation:_run_algorithm:148 - [Echo] Algorithm
finished
```

As we can see, the observer has executed the 'print' message me defied in the different trigger methods.

Now, we check if the observer has kept track of the events.

[8]:
```
observer.list_of_events
```

[8]:
```
['received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'received',
 'terminated']
```

Source code for pydistsim.demo_algorithms.santoro2007.mega_merger.algorithm

```python
import math
import random
from collections.abc import Callable
from dataclasses import dataclass
from enum import StrEnum
from typing import Generic, TypedDict, TypeVar

from pydistsim.algorithm import NodeAlgorithm, StatusValues
from pydistsim.algorithm.node_wrapper import DMANodeAccess, NeighborLabel
from pydistsim.demo_algorithms.santoro2007.mega_merger.labels import (
    COUNTRIES,
    IATA_AIRPORTS,
)
from pydistsim.logging import logger
from pydistsim.message import Message
from pydistsim.restrictions.communication import BidirectionalLinks
from pydistsim.restrictions.reliability import TotalReliability
from pydistsim.restrictions.topological import Connectivity

PARENT = "parent"
```

[docs]

```python
class MMStatus(StatusValues):
    INITIATOR = "INITIATOR"
    SLEEPING = "SLEEPING"
    ASKING_OUTSIDE = "ASKING_OUTSIDE"
    COMPUTING_MIN_WEIGHTS = "COMPUTING_MIN_WEIGHTS"
    WAITING_PARENT = "WAITING_PARENT"
    ELECTED = "ELECTED"
    NOT_ELECTED = "NOT_ELECTED"
```

[docs]

```python
class MMHeaders(StrEnum):
    ARE_YOU_OUTSIDE = "ARE_YOU_OUTSIDE"
    INTERNAL = "INTERNAL"
    EXTERNAL = "EXTERNAL"
    LET_US_MERGE = "LET_US_MERGE"
    MERGE_ME = "MERGE_ME"
    NOTIFICATION = "NOTIFICATION"
    MIN_LINK_WEIGHT = "MIN_LINK_WEIGHT"
    TERMINATION = "TERMINATION"
```

[docs]

```python
@dataclass(slots=True, frozen=True)
class City:
    name: int
    level: int
    is_downtown: bool

    def make_child(self):
        return City(name=self.name, level=self.level, is_downtown=False)

    def __eq__(self, other):
        if self is other:
            return True
        if other is None:
            return False
        if not isinstance(other, City):
            return False

        return self.name == other.name
```

[docs]

```python
class MMNode(DMANodeAccess):
    """
    MMNode class represents a node in the Mega Merger algorithm.

    Attributes:
        id (int): Unique identifier for the node.
        city (City): The city associated with the node.
        weight (dict[NeighborLabel, int]): Weights of the links to neighboring nodes.

        internal_links (set[NeighborLabel]): Set of internal links to neighboring nodes.
        children (set[NeighborLabel]): Set of child nodes.
        tree_key (dict[str, NeighborLabel]): Dictionary mapping tree keys to neighbor labels.
        blocked_messages (dict[tuple[NeighborLabel, MMHeaders], Message]): Messages that are blocked
and waiting to be
            processed.
        external_received (bool): Flag indicating if an external message has been received since the
last reset.
```

```
        weight_received (set[NeighborLabel]): Set of neighbors from which weight has been received
since the last round
            of asking for the minimum weight.
        min_weight (int): Minimum weight among the links received in the last round.
        asking_min_weight (bool): Flag indicating if the node is asking for the minimum weight.
        last_min_weight_sent (int): Last minimum weight sent by the node.
        link_trying_merge (NeighborLabel): Link that is trying to merge.
        link_asking_outside (NeighborLabel): Link that is asking for an outside connection.
        link_with_min_weight (NeighborLabel): Link directing to the minimum weight (might be this
link or another down
            the tree).
    """

    id: int
    city: City
    weight: dict[NeighborLabel, int]

    internal_links: set[NeighborLabel]
    children: set[NeighborLabel]
    tree_key: dict[str, NeighborLabel]
    blocked_messages: dict[tuple[NeighborLabel, MMHeaders], Message]

    external_received: bool
    weight_received: set[NeighborLabel]
    min_weight: int
    asking_min_weight: bool
    last_min_weight_sent: int
    link_trying_merge: NeighborLabel
    link_asking_outside: NeighborLabel
    link_with_min_weight: NeighborLabel



T = TypeVar("T")



                                                                                        [docs]
class MegaMergerParameters(TypedDict, Generic[T]):
    """
    Class to specify the parameters for the Mega Merger algorithm.

    The list of weights and cities must have at least the same size of edges and nodes in the
network, respectively.
    Additionally, both list must be disjoint, i.e., no city can be a weight and vice-versa.
    Obviously, the `INF_WEIGHT` must be bigger than any weight or city and must be comparable in type
to both
    (`>` and `<` must work).

    Attributes:
        percentage_of_initiators (float): Percentage of initiators in the network.
        INF_WEIGHT (T): Value representing infinity.
        WEIGHT_LIST (list[T]): List of weights for the links.
        CITY_LIST (list[T]): List of cities in the network.
    """

    percentage_of_initiators: float
    INF_WEIGHT: T
    WEIGHT_LIST: list[T] | Callable[[int], T]
    CITY_LIST: list[T] | Callable[[int], T]



                                                                                        [docs]
class ExampleParameters:
    countries_parameters: MegaMergerParameters = {
        "percentage_of_initiators": 0.4,
        "WEIGHT_LIST": IATA_AIRPORTS,  # 3-letter airport code
        "CITY_LIST": COUNTRIES,  # 2-letter country code
        "INF_WEIGHT": "ZZZZ",  # comparable to both WEIGHT_LIST and CITY_LIST
    }

    numerical_parameters: MegaMergerParameters = {
        "percentage_of_initiators": 0.5,
        "CITY_LIST": lambda i: i + 1,  # positives
        "WEIGHT_LIST": lambda i: -i - 1,  # negatives
        "INF_WEIGHT": 9999999,  # bigger than any weight or city
    }



                                                                                        [docs]
class MegaMergerAlgorithm(NodeAlgorithm):
    """
    Mega-Merger
    ===========

    This algorithm is a distributed algorithm that elects a single node in a network.
    The way it works is by creating cities (a grouping of nodes) and merging them together, while
    maintaining a tree structure with a root (the downtown node) that makes the decisions.
```

```
    Ultimately, when the cities are merged, the downtown node is elected as the root of the tree.

    The algorithm is based on the descriptions in the book `Design and Analysis of Distributed
    Algorithms <http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471719978,descCd-
description.html>`_
    by Nicola Santoro.

    Parameters
    ----------

    percentage_of_initiators : float
        The percentage of initiators in the network.

    INF_WEIGHT : T
        Value representing infinity.

    WEIGHT_LIST : list[T] | Callable[[int], T]
        Mapping of weights for the links.

    CITY_LIST : list[T] | Callable[[int], T]
        Mapping of cities in the network.
    """

    default_params: MegaMergerParameters = ExampleParameters.countries_parameters

    Status = MMStatus

    S_init = (MMStatus.INITIATOR, MMStatus.SLEEPING)
    S_term = (MMStatus.ELECTED, MMStatus.NOT_ELECTED)

    algorithm_restrictions = (
        BidirectionalLinks,
        TotalReliability,
        Connectivity,
    )

    def _create_wrapper_manager(self):
        return self.NODE_WRAPPER_MANAGER_TYPE(self.network, MMNode)


                                                                                                [docs]
    def initializer(self):
        # Initialize the nodes and the weights dictionary
        nodes: list[MMNode] = list(self.nwm.nodes())
        random.shuffle(nodes)
        for i, node in enumerate(nodes):
            node.weight = {}
            node.id = self.CITY_LIST(i) if callable(self.CITY_LIST) else self.CITY_LIST[i]
            node.status = MMStatus.SLEEPING

        # Initialize the weights
        edges = list(self.nwm.edges())
        random.shuffle(edges)
        for i, ((u_own_view, us_view_of_v), (v_own_view, vs_view_of_u)) in enumerate(edges):
            u_own_view.weight[us_view_of_v] = v_own_view.weight[vs_view_of_u] = (
                self.WEIGHT_LIST(i) if callable(self.WEIGHT_LIST) else self.WEIGHT_LIST[i]
            )

        # Choose the initiators
        quantity_of_initiators = math.floor(self.percentage_of_initiators * len(self.network) + 1)
        for node in set(
            random.choices(
                tuple(self.network.nodes()),
                k=quantity_of_initiators,
            )
        ):
            node.status = MMStatus.INITIATOR
            node.push_to_inbox(Message(meta_header=NodeAlgorithm.INI, destination=node))



    @MMStatus.INITIATOR
    def default(self, node: MMNode, message: Message):
        self.on_awakening(node, message if message.meta_header != NodeAlgorithm.INI else None)

    @MMStatus.SLEEPING
    def receiving(self, node: MMNode, message: Message):  # noqa: F811
        self.on_awakening(node, message)

    def on_awakening(self, node: MMNode, message: Message = None):
        node.status = MMStatus.COMPUTING_MIN_WEIGHTS

        node.city = City(name=node.id, level=1, is_downtown=True)
        node.internal_links = set()
        node.blocked_messages = {}
        node.tree_key = {PARENT: None}
        node.children = set()
        node.external_received = False
        node.weight_received = set()
        node.link_trying_merge = None
        node.link_asking_outside = None
        node.link_with_min_weight = None
        node.min_weight = self.INF_WEIGHT
```

```
        if self.get_external_links(node):
            node.status = MMStatus.COMPUTING_MIN_WEIGHTS
            node.asking_min_weight = True
            self.fire_next_are_you_outside(node)

            if message:
                self.receiving_COMPUTING_MIN_WEIGHTS(
                    node, message
                )  # undirected call to `receiving` when status is COMPUTING_MIN_WEIGHTS

        else:
            self.when_termination(node)

    @MMStatus.ASKING_OUTSIDE
    def receiving(self, node: MMNode, message: Message):  # noqa: F811
        self.log_received(node, message)

        match message.header:
            case MMHeaders.ARE_YOU_OUTSIDE:
                self.on_are_you_outside(node, message)
            case MMHeaders.EXTERNAL:
                self.on_external(node, message)
            case MMHeaders.INTERNAL:
                self.on_internal(node, message)
            case MMHeaders.LET_US_MERGE:
                self.on_let_us_merge(node, message)
            case MMHeaders.MIN_LINK_WEIGHT:
                self.on_min_link_weight(node, message)
            case MMHeaders.NOTIFICATION:
                self.on_notification(node, message)
            case _:
                logger.debug(f"ASKING_OUTSIDE: {message.header}")

    @MMStatus.COMPUTING_MIN_WEIGHTS
    def receiving(self, node: MMNode, message: Message):  # noqa: F811
        self.log_received(node, message)

        match message.header:
            case MMHeaders.ARE_YOU_OUTSIDE:
                self.on_are_you_outside(node, message)
            case MMHeaders.LET_US_MERGE:
                self.on_let_us_merge(node, message)
            case MMHeaders.MERGE_ME:
                self.on_merge_me(node, message)
            case MMHeaders.NOTIFICATION:
                self.on_notification(node, message)
            case MMHeaders.MIN_LINK_WEIGHT:
                if node.asking_min_weight:
                    self.on_min_link_weight(node, message)
                else:
                    self.log_ignore(node, message)
            case MMHeaders.TERMINATION:
                self.on_termination(node, message)
            case _:
                logger.debug(f"COMPUTING_MIN_WEIGHTS: {message.header}")

    @MMStatus.NOT_ELECTED
    def receiving(self, node: MMNode, message: Message):  # noqa: F811
        self.log_received(node, message)

        match message.header:
            case MMHeaders.NOTIFICATION:
                self.on_notification(node, message)  # a terminated node may still change its city
            case _:
                logger.debug(f"NOT_ELECTED: {message.header}")

    @MMStatus.WAITING_PARENT
    def receiving(self, node: MMNode, message: Message):  # noqa: F811
        self.log_received(node, message)

        match message.header:
            case MMHeaders.ARE_YOU_OUTSIDE:
                self.on_are_you_outside(node, message)
            case MMHeaders.LET_US_MERGE:
                self.on_let_us_merge(node, message)
            case MMHeaders.MERGE_ME:
                self.on_merge_me(node, message)
            case MMHeaders.NOTIFICATION:
                self.on_notification(node, message)
            case MMHeaders.TERMINATION:
                self.on_termination(node, message)
            case _:
                logger.debug(f"WAITING_PARENT: {message.header}")

    def log_received(self, node: MMNode, message: Message):
        logger.info(f"Receiving\t{message.header} ({message.source.unbox()} --> {node.unbox()}, data=
{message.data})")

    def log_ignore(self, node: MMNode, message: Message):
        logger.info(f"Ignoring\t{message.header} ({message.source.unbox()} --> {node.unbox()}, data=
{message.data})")


                                                                                                [docs]
    def on_are_you_outside(self, node: MMNode, message: Message):
```

```
        """
        Handles the 'are you outside' message received by a node.

