ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Simulación, análisis y evaluación de algoritmos epidémicos

AUTOR: Carlos Barroso Fernández

TITULACIÓN: Grado en Ingeniería Telemática

TUTOR: José Luis López Presa

DEPARTAMENTO: Departamento de Ingeniería Telemática y Electrónica

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: Margarita Millán Valenzuela

TUTOR: José Luis López Presa

SECRETARIO: Hugo Alexer Parada Gélvez

Fecha de lectura:

Calificación:

El Secretario,

AGRADECIMIENTOS

Es justo dejar una pequeña nota de agradecimiento a todas las personas que han estado para mi a lo largo de la realización de este trabajo. En especial, a mi tutor José Luis, por su constante apoyo y paciencia, así como por haber sabido guiarme en todo momento a pesar de que a veces me obstinaba bastante. Has sido un gran referente.

Así mismo, he de agradecer también a Ernesto y Ramón los cuales me han acompañado durante la realización de este proyecto. Ellos me han facilitado el desarrollo de esta investigación con su dedicación y trabajo, ofreciéndome sus mejores consejos en todo momento.

También quiero dar las gracias a mi familia, por apoyarme en todas mis decisiones y estar siempre a mi lado.

Y por último y más importante, quiero agradecer a Carla por el apoyo incondicional, por su esfuerzo por hacerme mejor y por aguantarme, que tiene tela. A pesar de que no entendía mucho, me ha estado ayudando con todo lo que podía y más.

Simulación, análisis y evaluación de algoritmos epidémicos

PALABRAS CLAVE

Algoritmo epidémico, simulación, Peersim, redes con equipos de bajas capacidades, R, optimización de memoria y energía.

RESUMEN

El objetivo principal de cualquier red es el intercambio de información entre sus diferentes elementos. Actualmente, la gran demanda existente exige a las redes incrementar su capacidad día a día para poder intercambiar cada vez más información y más rápido. Pero no se queda ahí, ya que se prevé que sigan aumentando aún más las exigencias. Para poder satisfacer toda esta demanda, no solo se ha de incrementar la capacidad de los equipos y los enlaces para que puedan soportar más información, sino que también hay que pulir los algoritmos que rigen la lógica detrás del intercambio de esta información.

Entre los muchos protocolos que existen, los epidémicos (*gossiping* o *epidemic*, en inglés) destacan por utilizar los modelos matemáticos detrás de las expansiones de los rumores o las epidemias (son muy similares) para difundir la información entre todos los elementos de la red. Estos protocolos se centran en conseguir una difusión más eficiente sin perder fiabilidad. Además, son muy tolerantes a fallos y muy flexibles.

Por otro lado, las redes con equipos de bajas capacidades, como por ejemplo las redes de sensores, se encuentran actualmente en el foco de muchos proyectos de investigación, debido a su relación con tecnologías punteras como el Internet de las Cosas (IoT, por sus siglas en inglés). En este proyecto se aborda la simulación de varios algoritmos epidémicos, desarrollados previamente y pensados para este tipo de redes, a fin de comprobar su competitividad y analizar su respuesta de manera práctica, sometiéndolos a distintas pruebas.

En primer lugar, se analiza el marco tecnológico de este trabajo, comenzando por profundizar en las redes propuestas, los algoritmos epidémicos y los protocolos relacionados con ellos que existen hoy en día; y concluyendo con una descripción de los simuladores de redes y procesadores de datos, poniendo el foco en Peersim y R, respectivamente (al ser los utilizados en este trabajo). Seguido de esto, se describe el código implementado en la programación de los protocolos y la preparación de todo el entorno para su posterior simulación. Una vez acabado de revisar el código, se analizan las ejecuciones realizadas con el simulador, comparando los diferentes algoritmos entre sí y con otros algoritmos de referencia, con el fin de extraer conclusiones que se contrastarán con las previsiones teóricas. Se incluye un apéndice en el que se describen todas las herramientas utilizadas durante el trabajo, junto con la justificación de su uso.

Tras todo este proceso, los resultados obtenidos de comparar los resultados de las diferentes simulaciones indican que los algoritmos se comportan como se esperaba en la teoría, demostrando sus capacidades de ahorro de memoria y energía sin afectar sobremanera al rendimiento temporal.

Simulation, analysis and evaluation of epidemic algorithms

KEYWORDS

Epidemic algorithm, simulation, Peersim, low-capacity equipment networks, R, memory and energy optimization.

ABSTRACT

The main objective of any network is the exchange of information between its different elements. Currently, the great existing demand forces the networks to increase their capacity day by day to be able to exchange more and more information and in a faster way. However, it does not stop there, as the demands are expected to continue to increase. To meet all this demand, not only must the capacity of the equipment and the links be increased so that they can support more information, but the algorithms that govern the logic behind the exchange of this information must also be polished.

Among the many protocols that exist, epidemic (or gossiping) ones stand out for using the mathematical models behind the spread of rumors or epidemics (they are very similar) to spread information among all elements of the network. These protocols are focused on achieving more efficient diffusion without losing reliability. In addition, they are highly fault tolerant and flexible.

On the other hand, networks with low-capacity equipment, such as sensor networks, are currently the target of many research projects, due to their relationship with cutting-edge technologies such as the Internet of Things (IoT). This project deals with the simulation of several previously developed epidemic algorithms designed for this type of networks, in order to verify their competitiveness and analyze their response in a practical way, subjecting them to different tests.

In the first place, the technological framework of this work is analyzed, beginning by delving into the proposed networks, the current epidemic algorithms and their related protocols; and conclude with a description of the network simulators and data processors, focusing on Peersim and R, respectively (as used in this work). Next, the code that implements the protocols is described. Then, the preparation of the entire environment for later simulation is depicted. Once the code has been reviewed, the executions carried out with the simulator are analyzed, comparing the different algorithms with each other and with reference algorithms, in order to draw conclusions that will be contrasted with the theoretical forecasts. An appendix is included which describes all the tools used during the job and why they were chosen. An appendix is included in which all the tools used during the work are described, together with the justification for their use.

After all this process, the results obtained by comparing the results of the different simulations indicate that the algorithms behave as expected in theory, demonstrating their ability to save memory and power without greatly affecting performance.

Índice general

Índice de figuras

Siglas y acrónimos

5G Fifth generation.

BRITE Boston University Representative Internet Topology Generator..

GNU GNU's Not Unix.

IDE Integrated Development Environment.

IEC International Electrotechnical Commission.

IoT Internet of Things.

ISO Internacional Organization for Standardization.

NoSQL Not Only SQL.

P2P Peer-to-peer.

RFID Radio Frequency Identification.

SI Susceptible-Infected.

SIR Susceptible-Infected-Removed.

SIS Susceptible-Infected-Susceptible.

SQL Structured Query Language.

UML Unified Modeling Language.

Capítulo 1

Introducción

1.1. Contexto y motivación

Al comparar las epidemias con los rumores, podríamos decir que tienen muchas cosas en común y esto se basa, sobre todo, en que la lógica que rige la expansión de ambos es extremadamente similar: en un corto periodo de tiempo, distribuir algo, ya sea un rumor o una enfermedad, entre todo el entorno a través de ir enviándolo vecino a vecino.

Si nos fijamos en estas características (sencillez, robustez, escalabilidad, rápida expansión, descentralización, etc.), nos damos cuenta de que son ideales para utilizarlas en entornos de redes, y es precisamente a raíz de este hecho donde nacen los protocolos epidémicos que distribuyen información siguiendo los patrones y modelos matemáticos definidos por las epidemias y los rumores [3]. Más concretamente, estos destacan en la diseminación de información en redes entre pares o P2P (*peer-to-peer*, en inglés) [4].

Sin embargo, esta no es la única área donde se aplican estos protocolos. Debido a sus características, también se utilizan, por ejemplo, para la agregación de datos, control de errores, generación de topologías para redes superpuestas o gestión de membresías [5]. Estos protocolos están en auge debido a la gran expansión y aumento de complejidad de las redes distribuidas [6].

Los algoritmos epidémicos se pueden clasificar según los tipos de mensajes que intercambian los nodos, lo que genera tres grupos simples (*Eager push*, *Lazy push* y *Pull*, por sus nombres en inglés) que luego pueden ser combinados para potenciar sus ventajas. También poseen tres propiedades que se pueden ajustar para conseguir distintos objetivos y funciones con respecto a la red en la que se aplican. Entre ellas, cabe destacar la frecuencia de las rondas, qué información y a qué número de vecinos se envía dicha información en cada ronda, y sus características de terminación.

Como hemos podido ver, los protocolos epidémicos tienen un amplio abanico de posibilidades para poder desarrollarse, ya que su aplicación no está restringida a unos ámbitos de aplicación o entornos concretos. Por lo tanto, existe mucho aún por investigar y mejorar en este campo.

Gracias a las características que poseen estos protocolos, pueden ser muy eficaces a la hora de distribuir información en redes con bajas capacidades, es decir, con equipos muy limitados en recursos y procesos no fiables. Además, se ha aumentado el interés en este tipo de redes debido a su relación con tecnologías como la 5ª generación de telefonía

móvil (5G) [7] o el Internet de las Cosas, *Internet of Things* (IoT), en inglés [8]. La motivación de este proyecto se basa en encontrar algoritmos que sean competentes en el ámbito de estas redes, abriendo nuevas posibilidades a protocolos epidémicos.

1.2. Objetivos

Este trabajo forma parte de una investigación de mayor alcance, cuyo objetivo es la creación de algoritmos que sean simples y eficientes en cuanto al uso de recursos (tanto memoria como energía), para que puedan ser implantados en redes de escasa capacidad. De las características vistas en la sección anterior, se pretende reducir la frecuencia y disminuir la cantidad de información y vecinos a los que se envía dicha información en cada ronda, sin que esto produzca retrasos problemáticos en el proceso. Además, se busca que los procesos logren la quiescencia (estado de reposo) lo antes posible después de que la red alcance el conocimiento máximo (la información se haya propagado al mayor número de nodos posible).

El objetivo concreto que tiene este trabajo es la simulación y el análisis de los resultados obtenidos, de varios algoritmos epidémicos que ya han sido formulados [2], y compararlos, además, con algún algoritmo de referencia comprobando su utilidad. Para ello, se va a utilizar el simulador de eventos discretos Peersim [9]; más concretamente, una versión que añade mejoras en el rendimiento y la usabilidad [10]. El proceso se divide en las siguientes fases:

1. Preparación y familiarización con el simulador y los algoritmos.
2. Implementación del código necesario para simular los algoritmos, utilizando las herramientas ofrecidas por el marco de trabajo del simulador.
3. Estudio de los parámetros más interesantes para realizar las simulaciones.
4. Procesado de los datos obtenidos para poder analizarlos posteriormente de manera más eficaz.
5. Análisis de los resultados y comparación de dichos datos entre sí y con otros algoritmos similares.
6. Extracción de conclusiones finales de todo el proceso.

1.3. Estructura del documento

En esta sección se realizará un breve resumen sobre el contenido de los diferentes capítulos que conforman dicha memoria con el objetivo de hacerle más fácil la movilidad por esta al lector.

En primer lugar, el Capítulo 1 introduce el contexto de todo el proyecto, la motivación y los objetivos, tanto los generales del proyecto como los de este trabajo concreto, finalizando con esta breve descripción de la estructura del documento.

A continuación, el Capítulo 2 presenta las tecnologías involucradas en el desarrollo de este trabajo, describiendo los aspectos más relevantes. Entre ellas se incluyen las redes de equipos de bajas capacidades, los algoritmos epidémicos, los simuladores y los procesadores de información.

Luego, el simulador y todo el código implementado para realizar las simulaciones es descrito en el Capítulo 3, tanto el necesario para crear el entorno de simulación, como el propio de los protocolos que implementan a cada algoritmo, concluyendo con la descripción de un protocolo de difusión por inundación (*broadcast*, en inglés) utilizado como referencia en las comparaciones.

El Capítulo 4 detalla las simulaciones realizadas con los parámetros de configuración específicos elegidos, se habla sobre el procesado que sufren los datos resultantes de estas, y se comparan las gráficas generadas tras el procesado, comentando y analizando los resultados obtenidos.

Por último, el Capítulo 5 expone las conclusiones generales de las simulaciones y de todo el trabajo realizado, añadiendo consejos para trabajos similares y proponiendo futuras líneas que se pueden seguir.

Tras este Capítulo se incluye el Anexo A, el cual menciona todas las herramientas que se han utilizado durante todo el trabajo, incluido el desarrollo de esta memoria, añadiendo las justificaciones correspondientes del uso de las que no han sido explicadas en otros capítulos.

Capítulo 2

Estado del arte

En este capítulo vamos a ver el estado de las tecnologías y conocimientos necesarios para la realización de este trabajo. El bloque está dividido en cuatro secciones: en primer lugar, un repaso a las redes de capacidades limitadas, para centrar el contexto; seguido por los protocolos epidémicos, desde su inicio a partir de modelos matemáticos hasta las aplicaciones más comunes; tras esto, los simuladores de redes, enfocándose en Peersim [9] (el utilizado en el trabajo) y, por último, las herramientas de procesado de datos, destacando el lenguaje R [11] entre ellas.

2.1. Redes de bajas capacidades

Al hablar de redes con bajas capacidades, estamos refiriéndonos a redes en las que los equipos que las componen tienen recursos muy limitados, por lo que los procesos que ejecuten tienen que tener en cuenta esta desventaja. Estos procesos suelen ser muy simples, de forma que no impliquen almacenar grandes cantidades de información en memoria (ya que muchas veces no se dispone de ella) y muy optimizados en el uso de la energía. Además, es muy probable que la fiabilidad de la red no sea muy alta, tanto por parte de los equipos como de los enlaces, por lo que los procesos también deben ser capaces de lidiar con estos problemas. Ejemplos de este tipo de redes pueden ser las redes de sensores o las redes de dispositivos RFID [12].

Las redes de sensores [13] quedan definidas por su propio nombre. Son redes en las que los nodos son sensores que miden algún parámetro del entorno y lo comparten con la red. Los sensores son equipos muy básicos y no tienen muchos recursos. Sin embargo, juntos pueden realizar procesos relativamente complejos. Es muy común encontrarse actuadores en este tipo de redes, para realizar funciones derivadas de la información obtenida por los sensores. Un ejemplo son los sensores de humedad en un cultivo, que cuando detectan que la humedad está por debajo de un límite, activan los aspersores.

Por otro lado, están las redes de dispositivos RFID [14] o identificación por radiofrecuencia (*Radio Frequency IDentification*, en inglés). Esta tecnología se basa en la comunicación inalámbrica de dos partes: un lector que lee y procesa información, y un emisor que almacena y emite el identificador. Se puede asemejar a los códigos de barras, donde el lector de códigos recibe la imagen de la etiqueta que identifica el artículo.

2.2. Protocolos epidémicos

Como se ha comentado con anterioridad, estos protocolos parten de los modelos matemáticos y algoritmos obtenidos de la expansión de epidemias, siendo éste su origen. Antes de comentar las características de estos protocolos, la siguiente sección repasará los modelos matemáticos de los cuales provienen.

2.2.1. Modelos matemáticos

La matemática es una ciencia muy aplicada en todos los campos, ya que otorga muchas posibilidades nuevas de estudio al enmarcar conceptos y procesos en modelos y algoritmos que se puedan entender y estudiar de una manera más sencilla y desde diferentes perspectivas. Además, a la hora de generar y estudiar estos modelos, podemos encontrar similitudes con otros casos de uso completamente distintos pero que se comportan de manera similar, pudiendo aplicar soluciones de uno al otro para solventar posibles problemas [15]. Este es justo el caso en el que nos encontramos, aplicando modelos matemáticos de epidemias al entorno de las redes de comunicaciones.

Existe una rama de la epidemiología dedicada al estudio de modelos matemáticos que se ajustan a la expansión de epidemias, tanto para realizar predicciones y modelos de contingencia frente a futuras amenazas, como para la aplicación de estos en otros campos de carácter similar. De hecho, descifrar los misterios detrás de las epidemias, su expansión y su fin, ha sido objetivo de científicos del área desde hace mucho tiempo y los resultados obtenidos al aplicar las matemáticas al problema han sido muy satisfactorios [16, 17].

A pesar de que cada enfermedad tiene sus peculiaridades, existen modelos básicos que resultan muy fiables en la mayoría de los casos. Los modelos más avanzados exigen mecanismos más complejos y llegan a abarcar muchos casos particulares. Aunque los modelos pueden no generar resultados precisos sobre cómo va a evolucionar una enfermedad, otorgan una estimación sobre su desarrollo que puede ser de vital importancia. Para poder explicar estos modelos, es necesario hablar primero de los elementos que intervienen en el proceso.

Por un lado tenemos los vértices o nodos, que representan a cada individuo de la población, es decir, los portadores de la enfermedad (las personas, por ejemplo). Según su estado con respecto a la enfermedad, pueden ser de varios tipos:

- Susceptible o S (por *Susceptible*, del inglés): El individuo es susceptible de ser infectado por la enfermedad.
- Infectado o I (por *Infected*, del inglés): El individuo está infectado y puede transmitir la enfermedad, contagiando a otros.
- Eliminado o R (por *Removed*, del inglés): Este es un estado más complejo. El grupo lo forman los individuos que no transmiten la enfermedad y no van a poder ser infectados, ya sea porque han muerto, han adquirido inmunidad o están aislados.

Estos tres estados son los más básicos y son los usados en los modelos simples. En los más complejos, pueden aparecer nuevos estados como, por ejemplo, los individuos que comienzan presentando una inmunidad que tras un tiempo pierden (M), o los Expuestos (E de *Exposed*, en inglés), que están contagiados, pero no pueden transmitir la enfermedad ya que esta se encuentra en un estado de incubación.

Por otro lado, los ejes, que representan las conexiones que tiene cada individuo con otros. La enfermedad utiliza estos canales para infectar a más individuos, propagándose a través de ellos. Juntando estos dos elementos (los vértices y los ejes) podemos formar grafos que representan a la población, los cuales son muy útiles a la hora de visualizar la expansión de la enfermedad.

Una vez explicado esto, vamos a analizar rápidamente los modelos básicos utilizados, que son los más interesantes para el trabajo. Estos modelos realizan muchas simplificaciones de las poblaciones, pero forman la base de los modelos más complejos [18, 16].

Modelo SI

Es el modelo más simple de todos (Figura 2.1). Supone una población estática, sin nacimientos ni muertes, y en la que no hay recuperación de los individuos tras su infección.

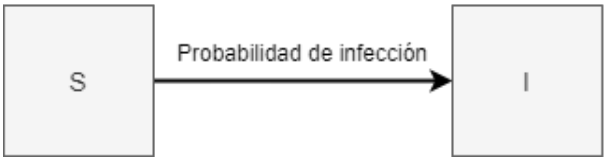


Figura 2.1: Modelo SI.

Las variables que entran en juego en este caso son los individuos de la población, las conexiones existentes entre ellos y la probabilidad de que un individuo contagiado transmita la enfermedad a uno sano en un contacto. Estos parámetros otorgados por cada enfermedad que se quiera analizar con este modelo (porque encaje con la descripción del mismo) son los que se van a añadir a las ecuaciones diferenciales propuestas para obtener las gráficas del desarrollo de la misma.

Se parte de una población, con un grupo de individuos susceptibles y otro grupo de infectados. En función de las conexiones y la probabilidad de contagio, el grupo de individuos susceptibles va perdiendo miembros al ser contagiados, pasando a pertenecer al otro grupo. A largo plazo, todos los individuos de la población estarían infectados (exceptuando los aislados), hecho que ocurriría siempre. Esta incongruencia se debe a las simplificaciones planteadas por este modelo.

Modelo SIS

Partiendo del modelo anterior, este nuevo modelo (Figura 2.2) añade una condición adicional: que los individuos infectados puedan recuperarse de la enfermedad, pasando de nuevo a ser susceptibles. Por lo tanto, si comenzamos en la misma situación que el anterior, con los dos grupos de individuos S e I, éstos irán saltando de un grupo a otro, en función de la probabilidad de contagio y las condiciones para la recuperación. Dependiendo de los valores de estos dos parámetros la enfermedad puede evolucionar hasta diferentes estados, como puede ser la extinción de la enfermedad o un equilibrio fijo en el número de infectados y susceptibles.

Figura 2.2: Modelo SIS.

Modelo SIR

Este modelo (Figura 2.3) añade un nuevo estado a los individuos. Parte del planteamiento del anterior modelo, de que tras un periodo infectados, los individuos dejan de estarlo pero, en este caso, en vez de volver al estado S, llegan al estado R, ya descrito, en el cual no van a infectar a otros ni van a poder ser infectados, porque tras superar la enfermedad desarrollan una inmunidad a la misma, o por la propia muerte del individuo causada por la enfermedad.

Figura 2.3: Modelo SIR.

Como hemos dicho, estos son los modelos más simples, que sientan las bases para definir el resto, como pueden ser los modelos SIRS, SEIR, SEIS y MSIR, o incluso ampliaciones de los modelos SIS y SIR que contemplan condicionantes adicionales. Estos modelos van añadiendo más características que suponen, en algunos casos, nuevos estados de los individuos [18].

Tras esto, se procede a desarrollar la aplicación de estos modelos sobre las redes de comunicaciones, repasando los orígenes de estos protocolos y dando algunos ejemplos.

2.2.2. Descripción

Como ya hemos comentado previamente, estos protocolos son los que utilizan los modelos matemáticos y los algoritmos de la expansión de las epidemias, que a su vez son muy similares a los obtenidos de la expansión de rumores. Es por esto que, a la hora de hablar sobre ellos, aparecen términos de ambos mundos (como pueden ser los nodos infectados o los rumores), además de los propios de las redes. Como podemos ver, ésta es una definición muy vaga, que no fija ningún tipo de límite ni restricción. Esto es así porque, más que un marco físico en el que se incluyen protocolos con características similares, lo que comparten estos protocolos es la filosofía que persiguen. No obstante, tras analizar las similitudes entre diferentes protocolos catalogados como epidémicos, se pueden extraer algunos aspectos clave de los mismos [5]:

- Intercambios de información entre dos nodos, teniendo en cuenta que la información

puede viajar en los dos sentidos,

- de manera repetida, pues se producen muchos intercambios de información a lo largo del tiempo,
- y eligiendo a los vecinos con los que se va a intercambiar información de manera aleatoria, sin seguir un modelo determinista.

El área de aplicación más común de estos algoritmos (y, como veremos más adelante, su primer área) es la diseminación de información en redes entre pares (*peer-to-peer*, en inglés) [4, 19, 3, 20]. Otras áreas en las que destacan estos algoritmos son: la agregación de datos [3], la detección de errores, la construcción de redes superpuestas y la gestión de las membresías de la red, entre otras [6]. Como vemos, estos algoritmos destacan a la hora de distribuir información a lo largo de toda la red, sobre todo en redes en las cuales no se pueden aplicar los protocolos tradicionales debido a su gran tamaño, su dinamismo o su propensión a errores, gracias a la resiliencia que le otorga el alto grado de redundancia y a que no necesitan conocer la topología completa de la red.

El modelo que tienden a seguir todos los protocolos es el SIR (Figura 2.3), funcionando de la siguiente manera: la información que se quiere transmitir representaría a la enfermedad, por lo que los nodos que la conozcan son los del estado I; los que aún no la conocen, S; y se llega a R una vez que se finaliza el proceso. No obstante, utilizar este modelo no es una condición obligatoria, pudiéndose utilizar el que mejor se adapte a las características propias de cada caso.

2.2.3. Surgimiento y desarrollo

Antes de continuar hablando a cerca de las características de estos protocolos, vamos a dedicar un momento a repasar su origen, el motivo por el cual se desarrollaron y la función que cumplían, para luego ir avanzando en su desarrollo y ver cómo se han ido adaptando a más entornos [5].

En [21], Demers et al. describieron por primera vez los algoritmos epidémicos. Ahí proponían una solución para diseminar la información de una red distribuida de servidores de bases de datos que, eventualmente, lograba la consistencia, si no se producían actualizaciones, garantizando que todas las bases de datos tuvieran la misma información.

Siguiendo por esta línea, en la década de los 90 se aplicaron estas ideas al campo de la distribución por inundación de información (*broadcast*, en inglés), solucionando los problemas que se planteaban en primera instancia y optimizando los procesos. A comienzos de este siglo, las áreas de aplicación de estos protocolos aumentaron enormemente gracias a las características que poseen estos algoritmos, pero no se les daba demasiada importancia. Con el auge de los sistemas distribuidos, cada vez más exigentes, ha sido necesario revisar todas las tecnologías disponibles para solventar los problemas dispuestos, entre las que se encontraban estos algoritmos, por lo cual han tenido un resurgimiento. Actualmente, se consideran una tecnología madura en algunos ámbitos [6].

2.2.4. Clasificación y ejemplos

Tras este inciso para repasar su historia, volvemos con las características de estos protocolos, pasando ahora a hablar de su clasificación. Según el tipo de mensaje que se intercambien los nodos de la red, se pueden clasificar a los protocolos en varios grupos diferentes que vamos a definir a continuación [22].

Eager push

Los nodos van a emitir directamente, en cuanto reciban la información, los mensajes a sus vecinos, sin esperar a que estos la pidan. Estos mensajes van a contener la información al completo. Comparado con los otros, este produce más tráfico redundante, pero su latencia es menor. Además, requiere un menor uso de memoria por parte del nodo.

Pull

En este caso, los nodos están enviando periódicamente a sus vecinos que le informen de la nueva información que posean. En cuanto el nodo detecta información desconocida para él, pide a ese vecino que se la envíe. Esto reduce el tráfico redundante, pero aumenta los tiempos de latencia con respecto al anterior y necesita más capacidad de memoria.

Lazy push

Este es una mezcla de los dos anteriores, ya que los nodos, según reciben la información, van a emitir un mensaje a sus vecinos. Pero este mensaje va a contener solo el identificador de dicha información. Si algún vecino recibe un identificador desconocido, pedirá al nodo que le envíe la información completa. Las características de este grupo son muy similares a las del anterior, disminuyendo el tráfico redundante y aumentando la latencia y el uso de memoria frente al primero.

A partir de estos tres tipos, se pueden crear nuevos, mezclándolos entre ellos de la manera más adecuada para sacar el máximo partido a las características particulares de cada uno. Vamos a definir ahora dos de los más destacados, pero las combinaciones son muy amplias, según las necesidades de cada caso.

Eager push and Pull

El proceso se divide en dos partes, empezando con el tipo *Eager push* con la intención de diseminar la información lo más rápido posible a los máximos nodos, para finalizar con el *Pull* para que los nodos que aún no conozcan esta información, la consigan. Esta última fase decrementa la redundancia de mensajes, sin afectar a penas a la eficiencia del algoritmo.

Eager push and Lazy push

Cada nodo comienza emitiendo su información a sus vecinos con el estilo *Eager push* hasta que llega a un límite marcado. Al resto de vecinos los entrega la información con el método *Lazy push.*

Parámetros generales

Otra manera de diferenciar este tipo de algoritmos es según las redes a las que se deben aplicar, por las restricciones que cada uno impone. Las tres más comunes son el tipo de sincronía, la fiabilidad y el conocimiento de la membresía. Según su sincronía, podemos tener sistemas síncronos (cuando las velocidades de todos los procesos son conocidas y están sincronizadas), asíncronos (cuando no lo están) o parcialmente síncronos (cuando las velocidades son conocidas, pero no están sincronizadas). Hablando de fiabilidad, se distinguen dos tipos, de enlace y de proceso. La fiabilidad de proceso implica que los procesos fallen (no fiables) o no fallen (fiables). Con respecto a la fiabilidad de enlace, el

caso es similar, teniendo enlaces fiables si no pierden mensajes en el proceso de transmisión y no fiables o con pérdidas si sí que lo hacen. Por último, el conocimiento de la membresía es el conocimiento que tiene un proceso del resto de procesos de la red, pudiendo ser total o parcial. El conocimiento total se suele dar en redes pequeñas, ya que es muy costoso de escalar, y suele implicar que un proceso tenga enlaces directos contra el resto. En el conocimiento parcial solo se conoce a los procesos directamente conectados, llamados comúnmente vecinos.

Los algoritmos epidémicos logran variar las características mencionadas mediante el ajuste de tres parámetros: la frecuencia de los procesos de envío, la selección de información y vecinos a los que se va a notificar en un proceso de envío, y las condiciones de terminación.

Una vez revisadas algunas de las clasificaciones más comunes de algoritmos epidémicos, se exponen a continuación algunos ejemplos reales de estos protocolos.

Ejemplos

En primera instancia, estos algoritmos se desarrollaron para replicar la información en las bases de datos distribuidas, por lo que tienen un largo recorrido en el campo de la diseminación de información, acumulando una gran cantidad de implementaciones a lo largo de todo ese tiempo. Podemos destacar las aplicaciones de compartición de archivos, siendo el caso más famoso BitTorrent [23], las cuales crean redes entre pares (*peer-to-peer*, en inglés) que intercambian información entre sus miembros de manera aleatoria. En arquitecturas NoSQL (sistemas de gestión de bases de datos que no utilizan SQL) se aplican estos algoritmos en favor de conseguir una mejor escalabilidad, como es el caso de Cassandra [24], incluso en algunos centros de datos de proveedores *cloud* parece (no se está seguro debido a que no siempre lo publican oficialmente) que utilizan este tipo de algoritmos, como es el ejemplo de Amazon [25]. Además de estos ejemplos, podemos añadir multitud de algoritmos que se engloban dentro de este grupo, como pueden ser HyParView [22], Narada [26] o NeEM [27], entre muchos otros.

