Tesis de Licenciatura

Análisis e implementación de Vivaldi como mecanismo para reducir el tráfico entre ISPs en BitTorrent

Maximiliano Geier
LU: 477/04
mgeier@dc.uba.ar

Directores:
Dr. Claudio E. Righetti
Dr. Esteban Mocskos

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Julio de 2011

Resumen

El software P2P representa un alto porcentaje del tráfico total de Internet. Los protocolos utilizados están diseñados sin tener en cuenta la proximidad de los participantes en la toma de decisiones de los algoritmos involucrados. Por este motivo, los sistemas resultantes pueden no ser muy eficientes en cuanto a la duplicación del tráfico, y esto puede no hacer viable el modelo de negocios actual de los ISPs.

Para mitigar este problema, diversos mecanismos han sido estudiados por la comunidad de investigadores y operadores de red. En particular, en nuestro trabajo utilizamos el sistema de coordenadas sintético Vivaldi para estimar latencias entre los nodos participantes, y de esta manera poder dirigir el tráfico de modo que su distribución sea más eficiente.

Construimos una implementación del algoritmo de Vivaldi y algunas mejoras mencionadas en la bibliografía. Además, modificamos un cliente de BitTorrent para que utilice Vivaldi para dirigir el tráfico. Luego, proponemos una metodología para analizar la viabilidad de nuestra solución. Nuestra propuesta utiliza la plataforma de prueba Emulab para realizar pruebas sobre redes con parámetros fijos de conectividad. Utilizamos 3 tamaños de redes diferentes, de 10, 33 y 64 nodos, construidas de forma tal que puedan identificarse grupos de nodos separados por su latencia entre sí, y capturamos los valores de coordenadas de Vivaldi a lo largo de 4000 segundos para distintos valores de los parámetros c_c y c_e, y con y sin la mejora *neighbor decay*. Con estos datos comparamos dos clusterizaciones diferentes, una construida con k-means sobre los valores de las coordenadas, y otra utilizando mediciones explícitas de latencia. Analizamos la influencia de los parámetros en el comportamiento de los resultados a lo largo del tiempo, observando dos características de la clusterización por k-means: la evolución del radio de los clusters y el porcentaje de error con respecto a la otra clusterización. Observamos que los valores más pequeños estudiados en la bibliografía no ofrecen una velocidad de convergencia razonable para montar Vivaldi sobre tráfico P2P, pero que utilizar parámetros menos conservadores sí convierte al algoritmo en un método viable para modelar latencias entre los nodos participantes. Por otro lado, el uso de neighbor decay no mostró aportar beneficio alguno.

En último lugar, realizamos un análisis preliminar de los resultados de montar Vivaldi sobre el tráfico de BitTorrent, utilizando la misma metodología que en el caso anterior. Analizamos un patrón fijo de vida de un torrent sobre la misma red pequeña de 10 nodos. Mostramos una degradación importante en la calidad de los resultados con respecto al caso sin BitTorrent, que atribuimos a una modificación en el patrón de tráfico inherente al protocolo. Comparamos estos resultados con una clusterización obtenida con valores aleatorios de coordenadas, donde mostramos una mejora del 12 % con respecto a no utilizar Vivaldi. Además, evaluamos el uso de neighbor decay, que no logra modificar sustancialmente esta situación. Finalmente, concluimos con un marco de trabajo a futuro que nos permita analizar posibles soluciones a los problemas encontrados.

Analysis and implementation of Vivaldi as a mechanism to reduce traffic among peers in different ISPs using BitTorrent

Maximiliano Geier

July 15, 2011

Abstract

P2P software represents a high percentage of the total traffic on the Internet. Protocols are designed without taking into account the proximity of peers in the decision-making process. This is why resulting systems may not be efficient in terms of traffic duplication, which could harm the current business model of ISPs.

In order to mitigate this problem, several mechanisms have been studied by the research community and network operators. In particular, our work makes use of the synthetic network coordinate system Vivaldi to estimate latencies among participating nodes, directing traffic in such a way that its distribution is more efficient.