        This method processes the message to determine if the node is part of the same city as the
sender
        or if it is outside. Depending on the city levels and the source of the message, it will
either
        handle the message internally, send an internal or external response, or block the message.
        """
        other_city: City = message.data
        my_city: City = node.city

        if my_city == other_city:
            if node.link_asking_outside == message.source:
                self.on_internal(node, message)
            else:
                self.send(node_source=node, data=None, destination=message.source,
header=MMHeaders.INTERNAL)
                self.add_internal_link(node, message.source)
        elif my_city.level >= other_city.level:
            self.send(node_source=node, data=None, destination=message.source,
header=MMHeaders.EXTERNAL)
        else:
            self.block_msg(node, message)
```

[docs]

```
    def on_external(self, node: MMNode, message: Message):
        """
        Handles the reception of an external message by a node.

        This method is called when a node receives a message confirming the link leads to an external
node.
        It updates the node's state and processes the message according to the
        Mega Merger algorithm.

        Behavior:
            - Sets the node's `external_received` flag to True.
            - If the message source is not the expected link, logs a debug message and ignores the
message.
            - Updates the node's status to `COMPUTING_MIN_WEIGHTS` (prev. was `ASKING_OUTSIDE`).
            - Updates the link with the minimum weight.
            - If the node is done asking outside:
                - If the node is the downtown, sends a "let us merge" message.
                - If the node is not the root, sends the minimum link weight to the parent node and
updates the
                node's status.
            - If the minimum weight is infinite, triggers partial termination.
        """
        node.external_received = True

        if node.link_asking_outside != message.source:
            logger.debug(f"External message from {message.source} not expected")
            self.log_ignore(node, message)
            return

        node.link_asking_outside = None
        node.status = MMStatus.COMPUTING_MIN_WEIGHTS

        self.update_link_with_min_weight(node, message.source, node.weight[message.source])

        if self.is_done_asking_outside(node):  # DONE ASKING OUTSIDE
            if not node.tree_key[PARENT]:  # ROOT
                self.send_let_us_merge(node, node.city.make_child(), node.min_weight)

            else:  # NOT ROOT
                self.send(
                    node_source=node,
                    data=node.min_weight,
                    destination=node.tree_key[PARENT],
                    header=MMHeaders.MIN_LINK_WEIGHT,
                )
                node.asking_min_weight = False
                node.status = MMStatus.WAITING_PARENT
                node.last_min_weight_sent = node.min_weight
                if node.min_weight == self.INF_WEIGHT:
                    self.partial_termination(node)
```

[docs]

```
    def partial_termination(self, node):
        """
        Handles the partial termination of a node in the Mega Merger algorithm.

        This method sets the status of the given node to NOT_ELECTED and sends a
        termination message to all its children nodes.

        Partial termination means that the node is not elected as the downtown node and neither
        it nor its children will make any further progress in the algorithm.
        """
        node.status = MMStatus.NOT_ELECTED
```

```
        self.send(node_source=node, data=None, destination=node.children,
header=MMHeaders.TERMINATION)
```

[docs]

```
    def on_internal(self, node: MMNode, message: Message):
        """
        Handles the internal message received by a node.

        This method processes an internal message received by a node and updates the node's state
accordingly.
        It performs the following actions:
        - Sets as internal the link between the node and the message source.
        - Checks if the message source is the expected link asking outside.
            - If not, logs a debug message and ignores the message.
        - Updates the node's status to COMPUTING_MIN_WEIGHTS if the message source is the expected.
        - If the node is done asking outside:
            - If the node is the root, triggers a new merge or the global termination.
            - If the node is not the root, sends the minimum link weight to the parent node.
        """

        self.add_internal_link(node, message.source)

        if node.link_asking_outside != message.source:
            logger.debug(f"Internal message from {message.source} not expected")
            self.log_ignore(node, message)
            return
        node.link_asking_outside = None
        node.status = MMStatus.COMPUTING_MIN_WEIGHTS

        if self.is_done_asking_outside(node):
            if not node.tree_key[PARENT]:  # ROOT
                if self.is_termination(node):
                    self.when_termination(node)
                else:
                    self.send_let_us_merge(node, node.city.make_child(), node.min_weight)

            else:  # NOT ROOT
                w = node.min_weight if node.min_weight else self.INF_WEIGHT
                self.send(
                    node_source=node,
                    data=w,
                    destination=node.tree_key[PARENT],
                    header=MMHeaders.MIN_LINK_WEIGHT,
                )
                node.status = MMStatus.WAITING_PARENT
                node.asking_min_weight = False
                node.last_min_weight_sent = w
                if w == self.INF_WEIGHT:
                    node.status = MMStatus.NOT_ELECTED

        elif len(self.get_external_links(node)) > 0:
            node.asking_min_weight = True
            self.fire_next_are_you_outside(node)
```

[docs]

```
    def on_let_us_merge(self, node: MMNode, message: Message):
        """
        Handles the LET_US_MERGE message received by a node.

        This method processes the LET_US_MERGE message and determines the appropriate action based on
the
        relationship between the sender and the receiver nodes, as well as their respective cities
and levels.

        Actions:
        - If the sender's city is the same as the receiver's city, sends the LET_US_MERGE message
down the tree.
        - If the merge is considered friendly, it triggers a friendly merge.
        - If the receiver's city level is higher than the sender's, it triggers a forced merge.
        - If the receiver's city level is the same as the sender's, it blocks the message.
        - Otherwise, it ignores the message.
        """
        logger.info(f"LET_US_MERGE: {message.source.unbox()} -> {node.unbox()}")

        sender_city, other_id, road_weight = message.data
        my_city = node.city

        if my_city == sender_city:
            if node.link_with_min_weight and node.min_weight != self.INF_WEIGHT:
                self.send_let_us_merge(node, sender_city, road_weight)

        elif self.is_friendly_merge(node, message) and message.source in
self.get_external_links(node):
            self.when_friendly_merge(node, message)
        elif my_city.level > sender_city.level and message.source in self.get_external_links(node):
            self.send(
                node_source=node,
                data=(my_city.make_child(), not self.is_done_asking_outside(node)),
                destination=message.source,
```

```
                header=MMHeaders.MERGE_ME,
            )
            self.add_child(node, message.source)

            self.add_internal_link(node, message.source)
        elif my_city.level == sender_city.level and message.source in self.get_external_links(node):
            self.block_msg(node, message)
        else:
            self.log_ignore(node, message)



    def add_child(self, node, child):
        node.children.add(child)


                                                                                                    [docs]
    def send_let_us_merge(self, node, new_city, road_weight):
        """
        Sends a 'LET_US_MERGE' message from the given node to its link with the minimum weight.

        This method handles both border and internal nodes. For border nodes, it checks if the
        minimum weight of the node matches the provided road weight before sending the message.
        If the weights do not match, an error is logged and the method returns without sending
        the message. For internal nodes, it sends the message directly.
        """
        # BORDER NODE
        if node.link_with_min_weight in self.get_external_links(node):
            if node.min_weight != road_weight:
                logger.error(f"MIN_WEIGHT != road_weight: {node.min_weight} != {road_weight}")
                return

            # INTERNAL NODE
            self.send(
                node_source=node,
                data=(new_city, node.id, node.min_weight),
                destination=node.link_with_min_weight,
                header=MMHeaders.LET_US_MERGE,
            )

            node.link_trying_merge = node.link_with_min_weight

            def let_us_merge_processor(blocked_message: Message, deleter: Callable[[], None]):
                if blocked_message.source in self.get_internal_links(node):  # Already merged
                    deleter()

                elif self.is_friendly_merge(node, blocked_message):
                    deleter()
                    self.when_friendly_merge(node, blocked_message)
                    return False

                return True

            self.unblock_by_header(node, MMHeaders.LET_US_MERGE, let_us_merge_processor)
        else:
            # INTERNAL NODE
            self.send(
                node_source=node,
                data=(new_city, node.id, node.min_weight),
                destination=node.link_with_min_weight,
                header=MMHeaders.LET_US_MERGE,
            )



                                                                                                    [docs]
    def on_merge_me(self, node: MMNode, message: Message):
        """
        Handles the MERGE_ME message received by a node.

        This method processes the MERGE_ME message, which indicates a forced merge between the sender
and the receiver.
        It triggers the merge and updates the node's state accordingly.
        """

        new_city, to_ask_min_weight = message.data
        new_city: City

        if new_city == node.city:
            self.log_ignore(node, message)
            return

        self.set_city(node, new_city)
        self.add_internal_link(node, message.source)

        if node.tree_key[PARENT]:
            node.children.add(node.tree_key[PARENT])
        self.set_parent(node, message.source)

        self.send(
            node_source=node,
            data=(new_city.make_child(), to_ask_min_weight),
            destination=node.children,
            header=MMHeaders.NOTIFICATION,
```

```
        )

        self.when_city_change(node, new_city, to_ask_min_weight)



[docs]
    def on_notification(self, node: MMNode, message: Message):
        """
        Handles the notification message received by a node.

        This method processes a notification message, updates the node's city,
        and manages the parent-child relationships within the node's tree structure.
        It also sends the notification to the node's children and triggers any
        necessary actions when the city changes.
        """
        new_city, to_ask_min_weight = message.data
        new_city: City

        if new_city == node.city:
            self.log_ignore(node, message)
            return

        self.set_city(node, new_city)

        if message.source in node.children:

            if node.tree_key[PARENT]:
                node.children.add(node.tree_key[PARENT])

            self.set_parent(node, message.source)
            node.children.remove(message.source)

        self.send(
            node_source=node,
            data=(new_city, to_ask_min_weight),
            destination=node.children,
            header=MMHeaders.NOTIFICATION,
        )

        self.when_city_change(node, new_city, to_ask_min_weight)



[docs]
    def when_city_change(self, node, new_city, to_ask_min_weight):
        """
        Handles the event when a node's city changes.

        This method resets the node's metadata and processes blocked messages
        related to merging and external status checks. Depending on the new city's
        level and the message's city level, it either merges the node with the
        source of the message or marks the source as internal or external.

        Internal Processors:
        - let_us_merge_processor: Processes messages with the header LET_US_MERGE.
        - are_you_outside_processor: Processes messages with the header ARE_YOU_OUTSIDE.

        If `to_ask_min_weight` is True, the node's status is set to COMPUTING_MIN_WEIGHTS,
        and it initiates the process to ask for the minimum weight from external links.
        """
        self.reset_meta(node)

        def let_us_merge_processor(blocked_message: Message, deleter: Callable[[], None]):
            msg_city, other_id, road_weight = blocked_message.data

            if blocked_message.source in self.get_internal_links(node):  # Already merged
                deleter()

            elif new_city.level > msg_city.level and new_city != msg_city:
                deleter()

                self.send(
                    node_source=node,
                    data=(new_city, to_ask_min_weight),
                    destination=blocked_message.source,
                    header=MMHeaders.MERGE_ME,
                )
                self.add_child(node, blocked_message.source)
                self.add_internal_link(node, blocked_message.source)

            return True

        self.unblock_by_header(node, MMHeaders.LET_US_MERGE, let_us_merge_processor)

        def are_you_outside_processor(blocked_message: Message, deleter: Callable[[], None]):
            msg_city = blocked_message.data

            if new_city == msg_city:
                deleter()

                self.send(node_source=node, data=None, destination=blocked_message.source,
header=MMHeaders.INTERNAL)
```

```
                self.add_internal_link(node, blocked_message.source)
            elif new_city.level >= msg_city.level:
                deleter()

                self.send(node_source=node, data=None, destination=blocked_message.source,
header=MMHeaders.EXTERNAL)

            return True

        self.unblock_by_header(node, MMHeaders.ARE_YOU_OUTSIDE, are_you_outside_processor)

        if to_ask_min_weight:
            node.status = MMStatus.COMPUTING_MIN_WEIGHTS
            node.asking_min_weight = True
            if len(self.get_external_links(node)) > 0:
                self.fire_next_are_you_outside(node)
            else:
                if node.tree_key[PARENT] and self.is_done_asking_outside(node):
                    self.send_INF_WEIGHT(node)
                    node.status = MMStatus.COMPUTING_MIN_WEIGHTS
                    node.asking_min_weight = False
                    node.last_min_weight_link = self.INF_WEIGHT
                    node.status = MMStatus.NOT_ELECTED
```

[docs]

```
    def on_min_link_weight(self, node: MMNode, message: Message):
        """
        Handles the event when a minimum link weight message is received by a node.