Pasamos ahora a hablar de las simulaciones, lo que son, lo que conllevan y las herramientas necesarias para llevarlas a cabo, centrándonos en el simulador que vamos a utilizar.

2.3. Herramientas de simulación

Las simulaciones son una pieza fundamental del desarrollo de cualquier tecnología, ya que nos permiten comprobar de manera práctica su funcionamiento sin necesidad de montar el escenario para el que se va a aplicar; esto supone un gran desembolso económico, y se amplifica cada vez más según aumenta el número de elementos involucrados en el proceso. Pero actualmente existen multitud de simuladores disponibles, y elegir el que mejor se adapte a un determinado entorno, puede llegar a ser una ardua tarea.

Un tipo específico de simuladores son los simuladores de eventos discretos [28]. Estos funcionan a través de eventos que van ocurriendo en el tiempo sin necesidad de que exista un flujo continuo de los mismos, de tal manera que se encolan los eventos para los instantes de tiempo que se requiera y solo se procesan esos eventos en orden de tiempo, siendo ignorados los instantes que no tengan eventos (Figura 2.4). Este tipo de simuladores destaca para las simulaciones de redes y protocolos, generando eventos para simular el envío y recepción de mensajes entre los nodos. Entre los más utilizados se encuentran OMNeT++ [29], NS-2 [1] (o su nueva versión NS-3 [30]) y Peersim [9].

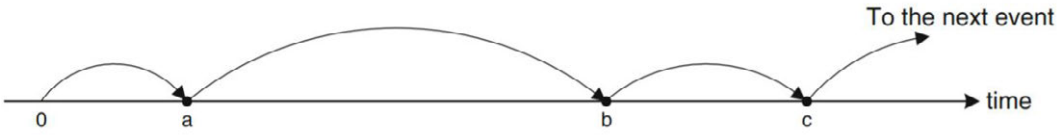


Figura 2.4: Simulador de eventos discretos [1].

De cara a elegir un simulador, es muy importante dedicarle el tiempo suficiente para elegir el adecuado, ya que puede suponer una gran diferencia de tiempo y esfuerzo el usarlo o no usarlo. Afortunadamente, para ayudarnos en esta tarea, existen estudios que comparan entre los simuladores más destacados, teniendo en cuenta parámetros como el rendimiento, el consumo de memoria, la facilidad de uso, la adaptabilidad y la funcionalidad, entre otros [31, 32], incluso existe una familia de normas ISO, la ISO/IEC 25000 [33], que define un modelo de calidad y evaluación de herramientas *software*.

En el caso de este trabajo, las características que debe tener el simulador son: un marco de trabajo simple, que permita operar de manera sencilla sobre él; alta flexibilidad de cara a la implementación, para poder añadir las peculiaridades de cada algoritmo y programar todas las pruebas necesarias para extraer la información que se requiera; y un entorno que se asemeje a la realidad para que sea más sencilla la comparación y comprensión.

2.3.1. Peersim

Peersim [34, 9] es un simulador de eventos discretos, entre pares y de *software* libre que está enteramente programado en el lenguaje de programación Java [35]. Una de sus características principales que lo hace destacar por encima de sus competidores es la capacidad que tiene para simular redes de este estilo con grandes cantidades de nodos sin que afecte al rendimiento de manera exponencial. Posee dos motores diferentes, uno basado en el manejo de eventos para ejecuciones asíncronas, que es más realista, y otro basado en ciclos, que abandona todo lo relacionado con el tiempo y simplifica los procesos para optimizar al máximo la ejecución. El hecho de que esté programado en un único lenguaje, y que este lenguaje sea Java (uno de los lenguajes de programación más conocidos y utilizados) nos permite mayor sencillez a la hora de trabajar sobre él. Sin embargo, el principal problema de este simulador es su interfaz de cara al usuario, que es algo tosca y difícil de usar debido a dos problemas fundamentalmente: la ausencia de interfaz gráfica y la simplificación de las interfaces base ofrecidas para la codificación de los modelos del usuario, que aunque le otorga más flexibilidad, deja encargado a este de programar parte de la lógica de bajo nivel. Esta simplificación es producida porque las interfaces deben poderse utilizar para los dos motores que posee el simulador.

Al estar programado en un solo lenguaje, de nuevo nos proporciona otra ventaja adicional, y es que las actualizaciones y mejoras por parte de los usuarios son más fáciles de implementar, incluso existe un portal en la página oficial del software donde están recopiladas algunas de ellas [36].

Una de estas modificaciones del simulador es la realizada por Fernando Díez [10], que pretende mejorar el problema particular que tiene este: la interfaz de cara al usuario. Para ello redefine las clases, ordenándolas por paquetes según su funcionalidad, dando al usuario una experiencia más cómoda de cara a que desarrolle sus modelos. También incluye el paquete BRITE [37], que es un marco de trabajo para la generación de redes y topologías, incluyendo herramientas para importar y exportar en diversos formatos.

Implementa algunos generadores y formatos de red básicos, pero otorga interfaces para codificar uno propio en caso de ser necesario.

Además, esta modificación implementa una mejora en el motor de simulación por eventos discretos, mediante la modificación de su estructura de almacenamiento de eventos para optimizar las acciones de encolado y desencolado, con el objetivo de obtener el máximo partido de este. El proceso concluye con un simulador más eficiente.

2.4. Procesadores de datos

Por último, en este capítulo se habla del procesado de datos, como paso posterior a realizar las simulaciones, analizando todo el proceso y dando algunos ejemplos, para centrarse finalmente en R [11].

Interpretar los resultados generados por las simulaciones puede ser algo tedioso, dependiendo de la cantidad de información que se extraiga y del formato con el que se represente. En casos donde la información es abundante y los datos obtenidos no son fácilmente legibles, se utilizan procesadores para obtener dicha información y agruparla adecuadamente. Con esto, los datos van a poder mostrarse en forma de gráficas que faciliten la labor de análisis de las personas. De hecho, es fundamental realizar estas gráficas, porque pueden revelar información que de otra manera sería imposible de obtener [38].

Como se puede observar, hay dos partes claramente diferenciadas. Por un lado, el procesado de la información del fichero de salida a un formato concreto, que entienda el graficador elegido (por ejemplo, agrupada en tablas). Esta parte suele estar integrada en el simulador pero no siempre está expresada en el formato que se necesita, por lo que se requiere de un transformador del formato (*parser*, en inglés). Por otro lado, se utiliza el graficador para expresar en gráficas esa información recogida.

Existen multitud de herramientas para realizar estos procesos, incluso puede que el propio simulador otorgue algunas facilidades para ello. Pero lo más común suele ser utilizar lenguajes de programación para el procesado de la información, junto con algún graficador de tablas, que a veces está integrado en el propio lenguaje a través de algún paquete. Los simuladores más usados suelen tener formatos estándar o herramientas específicas dedicadas [39].

A la hora de elegir las herramientas a utilizar, esta decisión no suele tener mucha relevancia en la duración del procesado de los datos, a no ser que se esté tratando con cargas elevadas de información, lo cual puede ralentizarlo enormemente. En estos casos sí es recomendable elegir alguna herramienta más especializada (y, sobre todo, optimizada) en el procesado de datos, como pueden ser los lenguajes de programación C [40], Fortran [41] y R [11].

Una parte muy importante que no puede pasar desapercibida, aplicada también a las simulaciones, es el hecho de realizar el proceso de forma ordenada, teniendo claro cual es el objetivo (los datos que quieres obtener y comparar). Esto ayuda a centrarse en el problema y obtener las conclusiones mucho más rápido, pero no implica que no se puedan realizar cambios, ya que a veces se obtienen resultados inesperados.

2.4.1. R

R [11] es un entorno de programación con su propio lenguaje (del mismo nombre) gratuito y de *software* libre orientado al análisis y procesado de datos estadísticos, que cuenta con potentes herramientas de cálculo y graficado. Es ampliamente conocido y cuenta con

una gran comunidad de usuarios, por lo que posee multitud de librerías que amplían su funcionalidad básica [42], incluso optimizando ciertas carencias del lenguaje. Fue creado en 1996 por Ross Ihaka y Robert Gentleman [43] a partir del lenguaje de programación S [44], y actualmente forma parte del proyecto GNU [45] de protección del *software* libre.

Las dos funciones principales por las que destaca es como *software* de análisis estadístico y como generador de gráficos. Por otro lado, uno de sus puntos más débiles es la velocidad, ya que el lenguaje de programación no es el más óptimo. Para solventar este problema, existen paquetes que han reprogramado las funciones más críticas y usadas para que sean ejecutadas en otro lenguaje (como C, por ejemplo), y posteriormente devuelvan el resultado a R.

En general, R [11] es una buena elección, simple y flexible, que deja a tu disposición muchas facilidades y herramientas (gracias a la gran variedad de paquetes con que cuenta), ya sea para utilizarlo como procesador de datos y estadísticas o como generador de gráficos. Solo en el caso de buscar el procesado más óptimo de la información, R [11] pierde valor frente a sus competidores, pero existen medios para optimizar los procesos mediante otros lenguajes de programación.

Capítulo 3

Código implementado

Como se ha comentado anteriormente en los objetivos, este trabajo se centra en la simulación y análisis de los algoritmos desarrollados por el grupo de investigación [2], para comprobar su funcionamiento en la práctica y comparar su utilidad frente a otros protocolos similares que intenten cumplir las mismas funciones.

El simulador utilizado para esta tarea es Peersim [9], más concretamente, una versión de este desarrollada por Fernando Díez [10], que es más eficiente y otorga un marco de trabajo más amigable. La decisión de elegir este simulador en concreto frente a otros reside en el potencial que tiene Peersim para simular redes con un gran número de nodos, sin que esto afecte al rendimiento de manera exponencial, y a la flexibilidad que ofrece de cara a programar protocolos y modelos de red sobre él. También es importante el hecho de que realiza la simulación de una manera realista, mediante clases que representan a los elementos presentes en las redes físicas. Que esté programado enteramente en Java hace que éste sea el único lenguaje necesario para poder utilizar y modificar el simulador, lo que supone otra gran ventaja.

Antes de realizar las simulaciones, se debe preparar al simulador con las particularidades propias requeridas, mediante la implementación de líneas de código. Este proceso se divide en tres apartados: en primer lugar, se crea el entorno, tanto la topología de la red como los nodos y los enlaces. Para ello, se usan las clases predefinidas por el simulador o se crean clases propias implementando las interfaces que Peersim ofrece. Después, se programan los protocolos, que contienen todos los procesos de los nodos para el tratamiento de la información y los mensajes (o paquetes), creándolos cuando sea necesario y distribuyéndolos por el enlace correspondiente. Por último, se implementa la extracción de la información necesaria para realizar posteriormente el análisis. Esto se puede llevar a cabo a través de la herramienta de registrado de información del simulador, la cual, mediante una clase estática y una interfaz, permite obtener de manera sencilla un fichero de registro, resultado de la ejecución, clasificado en niveles y con la información que se le entregue.

Referidos a las clases desarrolladas sobre el programa para poder ejecutar la simulación, se ha creado un paquete, llamado basicEnvironment, que posee la codificación del entorno y algunas clases abstractas para facilitar el desarrollo de cada algoritmo, teniendo en cuenta que se implementa cada uno en un protocolo distinto, dentro de su propio paquete. En el interior del paquete basicEnvironment existen 5 subpaquetes divididos por funcionalidad, la cual vamos a describir a continuación, incluyendo a la clase situada en el paquete raíz:

- raíz: en la raíz del paquete se sitúa una única clase, de nombre Main, que contiene

tanto la configuración del entorno previa al comienzo de la simulación como la generación de la topología y la instanciación de los objetos e implementa a la interfaz Control ofrecida por el simulador. Es una clase abstracta, para que cada algoritmo instancie su propia red.

- edge: contiene las clases para implementar la funcionalidad de los enlaces entre dos nodos. La clase principal implementa la interfaz Edge otorgada por Peersim.
- node: contiene las clases para implementar la funcionalidad de la red, tanto de los nodos en particular como la topología en general. Estas funcionalidades están desarrolladas por el simulador, por lo que se debe heredar de dichas clases (Node y GraphNetwork).
- packet: aquí se almacenan las clases que definen al objeto intercambiado entre los nodos (paquete o mensaje), incluyendo los datos y las cabeceras. La clase principal va a ser el objeto que se van a intercambiar los nodos de la red, por lo que debe seguir el formato establecido por el simulador.
- protocol: en este paquete se almacenan las clases correspondientes a los protocolos que van a ejecutarse en cada nodo. Son los responsables de la lógica de creación de mensajes (eligiendo el destino que corresponda) y envío de los mismos a través del enlace correspondiente, o procesado en el caso de que se reciba. Todos los protocolos deben implementar las interfaces globales otorgadas por Peersim para que puedan ser integrados en los nodos.
- statistics: este paquete contiene las clases necesarias para recabar información de la red a nivel global.

Por otra parte, los paquetes referidos a los algoritmos tienen por debajo una clase en el paquete raíz y dos subpaquetes, node y protocol, cuya funcionalidad es la ya comentada. La clase codificada en el subpaquete protocol hereda de la clase abstracta BrainProtocol (creada en el paquete basicEnvironment) e implementa la lógica del algoritmo en cuestión. La clase situada en la raíz del paquete y las del subpaquete node, heredan de la clase principal del paquete basicEnvironment situada en su raíz y de las clases del subpaquete node de basicEnvironment, respectivamente. En estas clases de los paquetes de los algoritmos se codifica la instanciación de los objetos propios de cada uno. Además, los algoritmos que lo necesiten pueden extender las funcionalidades ya implementadas en el paquete basicEnvironment, como es el caso del algoritmo 2.

Antes de explicar el código que se ha desarrollado para ejecutar las simulaciones, vamos a hablar de los parámetros de entrada (para configurar la ejecución) y los ficheros de salida (para obtener resultados y poder analizarlos) del simulador.

3.1. Uso del simulador

3.1.1. Parámetros de configuración

Los parámetros de entrada de la simulación son configurados en un fichero de texto plano que se pasa como único parámetro a la hora de ejecutar el programa (Figura 3.1). En este se pueden introducir todos los parámetros que se necesiten (uno en cada línea del fichero) con el siguiente formato:

'etiqueta del parámetro' 'valor del parámetro'

Es importante que el único espacio en blanco que haya sea entre la etiqueta y el valor.

Existe la posibilidad de que no se respete este formato, apilando varios parámetros en una línea, pero esto se usa para realizar varias simulaciones en una sola ejecución, característica que no se utiliza en el desarrollo de este trabajo.

```
init.algorithm alg0Broadcast.BC0Main
levelsFile misNiveles
logConfigurationFile outputConfiguration
simulation.endtime 10000000
timeUnit MICROSECONDS
waitTime 4000
statisticWaitTime 5000
bandWidth 1000000
network.numNodes 1000
network.numCorrects 950
network.conexionDensity 0.01
network.startTime 200
network.areaSize 100
network.subAreaSize 100
failType BOTH
failChance 0.005
failNumPackets 20
failTime 0
```

Figura 3.1: Fichero conf.txt.

Los parámetros mínimos requeridos por el simulador para que se pueda ejecutar son:

- **init.'identificador'**: se le indica el paquete y el nombre de la clase que implemente a la interfaz Control de Peersim, en este caso, las que hereden de Main. En esta clase se espera que se realice la configuración de parámetros previos al inicio de la simulación (como la generación de la red y la instanciación de todos sus elementos). El identificador es el nombre que se le da a esa simulación.
- **levelsFile**: ruta al fichero de los niveles de clasificación que utilizará la herramienta de registro de información.
- **logConfigurationFile**: ruta al archivo que filtra el fichero de registro de la ejecución.
- **simulation.endtime**: tiempo máximo que durará la simulación. Con añadir un número al parámetro no es suficiente para que esta termine, se debe ejecutar un evento de finalización para que lo haga realmente.
- **timeUnit**: unidad en la que se medirá el tiempo de la simulación. Este parámetro es opcional y por defecto se establece en segundos.

Además de estos parámetros, se han añadido los siguientes para realizar las modificaciones necesarias sobre la red:

- **waitTime**: unidades de tiempo que espera el protocolo para volver a enviar un mensaje, es decir, la frecuencia de los ciclos de envío. Nombrado también como τ.
- **statisticWaitTime**: unidades de tiempo entre cada ejecución del análisis del estado de la red para la recolección de datos.
- **bandWidth**: ancho de banda de los enlaces.
- **network.numNodes**: número de nodos de la red.

- **network.numCorrects**: número de nodos correctos (que no pueden fallar nunca).
- **network.conexionDensity**: probabilidad de que un nodo se conecte con otro.
- **network.startTime**: rango de tiempo en el que los nodos comienzan a transmitir.
- **network.areaSize**: tamaño del área donde situaremos los nodos.
- **network.subAreaSize**: tamaño de las porciones en las que dividiremos el área total, para la distribución de los nodos sobre todo el plano.
- **failType**: lugar que se contabiliza para establecer el fallo de los nodos. Puede ser en envío, recepción o ambos.
- **failChance**: probabilidad de fallo de los nodos.
- **failNumPackets**: número de paquetes que se deben enviar y/o recibir (dependiendo del tipo de fallo) para que los nodos comiencen a fallar.
- **failTime**: unidades de tiempo que transcurren hasta que los nodos pueden comenzar a fallar.

3.1.2. Características de la red

Para comprender mejor estos parámetros de configuración, se ha de explicar el tipo de red que se simula. La red posee enlaces punto a punto bidireccionales. Los nodos se clasifican en dos: correctos (que siempre van a funcionar) y no correctos (que pueden fallar en algún instante de tiempo). El hecho de que fallen y cómo lo hagan va a depender de los parámetros de configuración introducidos en el fichero conf.txt, sin embargo, nunca van a poder volver a levantarse. Los nodos van a poseer, al inicio de la simulación, una unidad de información, conocida como rumor, que se irá propagando entre ellos con el objetivo de que todos conozcan todos los rumores.

La topología es lo más aleatoria posible a excepción de una condición que se impone. Esta restricción se basa en que un nodo correcto cualquiera pueda llegar a todos los demás correctos mediante saltos en nodos correctos (como mínimo, esto no elimina la posibilidad de que se conecten de más formas usando los nodos no correctos entre ellos). Explicándolo de otra manera, tiene que existir un grafo conexo entre todos los nodos correctos de la red. Esta red se genera a través de una clase propia que se ha codificado en el simulador para este caso concreto, llamada RouterC2, dentro del paquete Model de BRITE [37]. Este es un paquete de generación de redes que ofrece multitud de herramientas e interfaces, y está integrado dentro del simulador. La clase codificada hereda los métodos de creación de nodos y fijación del ancho de banda de los enlaces de la clase abstracta RouterModel, por tanto, lo único que se codifica es la conexión entre los nodos. Los pasos para realizar dicha conexión son los siguientes: primeramente, se clasifican los nodos en correctos y no correctos; tras esto, se dibuja un grafo entre los nodos de manera aleatoria y se comprueba si el grafo de los nodos correctos (eliminando los que no lo son) es conexo; si es así, se crean los enlaces entre los nodos según el grafo dibujado, pero si no, se vuelve a dibujar otro grafo aleatorio entre todos los nodos hasta que se cumpla la condición marcada.

3.1.3. Herramienta de registro de información

Pasamos ahora a definir la herramienta de registrado de información de Peersim, que se encuentra dentro del paquete logging. Esta herramienta necesita unos niveles que recibe a través del fichero especificado en el parámetro levelsFile del fichero de configuración. Según

estos niveles, se clasifican las entradas al fichero de registro, resultado de la simulación, y se pueden filtrar a través de ellos mediante el fichero de filtro, especificado también en los parámetros de configuración, más concretamente con el parámetro logConfigurationFile. Hay dos maneras de escribir entradas en el fichero de registro: directamente a través de la clase PeersimLogger, llamando al método log; o usando este mismo método de la clase PeersimLogger, pero pasándole como parámetro un objeto que implementa la interfaz Loggable, para que ejecute el método log del propio objeto. De esta última manera se pueden apilar los niveles.

Una vez comprendidas las particularidades del simulador, se explica el código que se ha implementado sobre este para poder realizar la simulación, comenzando por las clases comunes a los algoritmos, las que describen el entorno (tanto los nodos y los enlaces, como la red en sí), y terminando con las no comunes (los protocolos que contienen el procesado y envío de los paquetes y de la información), del algoritmo más simple al más complejo, junto con una descripción de cada uno.

3.2. Entorno

Como se ha mencionado antes, el entorno hace referencia a todos los elementos necesarios para la correcta ejecución de los protocolos, dónde residen las implementaciones de los distintos algoritmos; estos son, los nodos, los enlaces, el diseño de la red, etc. Todo esto está codificado dentro del paquete basicEnvironment, con la estructura que ya hemos visto. Para explicar el contenido, se describen los subpaquetes uno por uno.

3.2.1. Subpaquete edge

Contiene el objeto que representa a los enlaces entre los nodos. La clase que posea esta funcionalidad debe heredar de la clase abstracta Edge ofrecida por Peersim, como se aprecia en el diagrama de clases de la Figura 3.2.

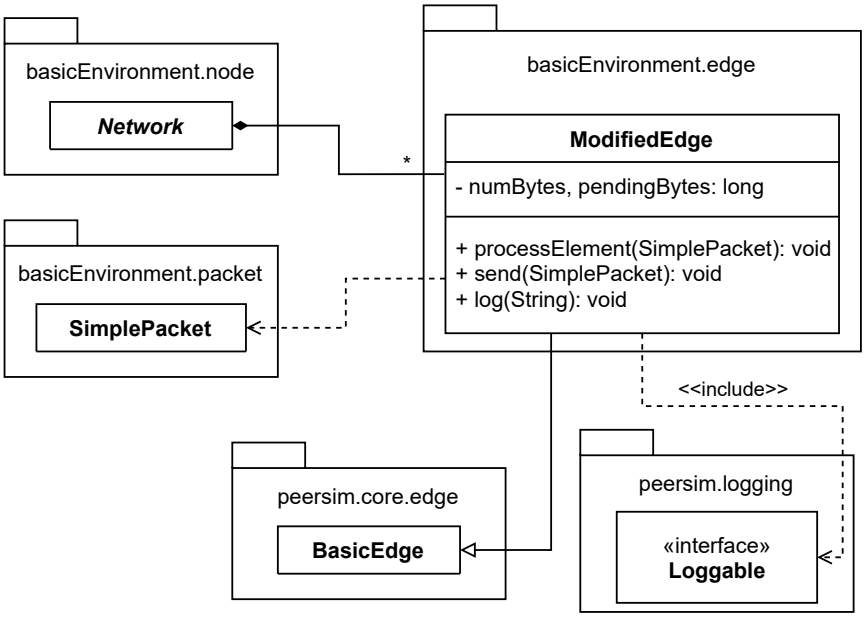


Figura 3.2: Diagrama de clases. Subpaquete edge.

Además, el simulador también ofrece la clase BasicEdge que implementa una funcionalidad básica de un enlace. Por lo tanto, para la codificación de la clase ModifiedEdge (el objeto enlace de la simulación) se hereda de esta clase, ya que su funcionalidad es más que suficiente teniendo en cuenta la red propuesta. Para obtener estadísticas de los enlaces, ModifiedEdge también implementa la interfaz Loggable, que es la principal razón de crear una nueva clase y no usar la ofrecida por el simulador.

Los únicos cambios realizados frente a la clase BasicEdge son la codificación del método log (heredado de la interfaz Loggable) y la creación de variables para el conteo de los paquetes y la cantidad de información transitados por el enlace.

3.2.2. Subpaquete node

Aquí se sitúan tanto la clase que representa a un nodo de la red como la clase que representa a la red en sí, que contiene las instancias de todos los nodos y enlaces. Ambas clases deben heredar de las clases abstractas Node y GraphNetwork, respectivamente, ofrecidas por el simulador. Todo esto está representado en la Figura 3.3 mediante el diagrama de clases del subpaquete.

La clase ModifiedNode es una clase abstracta que sirve de base para la creación de los nodos, que son únicos para cada algoritmo porque deben instanciar sus protocolos específicos. En ella se codifica la mayor parte de la funcionalidad del nodo de la red, dejando únicamente por codificar el método addSpecificProtocols.

Al igual que sucede con los enlaces, Peersim ofrece la codificación de una clase con la funcionalidad básica de un nodo, pero esta no cumple con las especificaciones requeridas para la red ya que no ofrece la posibilidad de que los nodos puedan cambiar de estado. Por lo tanto, la clase ModifiedNode implementa directamente a la clase abstracta Node (que representa un nodo de la red). Esta clase abstracta implementa la interfaz Fallible, que otorga un estado al nodo (*OK*, *FAIL* o *DEAD*, del inglés).

Con respecto a la codificación de la clase, el constructor lo hereda de la clase padre, ya que dónde realmente se inicializa el objeto es en el método init. Esto es así porque necesita saber de sus vecinos, por lo tanto, a la hora de instanciar los nodos, primero se crean todas las instancias de los objetos ModifiedNode (y de los enlaces) y luego se inicializa cada uno. Este proceso lo realiza la clase Network, como veremos a continuación. En el método init se instancian todos los objetos NodeGate (las puertas contra los otros nodos vecinos) y se conectan entre sí. Estas son utilizadas para transmitir y recibir los mensajes. Además, también se instancian los protocolos que van a ejecutarse sobre el nodo.

El resto de métodos codificados de la clase sirven para obtener información del nodo, excepto el método setFailState que cambia el estado del mismo. Todos ellos son métodos abstractos de Node.

El único método abstracto que se deja por codificar es addSpecificProtocols, como ya hemos dicho antes, el cual debe inicializar y añadir a la lista de protocolos del nodo, todos los protocolos necesarios que contengan el código del algoritmo en cuestión.

Por otro lado, la clase Network también es abstracta, en este caso, para instanciar los nodos correspondientes de cada algoritmo. La clase hereda de la clase abstracta GraphNetwork de Peersim, y lo único que se debe codificar es la instanciación de todos los elementos de la red, porque la topología ya está dibujada en el objeto Graph del paquete BRITE. El resto de métodos para la extracción de los elementos de la red se heredan de la clase padre.

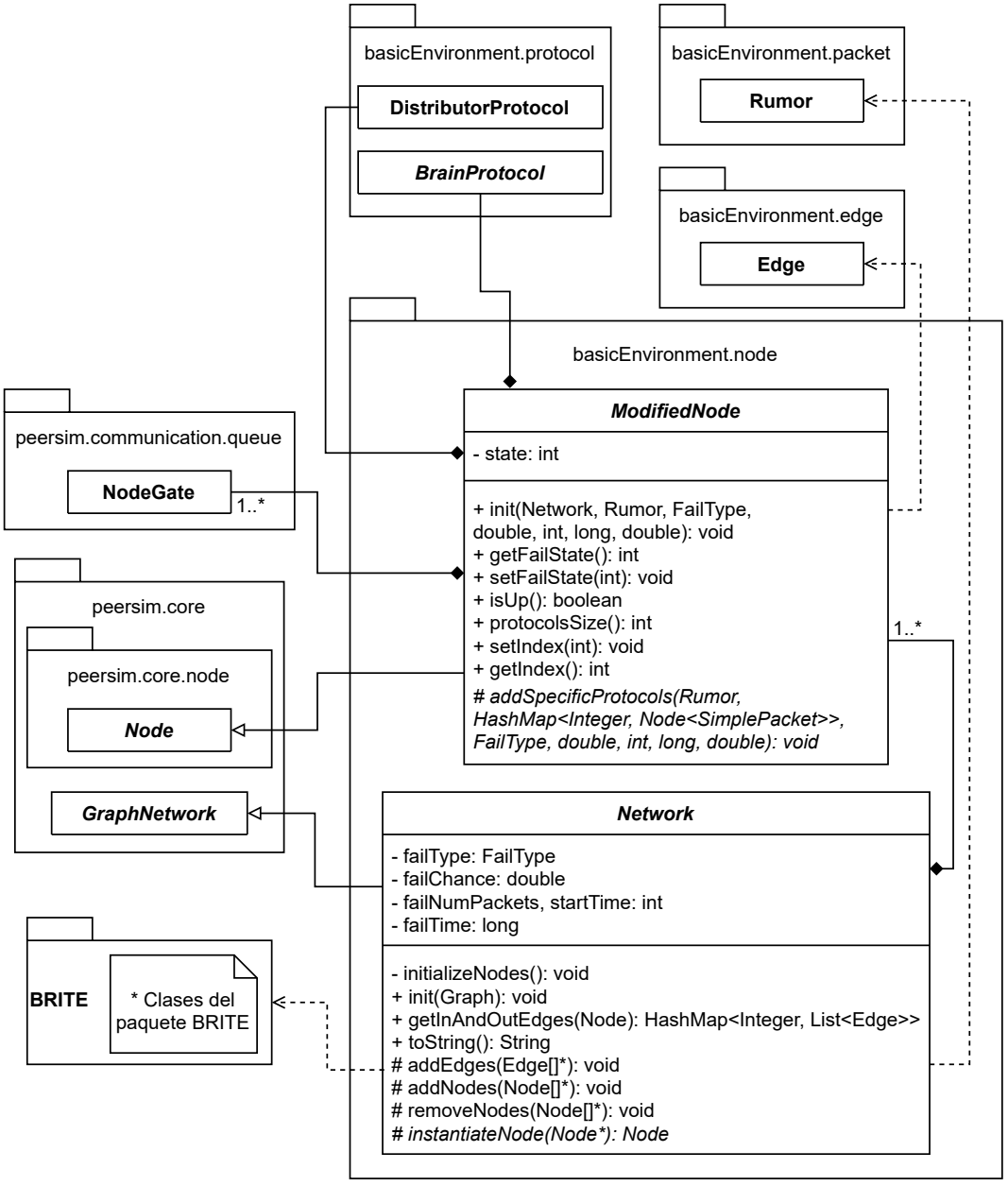


Figura 3.3: Diagrama de clases. Subpaquete node.