We have built an implementation of the Vivaldi algorithm and some improvements found in the bibliography. Moreover, we have modified a BitTorrent client to make it use Vivaldi to direct its traffic. We have then proposed a methodology to analyse the viability of our solution. Our proposal makes use of the Emulab testbed in order to make tests on networks with fixed conectivity parameters. We have made our tests on three networks of different sizes, 10, 33 and 64 nodes, which were built in such a way that it is possible to tell groups of nodes apart by their latencies, and we have captured the Vivaldi coordinate values during 4000 seconds for different values of the parameters c_c and c_e, with and without the *neighbor decay* improvement. Using this data, we have compared two different clusterizations, one built using k-means on the coordinate values, and the other built using explicit latency measurements. We have analysed the influence of the parameters on the behavior of the results over time, observing two characteristics of the k-means clusterization: the evolution of the cluster radii and the error percentage with respect to the other clusterization. We have observed that the smallest values which were studied in the bibliography do not converge reasonably fast in order to mount Vivaldi on P2P traffic, but using less conservative values does make the algorithm a viable method to model latencies among participating nodes. On the other hand, using neighbor decay did not seem to show any benefits.

Finally, we have made a preliminary analysis of the results of mounting Vivaldi on BitTorrent traffic, using the same methodology we have previously described. We have analysed a fixed BitTorrent life pattern on the same 10-node network. We show that the quality of the results suffers from an important degradation with respect to using Vivaldi alone, which we attributed to the fact that the traffic pattern had been modified. We have compared these results with a clusterization built using random coordinate values, and we show that Vivaldi improves 12% with respect to the former. Moreover, we have evaluated the use of neighbor decay, which could not sustantially improve this situation. Finally, we conclude with a future framework which enables us to analyse possible solutions to the problems we have found.

Índice general

Índice de figuras

Índice de algoritmos

1 Introducción

Los sistemas Peer-to-Peer (P2P) comenzaron a tomar popularidad como forma de distribución de contenido a partir de la creación del conocido software de descarga Napster [3]. Este software tuvo su época de esplendor a fines de la década de 1990, pero la idea perduró y sobrevivió en los años siguientes en cada uno de los sistemas originados a partir de él. La fuerte explosión de estos métodos de distribución de archivos significó un cambio de paradigma con respecto a cómo se veía el uso de Internet, y además un golpe importante al modelo de negocios que utilizaban hasta ese momento (y utilizan hoy en día en menor medida) los distribuidores de entretenimiento digital, ya que se comenzó a reemplazar al medio físico del CD de música o el DVD para las películas por una versión completamente digital. Este hecho influyó en el modelo de negocios de las distribuidoras ya que las copias dejaron de tener un costo de generación asociado.

Clase de protocolo	África (Sur)	América del Sur	Europa oriental	África (Norte)	Alemania	Europa (Sur)	Oriente medio	Europa (Suroeste)
P2P	*65,77%*	*65,21%*	*69,95%*	*42,51%*	*52,79%*	*55,12%*	*44,77%*	*54,46%*
Web	20,93%	18,17%	16,23%	32,65%	25,78%	25,11%	34,49%	23,29%
Streaming	5,83%	7,81%	7,34%	8,72%	7,17%	9,55%	4,64%	10,14%
VoIP	1,21%	0,84%	0,03%	1,12%	0,86%	0,67%	0,79%	1,67%
IM	0,04%	0,06%	0,00%	0,02%	0,16%	0,03%	0,50%	0,08%
Túneles	0,16%	0,10%	–	–	–	0,09%	2,74%	–
Estándar	1,31%	0,49%	–	0,89%	4,89%	0,52%	1,83%	1,23%
Juegos	–	0,04%	–	–	0,52%	0,05%	0,15%	–
Desconocido	4,76%	7,29%	6,45%	14,09%	7,84%	8,86%	10,09%	9,13%

Figura 1.1: Distribución del uso de Internet por clase de protocolo en diferentes zonas geográficas, 2008-2009 [1]

Actualmente los sistemas P2P representan una parte importante del tráfico total de Internet [1]. Podemos observar en la tabla de la figura 1.1 la distribución de distintas clases de tráfico según la zona geográfica. En el mismo apreciamos que el primer lugar lo tiene P2P en cada una de las zonas analizadas en el estudio. Esto ocurre incluso a pesar de que en los últimos años hay un decrecimiento del volumen de tráfico P2P, que se debe principalmente a nuevas tendencias en la distribución de contenido, tales como los sitios de descarga directa (Megaupload, Rapidshare, Hotfile, etc.) y los sitios dedicados al streaming de videos (el caso más destacado es YouTube). Esta modificación en los hábitos de los usuarios se debe principalmente al hecho de que los sistemas P2P suelen ser más complicados de utilizar, ya que requieren software especializado que debe ser instalado para su utilización, y además acarrean