        This method processes the received message, updates the node's state, and
        determines the next steps based on the node's role in the tree (root or non-root).
        """

        if message.source not in node.children:
            self.log_ignore(node, message)
            return

        node.weight_received.add(message.source)

        self.update_link_with_min_weight(node, message.source, message.data)

        if not self.is_done_asking_outside(node):
            return

        # DONE ASKING OUTSIDE
        if not node.tree_key[PARENT]:  # IM ROOT
            if self.is_termination(node):
                self.when_termination(node)
            else:
                logger.info(f"DOWNTOWN LET_US_MERGE {node.unbox()}")
                self.send_let_us_merge(node, node.city.make_child(), node.min_weight)
        else:
            self.send(
                node_source=node,
                data=node.min_weight,
                destination=node.tree_key[PARENT],
                header=MMHeaders.MIN_LINK_WEIGHT,
            )
            node.status = MMStatus.WAITING_PARENT
            node.asking_min_weight = False
            node.last_min_weight_sent = node.min_weight
            if not node.min_weight or node.min_weight == self.INF_WEIGHT:
                node.status = MMStatus.NOT_ELECTED



    def on_termination(self, node: MMNode, message: Message):
        self.when_termination(node)

    def when_termination(self, node: MMNode):
        if node.tree_key[PARENT] is None:
            node.status = MMStatus.ELECTED
            self.set_city(node, City(name=node.city.name, level=-1, is_downtown=True))
        else:
            node.status = MMStatus.NOT_ELECTED
            self.set_city(node, City(name=node.city.name, level=-1, is_downtown=False))

        # Unnecessary, as the children will already be partially terminated
        # self.send(node_source=node, data=None, destination=node.children,
header=MMHeaders.TERMINATION)

    def is_termination(self, node: MMNode):
        if node.tree_key[PARENT]:
            raise ValueError("This node is not the root of the tree")

        return self.is_done_asking_outside(node) and (node.min_weight is None or node.min_weight ==
self.INF_WEIGHT)

    def is_done_asking_outside(self, node: MMNode):
        return (node.external_received or len(self.get_external_links(node)) == 0) and (
            node.children.issubset(node.weight_received)
        )
```

```
    def update_link_with_min_weight(self, node: MMNode, neighbor: NeighborLabel, weight: int):
        if (node.link_with_min_weight is None) or weight < node.min_weight:
            node.link_with_min_weight = neighbor
            node.min_weight = weight
            return True

        return False

    def block_msg(self, node: MMNode, message: Message):
        node.blocked_messages[(message.source, message.header)] = message

    def unblock_by_header(
        self, node: MMNode, header: MMHeaders, unblocked_processor: Callable[[Message, Callable[[],
None]], bool]
    ):
        for key in tuple(filter(lambda s_h: s_h[1] == header, node.blocked_messages)):
            message = node.blocked_messages[key]

            if not unblocked_processor(
                blocked_message=message,
                deleter=lambda: node.blocked_messages.pop(key),
            ):
                break

    def is_friendly_merge(self, node: MMNode, message: Message):
        sender_city, other_id, road_weight = message.data
        sender_level = sender_city.level
        my_level = node.city.level

        return my_level == sender_level and message.source == node.link_trying_merge

    def when_friendly_merge(self, node: MMNode, message: Message):
        new_level = node.city.level + 1  # LEVEL INCREASED
        new_city_id = node.weight[message.source]
        other_city, other_id, road_weight = message.data

        if node.id < other_id:
            new_city = City(new_city_id, new_level, True)
            if node.tree_key[PARENT]:
                node.children.add(node.tree_key[PARENT])
                self.set_parent(node, None)
            node.children.add(message.source)
        else:
            new_city = City(new_city_id, new_level, False)
            if node.tree_key[PARENT]:
                node.children.add(node.tree_key[PARENT])
            self.set_parent(node, message.source)

        self.set_city(node, new_city)

        self.send(
            node_source=node,
            data=(new_city.make_child(), True),
            destination=node.children,  # check self.get_internal_links(node),
            header=MMHeaders.NOTIFICATION,
        )

        self.add_internal_link(node, message.source)

        self.when_city_change(node, new_city, True)


                                                                                    [docs]
    def fire_next_are_you_outside(self, node: MMNode):
        "Assumes there are external links"
        external_links = self.get_external_links(node)
        min_link, min_weight = min(((n, node.weight[n]) for n in external_links), key=lambda x: x[1])

        if node.link_asking_outside != min_link:
            self.send(
                node_source=node,
                data=node.city.make_child(),
                destination=min_link,
                header=MMHeaders.ARE_YOU_OUTSIDE,
            )
            node.link_asking_outside = min_link
            node.status = MMStatus.ASKING_OUTSIDE



    def send_INF_WEIGHT(self, node: MMNode):
        self.send(
            node_source=node,
            data=self.INF_WEIGHT,
            destination=node.tree_key[PARENT],
            header=MMHeaders.MIN_LINK_WEIGHT,
        )

    def reset_meta(self, node: MMNode):
        logger.info(f"Reset meta: {node.unbox()}")

        node.weight_received = set()
        node.link_with_min_weight = None
        node.link_trying_merge = None
```

```
        node.external_received = False
        node.min_weight = self.INF_WEIGHT

    def get_external_links(self, node: MMNode):
        return set(node.neighbors()) - node.internal_links

    def get_internal_links(self, node: MMNode):
        return node.internal_links

    def add_internal_link(self, node: MMNode, neighbor: NeighborLabel):
        node.internal_links.add(neighbor)

    def set_city(self, node: MMNode, city: City):
        logger.info(f"NEW CITY {node.unbox()}: {city}")

        node.city = city

    def set_parent(self, node: MMNode, parent: NeighborLabel):
        logger.info(
            f"NEW PARENT {node.unbox()}: new {parent.unbox() if parent else None}, previous:
{node.tree_key[PARENT].unbox() if node.tree_key[PARENT] else None}"
        )
        node.tree_key[PARENT] = parent
```

Visualizing Mega-Merger: a generalist protocol for distributed election

The implementation of Mega-Merger helps us visualize its behavior with 3 important parameters:

1. `WEIGHT_LIST` (`list[T]`): List of weights for the links.
2. `CITY_LIST` (`list[T]`): List of cities in the network.
3. `INF_WEIGHT` (`T`): Value representing infinity.

The specification of those parameters is as follows: The list of weights and cities must have at least the same size of edges and nodes in the network, respectively. Additionally, both list must be disjoint, i.e., no city can be a weight and vice-versa. Obviously, the `INF_WEIGHT` must be bigger than any weight or city and must be comparable in type to both (`>` and `<` must work).

This leaves us with one important question: how do we leverage this to visualize the behavior of Mega-Merger? The main idea is this: populate the lists with values that are somewhat meaningful to the analogy used to define Mega-Merger: cities and roads. Following this idea, we will use the following values:

1. `WEIGHT_LIST` : 3 letter strings corresponding to the IATA standard for airports names.
2. `CITY_LIST` : 2 letter strings corresponding to the ISO 3166-1 alpha-2 standard for countries names.
3. `INF_WEIGHT` : “ZZZZ”.

[2]:
```
from pydistsim.demo_algorithms.santoro2007.mega_merger.algorithm import MegaMergerAlgorithm,
ExampleParameters
from pydistsim.demo_algorithms.santoro2007.mega_merger.labels import COUNTRIES, IATA_AIRPORTS
from pydistsim.network import NetworkGenerator
from pydistsim.network.behavior import NetworkBehaviorModel
from pydistsim.simulation import Simulation
from pydistsim.gui import drawing as draw
```

Matplotlib configuration

[ ]:
```
import matplotlib.pyplot as plt


# This two lines are only necessary for the animation Js widget
%matplotlib notebook
plt.rcParams["animation.html"] = (
    "jshtml"  # so that the animation displays with the call of create_animation and no assignment
)

# This limits the size of the animation frames embedded in the notebook
plt.rcParams["animation.embed_limit"] = 1 * 1024  # max size of the animation in MB
```

Colormapping the nodes and weights

For a better visualization, we will need to be able to map the nodes to colors. The next lines will map the city names.

[3]:
```
import matplotlib.colors as colors

COLORS = list(colors.XKCD_COLORS.values()) * 50
NAMES = COUNTRIES + IATA_AIRPORTS

color_map = dict(zip(NAMES, COLORS))
```

Creating the animations

First, we create the network and the simulation object.

[4]:
```
net = NetworkGenerator.generate_hypercube_network(32)
net.behavioral_properties = NetworkBehaviorModel.IdealCommunication
sim = Simulation(net, ((MegaMergerAlgorithm, ExampleParameters.countries_parameters),))
```

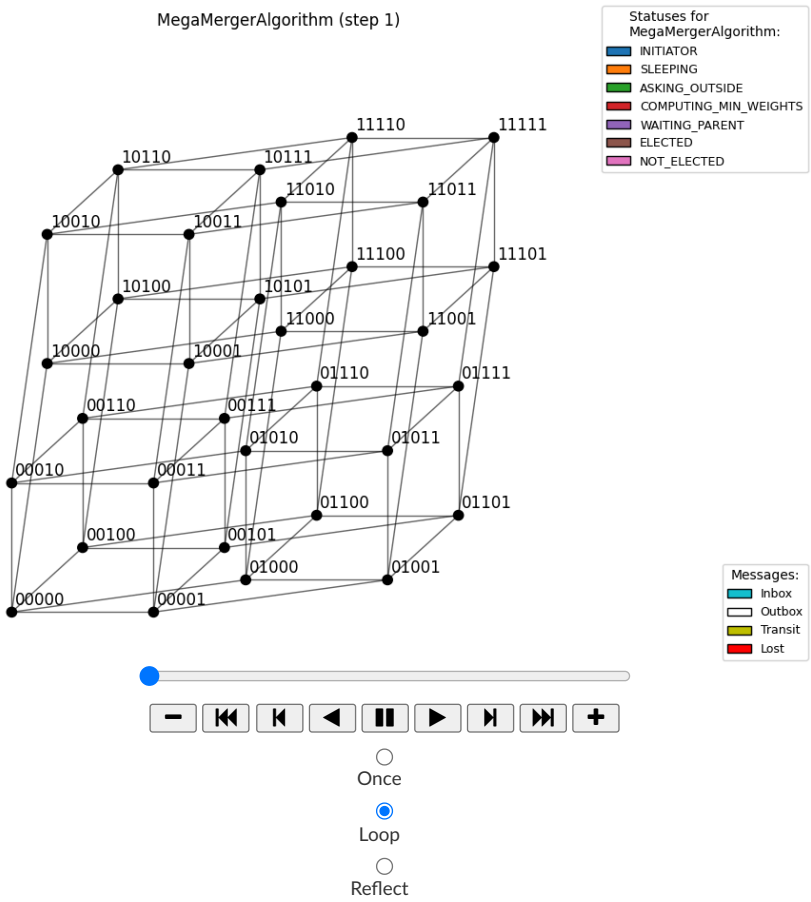
Now we simply create the animation object and display it. The arguments we pass are:

1. The simulation object.

2. The key where each node saves the tree information.
3. The size of the figure.
4. Percentage of the figure width that is occupied by the legend.
5. The size of the nodes.

[9]:
```
draw.create_animation(
    sim,
    tree_key="tree_key",
    figsize=(9, 7),
    space_for_legend=0.25,
    node_radius=5,
)
```

[9]:



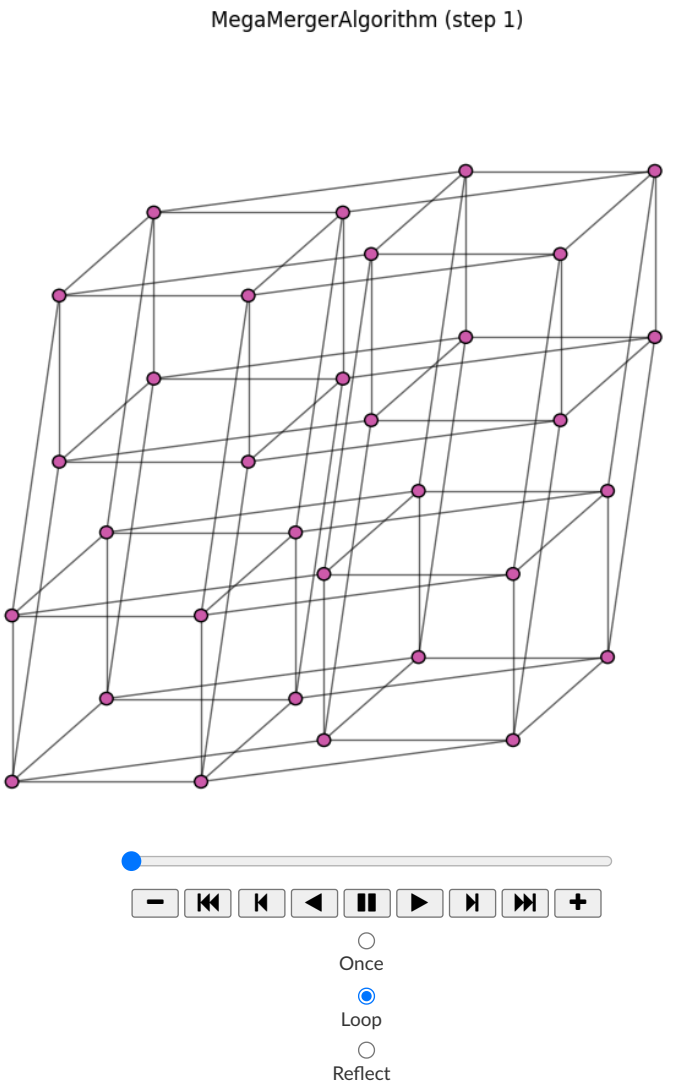
As we can see, the animation is not very informative, as the state of the node is not very representative of the progress of the algorithm. We can improve this by setting the color of the nodes to represent city they are in.

Additionally, we can set the downtown node to be a little bigger than the others, so it is easier to spot.

Since the state will not be displayed, we can remove the legend so it does not take up space.

[10]:
```
draw.create_animation(
    sim,
    tree_key="tree_key",
    figsize=(7, 7),
    node_radius=lambda node: 7 if "city" in node.memory and node.memory["city"].is_downtown else
5,
    message_size=4,
    show_legends=False,
    node_colors=lambda: {
        node: color_map[node.memory["city"].name if "city" in node.memory else "URU"] for node in
sim.network.nodes()
    },
    show_labels=False,
)
```

[10]:



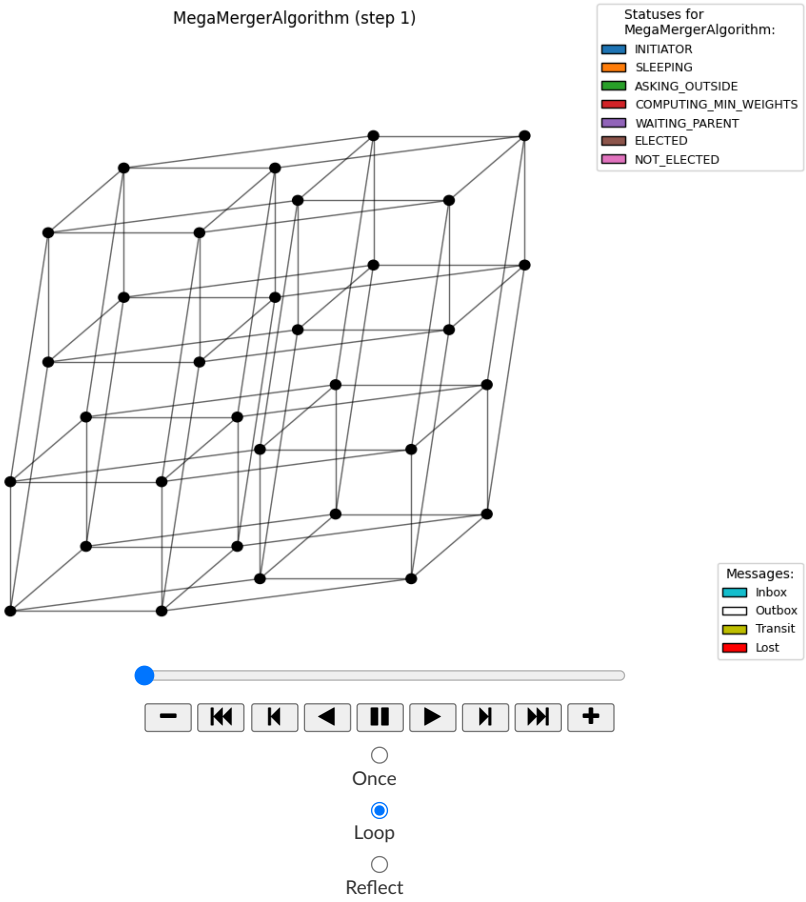
With this last configuration, we can see the progress of the algorithm in a more informative way. We can visually understand how the cities get bigger as the algorithm progresses.

In this next attempt, we try to combine the information gained from the previous two animations. We set the color of the nodes back to represent the state of the node, but we set the labels to represent the city the node is in. The size still marks which node is the downtown node for the city.

[11]:

```
draw.create_animation(
    sim,
    tree_key="tree_key",
    figsize=(9, 7),
    space_for_legend=0.2,
    node_radius=lambda node: 7 if "city" in node.memory and node.memory["city"].is_downtown else
5,
    message_size=2,
    node_labels=lambda: {
        node: f"{node.memory['city'].name}" if "city" in node.memory else "" for node in
sim.network.nodes()
    },
)
```

[11]:



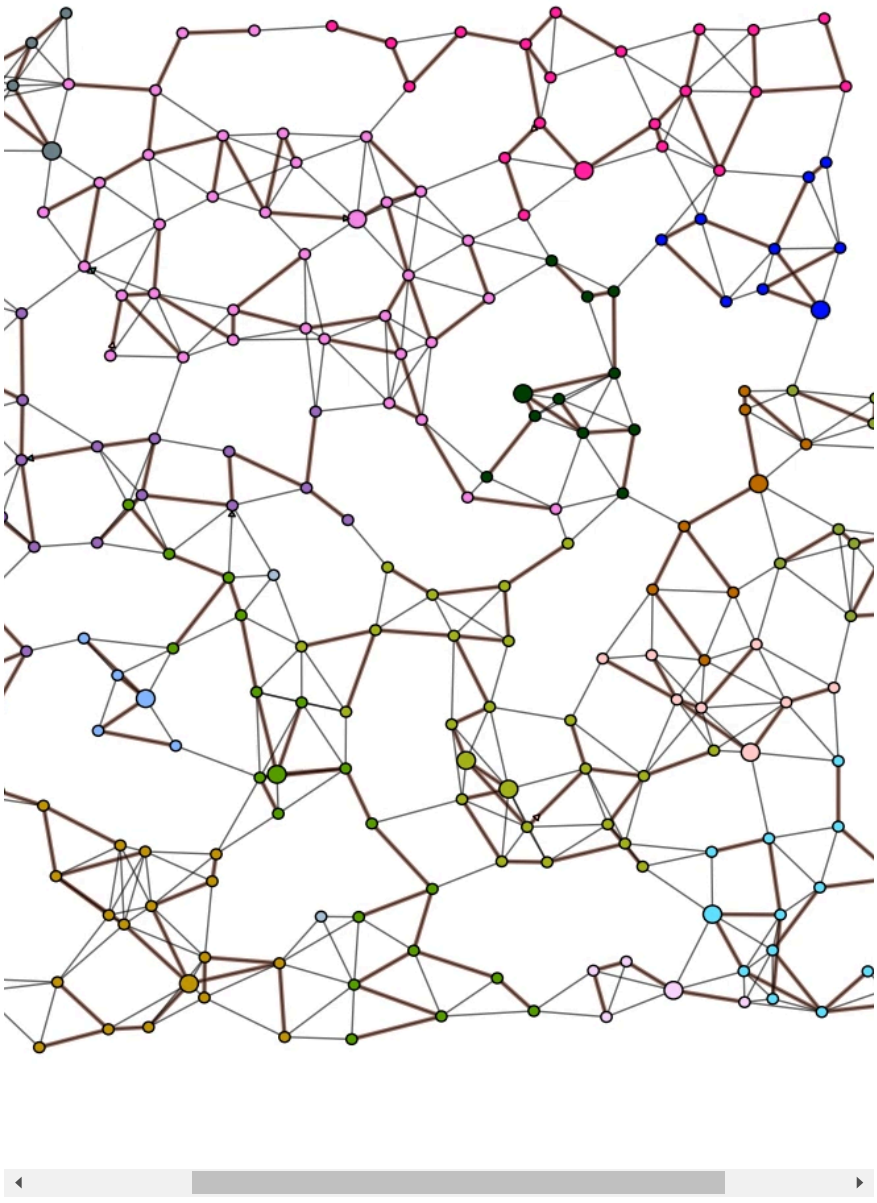
This ends up being the most informative animation, but it is also the most cluttered.

Lastly, we generate a new network, much bigger than the previous ones, and we animate it with the the configuration where the color represents the city.

[5]:
```
net = NetworkGenerator(
    250,
    degree=4.0,
    degree_tolerance=2.0,
).generate_homogeneous_network(randomness=0.06)
net.behavioral_properties = NetworkBehaviorModel.IdealCommunication

sim.network = net
sim.algorithms = ((MegaMergerAlgorithm, ExampleParameters.numerical_parameters),)
```

[7]:
```
from IPython.display import HTML

anim = draw.create_animation(
    sim,
    tree_key="tree_key",
    figsize=(10, 10),
    frame_limit=2001,
    node_radius=lambda node: 5 if "city" in node.memory and node.memory["city"].is_downtown else
3,
    message_size=2,
    show_legends=False,
    node_colors=lambda: {
        node: COLORS[int(node.memory["city"].name) if "city" in node.memory else 0] for node in
sim.network.nodes()
    },
    show_labels=False,
)

HTML(anim.to_html5_video())
```

[7]:



Ideal time and message cost analysis for Mega-Merger: comparing theoretical results with empirical simulations

[ ]:
```
%matplotlib inline

from pydistsim.demo_algorithms.santoro2007.mega_merger.algorithm import MegaMergerAlgorithm,
ExampleParameters
from pydistsim.benchmark import AlgorithmBenchmark
from pydistsim.network.behavior import NetworkBehaviorModel
```

Benchmarking: ideal communications

For this part, we will run a battery of simulations with the help of the benchmark module.

The parameters we will use are:

1. The algorithm to test.
2. The sizes of the networks to test.
3. The number of simulations to run for each network configuration (1 for now).
4. The network behavior (ideal communications obviously).

[3]:
```
from collections import defaultdict


benchmark_ideal = AlgorithmBenchmark(
    ((MegaMergerAlgorithm, ExampleParameters.numerical_parameters),),
    network_behavior=NetworkBehaviorModel.IdealCommunication,
    network_sizes=list(range(1, 30)) + list(range(30, 100, 10)) + list(range(100, 501, 50)),
    network_repeat_count=defaultdict(lambda: 1),  # 1 run for every network configurations
)