Toda la instanciación de los elementos de la red se realiza a través del método init, ya que en este método de la clase padre se llama a los métodos que crean los objetos de la red (que son codificados en la clase Network). Tras ejecutar el init del padre, se llama al método initializeRouters, que inicializa los nodos ya creados llamando al método init de cada nodo.

Durante la inicialización de los nodos, se les pasa por parámetro la pieza de información propia del nodo y las características de fallo de este (en caso de que sea no correcto), además del instante en el que deben mandar el primer paquete (se elige de manera aleatoria entre 0 y el límite marcado en los parámetros de configuración). Estos parámetros son necesarios para que lleguen a los protocolos del nodo.

En este caso, el único método abstracto que se deja para codificar en cada protocolo, es instantiateNode, el cual instancia un nodo y lo devuelve.

3.2.3. Subpaquete statistics

Como ya se ha comentado, este paquete contiene las clases necesarias para recoger la información a nivel global de la red. Para ello se utilizan dos clases: NetworkStatus y StatisticsEvent (Figura 3.4).

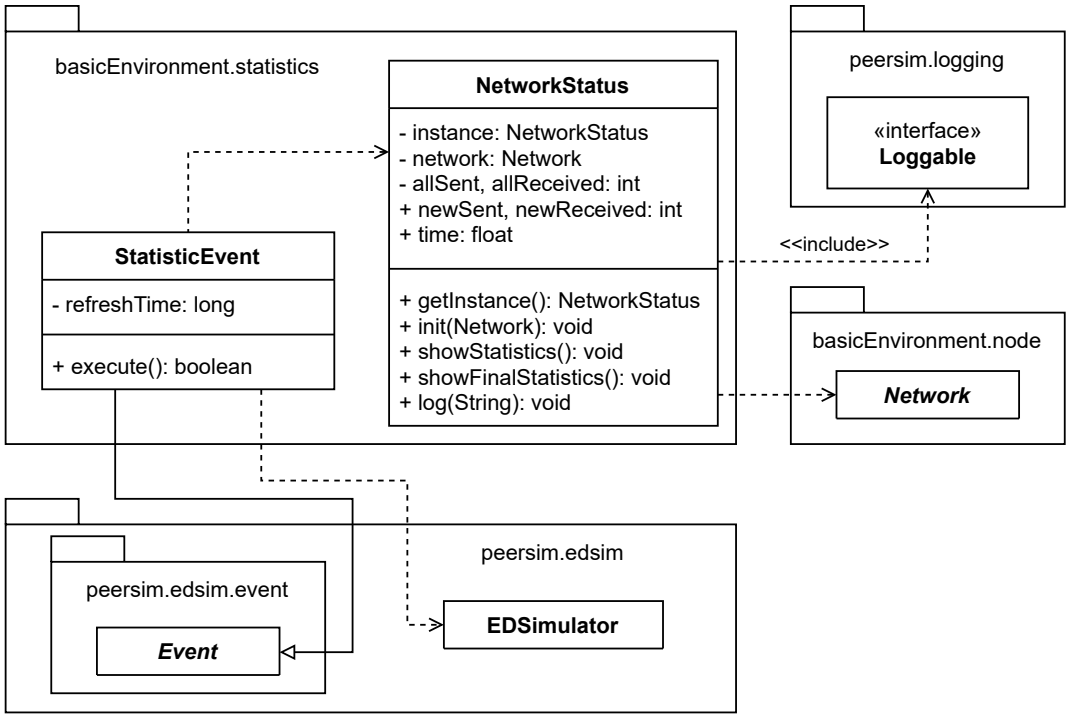


Figura 3.4: Diagrama de clases. Subpaquete statistics.

La primera es una clase que contiene las estadísticas generales de la red, como el número de paquetes totales que se han enviado, o el último instante de tiempo en el que se ha procesado un paquete. Esto requiere que la clase solo se instancie una vez, por lo que se ha desarrollado utilizando un diseño Singleton. También están codificados los métodos para escribir en el fichero de registro, resultado de la simulación, una imagen del estado actual

de la red, mostrando todas las estadísticas que se necesitan para el posterior análisis: la información por nodo, por enlace y la información global.

La segunda clase es un evento que se puede almacenar en la cola de eventos del simulador. Para poder implementar tus propios eventos, Peersim ofrece una clase abstracta llamada Event que debes implementar en tu clase evento. Lo único que debes codificar es el constructor y el método execute (que se ejecutará al desencolarse). En este caso, la función del evento consiste en llamar al método showStatistics de la clase NetworkStatus, que realiza la escritura sobre el fichero de registro del estado de la red, y volver a encolar otro StatisticsEvent en las unidades de tiempo marcadas en el fichero de configuración de inicio de la simulación (en el parámetro statisticWaitTime).

3.2.4. Subpaquete packet

En este paquete se almacenan todas las clases necesarias para crear el objeto que se van a intercambiar los nodos (los paquetes o mensajes), tanto la cabecera, como las unidades de información, es decir, los rumores (Figura 3.5).

Las clases se van a describir desde exterior hacia el interior del objeto intercambiado. La clase SimplePacket representa un paquete o mensaje que contiene toda la información intercambiada entre los nodos. Para ello, esta clase extiende de la clase abstracta Packet, de Peersim. La clase posee una cabecera, una lista de rumores y los métodos necesarios para serializar y deserializar la información. Para agregar las cabeceras, es necesario llamar al método appendHeaders después de instanciar el objeto de esta clase; se hace así para poder crear paquetes más complejos (con más cabeceras) a partir de este en los algoritmos que lo requieran. Además, se incluye un método para extraer la última cabecera de la lista, ya que el método otorgado por el simulador para obtener las cabeceras, extrae la primera que se añadió.

La cabecera del paquete es un objeto de la clase AddressHeader. Esta clase contiene dos números enteros que representan la fuente y el destino del paquete al que pertenece. La información del paquete está contenida en el objeto RumorList, que maneja una lista de objetos Rumor. Estos objetos son las unidades de información más básicas: contienen un identificador y un valor, representados en un número entero cada uno. Todas estas clases también son capaces de serializar y deserializar la información que contienen.

Además, todas las clases mencionadas implementan la interfaz Loggable (excepto la clase Rumor) para poder escribir la información que contienen en el fichero de registro resultado de la simulación.

3.2.5. Subpaquete protocol

Aquí se sitúan todas las clases relacionadas con los protocolos que se ejecutan en cada uno de los nodos como se observa en el diagrama de clases de la Figura 3.6.

La clase abstracta BrainProtocol, representa el protocolo donde reside la lógica de cada algoritmo, cómo se procesa cada paquete recibido, y la creación y envío de los nuevos paquetes. Por tanto, es necesaria la creación de clases, propias de cada algoritmo, que hereden de esta.

Hablando sobre su codificación, el constructor es el que inicializa todas las variables necesarias, pero no realiza ninguna configuración especial. Entre los métodos destacan:

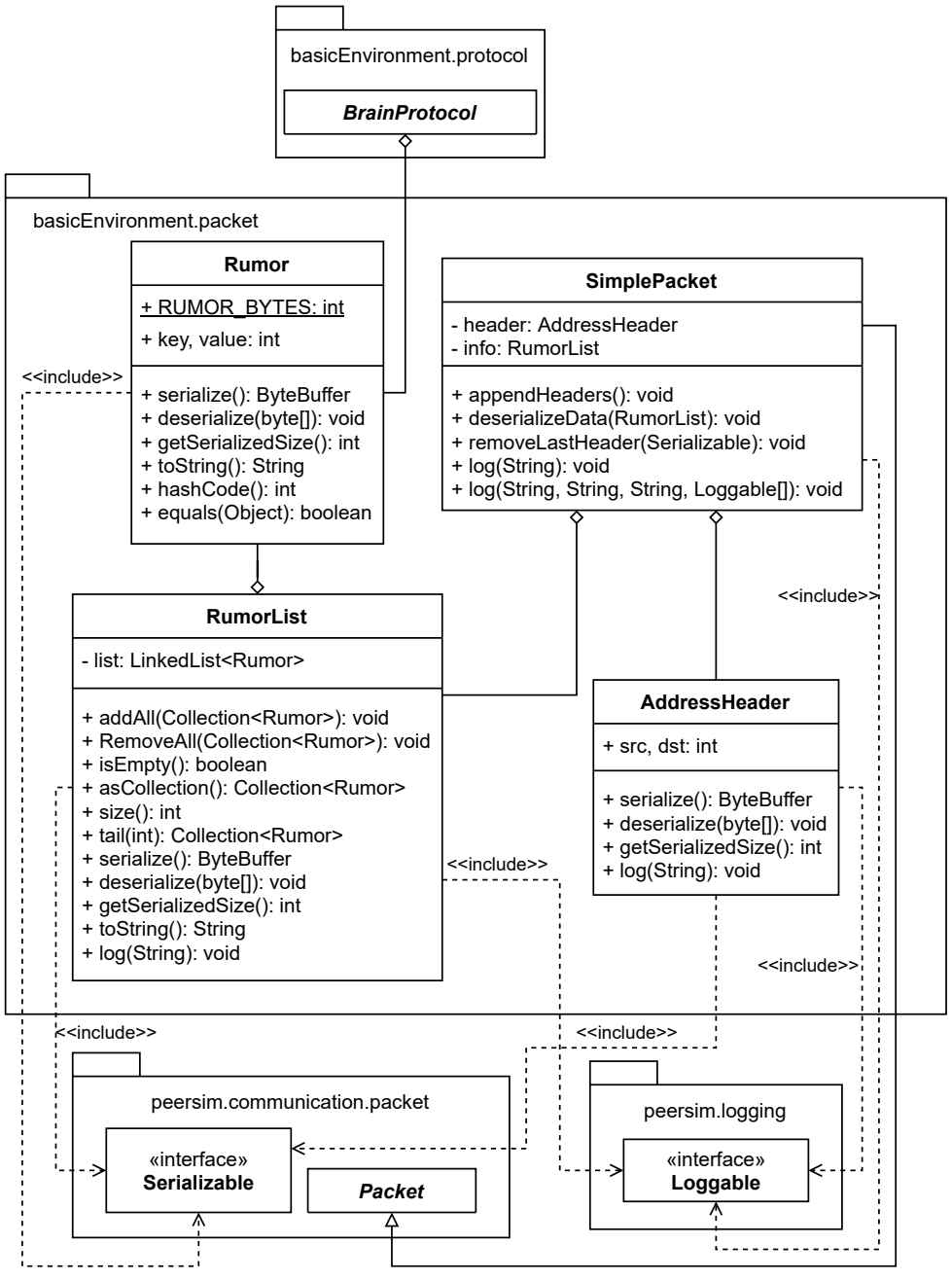


Figura 3.5: Diagrama de clases. Subpaquete packet.

incomingElement Es el método que se ejecuta cuando llega un paquete a una puerta del protocolo (en este caso, la puerta de conexión con el protocolo debajo en la pila de protocolos, ya que no tiene más). Lo único que realiza este método es el desencolado del paquete para que pueda ser procesado por el método processElement.

processElement Es un método abstracto, dónde se analiza el paquete recibido, almacenando los nuevos rumores en las estructuras correspondientes según las reglas del algoritmo en cuestión.

Antes de continuar con el siguiente método, se hace un inciso para explicar los ciclos de envío de información, característica que poseen todos los algoritmos. Estos ciclos se basan en esperar un determinado tiempo τ, desde que el nodo envía un mensaje hasta que envía el siguiente, para agrupar más información en los mensajes y reducir su número. El proceso se realiza encolando un evento de la clase ChooseNeighbourEvent en τ unidades de tiempo, el cual, cuando se desencola, ejecuta el método chooseNeighbour de la clase BrainProtocol.

chooseNeighbour En este método se elige a los vecinos a los que se va a mandar información, en función de los requisitos de cada algoritmo, y se encola un nuevo evento ChooseNeighbourEvent en el caso de que se realice dicho envío. Para que cada algoritmo incluya sus requisitos, se llama desde este método a informNeighbours. Este último es un método abstracto que codifica la elección de vecinos y envío de mensajes de cada algoritmo y devuelve un valor lógico en función de si se debe encolar o no el siguiente evento ChooseNeighbourEvent.

Además, se implementan métodos para incluir en el fichero de registro las estadísticas de cada nodo (como el número de mensajes/rumores que se han enviado y recibido), que provienen de la interfaz Loggable, la cual se implementa. Por último se añaden unos métodos 'protegidos' de utilidades para agilizar el desarrollo de las clases que hereden de esta. Estos métodos son:

- **countReceivingPacket** y **countSendingPacket**: utilizados para incrementar los atributos en función de los paquetes y rumores que se envían/reciben. Estos valores son los que luego se muestran en el fichero de registro.
- **translateStatus**: traduce el estado de un nodo a una cadena de texto.
- **fastChoose**: comprueba si se ha programado un proceso de envío de información o no, para ejecutar directamente el método chooseNeighbour en caso negativo.
- **enqueueEvent**: comprueba cuando se ha ejecutado el último envío y, en caso de que sea en un instante ya pasado, se programa un nuevo ChooseNeighbourEvent, respetando el tiempo τ.
- Métodos **log**: se codifican varios métodos para que se puedan escribir los parámetros necesarios desde las clases que desciendan de esta en el fichero de registro.

Resumiendo, los dos métodos abstractos que se dejan sin codificar para que se implementen los procesos de cada algoritmo son processElement e informNeighbours, que cuentan con el apoyo de los métodos arriba mencionados.

Como esta clase no realiza funciones de distribución de mensajes, se requiere de otros protocolos que los distribuyan. La clase DistributorProtocol realiza esta función, por lo que se sitúa debajo en la pila de protocolos. Gracias a la cabecera AddressHeader del mensaje que recibe, este protocolo puede reenviarlos por la puerta correspondiente. La clase posee una lista de objetos ProtocolGate asociadas cada una a un vecino del nodo, y

otra ProtocolGate más que se conecta con el protocolo de encima de la lista. Todas estas puertas de enlace se conectan con sus correspondientes destinos en el constructor de la clase.

Su método más destacado es processElement, el cual obtiene la dirección de destino del mensaje de la cabecera AddressHeader, y reenvía el paquete por la puerta del vecino destino o por la puerta hacia el siguiente protocolo por encima en la pila, en el caso que el destino sea el propio nodo.

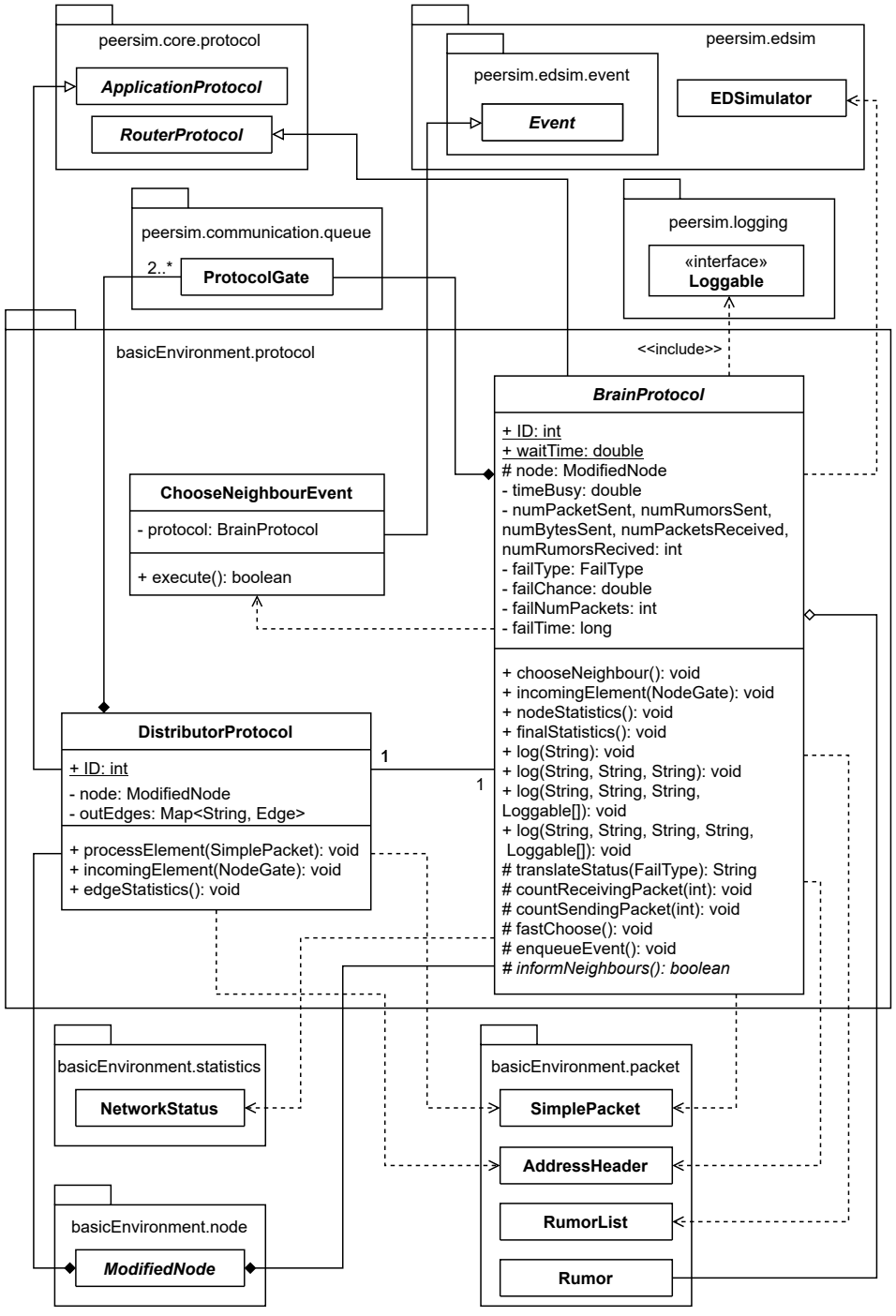


Figura 3.6: Diagrama de clases. Subpaquete protocol.

3.3. Algoritmo 1

Tras comentar todas las clases pertenecientes al paquete basicEnvironment, es el turno de los algoritmos, empezando por una descripción detallada de su funcionamiento, y luego detallando la clase que posee toda esa lógica. Todos ellos van a lograr la quiescencia, de una manera u otra, ya que es uno de los objetivos propuestos.

El primer algoritmo destaca por su simpleza, es de tipo *Push* y busca la eficiencia en el uso de la batería (Figura 3.7).

gossiping$_i(v_i)$:
(1) $known\_by_i[i] \leftarrow \{(v_i, i)\}$;
(2) $known\_by_i[k] \leftarrow \emptyset$, $\forall k \in Neighbors_i$;
(3) $N \leftarrow Neighbors_i$;
(4) **start** task $T1$.

T1:
(5) **when** $(SPREAD, new\_values_k)$ is received from process k **do**:
(6) $known\_by_i[i] \leftarrow known\_by_i[i] \cup new\_values_k$;
(7) $known\_by_i[k] \leftarrow known\_by_i[k] \cup new\_values_k$;
(8) $N \leftarrow \{j \in Neighbors_i : known\_by_i[j] \subset known\_by_i[i]\}$.

(9) **when** $N \neq \emptyset$ **do**:
(10) $send(SPREAD, known\_by_i[i] \setminus known\_by_i[s])$ to a random process $s \in N$;
(11) $known\_by_i[s] \leftarrow known\_by_i[i]$;
(12) $N \leftarrow N \setminus \{s\}$;
(13) **wait** τ units of time.

Figura 3.7: *Battery-efficient version: asynchronous gossip protocol for System S with unknown failures f_u (code for process i)* [2].

3.3.1. Descripción

Este algoritmo se divide en dos procesos, el de envío y el de recepción. Las variables utilizadas son: el conjunto de vecinos del nodo; una lista de tipo clave-valor, siendo la clave el identificador de un nodo (el propio y los vecinos) y el valor una lista con la información que sabe (o se supone que sabe) ese nodo; y una lista de vecinos pendientes de recibir información.

El proceso de recepción se ejecuta cuando el nodo recibe un mensaje. Los pasos a seguir son los siguientes: el nodo lee la información del mensaje y añade la desconocida a su lista, actualiza la lista con los rumores que se supone que sabe ese vecino y, por último, recalcula su lista de vecinos pendientes de recibir información, añadiendo a todos aquellos vecinos que crea que saben menos que él. Esto último da paso a que se inicie el proceso de envío, si no está ya ejecutándose.

Por el contrario, el proceso de envío se ejecuta mientras existan vecinos pendientes. El nodo extrae un vecino al azar de la lista de vecinos pendientes y le envía toda la información que el crea que ese vecino no conoce. Tras esto, actualiza la lista de rumores que sabe ese vecino y espera τ unidades de tiempo hasta volver a ejecutar otro ciclo de esta rutina.

La quiescencia se logra cuando el nodo cree que todos sus vecinos saben lo mismo que él, porque cuando esto ocurre el nodo deja de generar procesos de envío.

3.3.2. Clase Alg1Protocol

Esta clase simula un protocolo que implementa al algoritmo descrito antes, y para ello hereda de la clase BrainProtocol del paquete basicEnvironment, la cual se ha descrito previamente.

Los objetos utilizados para almacenar la información son, un mapa y dos colecciones. Las colecciones almacenan los rumores conocidos por el nodo y los vecinos que (se supone) conocen la misma información que este, y el mapa almacena los rumores que el nodo piensa que sabe cada uno de sus vecinos, asociados al identificador del vecino. Todas los atributos se inicializan en el constructor, que es la única funcionalidad que posee este.

Los métodos destacables codificados son processElement e informNeighbours, los dos métodos abstractos de BrainProtocol. El primero procesa la llegada de un mensaje, almacenando la información nueva y generando, en caso de que sea necesario, un evento de envío. El segundo elige aleatoriamente entre los candidatos a recibir información y se la envía, devolviendo un valor lógico que indica si es necesario encolar otro evento de envío.

Además, se implementa el código necesario para recopilar la información a través de la herramienta de registro. Para ello, se sobreescriben dos métodos, log y finalStatistics.

3.3.3. Resto de código

Además de esta clase, es necesario codificar tres clases que corresponden con las funciones del nodo y del objeto red y la clase principal. Esto es así porque, como ya se ha comentado antes al describirlas, se han creado tres clases abstractas (ModifiedNode, Network y Main) para que se termine de codificar su funcionalidad en cada algoritmo. Las clases que las completan son Alg1Node, Alg1Network y Alg1Main, respectivamente. El único método que se codifica en ellas es, en el caso de Alg1Node, addSpecificProtocols, que añade a la lista de protocolos del nodo una nueva instancia de Alg1Protocol; en el caso de Alg1Network, solo se codifica instantiateNode, el cual devuelve un nuevo objeto de la clase Alg1Node; y en el caso de Alg1Main, se codifica getNetwork, que devuelve un objeto Network instanciado a partir de la clase Alg1Network.

Este algoritmo no posee ninguna clase extra para ampliar la funcionalidad de las herramientas básicas ya codificadas en el paquete basicEnvironment, por lo que este es todo el código necesario para su funcionamiento.

3.4. Algoritmo 2

Siguiendo el mismo formato que en la sección anterior, se va a comenzar describiendo este nuevo algoritmo, para luego comentar el código de su implementación. Este (Figura 3.8) busca optimizar el uso de la memoria con respecto al anterior, conservando la sencillez, y la eficiencia en el uso de la batería. Es este caso, el algoritmo es de tipo *Push & Pull*, lo que aumenta su complejidad pero gana en eficacia.

3.4.1. Descripción

Este algoritmo es muy similar al anterior, y las diferencias principales residen en las estructuras de almacenamiento de la información y en los mensajes intercambiados. Las variables implicadas, en este caso, son las siguientes: la lista de vecinos, una lista ordenada de rumores, una lista que asigne a todos los nodos (ese nodo y sus vecinos) un número entero

gossiping$_i(v_i)$:
(1) $known\_by_i[1] \leftarrow (v_i, i)$;
(2) $index_i[i] \leftarrow 1$;
(3) $index_i[k] \leftarrow 0$, $\forall k \in Neighbors_i$;
(4) $pending\_OK_i \leftarrow \emptyset$;
(5) $N \leftarrow Neighbors_i$;
(6) **start** task $T1$.

T1:
(7) **when** $(SPREAD, new\_values_k)$ is received from process k **do**:
(8) $updates_i \leftarrow \{(v,q) \in new\_values_k : (v,q) \notin known\_by_i[.]\}$;
(9) **for_each** $((v,q) \in updates_i)$ **do**
(10) $index_i[i] \leftarrow index_i[i] + 1$;
(11) $known\_by_i[index_i[i]] \leftarrow (v,q)$;
(12) **end for_each**;
(13) $new\_values_i \leftarrow known\_by_i[index_i[k] + 1\ ..\ index_i[i]] \setminus new\_values_k$;
(14) $send(OK, new\_values_i)$ to process k;
(15) $index_i[k] \leftarrow index_i[i]$;
(16) $N = \{j \in Neighbors_i : (index_i[j] < index_i[i]) \wedge (j \notin pending\_OK_i)\}$.

(17) **when** (OK, new_values_k) is received from process k **do**:
(18) $pending\_OK_i \leftarrow pending\_OK_i \setminus \{k\}$;
(19) $updates_i \leftarrow \{(v,q) \in new\_values_k : (v,q) \notin known\_by_i[.]\}$;
(20) **for_each** $((v,q) \in updates_i)$ **do**
(21) $index_i[i] \leftarrow index_i[i] + 1$;
(22) $known\_by_i[index_i[i]] \leftarrow (v,q)$;
(23) **end for_each**;
(24) $N = \{j \in Neighbors_i : (index_i[j] < index_i[i]) \wedge (j \notin pending\_OK_i)\}$.

(25) **when** $N \neq \emptyset$ **do**:
(26) **let** s **be** a random process : $s \in N$;
(27) $new\_values_i \leftarrow known\_by_i[index_i[s] + 1\ ..\ index_i[i]]$;
(28) $send(SPREAD, new\_values_i)$ to s;
(29) $index_i[s] \leftarrow index_i[i]$;
(30) $pending\_OK_i \leftarrow pending\_OK_i \cup \{s\}$.

Figura 3.8: *Battery-efficient and memory-optimal version: asynchronous gossip protocol with linear buffering for System S and unknown failures f_u (code for process i)* [2].

que identifica la cantidad de información que conocen, una lista de vecinos pendientes de confirmación y otra lista con vecinos pendientes de recibir información.

En este algoritmo se contemplan dos tipos de mensajes, *SPREAD* y *OK*, en inglés. Los mensajes *SPREAD* tienen la misma función que los mensajes del anterior algoritmo, la cual es transmitir información nueva a los vecinos. Sin embargo, los mensajes *OK* se utilizan para realizar una confirmación de la recepción de un mensaje *SPREAD*, enviando además, toda la información, que cree el nodo, que no sabe el vecino que le envió el mensaje.

El proceso de envío de información es exactamente igual que en el anterior algoritmo, el nodo extrae un vecino aleatorio de la lista de pendientes de recibir información y le envía toda la información que se supone que no sabe a través de un mensaje *SPREAD*, añadiéndolo a la lista de vecinos pendientes de confirmación y actualizando la cantidad de información que este sabe. Tras esto se espera τ hasta la siguiente iteración, repitiendo el proceso. A la hora de la recepción de este mensaje, se añade toda la información nueva a la lista, se envía el mensaje *OK* al vecino que envió el mensaje *SPREAD* con toda la

información que ese nodo cree que aún no sabe, y se actualiza el número de rumores que este sabe. Además, se ejecuta un nuevo proceso de envío en este nodo (el que recibe el mensaje *SPREAD*) en caso de que sea necesario. Por último, en la recepción del mensaje *OK* se extrae al vecino de la lista de vecinos pendientes de confirmación, se analiza la información recibida, almacenando la nueva, y se genera un nuevo proceso de envío, en caso de que se requiera.