el problema de ser una fuente potencial de spyware y malware. Un claro ejemplo de este problema se ve en la existencia de diferentes versiones con adware que circulaban por Internet de KaZaA, el cliente de un popular P2P del mismo nombre. Aún así, el software P2P también mejoró en cuanto a usabilidad en los últimos años (por ejemplo, el cliente oficial de BitTorrent, μTorrent, no requiere instalación y se configura automáticamente para aceptar conexiones entrantes aunque su dirección IP se encuentre enmascarada[1]), el volumen de tráfico generado por las nuevas formas alternativas de distribución de contenido aún no está cerca de sobrepasar el del software P2P, por lo que sigue siendo prioritaria la necesidad de manejar el volumen de tráfico generado por estos últimos por parte de los proveedores de Internet. Entre la gran cantidad de sistemas P2P existentes, el más utilizado a nivel mundial es BitTorrent [4,5]. Es en este sistema en el cual basaremos nuestro modelo de referencia.

Un punto importante a atacar para reducir el tráfico de los P2Ps es el hecho de que los patrones de conexión obedecen a criterios que nada tienen que ver con la estructura de la red subyacente. Esto puede ser un problema, y para ejemplificarlo describiremos un escenario posible. Supongamos que dos participantes desean obtener un recurso de una red P2P y se encuentran topológicamente cerca el uno del otro (por ejemplo, en el mismo proveedor de Internet) pero los poseedores de ese recurso no. Un criterio eficiente desde el punto de vista del tráfico generado podría consistir en que ambos clientes que desean el recurso descarguen subconjuntos disjuntos de las partes del mismo y luego compartan las que tiene uno y otro entre sí, ya que de esta manera el tráfico duplicado recorre una menor cantidad de links. Por otro lado, también podrían pedir todas las partes a los proveedores lejanos. Si no se tiene en cuenta la topología de la red en la toma de decisiones puede ocurrir este último caso. Este tráfico duplicado es costoso en el caso de un proveedor residencial, ya que involucra atravesar infrastructura no propia, donde se cobra por tasa transferida [6]. Además, el tráfico dentro del mismo proveedor podría ser tratado de manera más eficiente (por ejemplo, con una tasa de subida preferencial), y de esta manera se beneficiaría también el usuario final del servicio.

Tener en cuenta la topología en un software de red que funciona en Internet no es una tarea sencilla. La información relevante no está disponible ya que no es necesaria para el funcionamiento de la red. Además, la topología es siempre cambiante, y no existe ningún mecanismo para descubrirla automáticamente. Una primera aproximación a este problema podría consistir en utilizar la información provista por el servicio de nombres DNS. Esto tiene varios problemas, ya que esa información puede ser imprecisa (los nombres de dominio en una organización no tienen por qué corresponderse con la estructura de sus redes) e incompleta (no todas las direcciones IP tienen un nombre reverso asociado). Existen diversas tecnologías para agregar datos de geolocalización a las entradas de DNS, pero tienen un costo de mantenimiento asociado a una componente humana, ya que es necesario configurar manualmente esos campos. Además, la geolocalización no nos dice nada acerca de la topología de la red, ya que entran en juego otros parámetros además de la distancia geográfica, como ser el tendido de cables o el uso de tecnología inalámbrica, la calidad de los links o la velocidad de los routers intermedios. También se puede utilizar Border Gateway Protocol (BGP), que provee información accesible por medio de consultas a través de DNS o WHOIS sobre los sistemas autónomos (AS) que conforman Internet. Todos estos mecanismos requieren hacer consultas explícitas a bases de

[1] La escasez de direcciones IP en IPv4 hace que ningún cliente residencial tenga más de una dirección, en un mundo en el que casi cualquier persona conectada a Internet tiene más de un dispositivo que se lo permite. Esto se "solucionó" con la instalación de routers residenciales que cumplen la función de enmascarar las direcciones internas de los múltiples dispositivos que puede haber en un hogar u oficina utilizando una única dirección IP ruteable.

datos localizadas por fuera del backbone del proveedor, que pueden ser costosas. Este problema motiva la creación de otras herramientas que descubran automáticamente la topología y que puedan estimar los datos necesarios sin necesidad de realizar consultas explícitas. Dentro de este grupo se encuentran los sistemas de coordenadas sintéticos como Vivaldi [7], que permite estimar en forma distribuida la latencia entre cualquier par de nodos entre todos los que ejecuten el algoritmo.