benchmark_ideal.run()
```

After the benchmark terminates, we can plot the results. But first, lets take a look at the raw data so we can understand what we may achieve.

[4]:
```
benchmark_ideal.get_results_dataframe(grouped=True)
```

[4]:

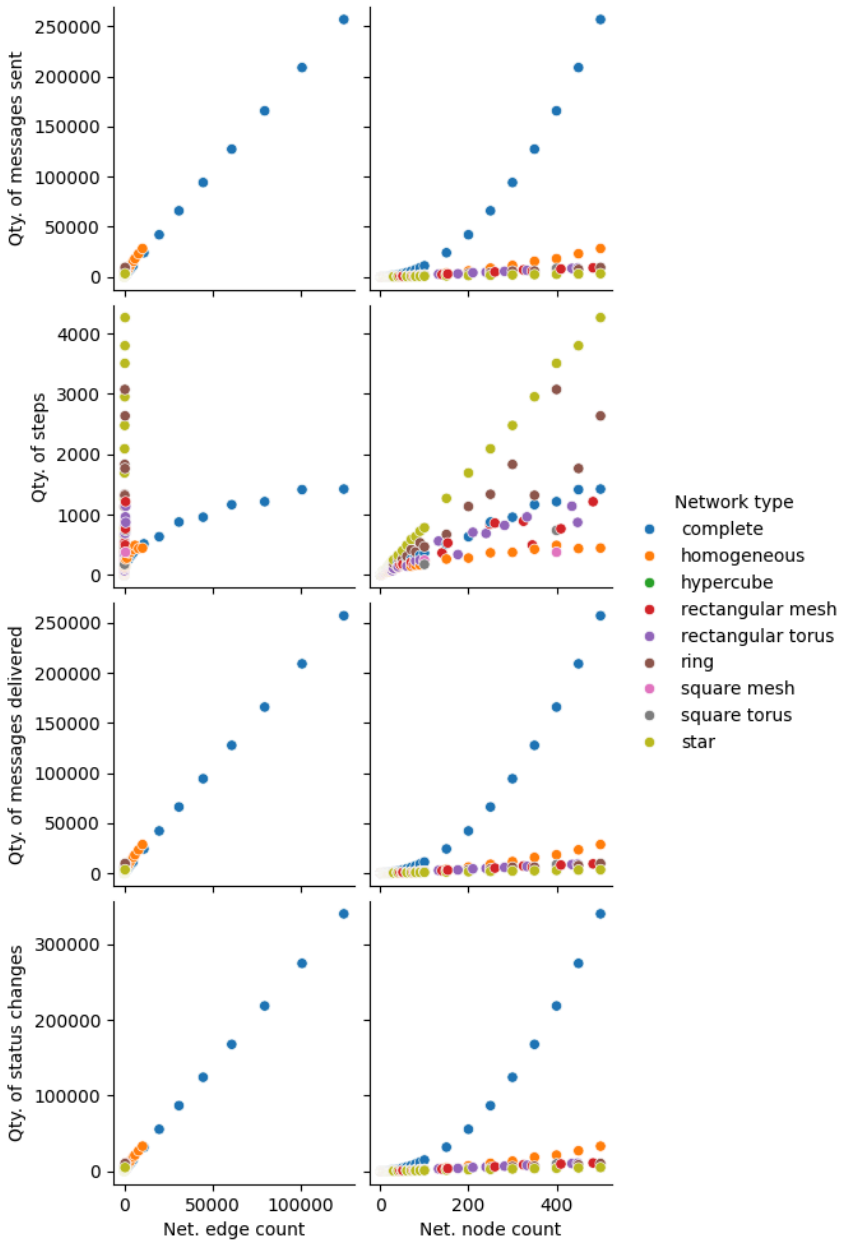
	Net. node count	Network type	Net. edge count	Qty. of messages sent	Qty. of messages delivered	Qty. of status changes	Qty. of steps
0	1	complete	0	0.0	1.0	4.0	3.0
1	1	homogeneous	0	0.0	1.0	4.0	3.0
2	1	hypercube	0	0.0	1.0	4.0	3.0
3	1	rectangular mesh	0	0.0	1.0	4.0	3.0
4	1	rectangular torus	1	1.0	2.0	7.0	5.0
...	...	...	...	...	...	...	...
260	483	rectangular mesh	802	8964.0	9146.0	10833.0	1215.0
261	500	complete	124750	256830.0	257033.0	339751.0	1423.0
262	500	homogeneous	10422	28268.0	28472.0	32991.0	447.0
263	500	ring	500	9337.0	9536.0	10145.0	2634.0
264	500	star	499	3112.0	3304.0	4805.0	4260.0

265 rows × 7 columns

Plot everything:

[5]:
```
benchmark_ideal.plot_analysis()
```

[5]:
```
<seaborn.axisgrid.PairGrid at 0x7f153ba8aa10>
```



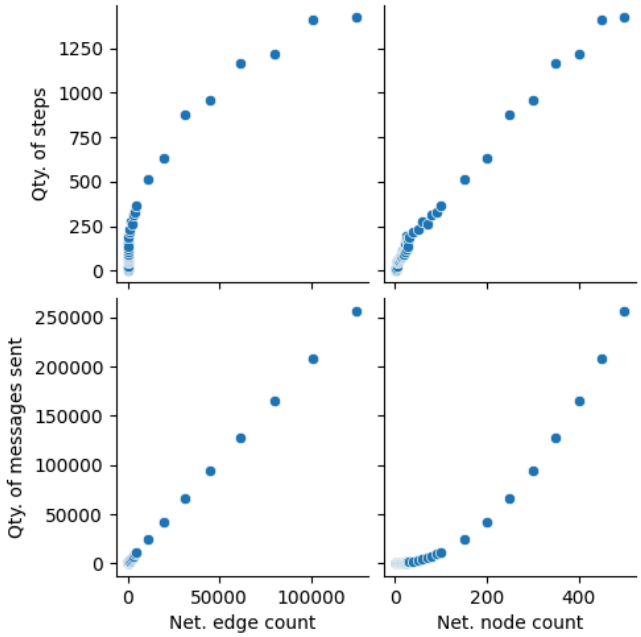
Plot only complete network results

Since the complete and homogeneous network results are the most interesting, we will plot them separately.

Complete network runs

[6]:
```
benchmark_ideal.plot_analysis(
    result_filter=lambda line: line["Network type"] == "complete", y_vars=["Qty. of messages sent", "Qty. of steps"]
)
```

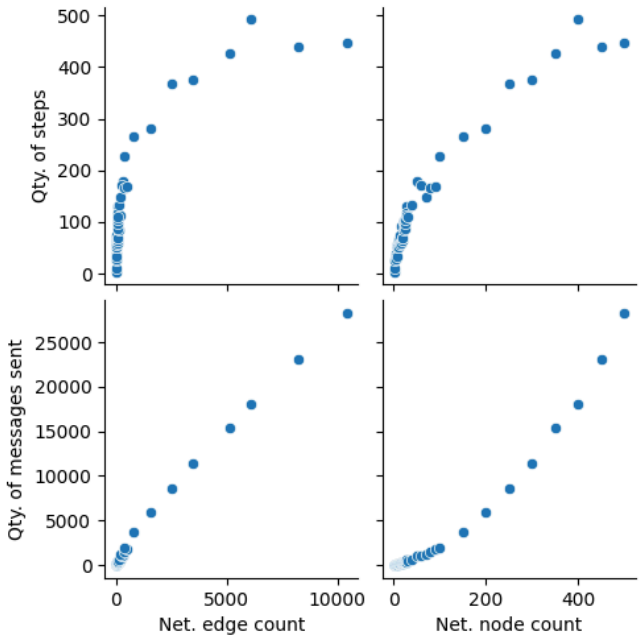
[6]: <seaborn.axisgrid.PairGrid at 0x7f153966e5d0>



Homogeneous network runs

[7]:
```
benchmark_ideal.plot_analysis(
    result_filter=lambda line: line["Network type"] == "homogeneous", y_vars=["Qty. of messages sent", "Qty. of steps"]
)
```

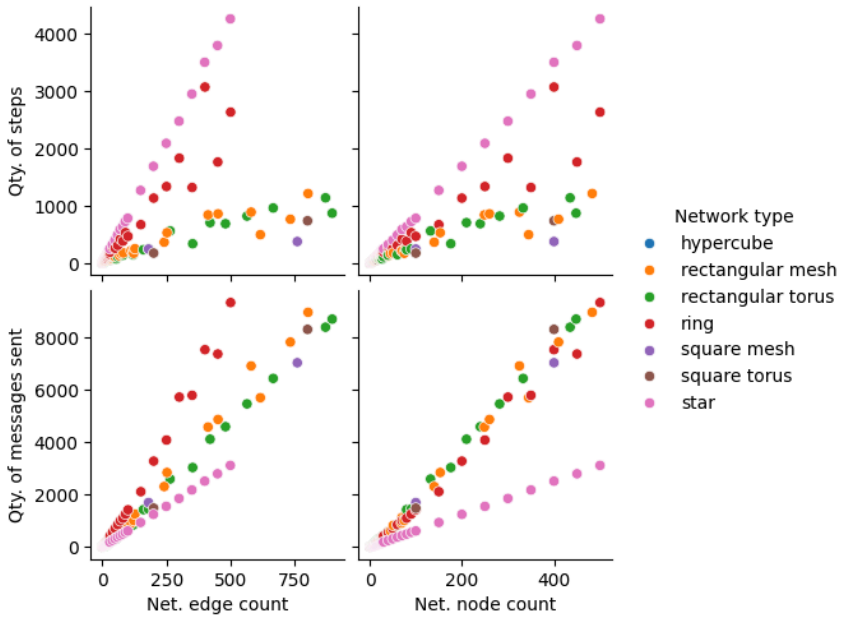
[7]: <seaborn.axisgrid.PairGrid at 0x7f153b9c5910>



Every other network type

[8]:
```
benchmark_ideal.plot_analysis(
    result_filter=lambda line: line["Network type"] not in ("complete", "homogeneous"),
    y_vars=["Qty. of messages sent", "Qty. of steps"],
)
```

[8]: <seaborn.axisgrid.PairGrid at 0x7f1538223310>



[9]:
```
# backup the results, since we will overwrite them
back = benchmark_ideal.results.copy()
```

Plotting against theoretical upper bound

Message count as a function of the number of edges

To be able to plot the theoretical upper bound, we first need to project it to a 2D space. We can do this by taking the quantity of nodes as the maximum possible for any given number of edges. Now the bound is a function of the number of edges only.

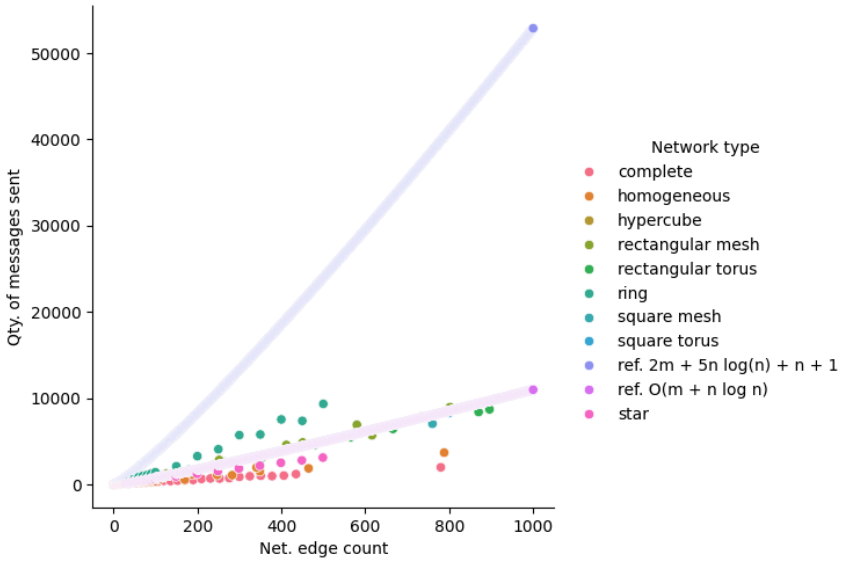
First we add the theoretical execution metrics to the benchmark results:

[11]:
```
from math import log2

for m in range(1, 1001):
    m = m * 1.0

    n = 1.0 * (m + 1)  # Maximum number of nodes for any connected network with m edges

    loose_bound = m + n * log2(n)  # O(m + n log n)
    benchmark_ideal.results.insert(
        0,
        {
            "Net. node count": n,
            "Net. edge count": m,
            "Network type": "ref. O(m + n log n)",
            "Qty. of messages sent": loose_bound,
            "Qty. of messages delivered": loose_bound,
            "Qty. of status changes": loose_bound,
            "Qty. of steps": loose_bound,
        },
    )

    tight_bound = 2 * m + 5 * n * log2(n) + n + 1
    benchmark_ideal.results.insert(
        0,
        {
            "Net. node count": n,
            "Net. edge count": m,
            "Network type": "ref. 2m + 5n log(n) + n + 1",
            "Qty. of messages sent": tight_bound,
            "Qty. of messages delivered": tight_bound,
            "Qty. of status changes": tight_bound,
            "Qty. of steps": tight_bound,
        },
    )
```

Plot the results

[12]:
```
benchmark_ideal.plot_analysis(
    result_filter=lambda line: line["Net. edge count"] <= 1000,
    y_vars=[
        "Qty. of messages sent",
    ],
    x_vars=["Net. edge count"],
    pairplot_kwargs={"height": 5},
)
```

[12]: <seaborn.axisgrid.PairGrid at 0x7f153361fbd0>



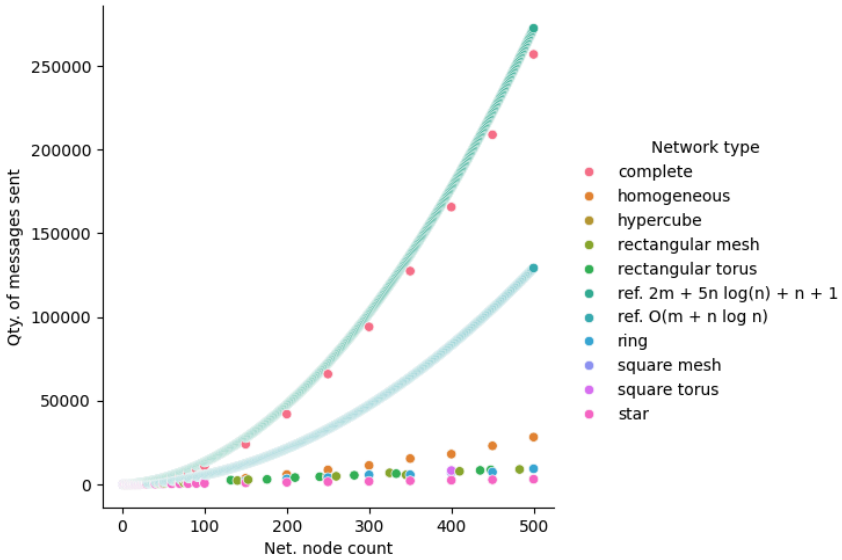
Again, we project the theoretical upper bound to a 2D space. Now the bound is a function of the number of nodes only.

[13]:
```
from math import log2