Al igual que el algoritmo 1, este también logra la quiescencia de la misma manera, dejando de enviar mensajes *SPREAD* cuando el nodo cree que todos sus vecinos saben lo mismo que él (o se encuentran en la lista de vecinos pendientes de confirmación, lo cual indica que ese vecino está caído).

3.4.2. Clase Alg2Protocol

Esta clase es análoga a Alg1Protocol, ya que es la clase que contiene la lógica de este algoritmo, representando el protocolo que ejecutará el nodo, y extiende de la clase BrainProtocol.

Los atributos utilizados, en este caso, para almacenar la información son: dos colecciones, una con los vecinos que poseen toda la información y otra con los pendientes de confirmación; una lista ordenada con todas las unidades de información; y un mapa con todos los vecinos por clave, y el número de rumores que el nodo cree que conoce, como valor.

Con estas estructuras de almacenamiento (las cuales se instancian todas en el constructor) se codifican los mismos métodos abstractos que en el anterior algoritmo (processElement, informNeighbours y los métodos para visualizar las estadísticas), los cuales poseen la misma funcionalidad.

3.4.3. Resto de código

Además de las clases mínimas para cualquier algoritmo (en este caso Alg2Main, Alg2Node y Alg2Network) cuya funcionalidad ya hemos descrito, para poder implementar la lógica de este algoritmo es necesario codificar un nuevo mensaje o paquete (el objeto intercambiado por los nodos) que agregue una cabecera para diferenciar los dos tipos de mensaje que coexisten en el protocolo.

Esta nueva clase se llama Alg2Packet, y extiende de la clase SimplePacket. Se codifica un nuevo atributo, de la clase TypeHeader, que se instancia en el constructor. Además se sobreescriben los métodos appendHeaders, para agregar ésta cabecera, y log, para poder registrarla. La clase TypeHeader contiene un número entero que representa el tipo de mensaje, y las clases necesarias para serializar, deserializar y escribir en el fichero de registro la información.

3.5. Algoritmo 3

Ahora se procederá a comentar el algoritmo 3 (Figura 3.9). Este comparte algunas características con sus dos predecesores, pero también implementa nuevas. Vuelve a ser un algoritmo de tipo *Push & Pull* pero, en este caso, se busca el uso más óptimo de la batería a través de un método diferente para lograr la quiescencia.

gossiping$_i(v_i)$:
(1) $known\_by_i[i] \leftarrow \{(v_i, i)\}$;
(2) $known\_by_i[k] \leftarrow \emptyset$, $\forall k \in Neighbors_i$;
(3) $pending\_OK_i \leftarrow \emptyset$;
(4) **let** N **be** a random set of up to $f+1$ processes $k \in Neighbors_i$;
(5) $send(SPREAD, known\_by_i[i])$ to each process $k \in N$;
(6) $pending\_OK_i \leftarrow pending\_OK_i \cup N$;
(7) **start** Task 1.

Task 1:
(8) **when** $(SPREAD, new\_values_k)$ is received from process k **do**:
(9) $known\_by_i[k] \leftarrow known\_by_i[k] \cup new\_values_k$;
(10) $known\_by_i[i] \leftarrow known\_by_i[i] \cup new\_values_k$;
(11) $send(OK, known\_by_i[i] \setminus known\_by_i[k])$ to process k.

(12) **when** (OK, new_values_k) is received from process k **do**:
(13) $pending\_OK_i \leftarrow pending\_OK_i \setminus \{k\}$;
(14) $known\_by_i[k] \leftarrow known\_by_i[k] \cup new\_values_k$;
(15) $known\_by_i[i] \leftarrow known\_by_i[i] \cup new\_values_k$;
(16) **let** N **be** a random set of up to $f+1$ processes $k \in Neighbors_i$ **such that** $((known\_by_i[k] \subset known\_by_i[i]) \wedge (k \notin pending\_OK_i))$;
(17) $send(SPREAD, known\_by_i[i] \setminus known\_by_i[k])$ to each process $k \in N$;
(18) $pending\_OK_i \leftarrow pending\_OK_i \cup N$.

Figura 3.9: *Battery-optimal version: asynchronous gossip protocol for System S with known failures $f_k + 1$ (code for process i)* [2].

3.5.1. Descripción

Este algoritmo vuelve a utilizar las estructuras para almacenar la información del algoritmo 1, y además mantiene los dos tipos de mensajes del algoritmo dos.

Las dos características más remarcables de este frente a los anteriores están muy relacionadas entre sí, siendo, por un lado, la manera de lograr la quiescencia: en este caso se realiza ejecutando el envío de mensajes *SPREAD* sólo en el caso de recibir algún *OK*, por lo que desaparece el proceso de envío de los otros dos algoritmos. Para asegurarse de que no se quede atascado el nodo (es decir, deje de enviar mensajes *SPREAD* porque el vecino que le debe contestar con el *OK* para que el algoritmo continúe está caído), entra en juego la siguiente característica, que es el *fanout* (en inglés): el conjunto de vecinos a los que se les va a enviar la información. En los algoritmos anteriores el *fanout* es de 1 porque solo envían información a un vecino a la vez. En este caso, el *fanout* mínimo necesario es el número de nodos no correctos de la red más uno (en caso de que el número de vecinos del nodo sea menor que este se enviará la información a todos los vecinos). Esto implica que el número de nodos no correctos de la red tiene que ser un valor conocido por todos.

Por tanto, el algoritmo queda de la siguiente manera: al iniciarse el proceso, se elige un grupo de vecinos a los que se les van a enviar (o *fanout*, cuyo valor dependerá de los nodos no correctos de la red) los primeros mensajes *SPREAD* y, además, se almacenan en la lista de pendientes de recibir información. A la recepción de un mensaje *SPREAD*, se almacena la nueva información, tanto propia, como la que sabe el emisor, y se envía un mensaje *OK*, con la información que se supone que no sabe, a este nodo. Cuando se recibe el mensaje *OK*, de nuevo, se almacena toda la información, se elimina a ese vecino de la lista de pendientes de recibir información, y se vuelve a elegir otro grupo de vecinos para enviarlos un mensaje *SPREAD* y añadirlos a la lista de pendientes de recibir información.

3.5.2. Clase Alg3Protocol

Como en los anteriores algoritmos, esta es la clase que simula el protocolo que posee la lógica del algoritmo 3, y también hereda de la clase BrainProtocol.

La codificación de esta sigue las líneas de las anteriores, codificando los métodos necesarios con las funcionalidades ya comentadas. Lo más destacable es que el método informNeighbours devuelve siempre el valor lógico FALSO (*FALSE*, en inglés), ya que en ningún caso se va a querer encolar directamente un nuevo evento de envío de información debido a la nueva manera de obtener la quiescencia.

3.5.3. Resto de código

Al igual que en la sección anterior, se siguen los mismos pasos que en los anteriores algoritmos a la hora de codificar las clases Alg3Node, Alg3Network y Alg3Main. Además, se reutiliza la clase Alg2Packet para simular el mensaje intercambiado por los nodos.

3.6. Algoritmo 4

Por último, de manera análoga a los anteriores, se va a explicar el algoritmo 4 (Figura 3.10), que también es del estilo *Push & Pull*. Este algoritmo solo realiza una pequeña modificación con respecto al anterior para optimizar, además del uso de la batería, la memoria.

3.6.1. Descripción

Como hemos comentado en la introducción de este algoritmo, es exactamente igual que el anterior, la única diferencia son las estructuras para almacenar la información que utiliza. Estas son las mismas que las del algoritmo dos, es decir, la lista de vecinos, una lista ordenada de rumores, una lista que asigne a todos los nodos (él mismo y sus vecinos) un número entero que identifica el número de rumores que conocen, una lista de vecinos pendientes de confirmación y otra lista con vecinos pendientes de recibir información.

Por lo demás, los procesos son exactamente los mismos que en algoritmo 3: coexisten los dos tipos de mensajes, *SPREAD* y *OK*; y para lograr la quiescencia se utiliza el mecanismo de enviar los *SPREAD* solo a la recepción del *OK*, añadiendo que el *fanout* sea igual al número de nodos no correctos de la red más 1.

3.6.2. Clase Alg4Protocol

La codificación de la clase que simula el protocolo con esta lógica, es muy similar a la anterior, ya que el algoritmo es muy parecido. Las únicas diferencias residen a la hora de tratar la información que almacena el protocolo, que ahora se hace de la misma manera que en el algoritmo 2. Por lo demás, el código no añade ninguna novedad con respecto a los anteriores.

3.6.3. Resto de código

En el resto de código tampoco se añade nada diferente: se codifican las clases necesarias (Alg4Node, Alg4Network y Alg4Main) con sus métodos correspondientes y se utiliza la clase Alg2Packet para simular los paquetes o mensajes intercambiados por los nodos.

gossiping$_i(v_i)$:
(1) $known\_by_i[1] \leftarrow (v_i, i)$;
(2) $index_i[i] \leftarrow 1$;
(3) $index_i[k] \leftarrow 0$, $\forall k \in Neighbors_i$;
(4) $pending\_OK_i \leftarrow \emptyset$;
(5) **let** N **be** a random set of up to $f+1$ processes $k \in Neighbors_i$;
(6) $send(SPREAD, known\_by_i[.])$ to each process $k \in N$;
(7) $pending\_OK_i \leftarrow pending\_OK_i \cup N$;
(8) $index_i[k] \leftarrow 1$, $\forall k \in N$;
(9) **start** Task 1.

Task 1:
(10) **when** $(SPREAD, new\_values_k)$ is received from process k **do**:
(11) $updates_i \leftarrow \{(v,q) \in new\_values_k : (v,q) \notin known\_by_i[.]\}$;
(12) **for_each** $((v,q) \in updates_i)$ **do**
(13) $index_i[i] \leftarrow index_i[i] + 1$;
(14) $known\_by_i[index_i[i]] \leftarrow (v,q)$;
(15) **end for_each**;
(16) $new\_values_i \leftarrow known\_by_i[index_i[k]+1\ ..\ index_i[i]] \setminus new\_values_k$;
(17) $send(OK, new\_values_i)$ to process k;
(18) $index_i[k] \leftarrow index_i[i]$.

(19) **when** (OK, new_values_k) is received from process k **do**:
(20) $pending\_OK_i \leftarrow pending\_OK_i \setminus \{k\}$;
(21) $updates_i \leftarrow \{(v,q) \in new\_values_k : (v,q) \notin known\_by_i[.]\}$;
(22) **for_each** $((v,q) \in updates_i)$ **do**
(23) $index_i[i] \leftarrow index_i[i] + 1$;
(24) $known\_by_i[index_i[i]] \leftarrow (v,q)$;
(25) **end for_each**;
(26) **let** N **be** a random set of up to $f+1$ processes $k \in Neighbors_i$
such that $((index_i[k] < index_i[i]) \wedge (k \notin pending\_OK_i))$;
(27) **for_each** $(s \in N)$ **do**
(28) $new\_values_i \leftarrow known\_by_i[index_i[s]+1\ ..\ index_i[i]]$;
(29) $send(SPREAD, new\_values_i)$ to s;
(30) $index_i[s] \leftarrow index_i[i]$;
(31) **end for_each**;
(32) $pending\_OK_i \leftarrow pending\_OK_i \cup N$.

Figura 3.10: *Battery-and-memory-optimal version: asynchronous gossip protocol with linear buffering for System S and known failures $f_k + 1$ (code for process i)* [2].

3.7. Protocolo de difusión por inundación

Además de estos cuatro algoritmos, que son los que interesa simular para ver su comportamiento, se ha programado un quinto protocolo para realizar comparaciones. Este es exactamente igual que el algoritmo 1 (usa sus mismas estructuras de almacenamiento y procesos similares) y sus clases se almacenan en el paquete alg0Broadcast.

La única diferencia con respecto al algoritmo 1 reside en el proceso de envío de información, ya que a la hora de realizar dicho envío, en vez de escoger un vecino aleatorio de entre los candidatos, se envía información a todos los vecinos que no sepan lo mismo que el nodo pero se sigue respetando el tiempo de espera τ. Se puede lograr simular un proceso de difusión por inundación completo si el valor de este τ fuese 0, lo que supondría que se envíe toda la información nueva a todos los vecinos que no la sepan, en el mismo instante en el que la recibe el nodo.

Con respecto al código, las clases siguen la línea que ya hemos visto en los otros algoritmos, llamándose en este caso BC0Protocol, BC0Node, BC0Network y BC0Main.

Capítulo 4

Simulación y resultados

Una vez preparado todo el código del programa, solo queda ejecutarlo para realizar las simulaciones. Pero, ¿qué parámetros de configuración (o variables) son más idóneos para realizar cada ejecución? Es cierto que cuantas más ejecuciones con parámetros diferentes se hagan y comparen, más conclusiones se pueden obtener. Aún así, no es posible realizar infinitas, por lo que se deben elegir cuidadosamente los parámetros de configuración, en función de los resultados que se esperan, de las características de los algoritmos y la red, y de los objetivos de los algoritmos.

Por esto, a continuación se indican cuáles son los parámetros elegidos para todas las ejecuciones, que se han realizado tras algunas pruebas. Luego se habla del procesado de los datos obtenidos y, por último, de las comparaciones entre los datos y conclusiones extraídas.

4.1. Parámetros de configuración

En primera instancia se realizaron algunas pruebas eligiendo parámetros sin mucho criterio, para ver la respuesta del simulador. Esto ayudó a concretar los más interesantes sobre los que se debía analizar con cautela.

4.1.1. Condiciones de partida

Tras analizar estas primeras ejecuciones de los algoritmos, se obtuvieron las siguientes conclusiones y objetivos a la hora de establecer los parámetros de las simulaciones:

- En primer lugar, si se analizan los algoritmos 1 y 3 (Secciones 3.3 y 3.5, respectivamente), las estructuras de almacenamiento no están optimizadas: cada nodo almacena los rumores de todos sus vecinos, más los suyos, esto es $n * v * r$ (siendo n el número de nodos, v el número de vecinos y r los bytes que ocupa un rumor, que en este caso ocupa 8 bytes). Teniendo en cuenta que el simulador engloba a todos los nodos, la memoria que este necesita almacenar es del orden de $n^2 * v * r$. Si queremos simular redes de 10000 nodos, la memoria asciende al orden de GigaBytes (GB) con solo 10 vecinos por nodo, lo que acarrea problemas de cara a las ejecuciones. Por esto, se establece el número de nodos en 1000, que es un número lo suficientemente grande para realizar las pruebas.
- Como se ha definido anteriormente, los algoritmos 3 y 4 (Secciones 3.5 y 3.6, respectivamente) envían información en cada ciclo a un conjunto de vecinos cuyo número

equivale al número de vecinos no correctos más uno. Si éste valor es mayor que el número de vecinos que poseen, los nodos enviarán la información a todos sus vecinos como si se tratase de un algoritmo de difusión por inundación (*broadcast*, en inglés). Esto le otorga velocidad a los algoritmos, pero los vuelve energéticamente ineficientes, por lo que es necesario hacer comparaciones entre redes con diferente número de nodos correctos (y el resto de parámetros iguales) para comprobar el peso que tiene sobre los algoritmos.

- Siguiendo la línea del punto anterior, para que los algoritmos se diferencien de los de difusión, el valor de τ (o frecuencia entre ciclos de envío) no puede ser demasiado pequeño. Además, se puede observar que en simulaciones con este valor demasiado bajo, las colas de los enlaces en el algoritmo 1 están muy saturadas y provocan retrasos importantes en el proceso. Pero tampoco puede ser muy grande, pues se ralentiza en exceso. Para averiguar más información sobre este parámetro se deben realizar simulaciones que nos ayuden en su estudio.
- Hablando del ancho de banda, éste ejerce mucha influencia sobre el tiempo de transmisión de los paquetes (como es de esperar), por lo que para poder realizar un análisis entre varias simulaciones de la eficiencia de los algoritmos, es necesario fijar un ancho de banda y no variarlo. En este caso, se ha decidido establecerlo en 1 Megabit (Mb) y seleccionar que las unidades de tiempo del simulador se correspondan a microsegundos (μs). Esto permite que se aprecien los tiempos de los procesos de transmisión.
- Debido a lo establecido en el punto anterior, las medidas de las variables se realizan cada 5000 unidades de tiempo. Este parámetro de configuración se ajusta para que el fichero de resultados que se crea tras cada ejecución no sea excesivamente grande, pero a su vez se tomen el número necesario de muestras para ver el progreso de las ejecuciones. Es muy importante tenerlo en cuenta a la hora de observar las gráficas obtenidas por el error que se genera en los datos relacionados con el tiempo.
- Con respecto al tipo de fallo de los nodos no correctos, es decir, a la configuración de cuándo se va a producir el cambio de estado (de OK a $DOWN$) en los nodos no correctos, no se observa a simple vista que alteren las características de los algoritmos, pero para asegurarse de que no produce ninguna interacción se ha decidido dejarlo igual en todas las simulaciones.
- Una de las características del sistema sobre el que se está simulando es que es asíncrono. Para demostrar esta característica, se ha programado que los nodos comiencen aleatoriamente en un rango determinado, marcado en los parámetros de configuración. Este parámetro se ha establecido en 200 unidades de tiempo del simulador en todas las ejecuciones para que no influya a la hora de compararlas.
- Según las definiciones de los algoritmos 1 y 2 (Secciones 3.3 y 3.4, respectivamente), por cada vecino que tenga un nodo, va a transmitir o recibir como mínimo un paquete (en el caso ideal). Por otro lado, se puede pensar que el número de vecinos influye directamente en la velocidad de propagación de la información. Debido a esto se pueden esperar grandes diferencias en simulaciones con muchos y pocos vecinos, pero es necesario realizar simulaciones para comprobarlo.

4.1.2. Simulaciones realizadas

Tras esas primeras ejecuciones de ajuste, se siguieron realizando pruebas, con la experiencia obtenida de las anteriores y con más criterio, para concluir las dudas que no se habían podido averiguar previamente. Las simulaciones se realizaron de la siguiente forma: en primer lugar se elegían los parámetros a simular y posteriormente se ejecutaban simulaciones de todos los algoritmos. Esto permite que se puedan comparar en todo momento las diferentes respuestas que genera cada uno.

La primera tanda de simulaciones se centró en estudiar el comportamiento de los algoritmos en función del número de vecinos que tuviera cada nodo y se dejaron el resto de parámetros fijos para que no ejercieran ninguna influencia. El valor de τ se decidió establecer en 4000 unidades de tiempo para estas pruebas, ya que este es un valor intermedio de este parámetro como se verá posteriormente. Se realizaron ejecuciones de este tipo con 5, 10, 50, 100 y 200 vecinos de media.

Tras esto, se decidió estudiar la influencia del valor de τ. Para ello, se realizaron simulaciones otorgándole los valores de 1000, 4000 y 9000 unidades de tiempo. Se realizaron en redes con 10 y 100 vecinos (de media), para comprobar si tenía algún efecto variar los dos parámetros a la vez, pero el resto de parámetros se dejaron iguales. Para estas dos primeras rondas de simulaciones, el número de nodos correctos de la red se estableció en 950.

Por último, con el fin de verificar el efecto que tiene la variación en el número de nodos correctos de la red sobre los algoritmos, se realizaron simulaciones alterando solamente este parámetro. Se decidió comparar la simulación que ya se había realizado de 10 vecinos por nodo, $\tau = 4000$ unidades de tiempo y 950 nodos correctos, con otra nueva exactamente igual, pero de 999 nodos correctos.

En total, se realizaron 50 simulaciones diferentes, sin contar todas las simulaciones previas usadas para comprobar la respuesta de los algoritmos y del simulador, que se usaron para obtener las conclusiones que se describen más adelante.

4.2. Análisis de los datos

Tras realizar las ejecuciones del simulador, obteniendo los diferentes ficheros de registro de los resultados, se deben analizar estos datos. Para interpretar los ficheros se utiliza un procesador externo a Peersim, que obtiene los datos relevantes y los almacena en unidades de información que se puedan graficar.

R [11] es un lenguaje de programación y un entorno para el análisis de datos estadísticos. La decisión de escogerlo para el procesado de los datos y el graficado de los mismos frente a otros procesadores se basa en que R permite realizar estas dos tareas de manera simple, sin necesidad de recurrir a un programa para la extracción de los datos, y otro para obtener las gráficas. Además tiene un gran potencial para realizar análisis estadísticos y nos otorga mucha flexibilidad de cara a la obtención de las gráficas.

Para transformar los ficheros de registro en tablas de datos legibles por R se ha codificado un *script*. Este recibe la ruta del fichero y genera una tabla o *frame*, en inglés, que contiene toda la información necesaria ordenada. A partir de ella, se crean las gráficas necesarias para el análisis de la información. El código para generarlas se encuentra en otro *script* diferente. La mayoría de las gráficas que se han utilizado en todo el proceso representan alguna variable en función del tiempo, y en ellas se analizan: los mensajes y rumores enviados y recibidos, tanto los totales como los nuevos entre cada comprobación del estado

de la red; la suma de rumores conocidos por todos los nodos, para ver el progreso del conocimiento de la red; la media de rumores por paquete, y el número de enlaces que están transmitiendo, o pendientes de transmitir, un mensaje.

4.3. Comparación de gráficas

En esta sección se explican las gráficas obtenidas de las simulaciones, junto con todos los detalles extraídos sobre ellas de las comparaciones realizadas. Para el propósito de este trabajo, las gráficas que más nos van a interesar son las que representan el crecimiento de los rumores y paquetes enviados y recibidos, por lo que son las únicas que se van a analizar para no saturar el documento. Hay que tener en cuenta, a la hora de comparar los resultados, el hecho de que no se están simulando sobre redes exactamente iguales (ya que está programado para que cada ejecución se realice sobre una red aleatoria nueva), por lo que pueden existir pequeñas variaciones en los valores medidos que correspondan a las diferencias de topología.

Para realizar este análisis, se va a seguir el mismo orden seguido al ejecutar las simulaciones, comenzando con las comparaciones entre vecinos, prosiguiendo con las que poseen distintos valores de τ y finalizando con las de diferente número de nodos correctos. Tras esto, se comparan las gráficas obtenidas de los diferentes algoritmos para simulaciones con los mismos parámetros de configuración, con el objetivo de examinar las diferencias entre ellos.

Primero se comparan las gráficas de las simulaciones realizadas para examinar la afectación de la variación del número de vecinos de la red en la respuesta de los algoritmos (Figuras 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7 y 4.8). Lo primero que se observa es que cuantos más vecinos posea la red, más se va a transmitir, tanto paquetes (o mensajes) como rumores, y ocurre con todos los algoritmos. A la vez que esto aumenta, también aumenta el tiempo que tarda en llegar a la quiescencia, pero se disminuye el tiempo en conseguir el conocimiento máximo en la red (todos los nodos conocen todos los rumores que tienen que conocer), aunque se ralentiza esta disminución según se aumentan los vecinos. Sin embargo, el algoritmo 1 no sigue este patrón, pues, a pesar de que se produce una reducción del tiempo en el salto de 5 (Figura 4.2a) a 10 (Figura 4.2b) vecinos, a partir de ese punto cuantos más vecinos más tarda en conseguir el conocimiento máximo (Figura 4.2).

Comparando las gráficas más detalladamente, se observa que el número de paquetes enviados, tras conseguir el conocimiento máximo de la red, aumenta de forma drástica según aumenta el número de vecinos, siendo en los algoritmos 1 (Figura 4.1) y 2 (Figura 4.3) dónde se produce de forma más pronunciada. Por otro lado, la cantidad de paquetes que se necesitan enviar para alcanzar el conocimiento máximo es estable en los algoritmos 1 (Figura 4.1) y 2 (Figura 4.3), pero aumenta mucho en los algoritmos 3 (Figura 4.5) y 4 (Figura 4.7). Si nos referimos a la cantidad de rumores enviados, el crecimiento es aún mayor (Figuras 4.6 y 4.8). Con respecto a la forma de las gráficas, tienen una forma constante (la cual aumenta según se añaden más vecinos). Dicha forma se comentará cuando se comparen los algoritmos entre sí.

Seguido de esto, se analizan las simulaciones realizadas para comparar el efecto que tiene sobre los algoritmos la variación del valor de τ. En este caso, son las Figuras 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22, 4.23 y 4.24, que corresponden a las gráficas de las simulaciones realizadas con 10 y 100 vecinos para los diferentes valores dados a τ. Como se puede observar, con respecto al tiempo que tardan en llegar al estado

de reposo, cuanto más alto es el valor de τ más se incrementa; y sucede lo contrario con los paquetes transmitidos, cuanto más alto es el valor menos se transmiten; hechos que son más destacables en los algoritmos 1 y 2 (Figuras 4.9, 4.11, 4.17 y 4.19). Sin embargo, los rumores transmitidos para los algoritmos 2 con 100 vecinos, 3 y 4 (en el 1 siguen el mismo patrón que los paquetes, como se ve en las Figuras 4.10 y 4.18) apenas varían para los distintos valores de τ (Figuras 4.12, 4.14, 4.22, 4.16, 4.24). Aunque, observando con detalle en la Figura 4.24, se aprecia un pequeño aumento de los rumores del algoritmo 4 con 100 vecinos según aumenta el valor de τ. Sin embargo, el algoritmo 2 con 100 vecinos se comporta como el algoritmo 1 (Figura 4.20).

Hablando del instante de conocimiento máximo, al igual que sucede con el tiempo, este sufre un pequeño aumento según crece τ. Los paquetes y rumores necesarios para llegar a este conocimiento están completamente ligados a la variación en la cantidad de paquetes y rumores transmitidos, respectivamente. Es decir, en los casos que hay mucha variación en el número de paquetes o rumores transmitidos, también hay mucha variación en el número de paquetes o rumores necesarios para lograr el conocimiento máximo. La forma de las gráficas no tiene ningún dato relevante, ya que todas son muy similares, pero un poco desplazadas o alargadas. Destaca el caso del algoritmo 1, que para el valor de $\tau = 4000$ unidades de tiempo (Figuras 4.9b y 4.17b) se puede apreciar como si la gráfica se encogiera (esto también ocurre con el algoritmo 2 en la simulación con 100 vecinos, Figura 4.19b, pero de forma más leve).

Continuando con las comparaciones entre las gráficas, ahora es el turno de las simulaciones con diferente número de nodos correctos (Figuras 4.25, 4.26, 4.27, 4.28, 4.29, 4.30, 4.31 y 4.32). El dato que más resalta es que para los algoritmos 1 y 2, las gráficas son prácticamente iguales (Figuras 4.25, 4.26, 4.27 y 4.28). Las dos diferencias más apreciables son que el número de mensajes y rumores enviados es prácticamente igual al de los recibidos en la simulación de 999 nodos correctos (Figuras 4.25b, 4.26b, 4.27b y 4.28b) y que se produce una pequeña disminución en los rumores transmitidos en el algoritmo 2 (Figura 4.28). En los algoritmos 3 y 4 la respuesta que producen al cambio en el número de nodos correctos es la misma: al aumentar el número de nodos correctos aumentan el tiempo que se tarda en alcanzar el conocimiento máximo y el estado de reposo, disminuye el número de paquetes transmitidos y necesarios para llegar al conocimiento máximo, y aumentan levemente los rumores transmitidos y necesarios para ese alcanzar el conocimiento.

Por último, se comparan cinco simulaciones con los mismos parámetros de configuración, pero de cada algoritmo que se ha programado, incluido el algoritmo de prueba (Sección 3.7), cuyas gráficas están representadas en las Figuras 4.33 y 4.34. De los parámetros elegidos para estas simulaciones destacamos los siguientes: el número de vecinos por nodo, establecido en 100 vecinos; el valor de τ, en 4000 unidades de tiempo; y el número de nodos correctos, en 950. Como se observa (y se veía en las anteriores figuras), se pueden agrupar los algoritmos según la forma de sus gráficas, teniendo por un lado a los algoritmos 1 y 2, cuyas gráficas tienen forma de 'S' alargada en el tiempo, y por el otro a los otros tres algoritmos, cuyas gráficas, aunque difíciles de describir, pueden asemejarse a unos escalones. A pesar de esto, si nos fijamos de cerca en las gráficas de los algoritmos 1 y 2 podemos apreciar también unos pequeños escalones, pero mucho más suaves (este hecho es más notorio en las gráficas de simulaciones con un valor de τ mayor y pocos vecinos por nodo, como en las Figuras 4.9c, 4.10c, 4.11c y 4.12c).