Como punto de partida expondremos el objetivo de nuestro trabajo y luego haremos una reseña del trabajo relacionado encontrado en la bibliografía para atacar nuestra problemática, de los sistemas P2P en general, sobre Vivaldi, el sistema de coordenadas sintético que utilizamos en nuestro trabajo, y finalmente sobre BitTorrent, el sistema P2P con el que trabajamos. El capítulo 2 contiene una reseña de las implementaciones realizadas a lo largo del trabajo y la descripción de la metodología utilizada. El capítulo 3 trata sobre el análisis de los resultados y el trabajo futuro, y finalmente en el capítulo 4 presentamos las conclusiones.

Adicionalmente, al final del trabajo se encuentran varios apéndices con documentación acerca de los aspectos más técnicos de la tesis: en el apéndice A podemos encontrar una referencia acerca del uso del software implementado, el apéndice B información adicional acerca de los formatos de representación utilizados por las distintas componentes de software presentes en nuestro trabajo, en el apéndice C encontramos la documentación de las clases implementadas en nuestra solución y finalmente en el apéndice D mostramos las descripciones formales de las redes utilizadas en los experimentos.

1.1 Objetivo

Nuestro objetivo es obtener una implementación de la versión distribuida del algoritmo Vivaldi presentado por Dabek et al. en [7], que nos permita obtener una estimación de la latencia entre pares de nodos de una red. La latencia nos ofrece una medida de distancia que podemos utilizar para estimar si dos clientes pertenecen a un mismo proveedor o no. Esta implementación luego la utilizamos para modificar una versión real de BitTorrent, a la que le incorporamos el cálculo de coordenadas de Vivaldi, que finalmente son utilizadas para dirigir el tráfico generado por el sistema P2P con el fin de reducir el ancho de banda transferido entre clientes de distintos proveedores. Analizamos el funcionamiento de nuestra implementación de Vivaldi en redes controladas de varios tamaños sobre la plataforma de prueba Emulab, hacemos pruebas preliminares de la implementación sobre BitTorrent, analizamos los problemas encontrados y sugerimos posibles caminos a seguir.

1.2 Trabajo relacionado

El uso de sistemas como Vivaldi para incorporar información de vecindad a aplicaciones de red no es una idea nueva. Existe una extensa bibliografía que hace énfasis en distintos aspectos de este problema sumamente complejo.

Por un lado, en cuanto a Vivaldi, no podemos dejar de mencionar el propio paper donde fue publicado el algoritmo, de Dabek *et al.* [7]. Además, el trabajo de Ledlie *et al.* [8], donde se presentan mejoras que tienen como objetivo mejorar su performance en redes reales, y Wang y Ng [9], donde se plantea un mecanismo para mejorar la estabilidad de las coordenadas. También existen análisis con respecto al sistema de coordenadas utilizado por el algoritmo,

como el de Lucángeli Obes [10] o el de Lumezanu y Spring [11]. Un excelente análisis de la performance del algoritmo de Vivaldi sobre simulaciones fue realizado por Moravek *et al.* [12].

Además de Vivaldi, existen múltiples otras formas de generar esta información de vecindad. Algunas de ellas también son sistemas de coordenadas, o también pueden ser estructuras que embeben la estructura de red subyacente, a saber: RON [13], IDMaps [14], GNP [15], y los basados en simulaciones, como Big-bang o el propio Vivaldi y sus adaptaciones Vivaldi con dos niveles [16], Vivaldi jerárquico [17] y Pharos [18]. Notamos a continuación las características principales de cada uno:

- RON: sistema utilizado para descubrir caminos alternativos entre WANs, utilizando la redundancia de links encontrada en Internet. Recolecta RTTs para construir los caminos.
- IDMaps: mapa de latencias de los nodos en Internet, localiza los nodos en un mapa según algunas heurísticas.
- GNP: sistema similar a IDMaps, ofrece heurísticas más eficientes.
- Vivaldi: algoritmo distribuido que calcula para cada nodo posiciones en un sistema de coordenadas dado que embeben la topología de la red donde se ejecuta simulando la evolución de un sistema físico de mallas y resortes, utiliza como medida de distancia la latencia entre nodos, intentando minimizar el error intrínseco por la existencia de violaciones de la desigualdad triangular propia de la forma en la que se conectan las redes (por ejemplo, por la existencia de múltiples caminos con diferentes valores de latencia).
    - Vivaldi con dos niveles: modificación de Vivaldi que clusteriza los nodos de la red y ejecuta Vivaldi normal dentro de cada cluster; los nodos del nivel superior mantienen su propio sistema de coordenadas que utilizan para estimar distancias entre nodos que corresponden a clusters diferentes.
    - Vivaldi jerárquico: adaptación de Vivaldi que utiliza un "sistema de coordenadas" jerárquico para estructurar la red, en lugar de utilizar algún sistema métrico.
    - Pharos: otra modificación de Vivaldi que utiliza un sistema similar al Vivaldi con dos niveles.
- Big-bang: sistema similar a Vivaldi pero que simula la evolución de un campo vectorial.

En cuanto a las posturas con respecto a cómo manejar el volumen de tráfico generado por el software P2P también hay diversos enfoques en la bibliografía. Uno de ellos es el adoptado en el trabajo de Xie et al. sobre P4P [19], que consiste en proponer acuerdo de cooperación entre ISPs y usuarios de software P2P. La idea detrás de P4P es exportar una API a los clientes P2P que les permita realizar consultas acerca de la topología de las redes donde se ejecutan. Esto les permite incorporar estos datos en la toma decisiones de los algoritmos utilizados, y de esta manera funcionar de una manera más "amigable" con los ISPs. Otro ejemplo de la misma postura es LANC, de Zhang et al. [20], que desarrolla una codificación de direcciones de red que incluye información de vecindad. Un acercamiento diferente consiste en utilizar un modelo estructural para predecir las latencias entre los distintos clientes. Este modelo se configura midiendo latencias explícitamente entre puntos fijos conocidos y luego utilizando

BGP[2] para predecir caminos sobre los nodos. El funcionamiento se explica en detalle en el trabajo de Madhyastha et al. [21].

La postura que tomamos en esta tesis consiste en modificar el funcionamiento de los clientes P2P para que utilicen información de vecindad recolectada de alguna manera para la toma decisiones. Esto no requiere ningún tipo de interacción con los proveedores, por lo que es sumamente más sencillo de implementar que un sistema como P4P, pero tiene la desventaja de que no se tiene a disposición información detallada sobre la estructura de red subyacente y por este motivo puede ser más complicado obtener resultados precisos.

Particularmente en el sistema P2P que utilizamos, BitTorrent, la toma de decisiones en función de la información de vecindad la podemos realizar a nivel del tracker, o bien a nivel del cliente. Modificar el tracker tiene como ventaja que es completamente transparente al cliente. Cualquiera de las implementaciones existentes puede ser utilizada sin cambios adicionales en un entorno con un tracker modificado, mientras se respete el protocolo. Por otro lado, el tracker sólo utiliza la información de vecindad recolectada cuando se genera tráfico hacia él, por lo que es menos precisa. Modificar el cliente tiene la desventaja de que el deployment es más costoso (requiere un nuevo cliente, una extensión en el protocolo para poder transmitir la información de vecindad, y modificar su funcionamiento para que se utilice para dirigir el tráfico), pero los datos se actualizan sin costo adicional por el propio tráfico que generan los clientes.

También podemos incluir en esta postura al trabajo de Choffnes y Bustamante sobre Ono [22], que consiste en utilizar la información provista por redes de distribución de contenido[3] (CDNs) para consultar la ubicación aproximada de un cliente determinado, en lugar de ubicarlos en un sistema de coordenadas. En este caso, se considera que dos clientes se encuentran cerca entre sí si ambos "ven" la misma copia local del servidor de la CDN.

1.3 Las redes P2P

1.3.1. Definición

P2P es una arquitectura donde cada nodo participante o *peer* puede desempeñar una o más funciones (posiblemente variables) en la red. A diferencia de la arquitectura Cliente-Servidor, donde encontramos dos roles claramente marcados: el *servidor*, quien ofrece un servicio, y el *cliente*, quien lo pide, no hay roles necesariamente definidos en P2P. Cualquier participante puede ser iniciador o receptor de pedidos. Esto permite implementar servicios con capacidades de adaptación impensables en el modelo Cliente-Servidor, con características como resistencia a fallos, adaptación automática a cambios en la topología, etc.