# restore the original results
benchmark_ideal.results = back.copy()

for n in range(1, 501):
    n = 1.0 * n
    m = n * (n - 1) / 2  # Maximum number of edges for any connected network with n nodes (a
complete graph)

    loose_bound = m + n * log2(n)  # O(m + n log n)
    benchmark_ideal.results.insert(
        0,
        {
            "Net. node count": n,
            "Net. edge count": m,
            "Network type": "ref. O(m + n log n)",
            "Qty. of messages sent": loose_bound,
            "Qty. of messages delivered": loose_bound,
            "Qty. of status changes": loose_bound,
            "Qty. of steps": loose_bound,
        },
    )

    tight_bound = 2 * m + 5 * n * log2(n) + n + 1
    benchmark_ideal.results.insert(
        0,
        {
            "Net. node count": n,
            "Net. edge count": m,
            "Network type": "ref. 2m + 5n log(n) + n + 1",
            "Qty. of messages sent": tight_bound,
            "Qty. of messages delivered": tight_bound,
            "Qty. of status changes": tight_bound,
            "Qty. of steps": tight_bound,
        },
    )
```

Plot the results

[14]:
```
benchmark_ideal.plot_analysis(
    y_vars=[
        "Qty. of messages sent",
    ],
    x_vars=["Net. node count"],
    pairplot_kwargs={"height": 5},
)
```

[14]: <seaborn.axisgrid.PairGrid at 0x7f15336cd7d0>



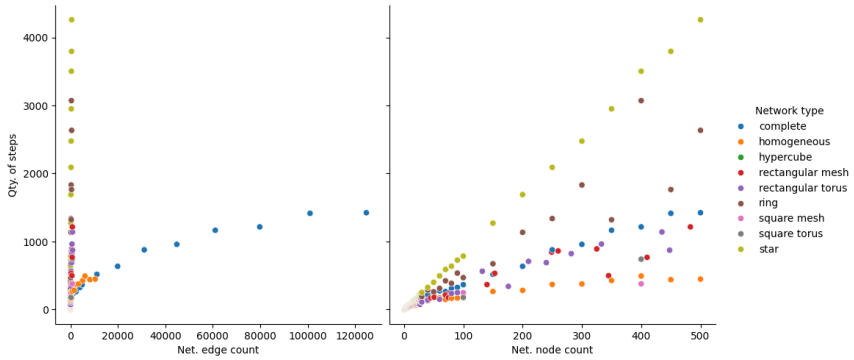
[15]:
```
# restore the original results
benchmark_ideal.results = back.copy()
```

Inferring time complexity

Let's focus on the quantity of steps as a function of the number of nodes and edges.

[19]:
```
benchmark_ideal.plot_analysis(
    y_vars=["Qty. of steps"],
    pairplot_kwargs={"height": 5},
)
```

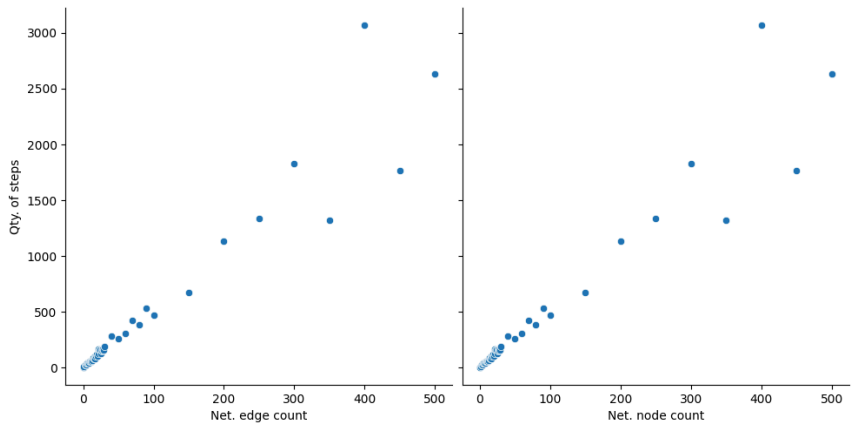
[19]: <seaborn.axisgrid.PairGrid at 0x7f153fab0090>



Since we are aiming to infer the worst time complexity of the algorithm, we must focus on the worst case scenario. This seems to be the case where the network is a ring:

[21]:
```
benchmark_ideal.plot_analysis(
    y_vars=["Qty. of steps"],
    result_filter=lambda line: line["Network type"] == "ring",
    pairplot_kwargs={"height": 5},
)
```

[21]: <seaborn.axisgrid.PairGrid at 0x7f153fa84d50>

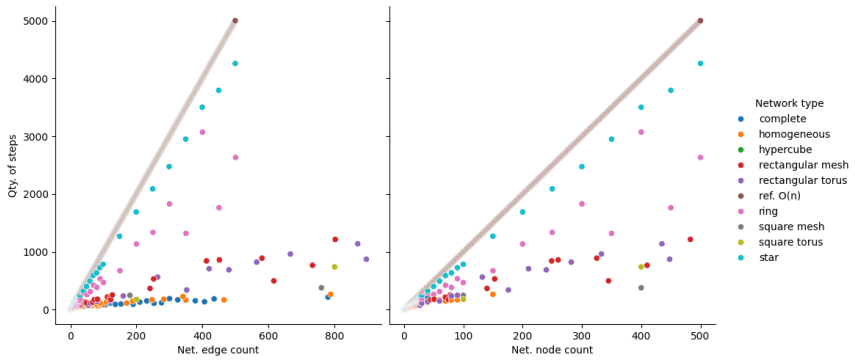


Obviously, the number of steps appears to be a linear function of the number of nodes and edges. Let's plot a linear regression to confirm this.

[29]:
```
from math import log2

# restore the original results
benchmark_ideal.results = back.copy()

for n in range(1, 501):
    n = 1.0 * n
    m = n - 1

    loose_bound = m + n * log2(n)  # O(m + n log n)
    benchmark_ideal.results.insert(
        0,
        {
            "Net. node count": n,
            "Net. edge count": m,
            "Network type": "ref. O(n)",
            "Qty. of messages sent": n,
            "Qty. of messages delivered": n,
            "Qty. of status changes": n,
            "Qty. of steps": 10 * n,
        },
    )
```

[34]:
```
benchmark_ideal.plot_analysis(
    y_vars=["Qty. of steps"],
    result_filter=lambda line: line["Net. edge count"] < 1_000,
    pairplot_kwargs={"height": 5},
)
```

[34]: <seaborn.axisgrid.PairGrid at 0x7f15563bf190>



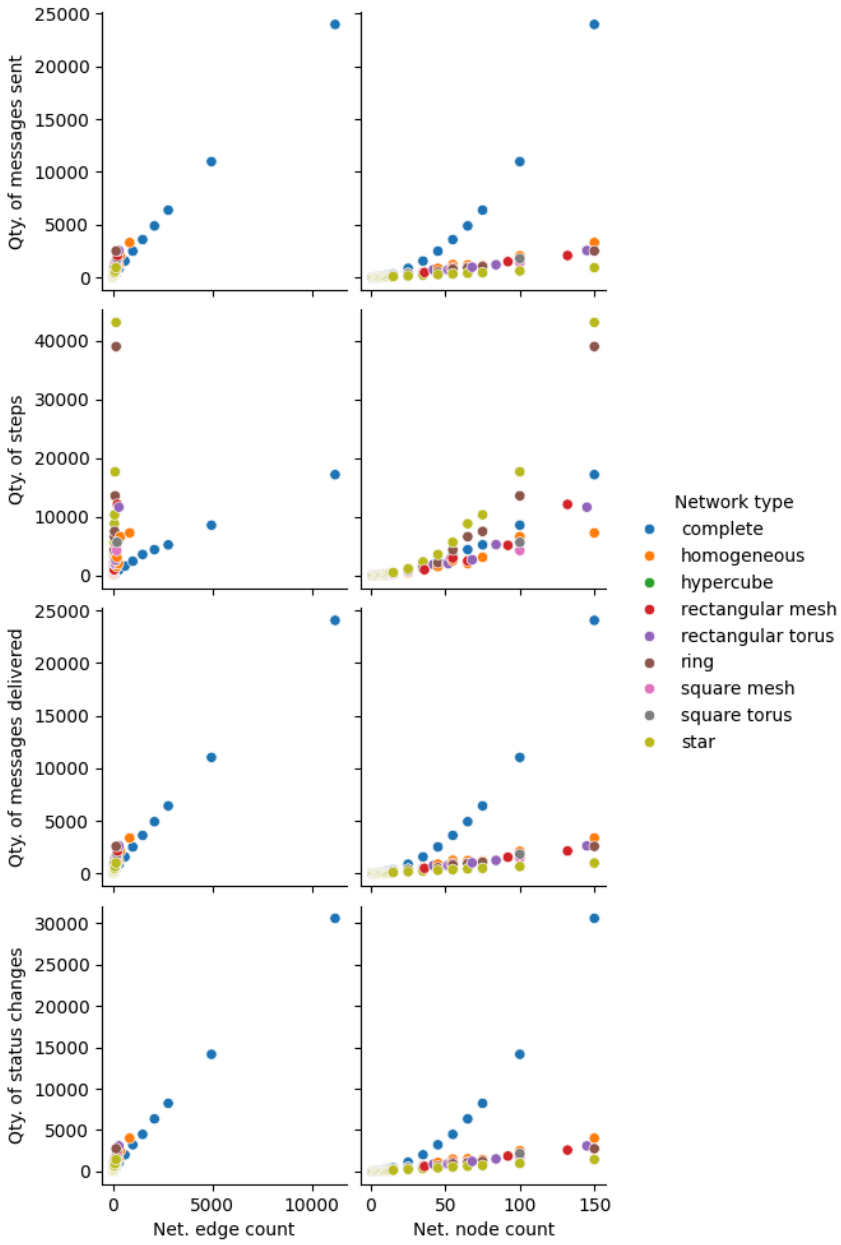
Benchmarking: communication with delays

Now we will run a very similar battery of simulations, but with a delay in the communications. We will not analyze the results in detail again.

[35]:
```
benchmark_with_delay = AlgorithmBenchmark(
    ((MegaMergerAlgorithm, ExampleParameters.numerical_parameters),),
    network_behavior=NetworkBehaviorModel.RandomDelayCommunication,
    network_sizes=list(range(1, 15)) + list(range(15, 76, 10)) + list(range(100, 200, 50)),
    network_repeat_count=defaultdict(lambda: 1),
)

benchmark_with_delay.run()
```

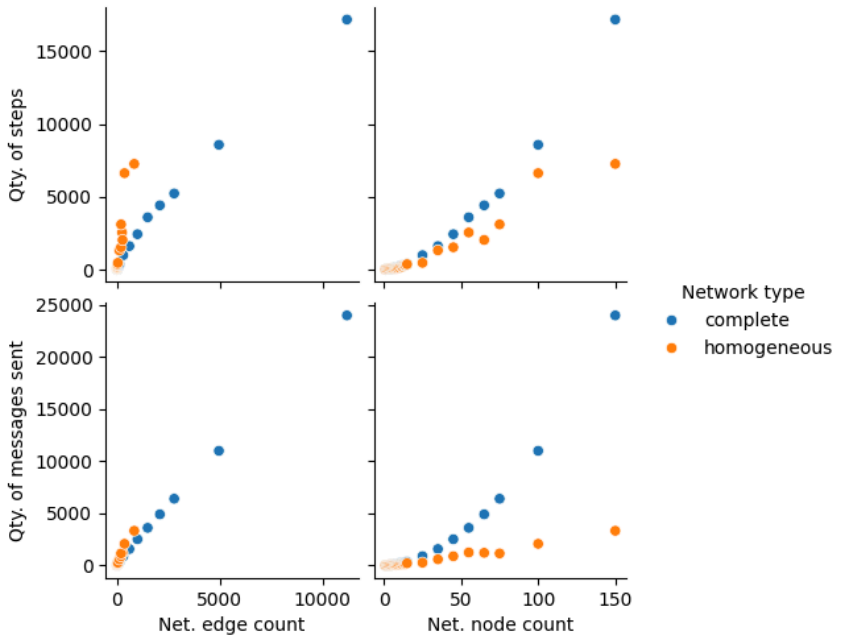
[36]:
```
benchmark_with_delay.plot_analysis()
```

[36]: <seaborn.axisgrid.PairGrid at 0x7f155cc80cd0>



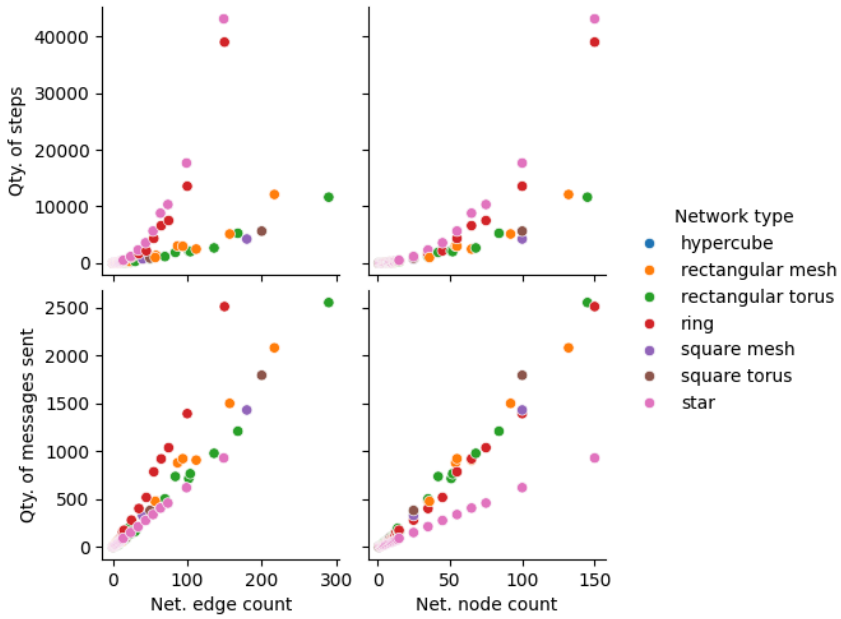
[37]:
```
benchmark_with_delay.plot_analysis(
    result_filter=lambda line: line["Network type"] in ("complete", "homogeneous"),
    y_vars=["Qty. of messages sent", "Qty. of steps"],
)
```

[37]: <seaborn.axisgrid.PairGrid at 0x7f15513a9150>



[38]:
```
benchmark_with_delay.plot_analysis(
    result_filter=lambda line: line["Network type"] not in ("complete", "homogeneous"),
    y_vars=["Qty. of messages sent", "Qty. of steps"],
)
```

[38]: <seaborn.axisgrid.PairGrid at 0x7f15586f37d0>



Testing the Mega-Merger implementation

To ensure the correctness of the algorithm, we will run a few tests.

Since the problem Mega-Merger solves is topologically independent Distributed Election, we will test the algorithm in a few different scenarios. The network will be of different sizes and shapes, and the proportion of initiators will vary.

The network behavior will be: message ordering, no message loss, random delay based on the network size and unsynchronized clocks.

[1]:
```
from random import uniform