Fijándose en las figuras, se pueden clasificar los algoritmos, del más eficiente al que menos, en función de cada variable que estudiamos:

- Según el tiempo que tardan en alcanzar el instante de conocimiento máximo y finalizar: el más rápido es el algoritmo de prueba, lo siguen los algoritmos 3 y 4 sin apenas diferencia, después está el 2 y a la cola el 1.
- Según la cantidad de rumores transmitidos: el que menos rumores transmite es el algoritmo 1, a continuación los algoritmos 2 y 3, seguidos por el 4, y el último el algoritmo de prueba.
- Según la cantidad de paquetes transmitidos, y la cantidad de paquetes y rumores necesarios para lograr el conocimiento máximo (se representan juntas porque los algoritmos se ordenan de la misma forma para las tres): vuelve a estar el algoritmo 1 el primero, detrás de este el 2, luego el 3 y el 4 (siendo un poco peor este último) y finaliza la lista el de prueba.
- Según la cantidad de paquetes recibidos desde que se alcanza el conocimiento máximo hasta que se llega a la quiescencia: el que menos recibe es el algoritmo 1, luego el 3 seguido del 2, después el 4 y por último el de prueba.
- Según la cantidad de paquetes enviados desde que se alcanza el conocimiento máximo hasta que se llega a la quiescencia: el que menos paquetes envía es el algoritmo de prueba, que no envía ninguno. Lo siguen los algoritmos 1 y 3, tras estos el 4 y finaliza la cola el 2.
- Según la cantidad de rumores recibidos desde que se alcanza el conocimiento máximo hasta que se llega a la quiescencia: en este caso, el que menos recibe es el 3, después el 4, el 1, el de prueba y acaba el 2.
- Según la cantidad de rumores enviados desde que se alcanza el conocimiento máximo hasta que se llega a la quiescencia: el algoritmo de prueba vuelve a ser el que menos envía, junto con el 3 (ninguno envía rumores). Tras ellos, el algoritmo 4, al que le siguen el 1 y el 2, finalizando este en el último lugar.

Toda esta información está resumida en la Tabla 4.1 para una mejor visualización de los resultados.

Tipo de clasificación	**1º**	**2º**	**3º**	**4º**	**5º**
Tiempo en finalizar	prueba	3 y 4	2	1	-
Tiempo en conseguir el conocimiento máximo	prueba	3 y 4	2	1	-
Rumores transmitidos	1	2 y 3	4	prueba	-
Paquetes transmitidos	1	2	3	4	prueba
Paquetes y rumores para el conocimiento máximo	1	2	3	4	prueba
Paquetes recibidos tras el conocimiento máximo	1	3	2	4	prueba
Paquetes enviados tras el conocimiento máximo	prueba	1 y 3	4	2	-
Rumores recibidos tras el conocimiento máximo	3	4	1	prueba	2
Rumores enviados tras el conocimiento máximo	prueba y 3	4	1	2	-

Cuadro 4.1: Clasificación de los algoritmos según diferentes características.

Volviendo a hablar de la forma de las gráficas, remarcar la similitud entre las de los algoritmos 1 y 2, y 3 y 4. No sólo tienen formas parecidas, si no que son muy semejantes en todas las medidas. Además, estas diferencias se van pronunciando según aumentan los vecinos y disminuyendo cuando se reducen, como se observa en las Figuras 4.3 y 4.7, o en todas las figuras del estudio de τ.

4.4. Resultados

Tras el análisis exhaustivo de todas las gráficas obtenidas de las simulaciones, se obtienen las conclusiones que se describen a continuación. Primero las referidas a la variación de los distintos parámetros de configuración, y cómo afecta a cada algoritmo en particular y luego las de los algoritmos entre sí.

4.4.1. Variación de parámetros de configuración

- La cantidad de vecinos por nodo media que posea la red afecta de manera considerable a todos los algoritmos, aumentando enormemente el número de rumores y paquetes que se envían. Si para un mismo número de nodos en la red se requieren más mensajes para llegar a la quiescencia, significa que se está enviando mucha más información redundante.
- Otra característica muy perjudicada cuando se aumenta el número de vecinos es el tiempo en alcanzar el estado de reposo, que está muy ligada a la anterior. Al tener que enviar más mensajes, se tarda más en finalizar el proceso.
- Sin embargo, el instante en el que se consigue el conocimiento máximo de la red, para los algoritmos 2, 3 y 4, disminuye ligeramente según se añaden vecinos, y cuantos más vecinos por nodo hay, menos influencia tiene. Esta respuesta es la esperada, pues cuantos más vecinos se posea, más fácil es que la información llegue a todos. Sorprendentemente, el algoritmo 1 da unos resultados inversos, realizando un pequeño aumento en el tiempo en el que se consigue este conocimiento según crece el número de vecinos de cada nodo, teniendo en cuenta que también tiene menos influencia cuanto mayor es el número de vecinos. A pesar de esta tendencia del algoritmo 1, cuando llega a demasiados pocos vecinos se produce el efecto contrario, aumentando considerablemente el tiempo, como se observa en la Figura 4.1a.
- Variado de los tres puntos anteriores, se puede deducir (y corroborarlo en las figuras) que cuantos más vecinos por nodo se añaden a la red más información redundante, tras haber logrado el conocimiento máximo, se envía. El ideal es que la red entre en estado de reposo lo antes posible después de que toda la información se haya difundido a todos los nodos. Por todo esto, se concluye que alrededor de los 10 vecinos por nodo de media es el valor óptimo para estos algoritmos.
- Si se incrementa el tiempo de espera entre ciclos o τ, como era de esperar, se produce un aumento en el tiempo que tarda el algoritmo en alcanzar el conocimiento máximo y en llegar al estado de reposo. Por otro lado, se reduce el número de mensajes e información enviada. Estos cambios son muy drásticos en el algoritmo 1, un poco menos en el algoritmo 2, y apenas se notan en los algoritmos 3 y 4. Por lo que se deduce que este parámetro apenas tiene influencia sobre estos dos últimos. Sin embargo, la respuesta tan pronunciada del algoritmo 1 se debe a que, para valores muy pequeños de τ, se comporta como un algoritmo de inundación, porque no le da tiempo a recibir información nueva antes de que decida volver a enviar. Además,

este algoritmo es totalmente dependiente de los ciclos para enviar información, no sucediendo así en los otros tres, los cuales pueden enviar mensajes de tipo *OK* en cualquier momento.

- Centrándonos en el número de rumores transmitidos, en los algoritmos 3 y 4 no hay diferencia aparente si variamos τ. Para el algoritmo 2 con 10 vecinos tampoco existe diferencia, pero si aumentamos los vecinos a 100, si que varía este número (Figura 4.20).
- Hablando de la saturación en los enlaces, el valor de τ tiene la mayor influencia sobre éste parámetro, como se puede deducir. Por un lado, al disminuir τ se envían más paquetes y más información, por lo que se va a producir más saturación. Por otro lado, como cuando se disminuye también se disminuye el tiempo, se van a tener que enviar los mensajes en un menor intervalo, produciendo más saturación aún.
- A parte de lo comentado, no se aprecian variaciones de interés a la hora de comparar la influencia de τ en simulaciones con diferente número de vecinos.
- Comparando las formas de las gráficas de cada algoritmo en las variaciones de τ para el algoritmo 1 y 2 (sobre todo en las gráficas de las simulaciones con 100 vecinos por nodo, Figuras 4.17 y 4.19), se puede intuir que alrededor de $\tau = 4000$ hay un punto en el que la relación entre las dos variables que mide la gráfica es más óptima.
- Como era de esperar, la influencia que tiene el número de nodos correctos sobre los algoritmos 1 y 2 es prácticamente nula. No ocurre esto con los algoritmos 3 y 4, que aumentan los tiempos, pero disminuye el número de paquetes transmitidos según aumentamos el número de correctos. Hay que tener en cuenta que estos algoritmos solo van a variar su funcionamiento en un determinado rango: $n - v \leq c \leq n$ siendo c el número de nodos correctos, n el número de nodos de la red y v el número de vecinos por nodo. Esto es así porque fuera de ese rango no se va a modificar el número de vecinos a los que se envía información en cada ciclo. Sin embargo, hay dos comportamientos que se salen de lo esperado, ambos referidos a los rumores. En primer lugar, al comparar las gráficas del algoritmo 2 (Figura 4.28) se aprecia una pequeña disminución en los rumores transmitidos al aumentar los correctos. Sin embargo, en las gráficas de los algoritmos 3 y 4 (Figuras 4.30 y 4.32) se puede ver el caso contrario, que al aumentar el número de correctos, aumenta ligeramente el número de rumores transmitidos.

4.4.2. Comparación entre algoritmos

Los algoritmos 1 y 2 son mejores en cuanto al envío de mensajes y cantidad de información, por lo que van a ser los energéticamente más eficientes (aunque el algoritmo 3 puede llegar a enviar la misma cantidad de información bajo la configuración adecuada). Comparando estos dos, la diferencia entre la cantidad de información que envía uno y otro es muy escasa, siendo el algoritmo 1 el que menos envía. Con respecto a los tiempos de finalización y de conocimiento máximo, el algoritmo dos es más rápido, pero con respecto a los paquetes transmitidos, el algoritmo 1 es el mejor, aumentando esta diferencia según aumenta el número de vecinos por nodo.

Por otro lado, los algoritmos 3 y 4 son muy similares, siendo el algoritmo 3 algo más óptimo en cuanto a los tiempos, la cantidad de información que envía y el número de mensajes. Esto es así, debido a que el algoritmo 4 simplifica el proceso de recepción de información en pos de optimizar al máximo la memoria utilizada. Comparados con los

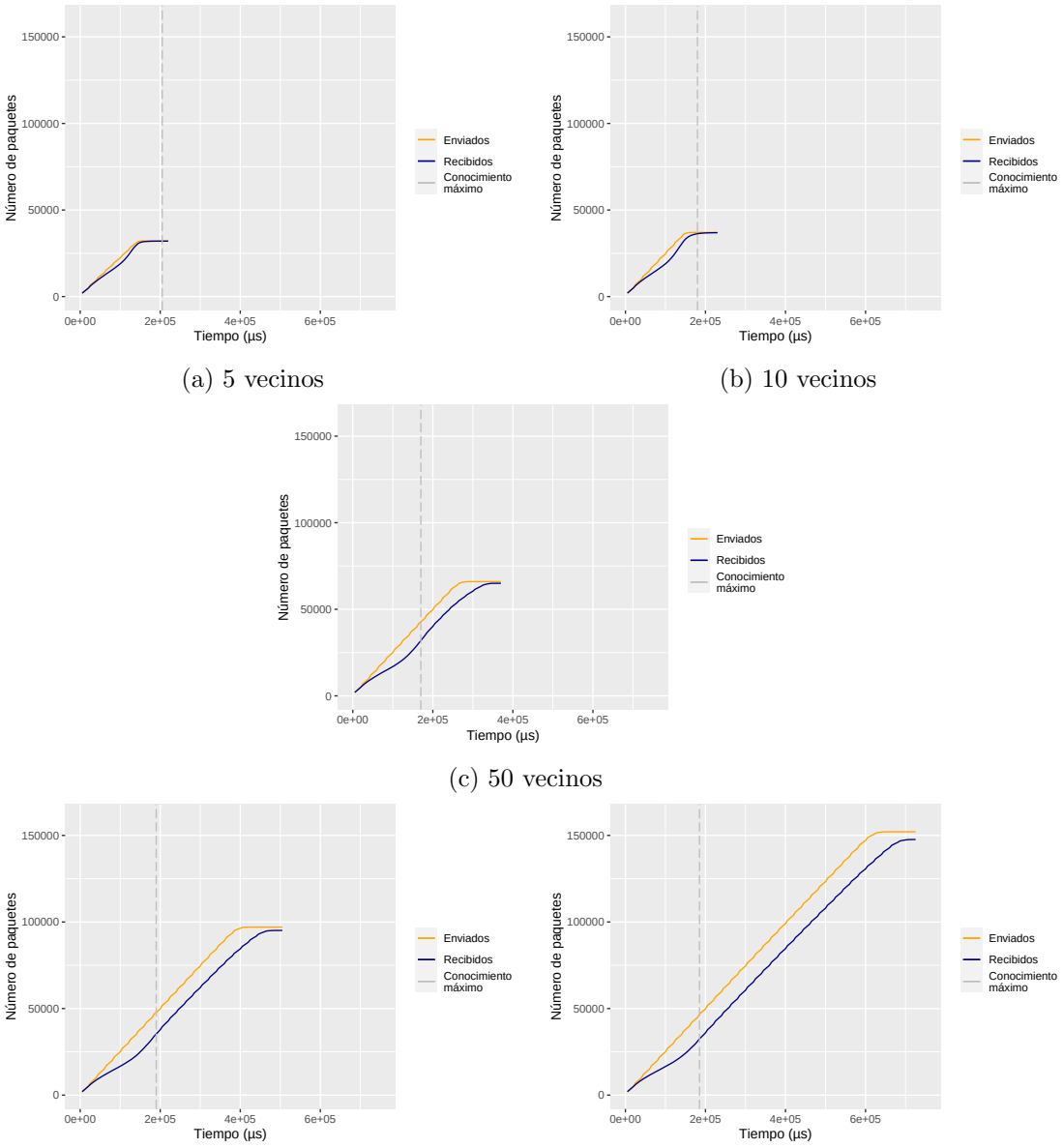
otros dos, estos algoritmos son más rápidos, es decir, consiguen llegar antes al instante de conocimiento máximo y al estado de reposo, aunque necesitan transmitir y recibir más mensajes y rumores para lograr ambos.

Con respecto al de prueba (el algoritmo de difusión por inundación) como era de esperar, este último es mucho más eficiente si hablamos de tiempos, pero hablando de los mensajes y la cantidad de información, el resto son mejores. En los casos en los que los algoritmos 3 y 4 envían información a todos sus vecinos (porque el número de nodos no correctos es mayor o igual), estos tienen las diferencias más reducidas frente al de prueba aunque siguen siendo relevantes. La Tabla 4.1 abrevia todas las comparaciones realizadas a los algoritmos, clasificándolos de más a menos óptimo.

De la forma de las gráficas de todos los algoritmos se observan unos ciclos, que parecen constantes, con periodos donde se recibe más información y periodos en los que menos. En los algoritmos 3, 4 y el de prueba son más irregulares, y en los otros dos son menos pronunciados. Además, con pocos vecinos se aprecian mucho peor (puede ser porque el algoritmo concluya en un solo ciclo). La existencia de estos ciclos puede estar relacionada con el tiempo de inicio de los nodos y con τ, pero se necesitan realizar más pruebas para comprobarlo.

Como resumen de todos los resultados obtenidos, se puede concluir que los algoritmos se comportan como se esperaba en la teoría, consiguiendo difundir la información de manera satisfactoria, en un tiempo razonable y ahorrando en el número de mensajes y rumores enviados. Sin embargo, no se dispone de suficiente información para destacar un algoritmo sobre otro, más aún sin conocer la importancia que se le puede dar al tiempo, a la memoria y a la energía. Con respecto a la influencia de los distintos parámetros de configuración sobre los que se ha estudiado, destaca la ventaja que supone disminuir el número de vecinos por cada nodo. Con esto se consigue que el instante en el que la red alcanza el conocimiento máximo y el tiempo en el que llega al estado de reposo sean muy parejos, disminuyendo el número de información redundante que se envía entre estos dos instantes de manera drástica. Hay que tener en cuenta que todas estas conclusiones obtenidas son sobre redes con una topología aleatoria.

A pesar de esto, es necesario continuar con el estudio de estos algoritmos desde diferentes frentes. En primer lugar, hay que concretar (a ser posible con ecuaciones) la influencia de la variación de los parámetros de configuración sobre los algoritmos para encontrar los valores óptimos. Por otro lado, siguen existiendo comportamientos que se deben estudiar para explicar su motivo; todos han sido mencionados en las secciones previas. Para concluir, también se debe buscar la influencia de otros parámetros de configuración, como puede ser la topología de la red que se simula.



(a) 5 vecinos

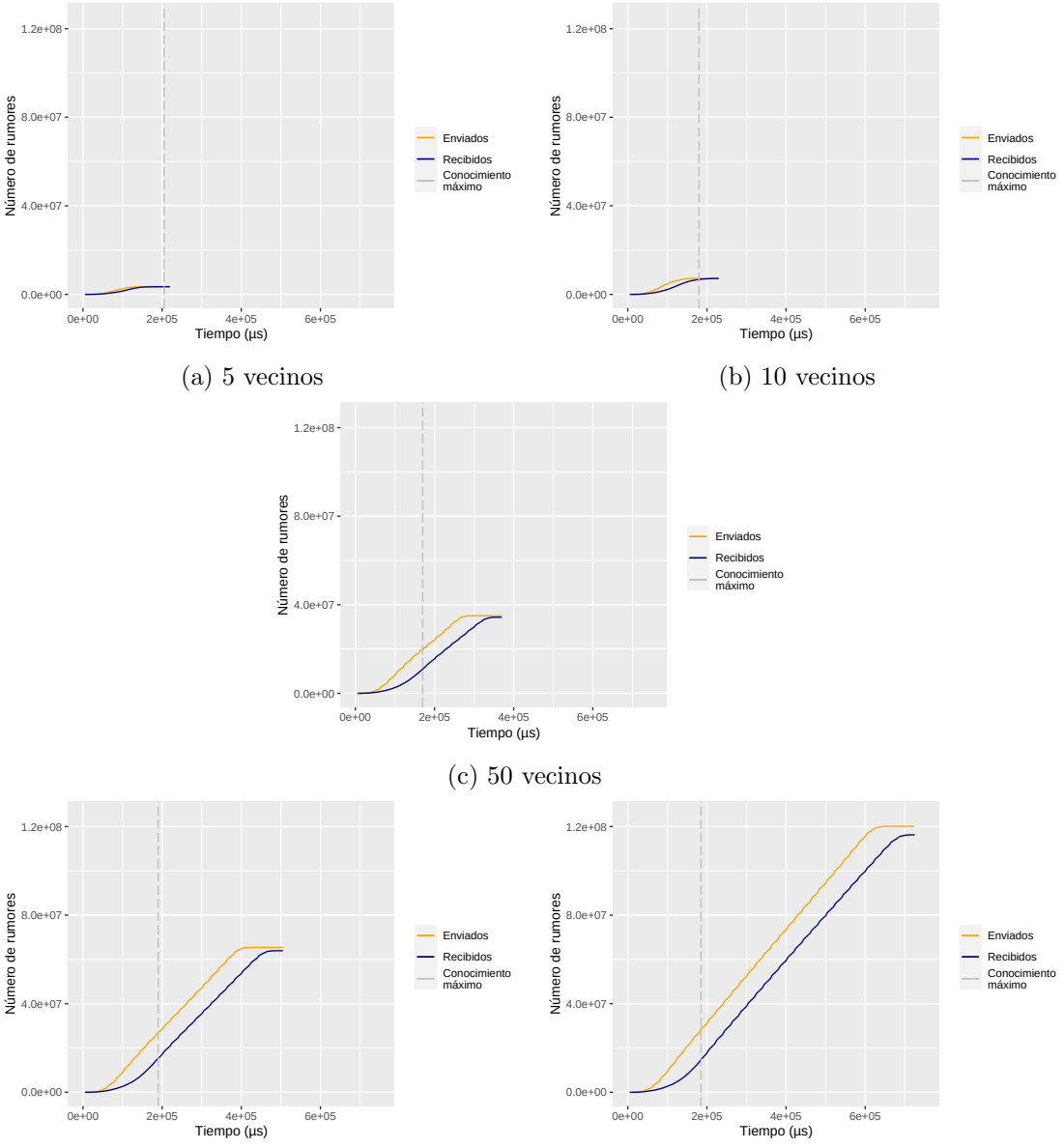
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.1: Paquetes transmitidos en función del tiempo sobre el algoritmo 1 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

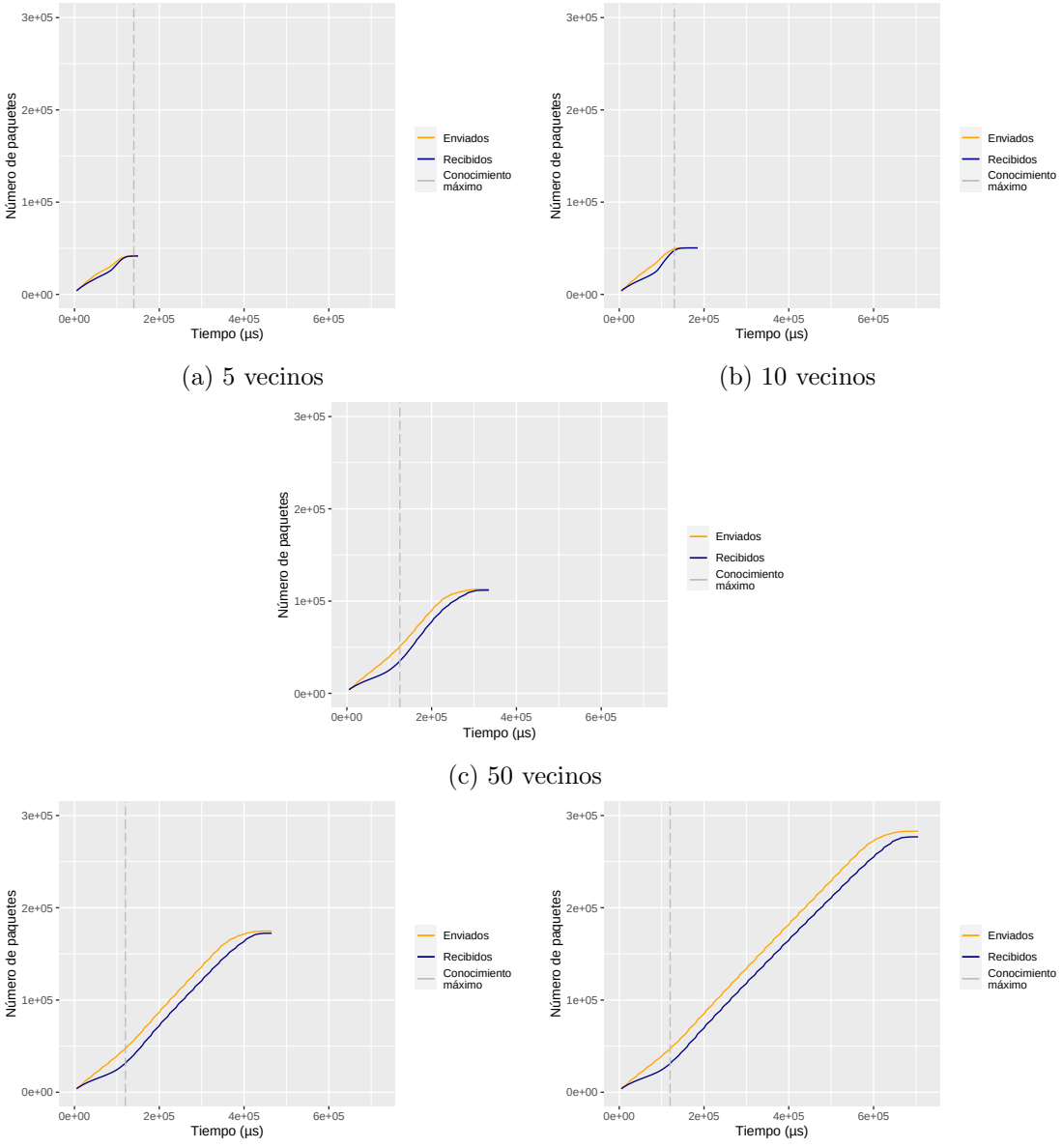
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.2: Rumores transmitidos en función del tiempo sobre el algoritmo 1 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

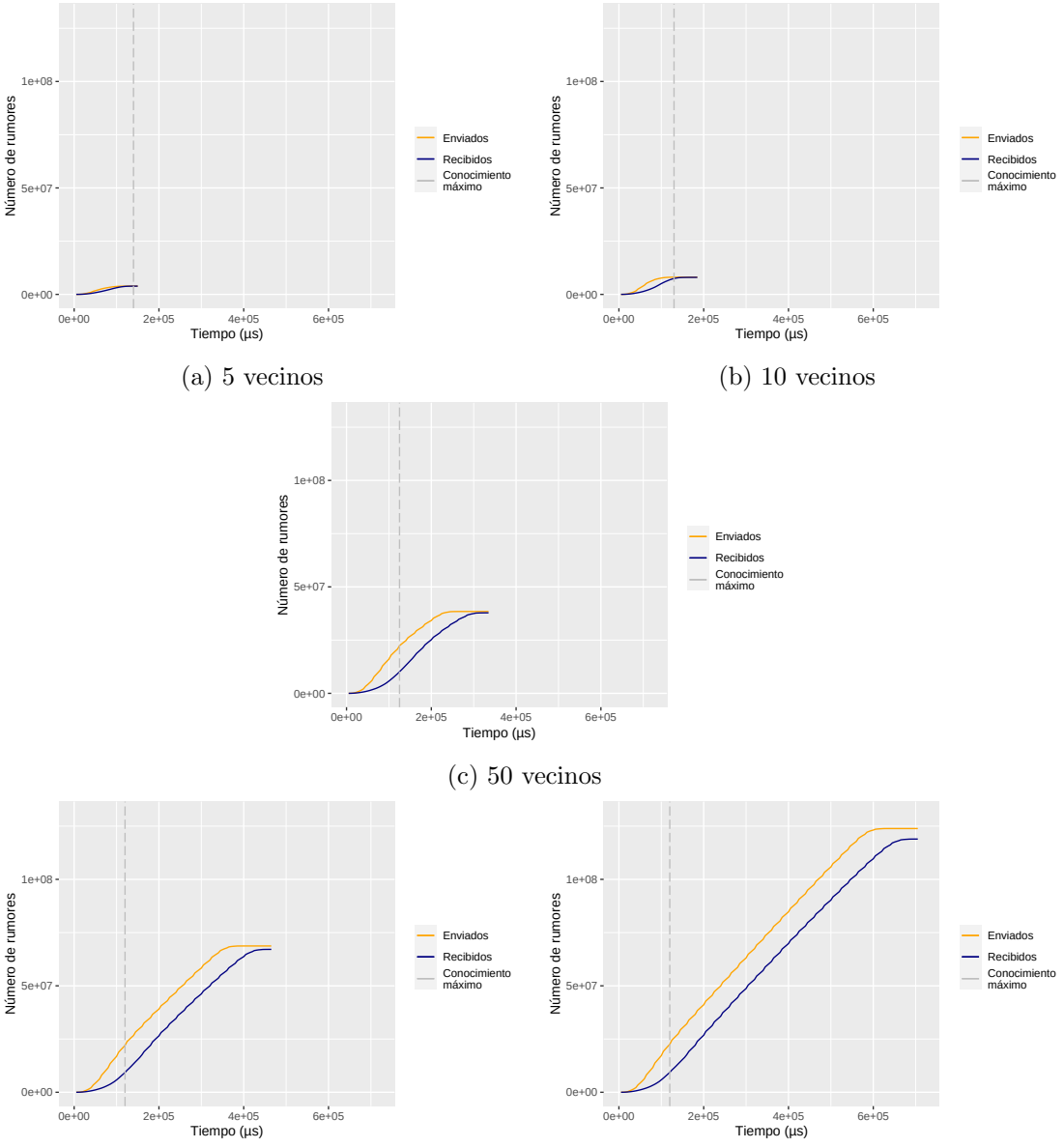
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.3: Paquetes transmitidos en función del tiempo sobre el algoritmo 2 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

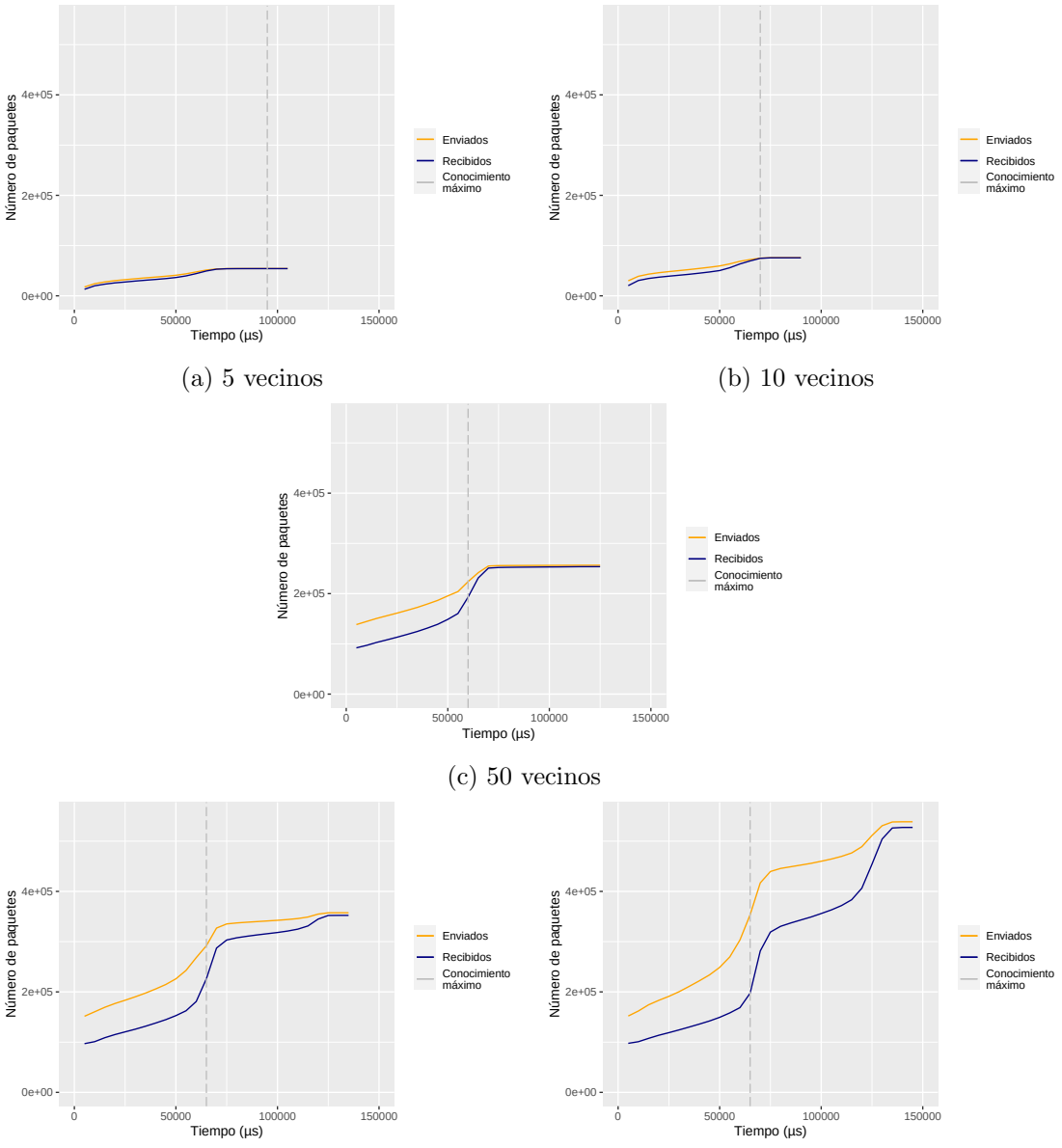
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.4: Rumores transmitidos en función del tiempo sobre el algoritmo 2 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