Entre las arquitecturas P2P podemos describir dos tipos:

- *Puras*: son aquellas que se ajustan perfectamente a la definición teórica, es decir, donde no hay roles definidos entre los participantes. Estas arquitecturas son las más versátiles, y es donde se pueden apreciar todas las características más interesantes que mencionamos.
- *Híbridas*: son aquellas que requieren de algún componente distiguido en la red con un rol especial. Por ejemplo, en la arquitectura del sistema P2P del que hablaremos en este trabajo, BitTorrent, existe un nodo con un rol particular al que se lo denomina

[2] Ver RFC 4271.
[3] En particular, utilizan Akamai [23].

tracker[4], cuya existencia es necesaria para que pueda dar comienzo la descarga, ya que provee información acerca de dónde buscar los datos. La desventaja de este modelo es que este nodo distinguido funciona como un punto único de falla en la red, pero por otro lado tiene la ventaja de que es más sencillo el mecanismo que da inicio a la descarga y el propio protocolo del sistema.

Por otro lado, según la forma en la que se encuentran conectados los *peers*, podemos clasificar los sistemas P2P entre estructurados y no estructurados [24].

Los no estructurados son aquellos donde los nodos no se encuentran conectados entre sí de ninguna manera en particular. En general, si se analiza la topología resultante, tiene la forma de una clique (un subconjunto, a veces aleatorio, de nodos completamente conectados entre sí). La mayoría de los P2Ps utilizados para la distribución de archivos entra en esta categoría. Algunos ejemplos son: Gnutella, eDonkey/eMule, Limewire, KaZaA, BitTorrent, entre otros. Una ventaja que tienen respecto de los estructurados es que el protocolo es más sencillo en cuanto al diseño y la implementación. También es más simple el mecanismo de búsqueda de contenido, pero en redes muy grandes pueden llegar a presentar problemas de escalabilidad.

Por otro lado, los estructurados, como su nombre lo indica, forman una topología de red entre los peers participantes. En esta categoría encontramos a la mayoría de los P2P académicos como Chord, CAN, Tapestry y Pastry, que utilizan un algoritmo de ruteo basado en Distributed Hash Table (DHT) para intercambiar mensajes eficientemente (realizando un máximo de $O(log(n))$ consultas antes de alcanzar al destinatario de un mensaje) entre los distintos peers. Como ejemplos no académicos de sistemas P2P estructurados tenemos extensiones para BitTorrent, LimeWire e eMule, que utilizan distintas implementaciones de Kademlia[5] para rutear cierto tráfico entre peers.

1.3.2. BitTorrent

BitTorrent fue concebido como un mecanismo de distribución de archivos. Si utilizamos HTTP para este mismo fin, y deseamos distribuir una copia de un archivo cuya transferencia demanda m bytes a n clientes diferentes, la cantidad total de bytes que debe enviar el emisor para completar todas las transferencias será $n \times m$ bytes, es decir, crece linealmente con la cantidad de copias transferidas. En cambio, utilizando BitTorrent, todos los clientes participan de la transferencia, distribuyendo partes (*pieces*) del archivo que ya tienen a otros clientes, e intercambiándolas por otras que no tienen, y de esta manera maximizan la utilización del ancho de banda disponible la mayor parte del tiempo. Esto se puede observar en la figura 1.2, donde se muestra una estadística realizada utilizando el software **SystemImager** [2]; el mismo cuenta con un módulo que utiliza BitTorrent para distribuir una imagen[6] de la instalación de una computadora en n máquinas simultáneamente, alternativamente a la utilización de multicast. Su principal ventaja con respecto a multicast es que es más versátil, ya que es más sencilla su implementación en entornos de red heterogéneos (por ejemplo, donde los clientes pertenecen a varias redes físicas diferentes). En presencia de más de un peer que posea el recurso de interés es además más eficiente, ya que permite distribuir uniformemente la carga entre cada poseedor (en multicast sería necesario crear varios grupos separados y distribuir a los clientes de antemano).

[4] Ver sección 1.5.4.

[5] DHT diseñada por Petar Maymounkov y David Mazières, cuenta con varias implementaciones libres.

[6] Copia fiel de la instalación de una máquina.