from pydistsim.demo_algorithms.santoro2007.mega_merger.algorithm import MegaMergerAlgorithm,
ExampleParameters
from pydistsim.network import NetworkGenerator
from pydistsim.network.behavior import NetworkBehaviorModel
from pydistsim.simulation import Simulation
```

[87]:
```
def test_net(net):
    # Numerical parameters: city names and weights are integers
    par = ExampleParameters.numerical_parameters.copy()

    # Randomize the parameters
    par.update({"percentage_of_initiators": uniform(0.1, 1)})

    # Create the simulation object
    sim = Simulation(net, ((MegaMergerAlgorithm, par),))

    # Run the simulation
    sim.run()

    assert all(node.status in MegaMergerAlgorithm.S_term for node in net)
    assert len({node.memory["city"].name for node in sim.network.nodes()}) == 1
    assert len({node for node in sim.network.nodes() if node.status ==
MegaMergerAlgorithm.Status.ELECTED}) == 1
```

Ring network

[88]:
```
%%time

for n in [1, 10, 40, 100]:
    net = NetworkGenerator.generate_ring_network(n)
    net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication
    test_net(net)
    print(f"Test passed for ring network with {n} nodes.")

print("\nAll tests passed for ring networks with 1, 10, 40 and 100 nodes.\n")
```

```
Test passed for ring network with 1 nodes.
Test passed for ring network with 10 nodes.
Test passed for ring network with 40 nodes.
Test passed for ring network with 100 nodes.

All tests passed for ring networks with 1, 10, 40 and 100 nodes.

CPU times: user 6.85 s, sys: 0 ns, total: 6.85 s
Wall time: 6.85 s
```

Complete network

[89]:
```
%%time

for n in range(2, 35):
    net = NetworkGenerator.generate_complete_network(n)
    net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication
    test_net(net)

print("All tests passed for complete networks with 2 up to 35 nodes.\n")
```

```
All tests passed for complete networks with 2 up to 35 nodes.

CPU times: user 9.52 s, sys: 0 ns, total: 9.52 s
Wall time: 9.52 s
```

Square mesh/grid network

[90]:
```
%%time

for n in range(2, 11):
    n = n * n

    net = NetworkGenerator.generate_mesh_network(n)
    net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication
    test_net(net)
    print(f"Test passed for square mesh network with {n} nodes.")

print("\nAll tests passed for square mesh networks with 4 up to 121 nodes.\n")
```

```
Test passed for square mesh network with 4 nodes.
Test passed for square mesh network with 9 nodes.
Test passed for square mesh network with 16 nodes.
Test passed for square mesh network with 25 nodes.
Test passed for square mesh network with 36 nodes.
Test passed for square mesh network with 49 nodes.
Test passed for square mesh network with 64 nodes.
Test passed for square mesh network with 81 nodes.
Test passed for square mesh network with 100 nodes.

All tests passed for square mesh networks with 4 up to 121 nodes.

CPU times: user 6.26 s, sys: 4.89 ms, total: 6.27 s
Wall time: 6.26 s
```

Path network (`1 x n` grid)

[91]:
```
%%time

for n in range(2, 51):
    net = NetworkGenerator.generate_mesh_network(a=1, b=n)
    net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication
    test_net(net)
    if n % 10 == 0:
        print(f"Test passed for path network with {n} nodes.")

print("\nAll tests passed for path networks with 1 up to 50 nodes.\n")
```

```
Test passed for path network with 10 nodes.
Test passed for path network with 20 nodes.
Test passed for path network with 30 nodes.
Test passed for path network with 40 nodes.
Test passed for path network with 50 nodes.

All tests passed for path networks with 1 up to 50 nodes.

CPU times: user 11.1 s, sys: 34.8 ms, total: 11.2 s
Wall time: 11.1 s
```

Star network

[92]:
```
%%time

for n in range(2, 45):
    net = NetworkGenerator.generate_star_network(n)
    net.behavioral_properties = NetworkBehaviorModel.RandomDelayCommunication
    test_net(net)
    if n % 10 == 0:
        print(f"Test passed for star network with {n} nodes.")

print("\nAll tests passed for star networks with 2 up to 44 nodes.\n")
```

```
Test passed for ring network with 10 nodes.
Test passed for ring network with 20 nodes.
Test passed for ring network with 30 nodes.
Test passed for ring network with 40 nodes.

All tests passed for ring networks with 2 up to 44 nodes.