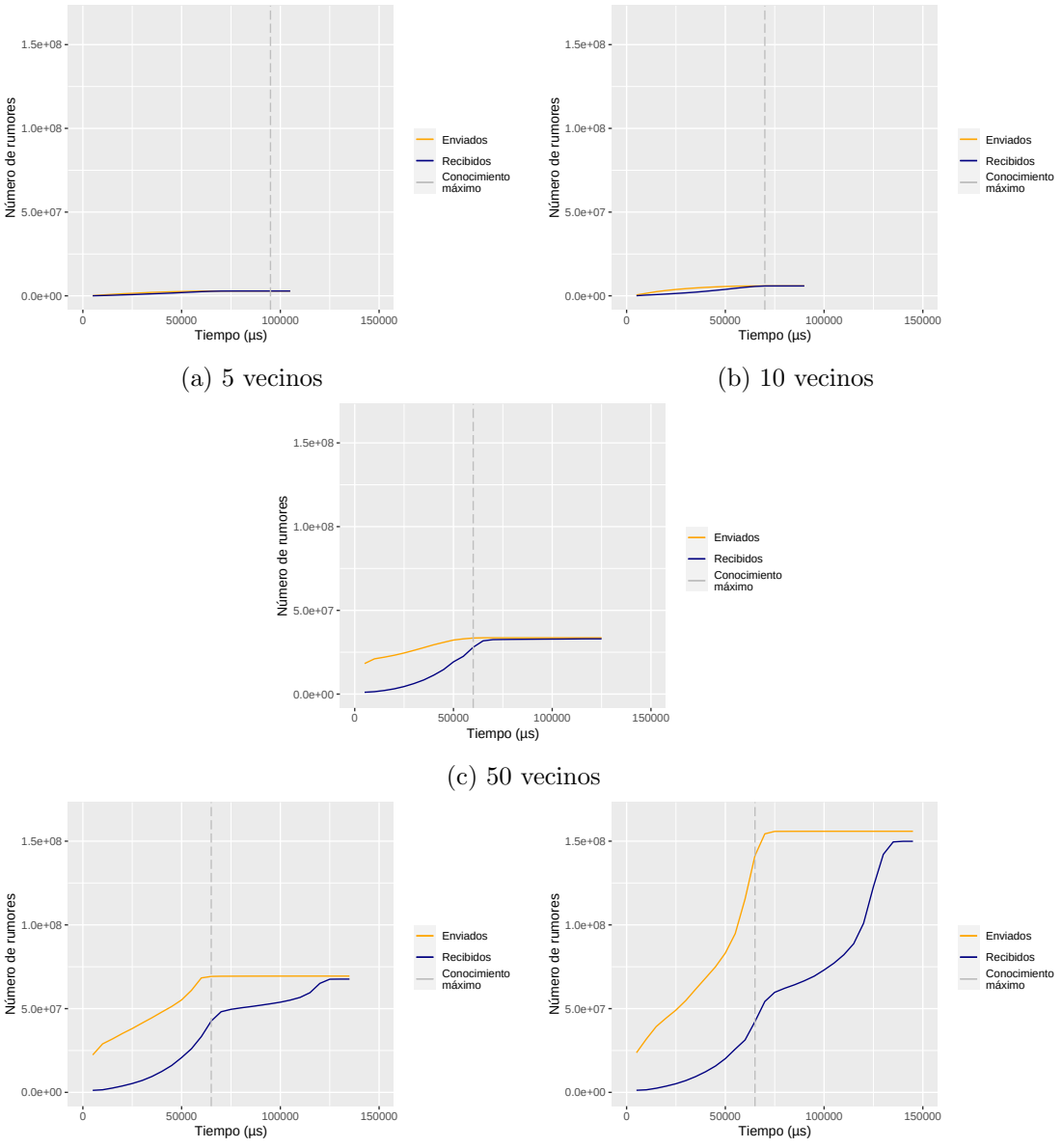
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.5: Paquetes transmitidos en función del tiempo sobre el algoritmo 3 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

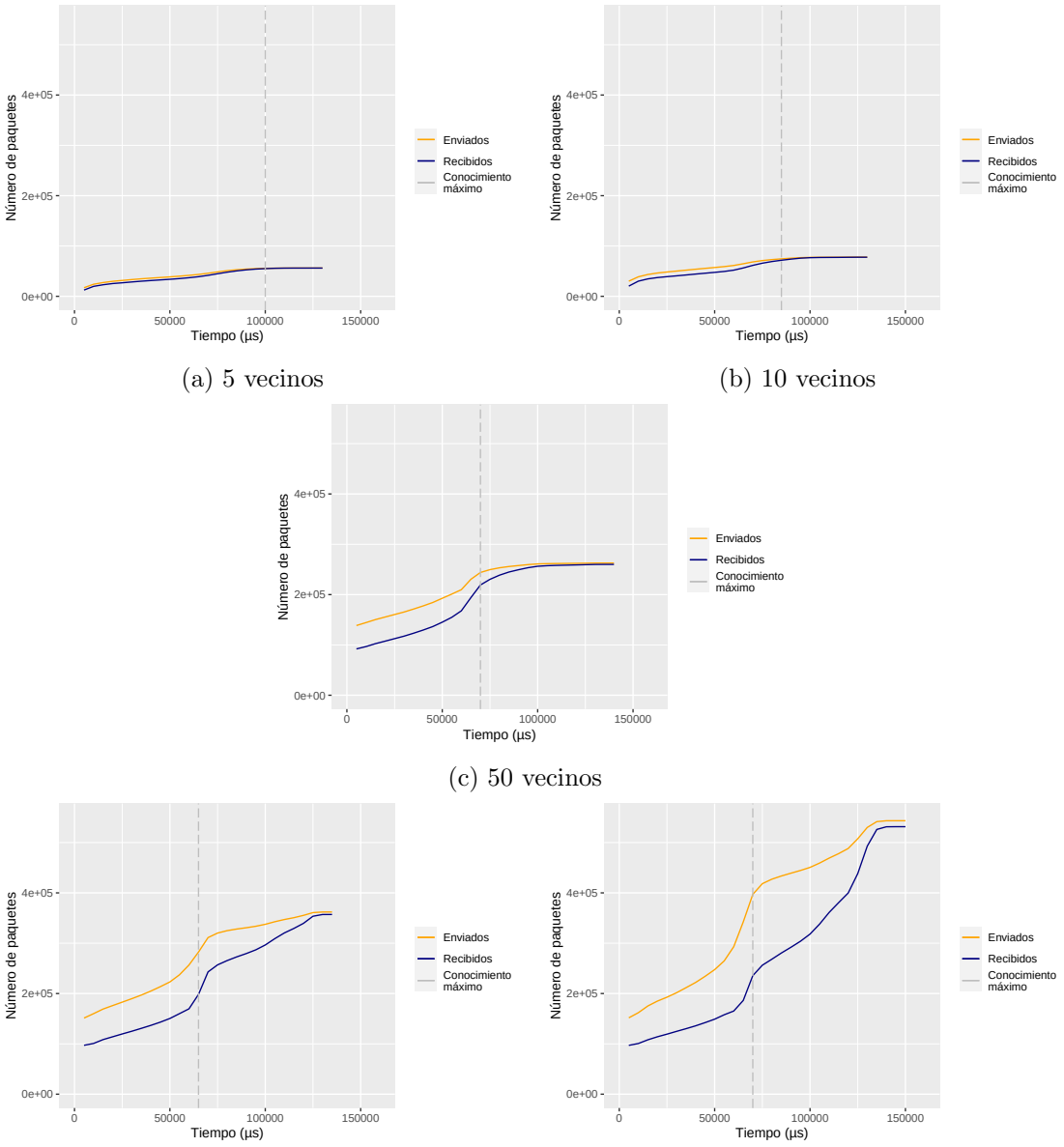
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.6: Rumores transmitidos en función del tiempo sobre el algoritmo 3 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

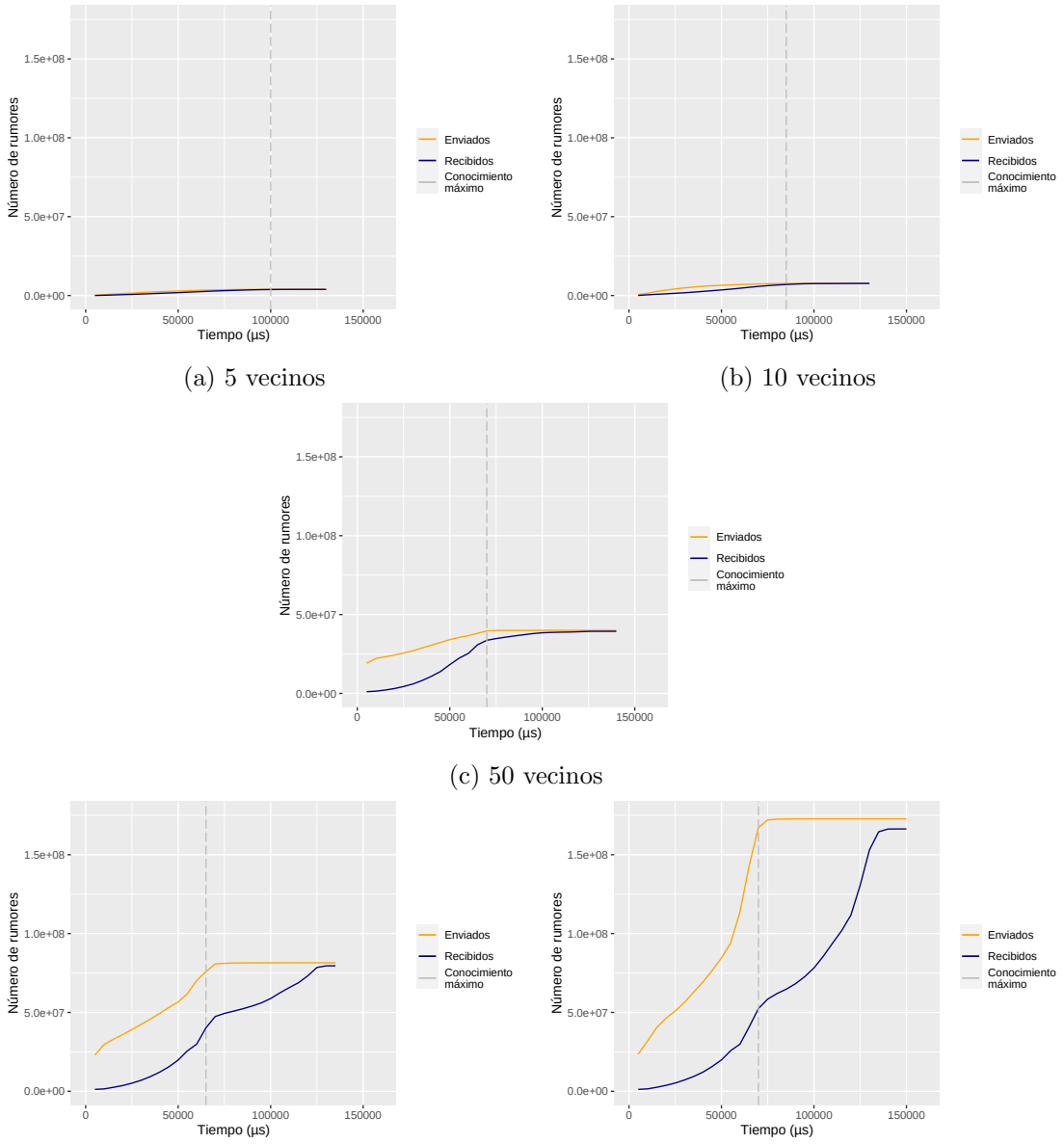
(b) 10 vecinos

(c) 50 vecinos

(d) 100 vecinos

(e) 200 vecinos

Figura 4.7: Paquetes transmitidos en función del tiempo sobre el algoritmo 4 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) 5 vecinos

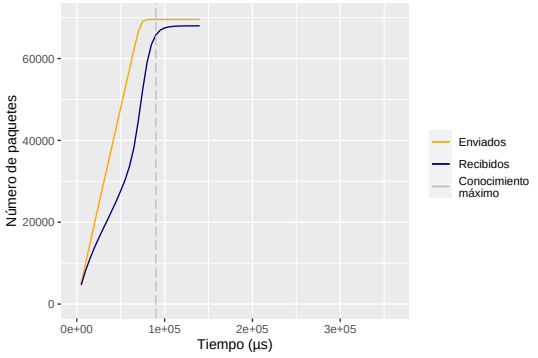
(b) 10 vecinos

(c) 50 vecinos

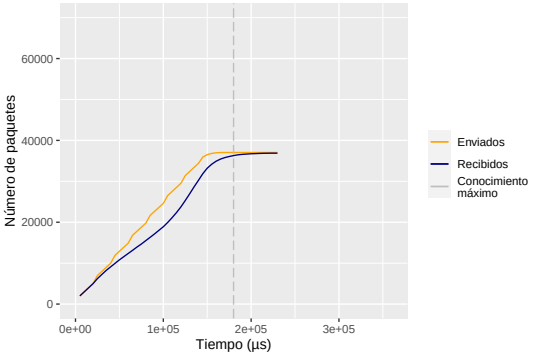
(d) 100 vecinos

(e) 200 vecinos

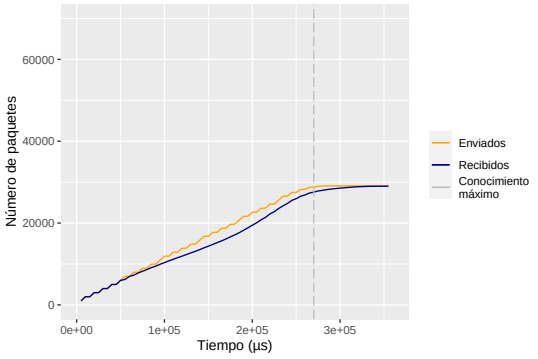
Figura 4.8: Rumores transmitidos en función del tiempo sobre el algoritmo 4 variando el número de vecinos. Simulación con 950 nodos correctos y $\tau = 4000$.



(a) $\tau = 1000$

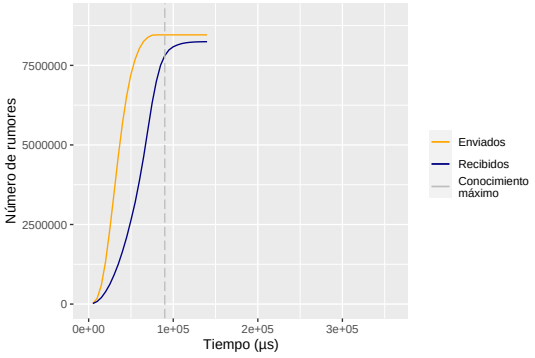


(b) $\tau = 4000$

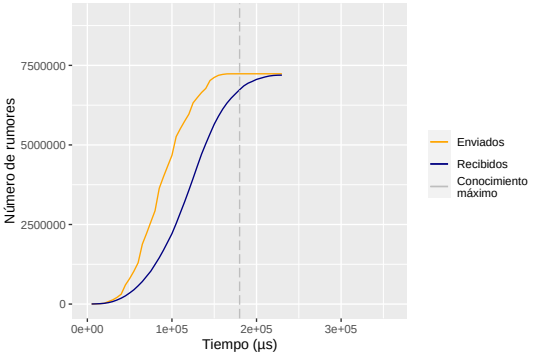


(c) $\tau = 9000$

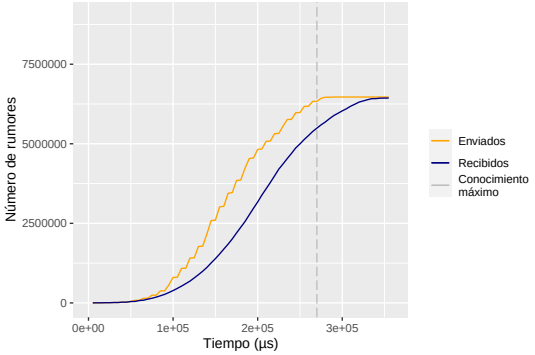
Figura 4.9: Paquetes transmitidos en función del tiempo sobre el algoritmo 1 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



(a) $\tau = 1000$

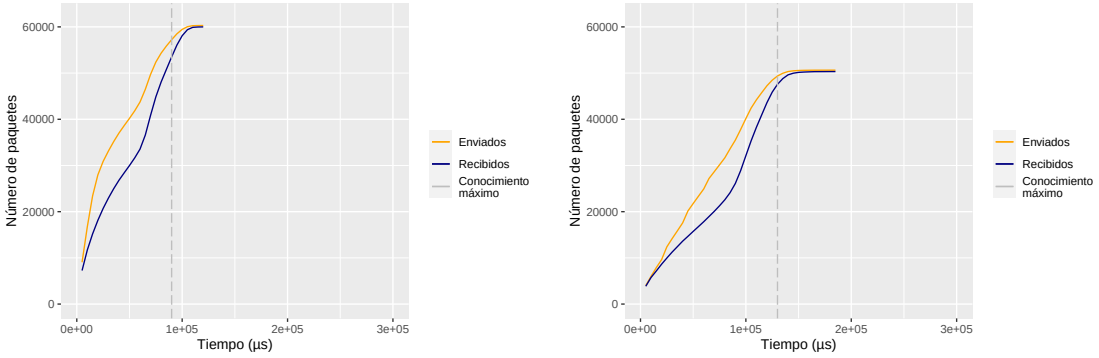


(b) $\tau = 4000$



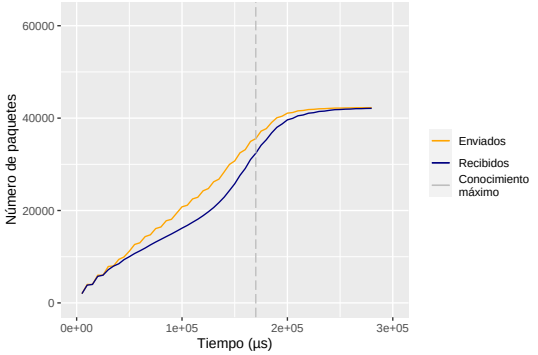
(c) $\tau = 9000$

Figura 4.10: Rumores transmitidos en función del tiempo sobre el algoritmo 1 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



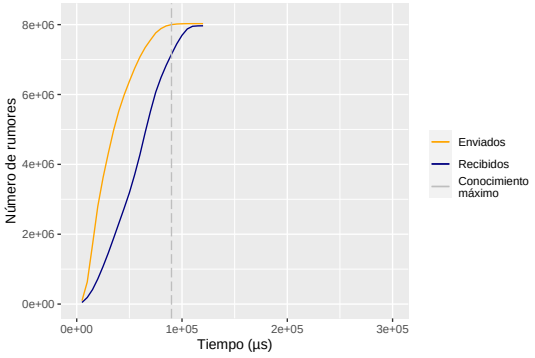
(a) $\tau = 1000$

(b) $\tau = 4000$

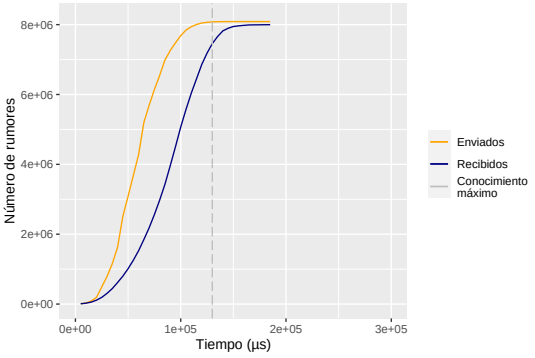


(c) $\tau = 9000$

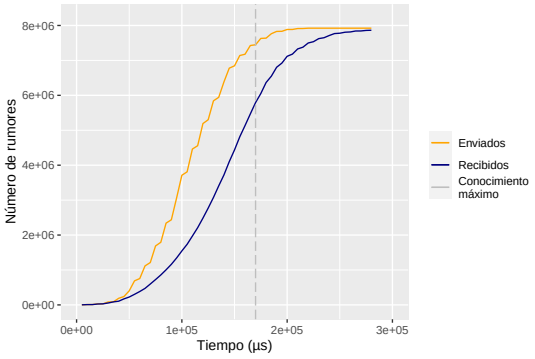
Figura 4.11: Paquetes transmitidos en función del tiempo sobre el algoritmo 2 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



(a) $\tau = 1000$

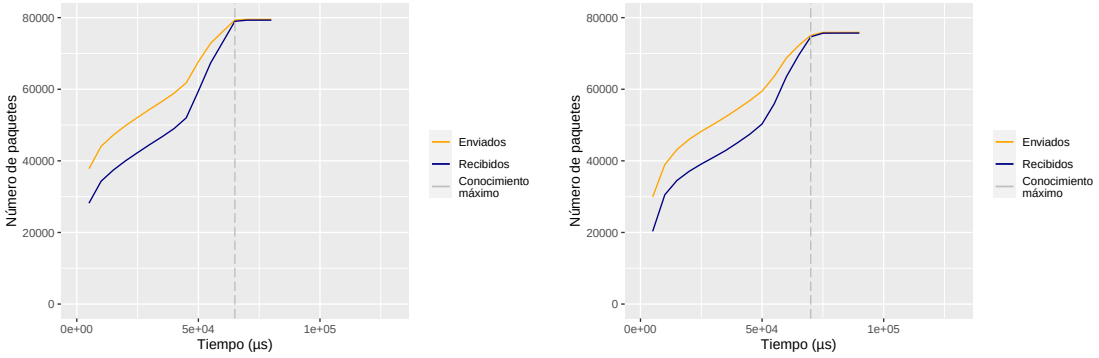


(b) $\tau = 4000$



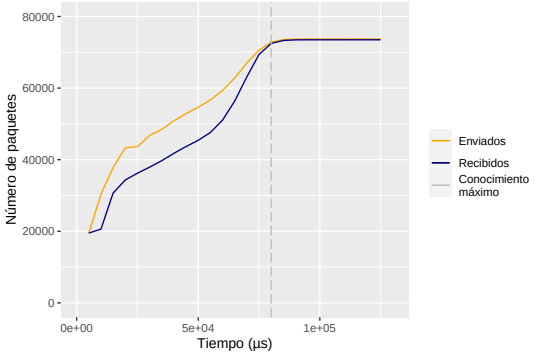
(c) $\tau = 9000$

Figura 4.12: Rumores transmitidos en función del tiempo sobre el algoritmo 2 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



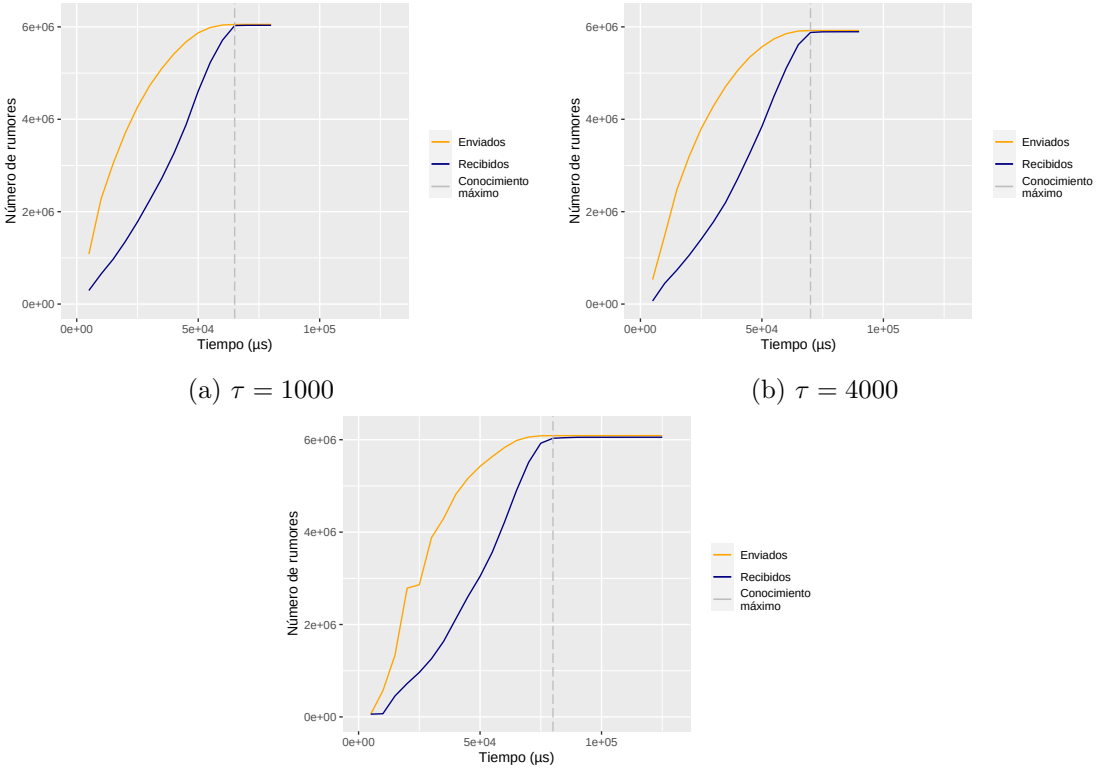
(a) $\tau = 1000$

(b) $\tau = 4000$



(c) $\tau = 9000$

Figura 4.13: Paquetes transmitidos en función del tiempo sobre el algoritmo 3 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.