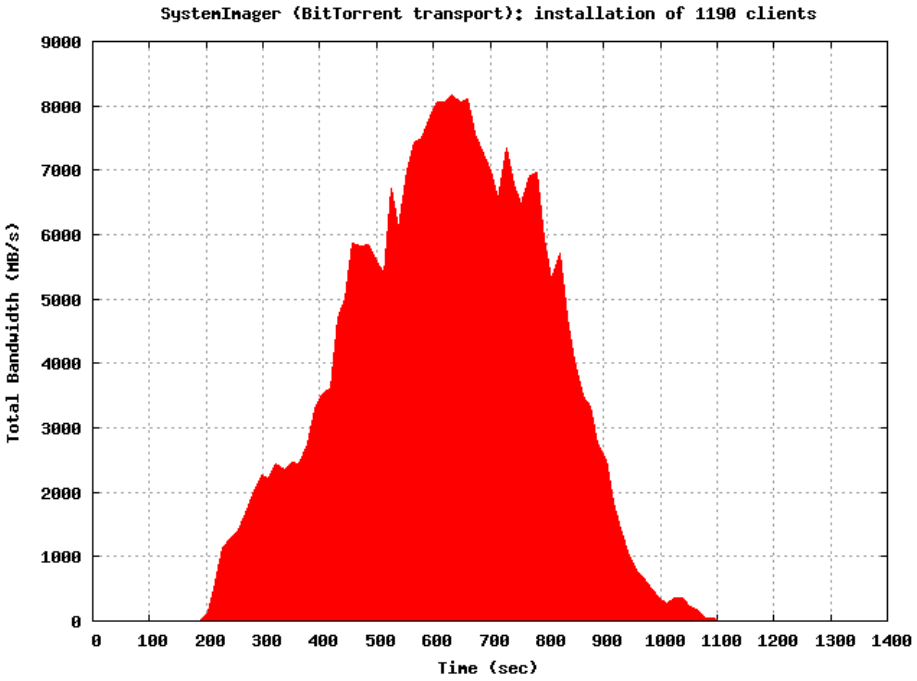


Figura 1.2: Ancho de banda medido en la instalación de los sistemas operativos de 1190 computadoras utilizando el módulo de BitTorrent del software SystemImager (gráfico tomado de [2])

Existe más software derivado que utiliza la tecnología de BitTorrent para distribuir contenidos, como por ejemplo **btwindeploy** (instalación desatendida de Windows en múltiples máquinas utilizando BitTorrent), **Vuze** (plataforma que utiliza el cliente Azureus para descargar contenidos multimedia) o **Bitlet**, que es un cliente de BitTorrent en sí mismo pero que permite hacer *streaming* de videos.

Volvemos con más detalle sobre BitTorrent en la sección 1.5.

1.4 Vivaldi

1.4.1. Motivación

Los sistemas de coordenadas sintéticos se pueden utilizar para predecir el Round-Trip Time (RTT)[7] entre dos hosts sin necesidad de realizar mediciones explícitas. Cuando se considera un sistema de coordenadas, es deseable tener en cuenta las características enumeradas a continuación.

- **Elección del espacio métrico**: el espacio elegido debe representar adecuadamente la estructura de la red subyacente. Por ejemplo, si la intención es representar las posiciones físicas de hosts en el planeta Tierra, sería razonable pensar en un sistema de coordenadas esférico, pero esto no tiene mucho sentido si lo que se desea medir es la latencia entre dos hosts, ya que hay que tomar en cuenta el camino recorrido por los paquetes que viajan de un host a otro, que no necesariamente guarda relación alguna con las posiciones físicas de los nodos.f

- **Escalabilidad**: debe ser mantenible para el caso en el que hay muchos hosts en la red, ya que de modo contrario no es utilizable en gran escala.

- **Descentralización**: los sistemas P2P son inherentemente descentralizados, por lo que no se puede aprovechar ningún sistema de coordenadas en el contexto de una red de estas características si no se puede ejecutar una versión distribuída del algoritmo que calcula las coordenadas.

- **Cantidad de tráfico de prueba**: si el volumen de tráfico necesario para cargar con la información de las coordenadas es elevado, es posible que agregar esta característica degrade de alguna otra forma el tráfico de la red, por lo que pierden utilidad rápidamente.