CPU times: user 7.53 s, sys: 10 µs, total: 7.53 s
Wall time: 7.53 s
```

Theoretical models for network algorithms

Distributed computing

Centralized computing

Build date: Nov 02, 2024

Release version: 2.1.1

The message passing model

Theoretical representation

The book titled “Design and Analysis of Distributed Algorithms” by Nicola Santoro provides an in-depth exploration of the fundamental concepts and principles underlying distributed computing systems. It begins by defining a distributed computing environment as a system composed of multiple computational entities, each of which may be referred to as a process, processor, switch, agent, or by other names depending on the context and the system being modeled. These entities work collaboratively, often communicating over a network, to achieve a common goal such as solving a problem or performing a specific task.

Each entity within a distributed computing environment possesses its own local memory, which includes various registers used to maintain its status and the values it processes. For instance, the status register (*status(x)*) indicates the current state of an entity, while the input value register (*value(x)*) stores values pertinent to the computations or tasks the entity is performing. Entities are equipped with the capabilities to perform local storage and processing, send and receive messages to and from other entities, reset alarm clocks, and alter their status based on certain conditions or events.

Actions and reactions

This model emphasizes that entities in a distributed environment react to external events. These events could include the arrival of a message, the ringing of an alarm clock, or spontaneous impulses generated within the system. When an entity encounters an event, it executes a sequence of operations known as an action, which is performed without interruption and within a finite time frame. The specific behavior of an entity is thus determined by its current status and the type of event it is reacting to. This behavior is often defined through a set of rules or protocols that outline the appropriate responses for each combination of status and event.

Communication

Communication between entities in a distributed computing environment is primarily achieved through the transmission and reception of messages. Each entity has a defined set of neighbors to which it can send messages (*out_neighbors(x)*) and from which it can receive messages (*out_neighbors(x)*). The communication topology of the environment is typically represented by a directed graph, where nodes symbolize entities and edges represent the communication links. This graphical representation helps in understanding the relationships and communication pathways between different entities within the system.

Axioms and restrictions

The model also outlines several axioms that are fundamental to the operation of such distributed computing environments. One such axiom is the principle of finite communication delays, which states that, in the absence of failures, any message transmitted between entities will be delivered

within a finite amount of time. Another axiom is local orientation, which implies that entities are capable of distinguishing between their neighbors, thereby enabling them to direct messages to specific entities and to know from which entity a message has been received.

In addition to these axioms, the book discusses various optional restrictions that may apply to distributed environments. Restrictions are additional properties or capabilities that define specific characteristics of the system, potentially limiting or enhancing its operation. These restrictions can pertain to communication properties, reliability, and synchrony, among other factors. For example, some systems may enforce message ordering, ensuring that messages sent to the same neighbor arrive in the order they were dispatched. Other systems may adhere to the principle of reciprocal communication, which requires that communication links be bidirectional, meaning that each entity capable of sending a message must also be able to receive a message in return.

Reliability is another critical aspect of distributed computing environments covered in the model. Systems may incorporate mechanisms for fault detection, allowing entities to detect the failure of links or other entities. Different levels of reliability can be defined based on the system's tolerance to failures. For instance, a system with guaranteed delivery ensures that messages are always delivered without any corruption. In contrast, a system characterized by partial reliability might guarantee that no failures occur during execution, whereas a totally reliable system would be completely free from failures.

Bottom line

In summary, the model provides a comprehensive abstraction of the various architecture components, entities, behaviors, communication protocols, axioms, and potential restrictions of distributed systems. It explores how entities operate within these environments, emphasizing predictable behavior through clearly defined rules and protocols that govern their interactions and responses to events. The focus on communication, reliability, and fault tolerance underscores the complexities and challenges inherent in designing and managing distributed computing systems.

PyDistSim equivalences

This framework provides a Python implementation of the message passing model described by Santoro. It offers a set of classes and functions that enable users to define and simulate distributed computing environments, create entities, specify communication topologies, and model interactions between entities. The framework is designed to facilitate the development and analysis of distributed algorithms, allowing users to explore various scenarios and test the behavior of entities under different conditions.

Protocols and algorithms can be implemented by subclassing the `NodeAlgorithm` class and defining the appropriate methods to handle every action.

The entities in the PyDistSim framework are represented by the `Node` class, which encapsulates the local memory and processing capabilities of an entity within a distributed environment. In particular, the `Node.status` attribute models the current state of each entity.

Restrictions can be enforced by adding them to the `NodeAlgorithm.algorithm_restrictions` attribute. This attribute is a list of classes (`Restriction`) representing all the restrictions that the algorithm must follow.

Centralized model

Theoretical representation

Centralized computing is a model in which all computational tasks, data storage, and resource management are handled by a single central server or a small cluster of servers. This contrasts sharply with distributed computing, where multiple independent entities work together across a network to perform tasks and store data.

In a centralized computing environment, all resources, such as processing power, memory, and storage, are consolidated in one central location. Users typically access these resources through terminals or thin clients, which depend entirely on the central server for processing power and data storage. This setup allows for a more straightforward architecture because all computing happens in one place, avoiding the complexities involved in synchronizing and managing multiple independent systems.

PyDistSim equivalences

Even though PyDistSim is not designed to simulate centralized systems, it can be used to simulate the setup phase of a distributed system, where a central entity distributes the initial state to all other entities.

Simple centralized algorithms can be implemented by subclassing the `NetworkAlgorithm` class.

Reference

This section contains detailed information about the classes and functions in the framework. Only refer to this section once you are familiar with the basic concepts of PyDistSim.

Build date: Nov 02, 2024

Release version: 2.1.1

Behavioral properties of a network

The communication properties of the network are defined in the `NetworkBehaviorModel` class. This class defines the three parameters that are used to model the communication properties of the network:

1. Whether or not the messages being sent at a given time will arrive in the same order they were sent (massage ordering).
2. The amount of delay that the messages will have when being sent.
3. The frequency at which the messages will be lost.
4. How rapidly the internal clocks of the nodes will increase.
5. How fast a node can process a message. This is used to model the case where a node can only process a certain number of messages per time unit.

To apply these properties to a network, simply set `NetworkType.behavioral_properties` to an instance of `NetworkBehaviorModel`.

Some predefined models are available as class attributes of the `NetworkBehaviorModel` class:

1. `NetworkBehaviorModel.IdealCommunication`
2. `NetworkBehaviorModel.UnorderedCommunication`
3. `NetworkBehaviorModel.ThrottledCommunication`
4. `NetworkBehaviorModel.UnorderedThrottledCommunication`
5. `NetworkBehaviorModel.RandomDelayCommunication`
6. `NetworkBehaviorModel.RandomDelayCommunicationSlowNodes`
7. `NetworkBehaviorModel.RandomDelayCommunicationVerySlowNodes`
8. `NetworkBehaviorModel.UnorderedRandomDelayCommunication`
9. `NetworkBehaviorModel.UnorderedRandomDelayCommunication`
10. `NetworkBehaviorModel.UnlikelyRandomLossCommunication`

To get maximum flexibility, delay and loss are defined as functions that take the message and the network as arguments. This means that the delay and loss can be defined as functions of the message and the network state, so this can be used to model more complex scenarios.

Logging

The logging module provides a flexible framework for emitting log messages from Python programs. The log message is passed to the `logger` object, which handles the formatting and output of the message. The logger is the entry point to the logging system.

By default, the logging module is disabled. To enable it, call the `enable_logger()` function. This will enable the logger and set the log level to WARNING. To change the log level, call the `set_log_level()` function.

Networks and the NetworkMixin

PyDistSim uses the library `networkx` to represent graphs. `networkx` provides a rich set of graph classes and methods to work with graphs.

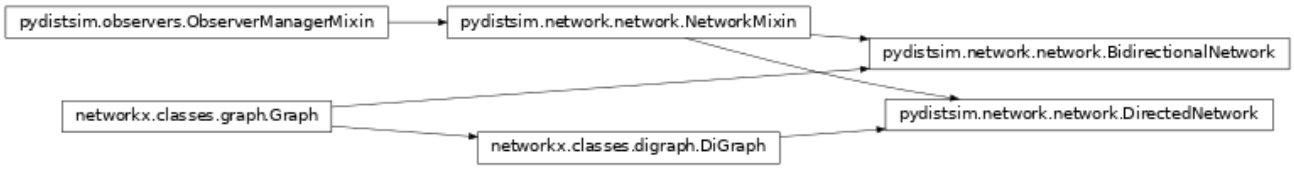
In order to extend the defined `networkx.Graph` and `networkx.DiGraph` in `networkx`, PyDistSim uses a mixin class.

This mixin class is called NetworkMixin and is defined in the network module. For the development of the framework, we have used this mixin to define `DirectedNetwork` and `BidirectionalNetwork`, which are subclasses of `networkx.Graph` and `networkx.DiGraph` respectively.

In broad terms, these NetworkMixin subclasses are responsible for the following:

- Managing the nodes and edges of the graph.
- Managing the data associated with the nodes and edges.
- Managing the network properties.

For class and method documentation refer to `NetworkMixin`.

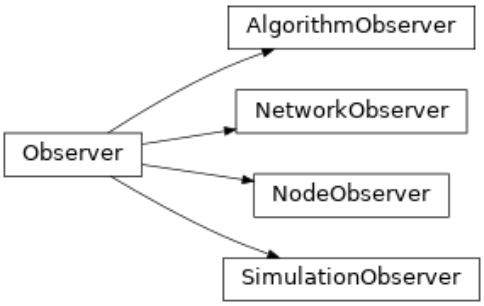


Inheritance diagram for DirectedNetwork and BidirectionalNetwork

Observers

PyDistSim provides a set of observers that can be used to monitor the simulation. Observers are used to collect data from the simulation, and can be used to plot the simulation results, or to analyze the simulation data.

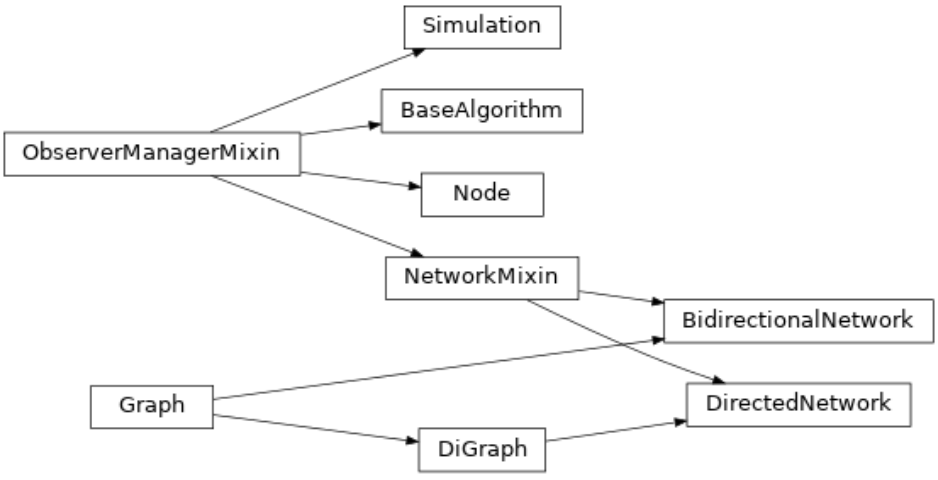
The observer classes are defined in the module `pydistsim.observers`. The observer base class is `Observer`, which defines the interface for all observers.



Observer inheritance diagram

The observable classes

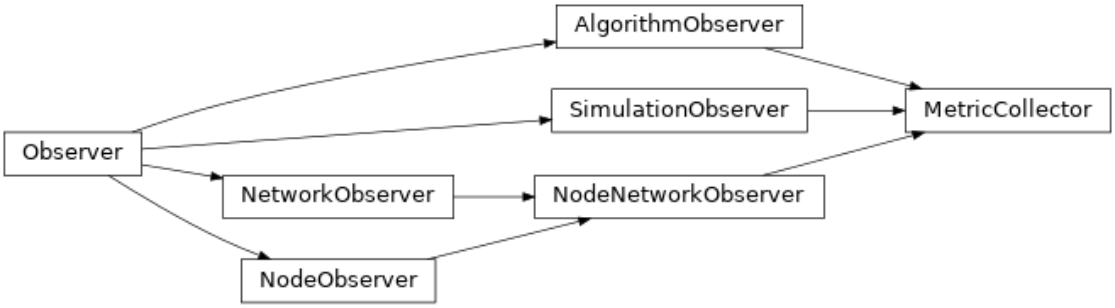
Standardize the use of observers in the framework, we implemented a mixin class `ObserverManagerMixin` that can be used to add observer functionality to any class. The mixin class provides methods to add and remove observers, and to notify the observers of changes in the observable object.



ObserverManagerMixin inheritance diagram

The metric collector

In addition to the observer classes, PyDistSim provides a special observer class called `MetricCollector`. The metric collector is used to collect metrics from the simulation, and to store the metrics in a data structure.



MetricCollector inheritance diagram

To enable the metric collection, the simulation object must be configured with a metric collector object like this:

```
metrics = MetricCollector()
sim = Simulation(net)
sim.add_observers(metrics)
```

To access the collected metrics, the `MetricCollector.create_report()` method can be used.

Subclassing `MetricCollector`

Extend this class and implement the desired event methods to collect custom metrics. For registering events, call the `MetricCollector._add_metric()` method. Add an instance of your custom collector to the simulation observers for it to work.

Even so, you can use the `events` attribute to register the **new** events you want to listen to. This includes new custom events that you would trigger from a custom algorithm.

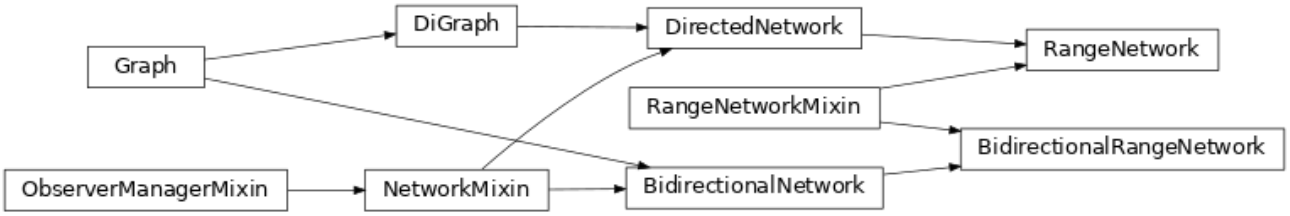
Example of implementing a custom metric collector:

```
class ExampleCustomMetricCollector(MetricCollector):
    events = ["example_custom_event"]

    class CustomMetricEventType(StrEnum):
        "Definition if this enum is optional. It helps to avoid typos in the event names."
        EXAMPLE_CUSTOM_EVENT_ZERO = "EXAMPLE_CUSTOM_EVENT_ZERO"
        ...

    def on_example_custom_event(self, a, b, c):
        self._add_metric(
            self.CustomMetricEventType.EXAMPLE_CUSTOM_EVENT_ZERO,
            {"a": a, "b": b, "c": c}
        )
```

Range Networks

PyDistSim implements an abstraction of range networks, which is a network where each node is connected to all nodes within a certain range. This is useful for simulating wireless networks, where the range of a node is limited by the power of its transmitter.

For reference, see `RangeNetwork` and `BidirectionalRangeNetwork`.

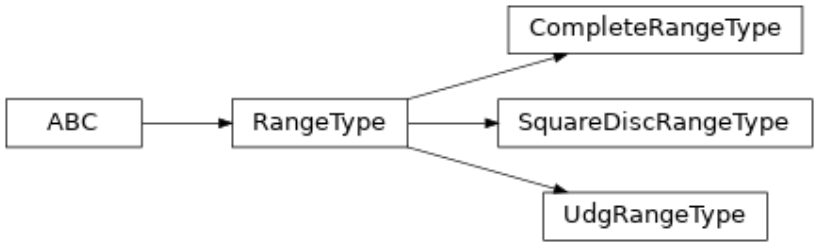


Inheritance diagram for RangeNetwork and BidirectionalRangeNetwork

Range Types

PyDistSim implements several range types, which define the behavior of the range network. The rage type defines which nodes are connected to each other, but use of distance is not mandatory. For example, the `CompleteRangeType` connects all nodes to all other nodes, regardless of distance.

To add a new range type, the developer needs to implement the `RangeType` interface, and override the method `RangeType.in_comm_range()`.



Inheritance diagram for RangeType

Sensors

Sensors provide a way for node to interact with its environment.

Sensors can also be used to satisfy algorithm prerequisites.

Generally sensors should incorporate some model of measurement insecurity that is inherent in real world sensors. This is implemented as a `ProbabilityFunction`.

Basic usage:

```
>>> node.compositeSensor = ('DistSensor','AoASensor')
>>> node.compositeSensor.sensors
(<pydistsim.network.sensor.DistSensor at 0x6d3fbb0>,
 <pydistsim.network.sensor.AoASensor at 0x6d3f950>)
```

To manually set sensor parameters first make an sensor instance:

```
>>> import scipy.stats
>>> aoa_sensor = AoASensor({'pf': scipy.stats.norm, 'scale': 10*pi/180 })
>>> node.compositeSensor = (aoa_sensor,)
```

class **Sensor** [source]

Abstract base class for all Sensors.

Sensor provides a certain capability for a node, providing information about the outside world. It could be a capability to detect neighbors, measure distance to them, or retrieve the environment temperature.

Parameters:
- **dict_args** (*dict*) – A dictionary containing the scale and probability function.
- **scale** (*float*) – The scale parameter for the probability function.
- **pf** (*rv_continuous or rv_discrete*) – The probability function (e.g. `scipy.stats.norm`).

classmethod **name**() [source]

Get the name of the Sensor class.

Returns: The name of the Sensor class.
Return type: str

abstract **read**()→ dict [source]

Read the sensor data.

This method should be overridden in a subclass.

Returns: The sensor data.
Return type: dict

Real world sensors

`AoASensor`	Provides azimuth between node and its neighbors.
`DistSensor`	Provides distance between node and its neighbors.

Knowledge sensors

`TruePosSensor`	Provides node's true position.

Composite sensor

class **`CompositeSensor`**(*node: Node, componentSensors: tuple[type[Sensor] | str] | None = None*) [source]

Wrap multiple sensors, coalesce results and return composite readout.

This class is not a sensor itself, i.e. subclass of `Sensor`, instead it serves as a placeholder for multiple sensors that can be attached to a `Node`.

Parameters:
- **node** (*Node*) – The Node that has this composite sensor attached to.
- **componentSensors** (*tuple[type[Sensor] | str]*) – Tuple of Sensor subclasses or their class names.

`get_sensor`(*name: str*) → Sensor [source]

Get a sensor by its name.

Parameters: **name** (*str*) – The name of the sensor.
Returns: The sensor object.
Return type: Sensor
Raises: **SensorError** – If multiple or no sensors are found with the given name.

`read`() [source]

Read measurements from all sensors.

Returns: A dictionary containing the measurements from all sensors.
Return type: dict

property **`sensors`**: *tuple[Sensor]*

Get the sensors associated with the object.

Returns: A tuple of Sensor objects.