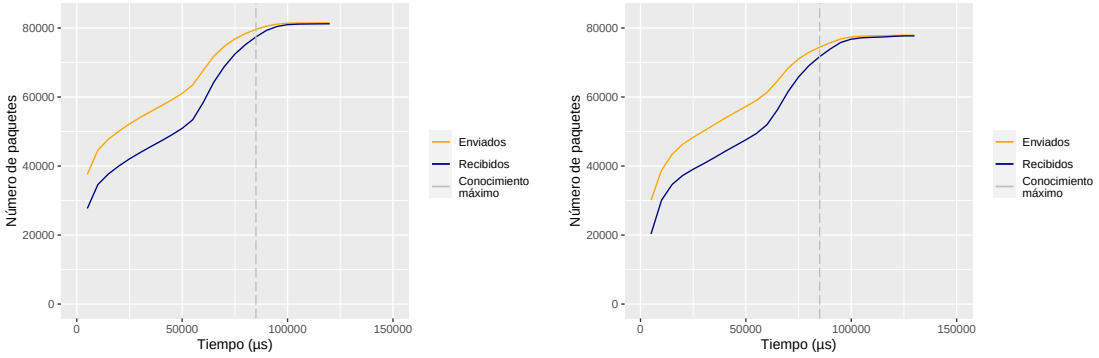


(a) $\tau = 1000$

(b) $\tau = 4000$

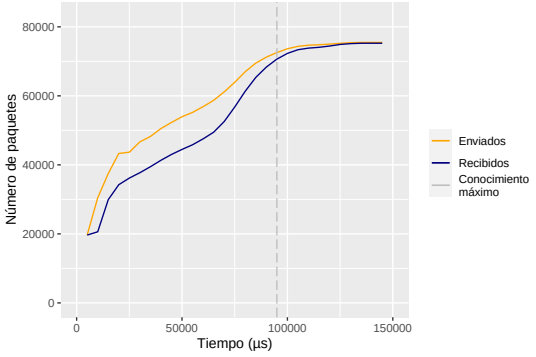
(c) $\tau = 9000$

Figura 4.14: Rumores transmitidos en función del tiempo sobre el algoritmo 3 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



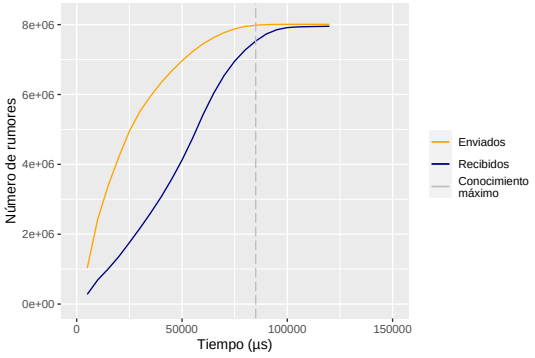
(a) $\tau = 1000$

(b) $\tau = 4000$

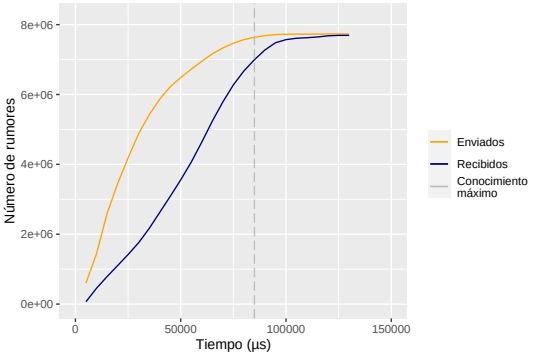


(c) $\tau = 9000$

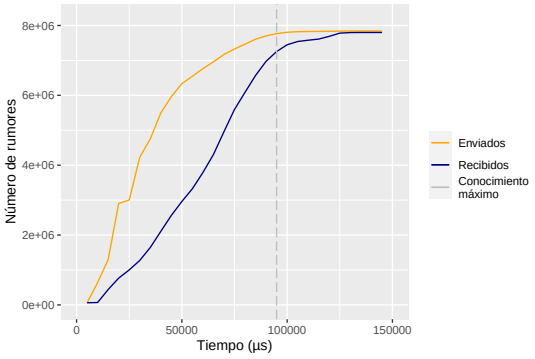
Figura 4.15: Paquetes transmitidos en función del tiempo sobre el algoritmo 4 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



(a) $\tau = 1000$

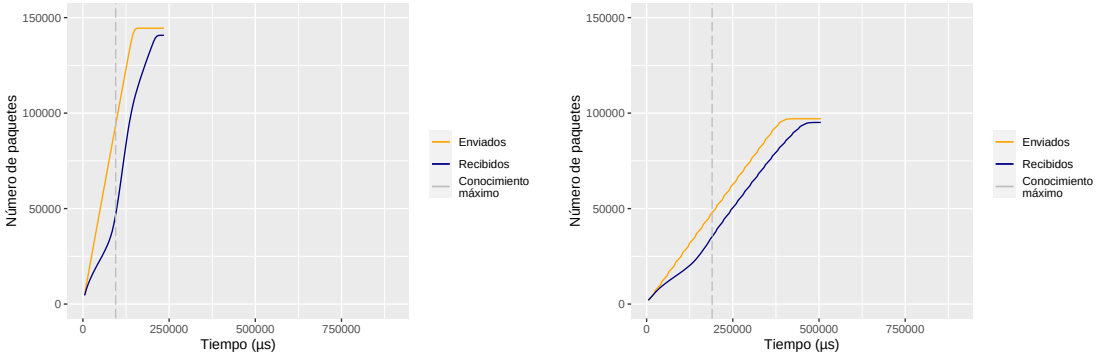


(b) $\tau = 4000$



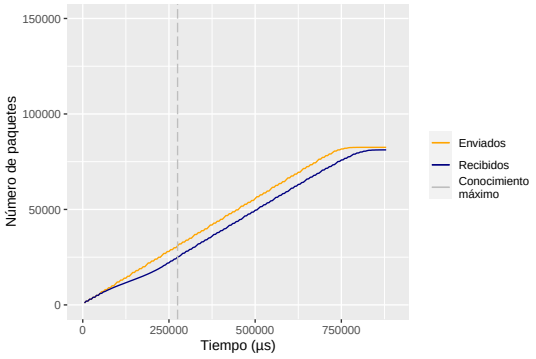
(c) $\tau = 9000$

Figura 4.16: Rumores transmitidos en función del tiempo sobre el algoritmo 4 variando el valor de τ. Simulación con 950 nodos correctos y 10 vecinos.



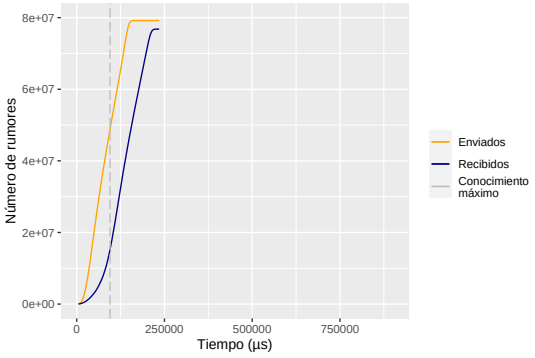
(a) $\tau = 1000$

(b) $\tau = 4000$

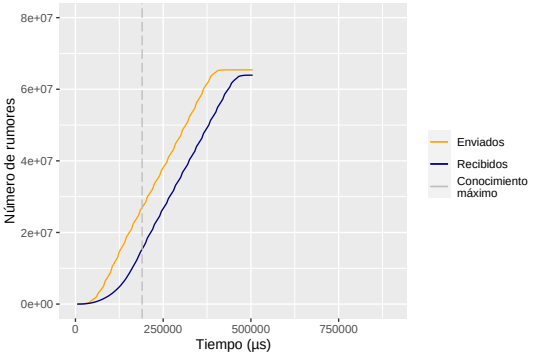


(c) $\tau = 9000$

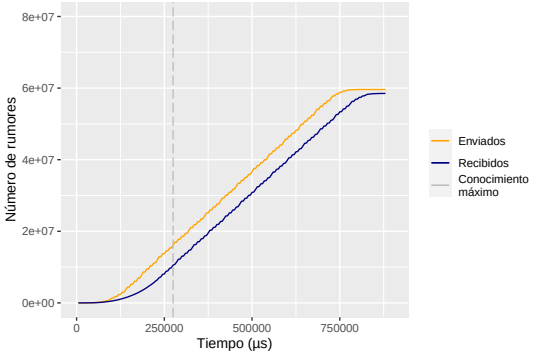
Figura 4.17: Paquetes transmitidos en función del tiempo sobre el algoritmo 1 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.



(a) $\tau = 1000$

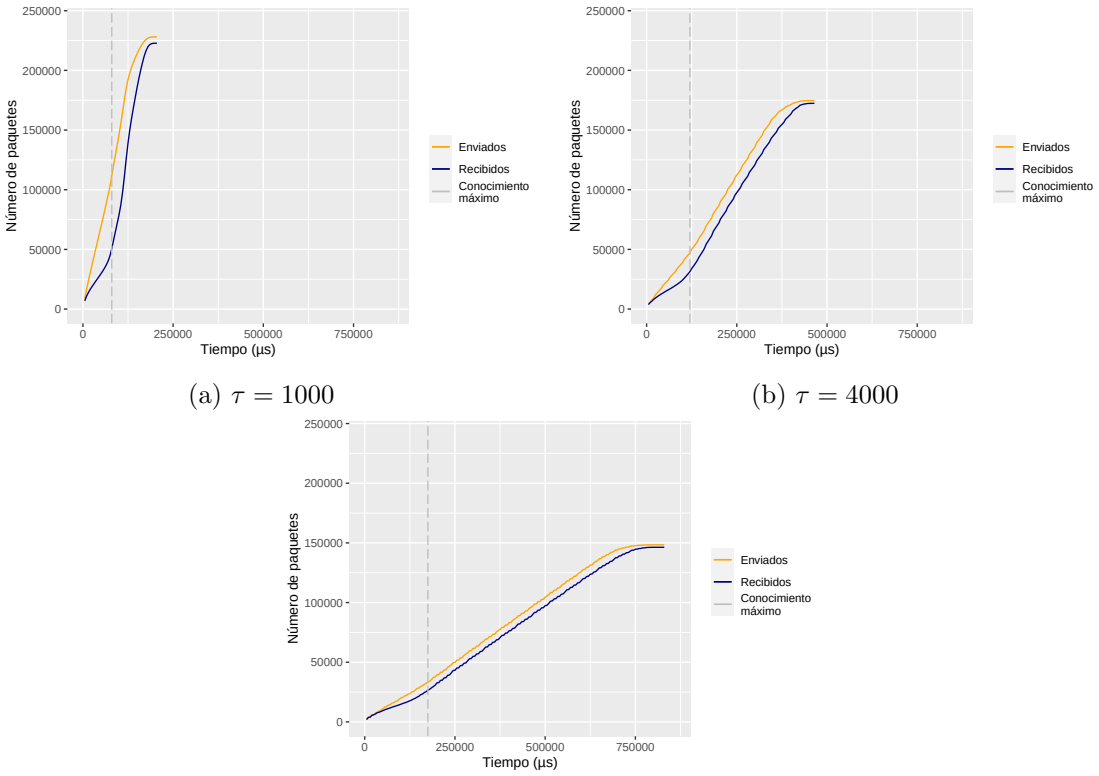


(b) $\tau = 4000$



(c) $\tau = 9000$

Figura 4.18: Rumores transmitidos en función del tiempo sobre el algoritmo 1 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.

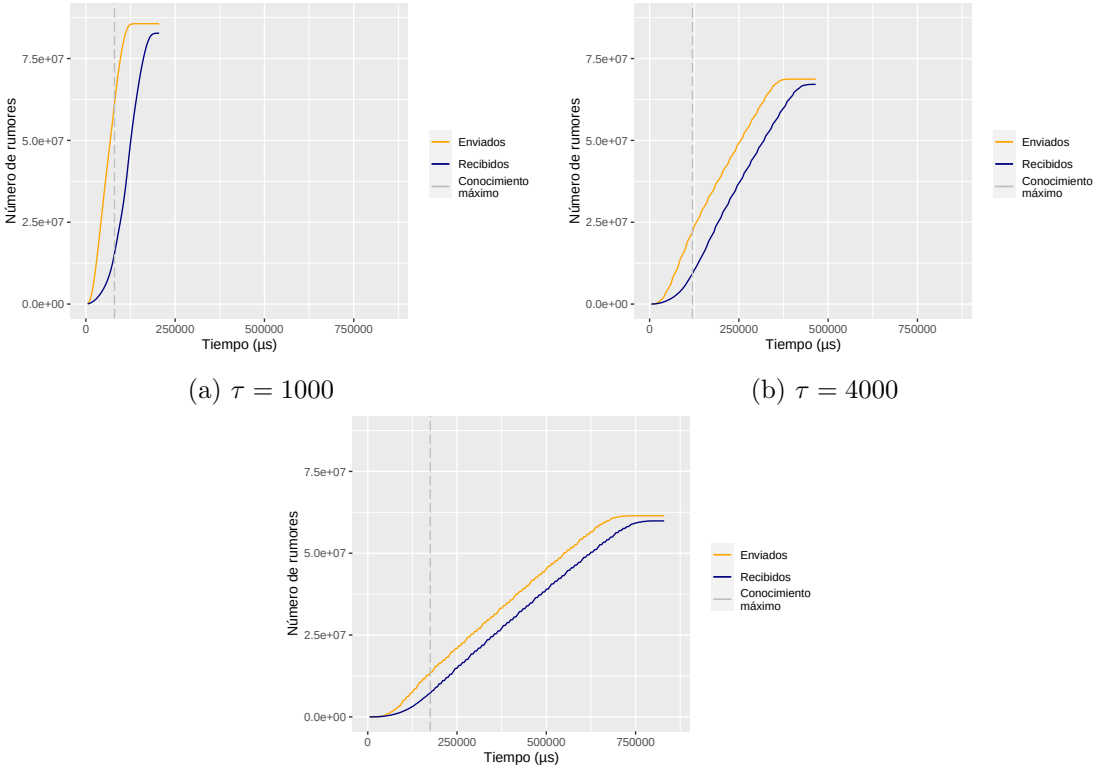


(a) $\tau = 1000$

(b) $\tau = 4000$

(c) $\tau = 9000$

Figura 4.19: Paquetes transmitidos en función del tiempo sobre el algoritmo 2 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.

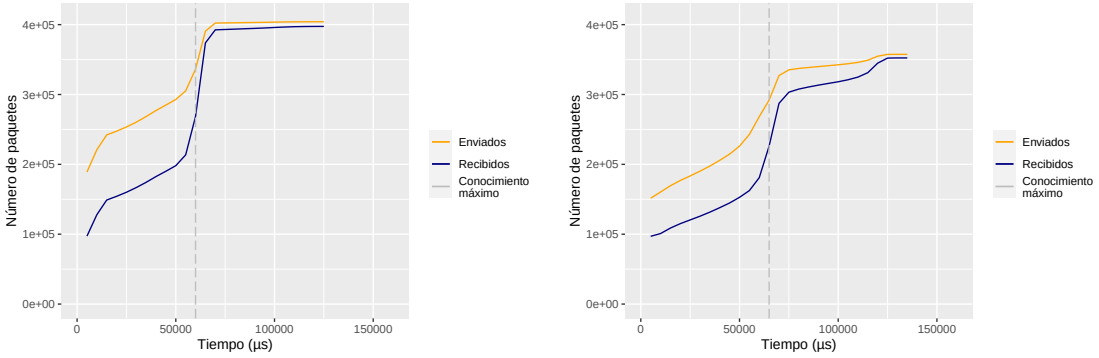


(a) $\tau = 1000$

(b) $\tau = 4000$

(c) $\tau = 9000$

Figura 4.20: Rumores transmitidos en función del tiempo sobre el algoritmo 2 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.

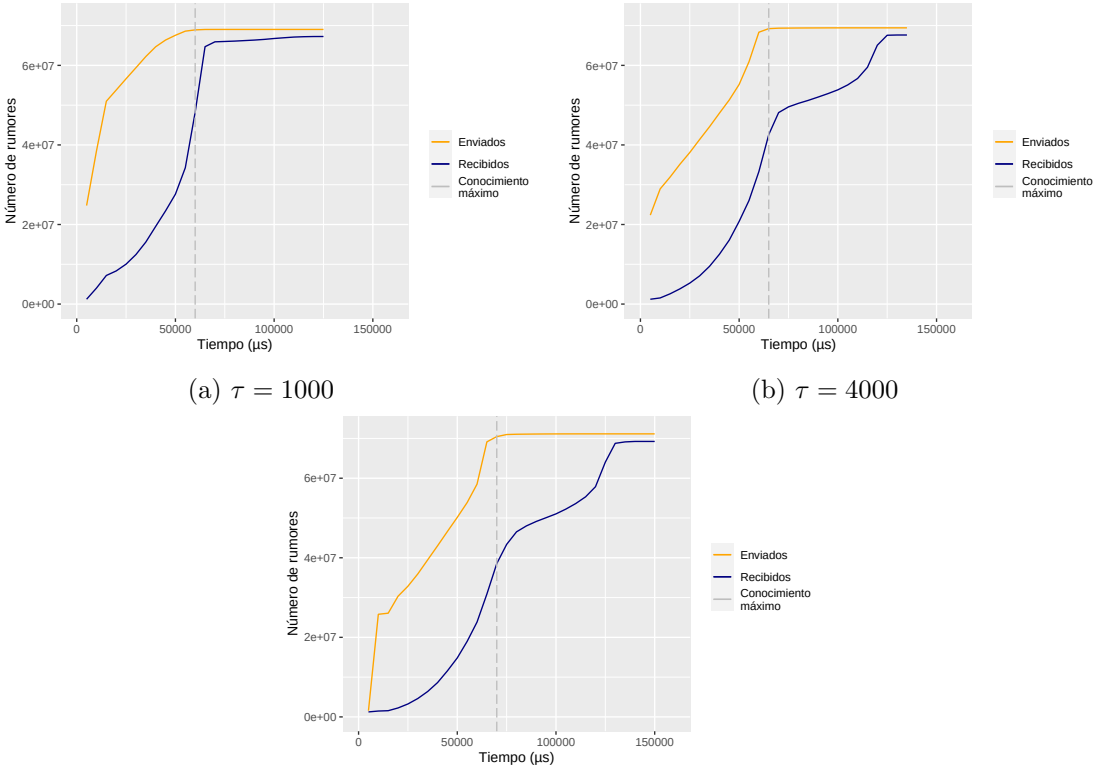


(a) $\tau = 1000$

(b) $\tau = 4000$

(c) $\tau = 9000$

Figura 4.21: Paquetes transmitidos en función del tiempo sobre el algoritmo 3 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.

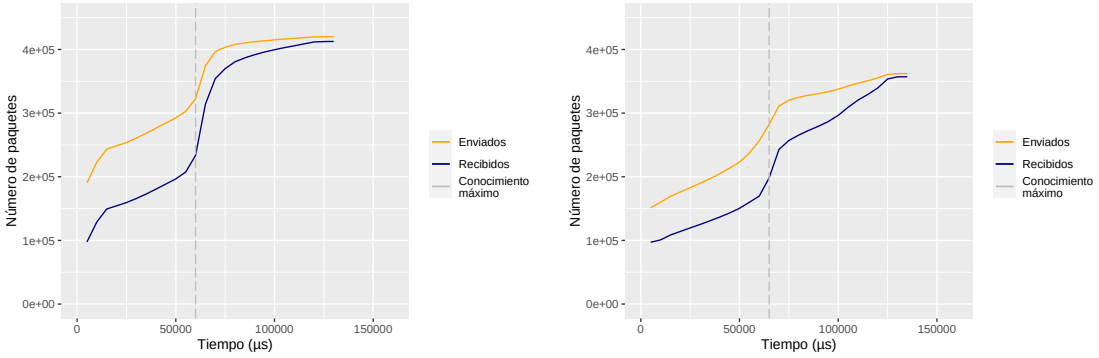


(a) $\tau = 1000$

(b) $\tau = 4000$

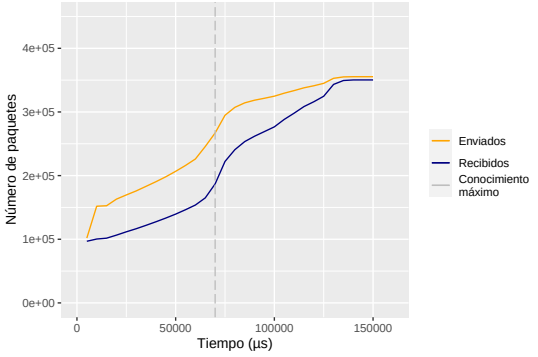
(c) $\tau = 9000$

Figura 4.22: Rumores transmitidos en función del tiempo sobre el algoritmo 3 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.



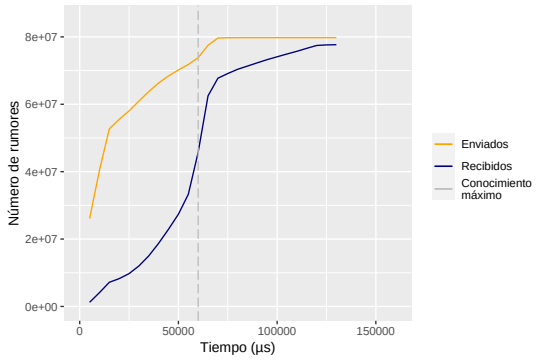
(a) $\tau = 1000$

(b) $\tau = 4000$

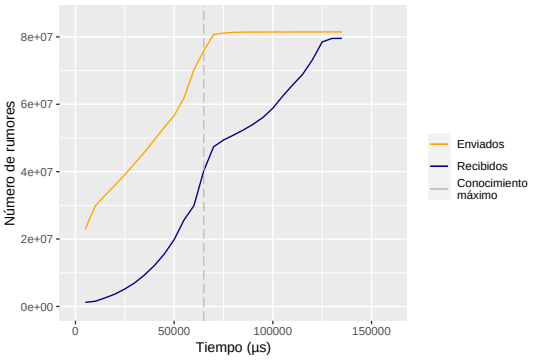


(c) $\tau = 9000$

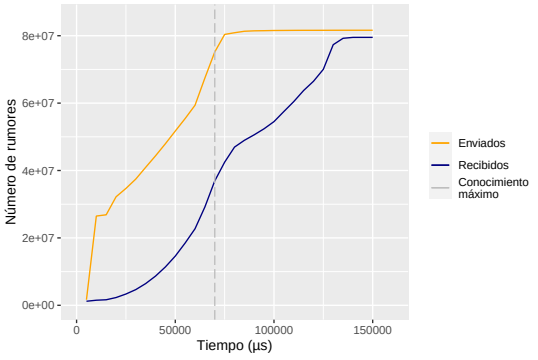
Figura 4.23: Paquetes transmitidos en función del tiempo sobre el algoritmo 4 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.



(a) $\tau = 1000$

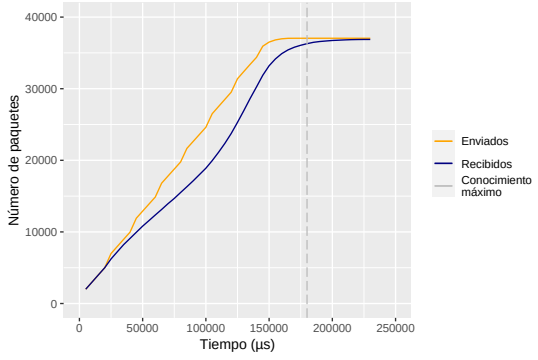


(b) $\tau = 4000$

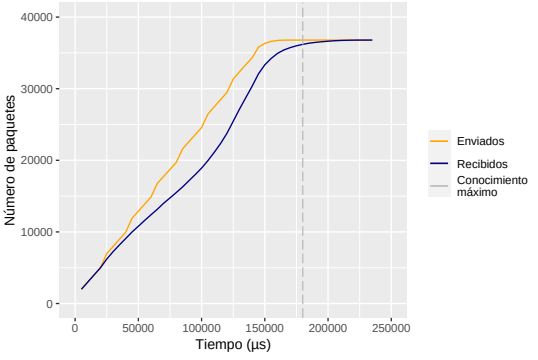


(c) $\tau = 9000$

Figura 4.24: Rumores transmitidos en función del tiempo sobre el algoritmo 4 variando el valor de τ. Simulación con 950 nodos correctos y 100 vecinos.

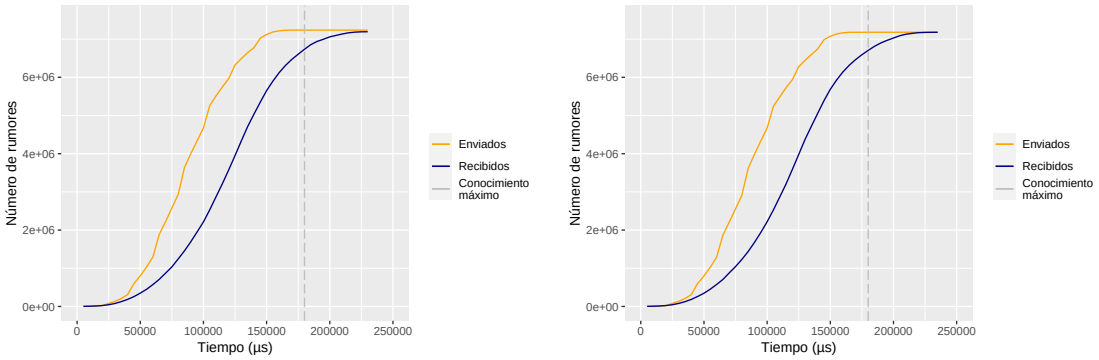


(a) 950 correctos



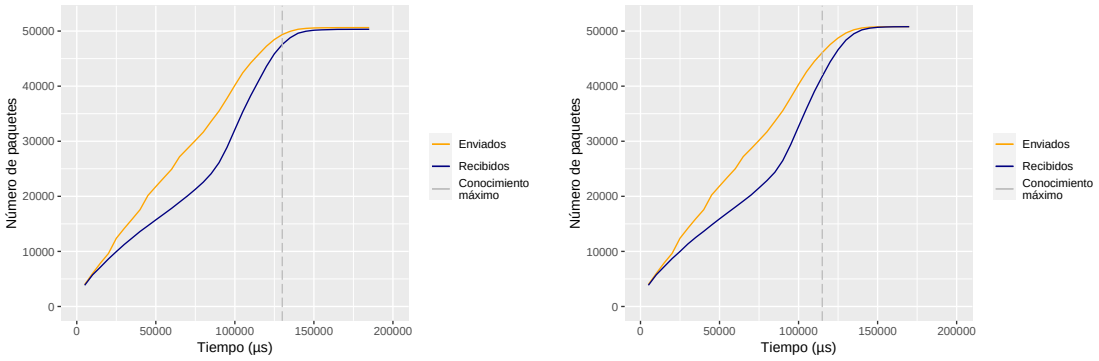
(b) 999 correctos

Figura 4.25: Paquetes transmitidos en función del tiempo sobre el algoritmo 1 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



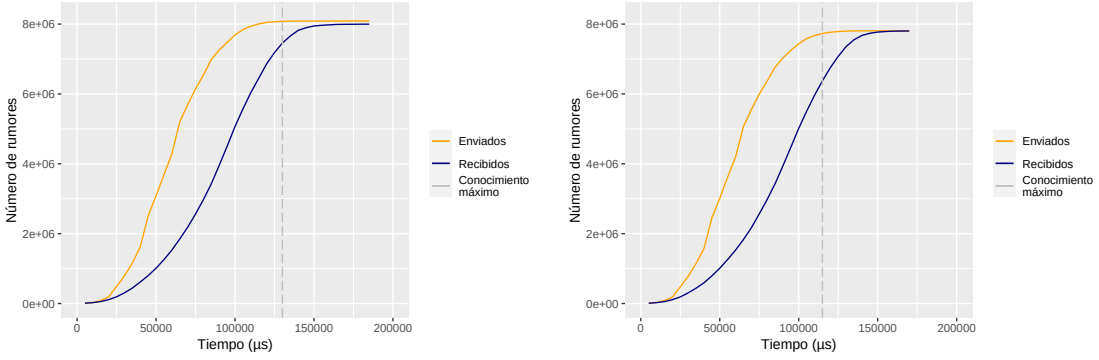
(a) 950 correctos (b) 999 correctos

Figura 4.26: Rumores transmitidos en función del tiempo sobre el algoritmo 1 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



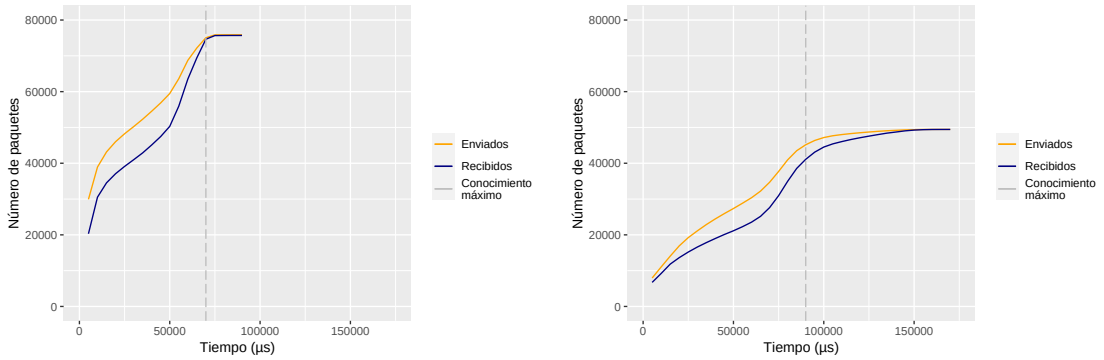
(a) 950 correctos (b) 999 correctos

Figura 4.27: Paquetes transmitidos en función del tiempo sobre el algoritmo 2 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



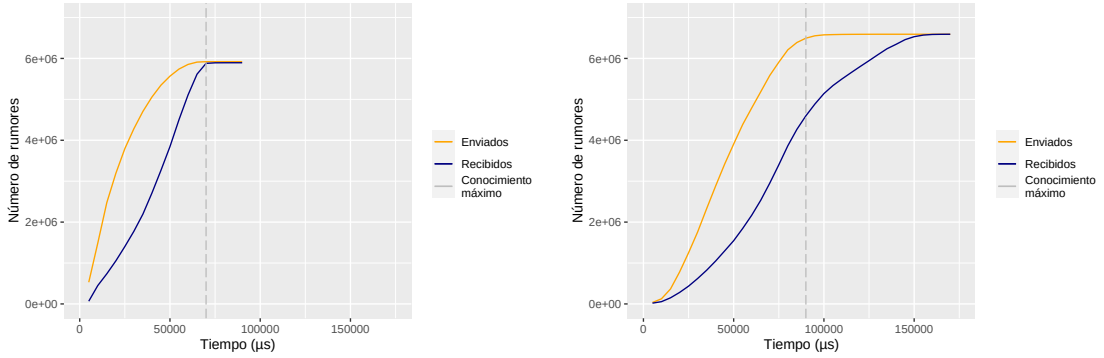
(a) 950 correctos (b) 999 correctos

Figura 4.28: Rumores transmitidos en función del tiempo sobre el algoritmo 2 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



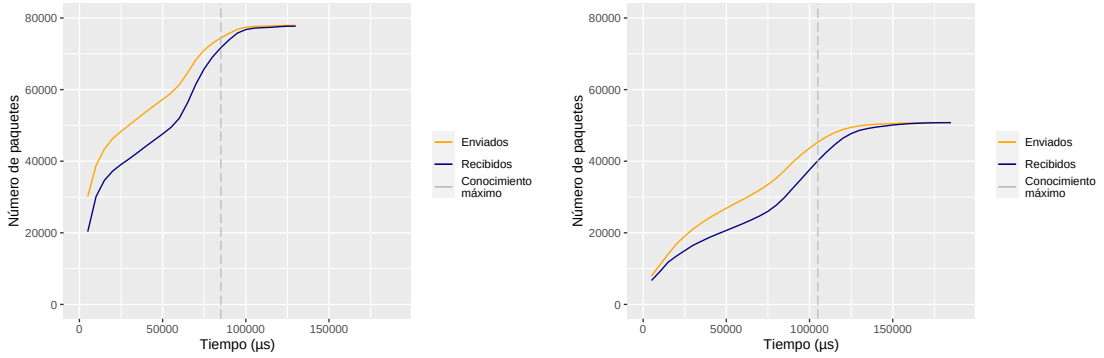
(a) 950 correctos (b) 999 correctos

Figura 4.29: Paquetes transmitidos en función del tiempo sobre el algoritmo 3 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



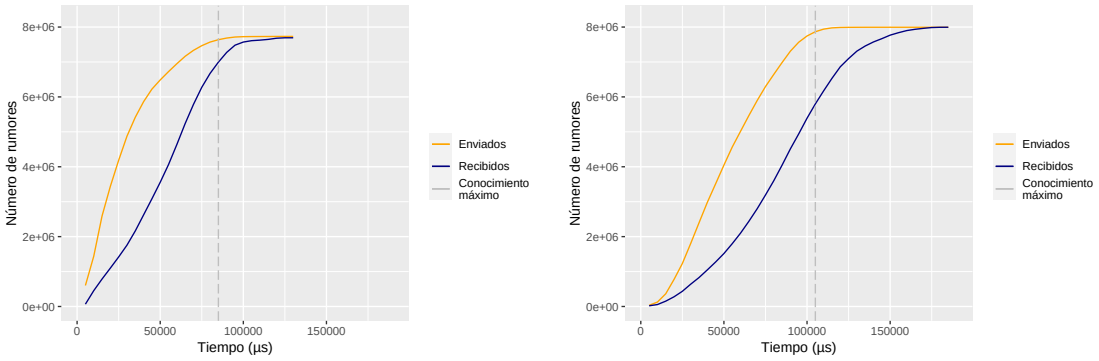
(a) 950 correctos (b) 999 correctos

Figura 4.30: Rumores transmitidos en función del tiempo sobre el algoritmo 3 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



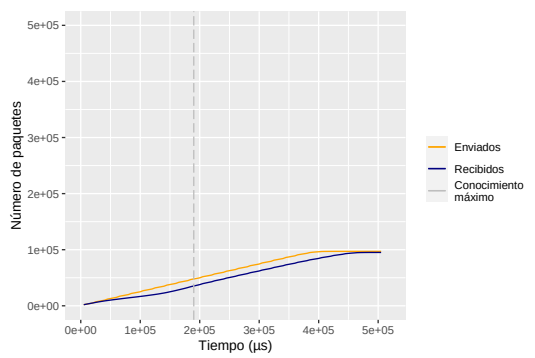
(a) 950 correctos (b) 999 correctos

Figura 4.31: Paquetes transmitidos en función del tiempo sobre el algoritmo 4 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.

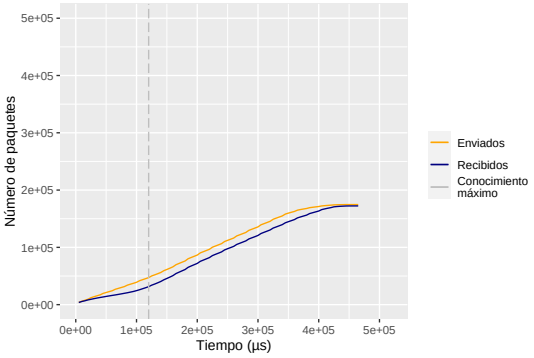


(a) 950 correctos (b) 999 correctos

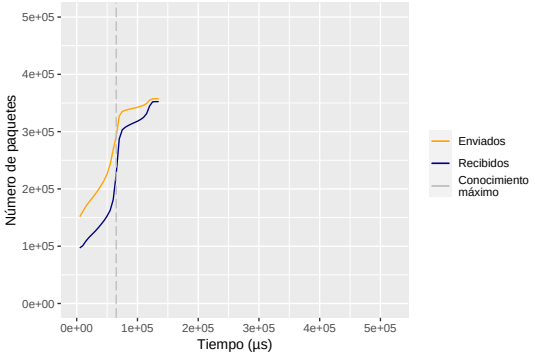
Figura 4.32: Rumores transmitidos en función del tiempo sobre el algoritmo 4 variando el número de nodos correctos. Simulación con 10 vecinos y $\tau = 4000$.



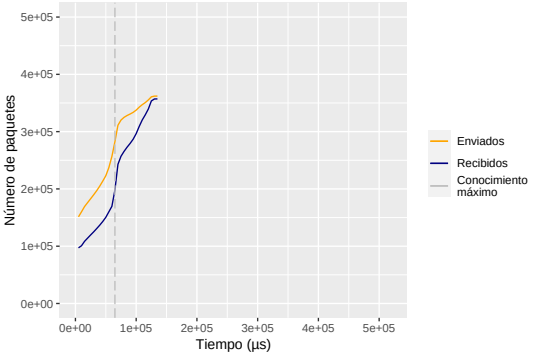
(a) Algoritmo 1



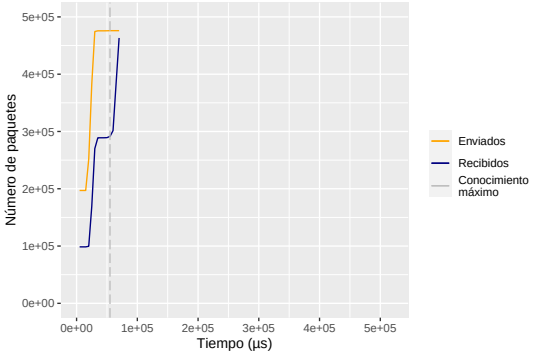
(b) Algoritmo 2



(c) Algoritmo 3

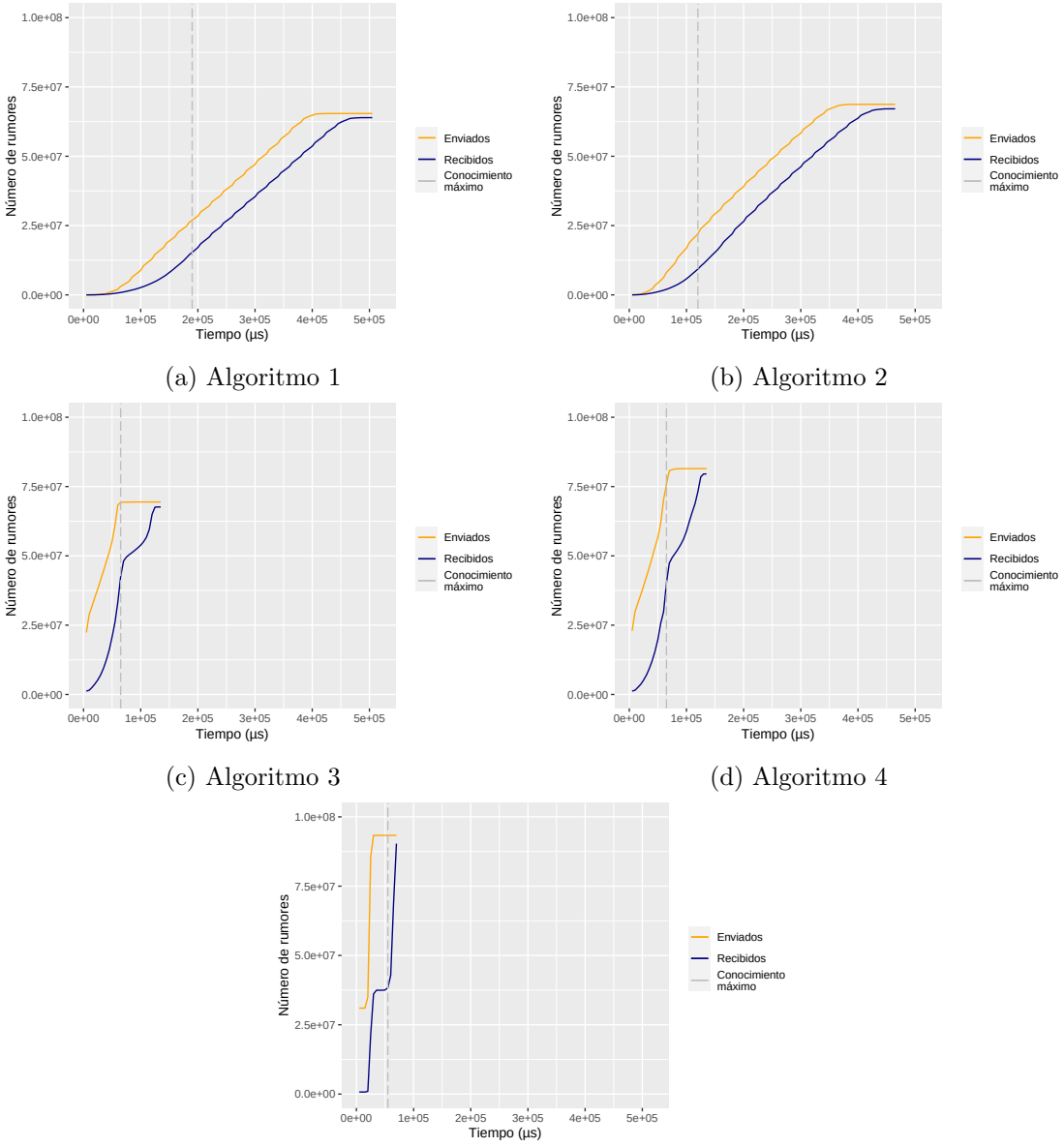


(d) Algoritmo 4



(e) Algoritmo de referencia

Figura 4.33: Paquetes transmitidos en función del tiempo sobre los cuatro algoritmos. Simulación con 100 vecinos, 950 nodos correctos y $\tau = 4000$.



(a) Algoritmo 1

(b) Algoritmo 2

(c) Algoritmo 3

(d) Algoritmo 4

(e) Algoritmo de referencia

Figura 4.34: Rumores transmitidos en función del tiempo sobre los cuatro algoritmos. Simulación con 100 vecinos, 950 nodos correctos y $\tau = 4000$.

Capítulo 5

Conclusiones

En este ultimo capítulo se van a comentar las conclusiones generales del trabajo como parte del proyecto de investigación y las realizaciones futuras que quedan pendientes, al no tener cabida en este debido a la limitación del tiempo.

Durante el desarrollo de este trabajo se han implementado en Java algoritmos desarrollados previamente por un grupo de investigación con la finalidad de realizar simulaciones para comprobar su funcionamiento. El simulador utilizado para ello ha sido una versión modificada de Peersim que mejora el rendimiento y la usabilidad. Además, se ha desarrollado un protocolo de difusión por inundación para utilizarlo como referencia a la hora de analizar los resultados. Tras esto, se han desarrollado dos *scripts* en R para extraer gráficas, de la información obtenida de las simulaciones, que puedan ser analizadas más fácilmente. Por último, se han comparado y analizado los resultados de distintas ejecuciones, seleccionadas por su interés, para comprobar el funcionamiento de los algoritmos.

De cara a los resultados obtenidos, los algoritmos se han comportado como se esperaba de manera teórica, sin provocar grandes sorpresas, aunque sí que se ha obtenido información nueva sobre ellos. La influencia directa del número de vecinos en la información redundante que se transmite y la escasa influencia que tiene la variación de τ sobre los algoritmos 3 y 4, son ejemplos de relaciones que no se habían planteado antes de las simulaciones.

Comparando los algoritmos entre sí, no se ha encontrado ninguno que fuese mejor en todos los aspectos que el resto. Dependiendo de las condiciones y de la importancia que se le de al tiempo, la energía y la memoria (ya que estas son las características que se busca optimizar), es más eficiente uno u otro, por lo que es necesario seguir realizando pruebas que revelen más información. Como era previsto, los algoritmos 1 y 2 ahorran más en paquetes transmitidos, y el 3 y 4 llegan al estado de reposo más rápido. Además, al ahorrar en memoria, los algoritmos 2 y 4 sacrifican eficiencia en las otras dos características.

Como conclusión general del trabajo, se ha cumplido con todos los objetivos propuestos, por lo que se puede decir que el resultado ha sido satisfactorio. A pesar de esto, han surgido inconvenientes que tienen origen en la escasa experiencia de la que se partía en el ámbito de las simulaciones de protocolos o algoritmos sobre una red, pero estos se han conseguido sobrepasar, logrando (como se ha dicho) alcanzar todos los objetivos. Tras el trabajo, queda toda la experiencia obtenida que se podrá aplicar de cara a continuar con el estudio de los algoritmos y en propuestas análogas.

Las posibles lineas de mejora para un trabajo similar, que se han encontrado a lo largo del desarrollo de este, son las descritas a continuación: de cara a optimizar el tiempo

para llevar a cabo el proyecto, es necesario otorgarle la suficiente dedicación al análisis del problema y a los objetivos que se buscan, para que a la hora de ejecutarlo no se realice ningún sobresfuerzo y los resultados obtenidos no tengan que ser procesados en exceso. Por otro lado, es muy importante tener en cuenta la influencia de la topología a la hora de realizar pruebas de protocolos o algoritmos, ya que se pueden comportar de formas muy distintas dependiendo de esta. Además, si se quiere comprobar la influencia de una variable concreta, es recomendable realizar las distintas simulaciones sobre la misma red. Esto permite minimizar las variaciones producidas por agentes externos.

Como tareas a futuro tras la realización de este trabajo, se dejan los siguientes puntos:

- Continuar realizando simulaciones y estudiar en profundidad las diferencias entre los algoritmos. Con las simulaciones realizadas se obtiene una idea de la respuesta de los algoritmos, pero es necesario continuar con un estudio detallado de sus características más relevantes (como la cantidad de paquetes enviados o el tiempo que tarda en llegar al estado de reposo) para establecer reglas o ecuaciones que puedan clasificarlos en diferentes escenarios o gráficas que indiquen cuándo son más óptimos. Por otra parte, se puede seguir profundizando sobre el efecto de otros parámetros de configuración en los algoritmos realizando más pruebas, por ejemplo, si influye el tipo de fallo que se establece en los nodos no correctos. También se pueden realizar simulaciones alterando nuevos parámetros en conjunto, para comprobar si existe resonancia entre ellos que alteren de forma drástica la respuesta de los algoritmos. Un ejemplo de esto puede ser variar el número de vecinos por nodo, pero manteniendo una misma proporción de vecinos frente a nodos no correctos (para que el *fanout* de los algoritmos 3 y 4 siempre tenga la misma proporción de vecinos).
- En la linea del punto anterior, se puede realizar un estudio más en profundidad de algunos escenarios con comportamientos extraños de los algoritmos de los que no se ha encontrado un patrón. Todos ellos se han comentado, pero podemos destacar los siguientes:
  - La influencia de los nodos no correctos sobre los rumores transmitidos en los algoritmos 2, 3 y 4. A pesar de que tras el estudio se han detectado alteraciones, es necesario realizar más simulaciones que indiquen claramente su tendencia, pues no concuerda con lo esperado.
  - El comportamiento del instante de conocimiento máximo del algoritmo 1 al variar el número de vecinos de cada nodo. En las comparaciones se ha detectado que este algoritmo no se comporta como el resto, por lo que se debe estudiar más en profundidad este caso.
  - La variación en el número de rumores transmitidos en el algoritmo 4 cuando se aumenta el valor de τ. En las pruebas realizadas da la sensación que este parámetro tiende a subir levemente cuanto mayor es el valor de τ, por lo que se debe analizar con mayor detalle para aclarar el suceso, que es contrario a lo esperado teóricamente.
- Añadir el código necesario para poder importar y exportar la topología de la red sobre la que se va a simular. Esto permite dos beneficios a la vez. Por un lado, se pueden comparar simulaciones con exactamente la misma red, lo que disminuye las variaciones que puedan surgir de esta fuente. Y por otro lado, se pueden comparar simulaciones que posean topologías específicas para comprobar los efectos de las distintas topologías de red sobre los algoritmos.

- Cambiar las características de la red sobre la que se simula. Todas las simulaciones se han realizado sobre la red descrita en el Capítulo 3. Sin referirnos a la topología (pues ya se habla en el punto anterior sobre ella) la única característica especial es que los nodos podían caerse. Se puede ampliar esto para comprobar la tolerancia a errores de los algoritmos, añadiendo enlaces no fiables, por ejemplo. También se puede programar que los nodos puedan volver a levantarse, si se habían caído, para comprobar la respuesta de los algoritmos.
- Optimizar el proceso de simulación, ya sea mediante variación en el código desarrollado o mediante otro simulador, para poder realizar simulaciones con más nodos. Debido a cómo ha sido desarrollado el código sobre el simulador (se buscaba que la simulación se asemejase lo máximo posible a lo que ocurre en la realidad), existen limitaciones en cuanto al uso de nodos por culpa de la memoria. Por tanto, para realizar simulaciones de más de 1000 nodos es necesario desarrollar un código centrado exclusivamente en la eficiencia de los procesos que se realizan. Además, se puede aprovechar el código desarrollado en este trabajo para ejecutar dos simulaciones con los mismos parámetros de configuración, una con cada programa. Los resultados deberían ser extremadamente similares (sino, es indicativo de que existen defectos en alguno).
- Tras el estudio de los algoritmos y su respuesta ha surgido la idea de uno nuevo. Este algoritmo busca beneficiarse del concepto del *fanout*, pero en vez de ligarlo al número de vecinos no correctos de la red, se define en proporción al número de vecinos del nodo (o de vecinos medios de la red). Para comprobar su eficiencia frente al resto, es necesario repetir el proceso de los otros algoritmos, programando un protocolo con esta lógica para que pueda ser simulado y comparado con el resto.

Apéndice A

Herramientas utilizadas

En este apéndice se van a mencionar todas las herramientas utilizadas durante el desarrollo de este proyecto, a pesar de que ya se han citado alguna de ellas en otros capítulos de esta memoria, incluyendo una breve explicación de su uso en caso de que no se haya explicado previamente.

Comenzamos por el código de las simulaciones. Como ya se ha explicado, el simulador utilizado es una versión mejorada de Peersim [9, 10]. Como Peersim está programado enteramente en Java [35], este es el lenguaje de programación utilizado al programar el código. Para facilitar esta tarea, se ha utilizado el entorno de desarrollo (IDE por sus siglas en inglés) desarrollado por Eclipse [46], de su mismo nombre. La decisión de utilizar este entorno proviene solamente por ser con el que más se está familiarizado.

Para el análisis y procesado de los datos se ha utilizado R [11] por los motivos ya expuestos en el capítulo 4, pero el entorno de desarrollo utilizado ha sido RStudio [47] en vez del nativo de R. El motivo de esto es que RStudio provee de un marco de trabajo más amigable que permite una mejor interacción y experiencia. Concretando más, para la creación de las gráficas se ha utilizado el paquete ggplot2 [48] por su flexibilidad, su simpleza y por que las gráficas que genera son más vistosas.

Por último, para el desarrollo de la memoria se ha empleado el sistema de composición de textos LaTeX [49], ya que ofrece control total sobre las herramientas y la información, y a que otorga una alta calidad tipográfica al documento. Como editor de este sistema, se ha elegido a Overleaf [50]. Este es un editor en línea de LaTeX que otorga la posibilidad de trabajo colaborativo. La creación de una cuenta es gratuita, aunque permite mejorarla previo pago para acceder a ciertas utilidades. Para la realización de los diagramas incluidos se ha utilizado el constructor de diagramas en línea de la página diagrams.net [51], ya que es gratuita, sin necesidad de registrarse e incluye varios paquetes con elementos y figuras del lenguaje unificado de modelado (UML por sus siglas en inglés) [52].

Bibliografía

[1] T. Issariyakul and E. Hossain, *Introduction to Network Simulator 2 (NS2).* Boston, MA: Springer US, 2009, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-0-387-71760-9_2

[2] E. Jiménez and J. L. López-Presa, "Efficient asynchronous gossip protocols for fault-prone sensor networks with memory and battery constraints," comunicación privada, 2021.

[3] M. Jelasity, *Gossip.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 139–162. [Online]. Available: https://doi.org/10.1007/978-3-642-17348-6_7

[4] A.-M. Kermarrec, L. Massoulie, and A. Ganesh, "Probabilistic reliable dissemination in large-scale systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 248–258, 2003.

[5] A. Montresor, *Gossip and Epidemic Protocols.* American Cancer Society, 2017, pp. 1–15. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/047134608X.W8353

[6] A.-M. Kermarrec and M. van Steen, "Gossiping in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, p. 2–7, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1317379.1317381

[7] S. R. S, T. Dragičević, P. Siano, and S. S. Prabaharan, "Future generation 5g wireless networks for smart grid: A comprehensive review," *Energies*, vol. 12, no. 11, 2019. [Online]. Available: https://www.mdpi.com/1996-1073/12/11/2140

[8] S. Madakam, V. Lake, V. Lake, V. Lake *et al.*, "Internet of things (iot): A literature review," *Journal of Computer and Communications*, vol. 3, no. 05, p. 164, 2015.

[9] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, 2009, pp. 99–100.

[10] F. Díez Muñoz, "Optimización de peersim: un simulador de eventos discretos para redes 5g," Master's thesis, Universidad Politécnica de Madrid, 2020.

[11] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2021. [Online]. Available: https://www.R-project.org/

[12] Y. Zhang, L. T. Yang, and J. Chen, *RFID and sensor networks: architectures, protocols, security, and integrations.* CRC Press, 2009.

[13] S. S. Iyengar, R. R. Brooks *et al.*, *Distributed sensor networks.* Chapman and Hall/CRC, 2004.

[14] M. Bhuptani and S. Moradpour, *RFID field guide: deploying radio frequency identification systems.* Prentice Hall PTR, 2005.

[15] J. X. Velasco-Hernandez, "Modelos matemáticos en epidemiología: enfoques y alcances," *Miscelánea Matemática*, vol. 44, pp. 11–27, 01 2007. [Online]. Available: https://www.researchgate.net/profile/Jorge-Velasco-Hernandez/publication/230765615_Modelos_matematicos_en_epidemiologia_enfoques_y_alcances/links/09e4150abf312de69a000000/Modelos-matematicos-en-epidemiologia-enfoques-y-alcances.pdf

[16] Y. Quintana Cruz, "Matemáticas y epidemiología: modelos y conclusiones," Trabajo Fin de Grado, Facultad de Ciencias de la Salud, Sección de Farmacia, Universidad de la Laguna, 2019. [Online]. Available: https://riull.ull.es/xmlui/handle/915/15998

[17] C. E. Álvarez Cabrera, E. J. Andrade Lotero, and V. Gauthier Umaña, "Modelos epidemiológicos en redes: una presentación introductoria," *Boletín de Matemáticas*, vol. 22, no. 1, pp. 21–37, 1 2015. [Online]. Available: https://revistas.unal.edu.co/index.php/bolma/article/view/51844

[18] F. Brauer, C. Castillo-Chavez, and Z. Feng, *Mathematical models in epidemiology.* Springer, 2019, vol. 32. [Online]. Available: https://doi.org/10.1007/978-1-4939-9828-9

[19] Z. Lavastida-López and Y. Almeida Cruz, "Propuesta de un modelo para el intercambio automático de información en redes p2p," in *VI Jornadas para el Desarrollo de Grandes Aplicaciones de Red (JDARE'09)*, Alicante, España, octubre 2009, pp. 333 – 351. [Online]. Available: https://www.dtic.ua.es/grupoM/recursos/articulos/JDARE-09-N.pdf

[20] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[21] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 1–12. [Online]. Available: https://doi.org/10.1145/41840.41841

[22] J. Leitão, J. Pereira, and L. Rodrigues, *Gossip-Based Broadcast.* Boston, MA: Springer US, 2010, pp. 831–860. [Online]. Available: https://doi.org/10.1007/978-0-387-09751-0_29

[23] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6. Berkeley, CA, USA, 2003, pp. 68–72.

[24] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, p. 35–40, Apr. 2010. [Online]. Available: https://doi.org/10.1145/1773912.1773922

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 205–220, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294281

[26] Y. hua Chu, S. Rao, S. Seshan, and H. Zhang, "A case for end system multicast,"

IEEE Journal on Selected Areas in Communications, vol. 20, no. 8, pp. 1456–1471, 2002.

[27] J. Pereira, U. do Minho, L. Rodrigues, U. de Lisboa, M. Monteiro, R. Oliveira, and A.-M. Kermarrec, "Neem: network-friendly epidemic multicast," in *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, 2003, pp. 15–24.

[28] E. Acea Zamora, "Simulación de eventos discretos," Ph.D. dissertation, Universidad Central"Marta Abreu"de Las Villas, 2010.

[29] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, 2008, pp. 1–10.

[30] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34. [Online]. Available: https://doi.org/10.1007/978-3-642-12331-3_2

[31] M. Montagud Aguar and F. Boronat Segui, "Aprendizaje mediante simulación de redes: Análisis, implantación y evaluación," *VAEP-RITA. Versión Abierta Español-Portugués*, vol. 1, no. 1, pp. 1–9, 2013.

[32] A. G. Dávalos, L. M. E. Paz, A. N. Cadavid, and A. V. Mejía, "Método de evaluación y selección de herramientas de simulación de redes," *Sistemas y Telemática*, vol. 9, no. 16, pp. 55–71, 2011.

[33] P. A. Roa, C. Morales, and P. Gutiérrez, "Norma iso/iec 25000," *Tecnología Investigación y Academia*, vol. 3, no. 2, p. 27–33, 12 2015. [Online]. Available: https://revistas.udistrital.edu.co/index.php/tia/article/view/8373

[34] I. Kazmi and S. F. Y. Bukhari, "Peersim: An efficient scalable testbed for heterogeneous cluster-based p2p network protocols," in *2011 UkSim 13th International Conference on Computer Modelling and Simulation*, 2011, pp. 420–425.

[35] "Java | Oracle," accessed: 2021-07-10. [Online]. Available: https://www.java.com/es/

[36] "PeerSim P2P Simulator," accessed: 2021-07-10. [Online]. Available: http://peersim.sourceforge.net/

[37] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2001, pp. 346–353.

[38] Y. Postigo and J. I. Pozo, "When a graph is worth more than a thousand data: Graph interpretation by adolescent students," *Journal for the Study of Education and Development*, vol. 23, no. 90, pp. 89–110, 2000. [Online]. Available: https://doi.org/10.1174/021037000760087982

[39] M. L. Almada and C. A. Talay, "Procesamiento gráfico de datos obtenidos en la simulación de redes. análisis comparativo de herramientas como complemento del simulador ns-2," *Informes Científicos Técnicos-UNPA*, vol. 11, no. 3, pp. 1–14, 2019.

[40] B. Kernighan and D. Ritchie, *El lenguaje de programación C*. Pearson Educación, 1991. [Online]. Available: https://books.google.es/books?id=OpJ_0zpF7jIC

[41] ISO Central Secretary, "Information technology — Programming languages — Fortran — Part 1: Base language — Technical Corrigendum 1," International Organiza-

tion for Standardization, Geneva, CH, Standard, 06 2021.

[42] P. Elosua, *Introducción al entorno R.* Servicio Editorial de la Universidad del País Vasco, 01 2011. [Online]. Available: https://web-argitalpena.adm.ehu.es/pdf/UWLGPS4979.pdf

[43] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *Journal of Computational and Graphical Statistics*, vol. 5, no. 3, pp. 299–314, 1996. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/1618600.1996.10474713

[44] R. A. Becker, J. M. Chambers, and A. R. Wilks, *The New S Language: A Programming Environment for Data Analysis and Graphics.* USA: Wadsworth and Brooks/Cole Advanced Books & Software, 1988.

[45] R. Stallman, "The GNU manifesto," *Dr. Dobb's Journal of Software Tools*, vol. 10, no. 3, pp. 30–??, Mar. 1985. [Online]. Available: https://www.gnu.org/gnu/manifesto.html

[46] "Enabling Open Innovation & Collaboration | The Eclipse Foundation," accessed: 2021-07-10. [Online]. Available: https://www.eclipse.org/

[47] RStudio Team, *RStudio: Integrated Development Environment for R*, RStudio, PBC, Boston, MA, 2021. [Online]. Available: http://www.rstudio.com/

[48] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York, 2016. [Online]. Available: https://ggplot2.tidyverse.org

[49] "LaTeX - A document preparation system," accessed: 2021-07-10. [Online]. Available: https://www.latex-project.org/

[50] "OverLeaf, Online LaTeX Editor," accessed: 2021-07-10. [Online]. Available: https://www.overleaf.com/

[51] "Diagram Software and Flowchart Maker," accessed: 2021-07-10. [Online]. Available: https://www.diagrams.net/

[52] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert, "Unified modeling language (UML) version 2.5.1," Object Management Group (OMG), Standard, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1