- **Rápida adaptación a cambios en la red**: la arquitectura de las redes en Internet es siempre cambiante, por lo que es importante que cualquier sistema de coordenadas que se utilice pueda corregirse, de modo que los valores obtenidos sean válidos.

El sistema de coordenadas elegido en este trabajo, Vivaldi, fue pensado puntualmente para cumplir con estas características [7].

[7] Tiempo que observa el emisor de un paquete entre que lo envía y obtiene una notificación de recepción de parte del receptor.

1.4.2. Algoritmo

Interpretación teórica

El objetivo del algoritmo de Vivaldi es asignarle a cada nodo de la red una coordenada dentro del sistema métrico utilizado de modo que la norma entre nodos represente de la manera más precisa posible la latencia entre los mismos.

Por la forma en la que se conectan las redes en Internet, no es posible especificar un sistema de coordenadas que represente con exactitud el RTT entre hosts [8], ya que existen casos, por ejemplo si hay redundancia de links, donde las latencias no preservan la propiedad de desigualdad triangular[8]. Lo que se busca en su lugar es que las coordenadas obtenidas minimicen el error total. Sean i y j dos nodos de la red, L_{ij} el RTT entre ambos y x_i y x_j sus respectivas coordenadas, podemos escribir la siguiente expresión para el error cuadrático:

$$E = \sum_i \sum_j (L_{ij} - \|x_i - x_j\|)^2$$

El algoritmo de Vivaldi busca una solución a este problema de minimización. El motivo por el que se eligió el error cuadrático frente a otras medidas de error no es casual: existe un algoritmo iterativo conocido que minimiza la misma cantidad y que es fácilmente paralelizable. Este algoritmo es el que se utiliza en la resolución del problema del sistema físico de la malla elástica. Al encontrar una solución a este problema se obtiene una solución de la minimización original.

Debemos expresar, entonces, nuestro problema en función de las variables del problema análogo. Las masas representarían a los hosts, mientras que los resortes medirían, en reposo, la latencia existente entre los hosts que unen. Además, suponemos que cada resorte tiene la misma constante de elasticidad $k = 1$. La longitud de cada uno en reposo, L_{ij}, no necesariamente es la misma que tendrá el resorte con todo el sistema en reposo. Esta última está dada por $\|x_i - x_j\|$, por lo que, si tomamos F_{ij} como el vector que indica la fuerza entre los nodos i y j, según la ley de Hooke [7] tenemos que:

$$F_{ij} = (L_{ij} - \|x_i - x_j\|)u(x_i - x_j)$$

En la ecuación anterior, F_{ij} indica la fuerza que ejerce el resorte sobre i y j, mientras que $L_{ij} - \|x_i - x_j\|$ indica el desplazamiento del resorte ij del reposo. La fuerza ejercida es proporcional a este desplazamiento, cuya dirección está dada por el vector unitario[9] $u(x_i - x_j)$. Luego, la fuerza neta en i está dada por la siguiente sumatoria:

$$F_i = \sum_{j \neq i} F_{ij}$$

Finalmente debemos simular la evolución temporal del sistema; esto se realiza utilizando la siguiente relación entre las coordenadas x_i y el tiempo t:

$$x_i = x_i + tF_i$$

El valor de t es un punto importante del algoritmo de Vivaldi, sobre el cual entraremos en detalle luego.

[8] $\|x_i + x_j\| \leq \|x_i\| + \|x_j\|$. Estos casos problemáticos son llamados Triangle Inequality Violations (TIVs) en la bibliografía.

[9] Vector u tal que $\|u\| = 1$.

En las siguientes secciones derivaremos la implementación utilizada en nuestro trabajo a partir de las ecuaciones anteriores.

Algoritmo centralizado

El algoritmo centralizado no ofrece ninguna utilidad real más que ilustrar el funcionamiento de la versión distribuida. Sin embargo, es un paso necesario para nuestro análisis ya que es más sencillo de analizar que la versión final [7]. Mostramos el pseudocódigo en el algoritmo 1.

Algoritmo 1 Vivaldi centralizado

```
{Input: L (matriz de latencias) y x (vector con las coordenadas actuales)}
{Output: x con coordenadas más precisas}
{ϵ: tolerancia de error máxima}
while error(L, x) > ϵ do
  for all i in L do
    F = 0
    for all j in L do
      {Computar la fuerza del resorte}
      e = L_ij − ‖x_i − x_j‖
      {Sumar la fuerza del vector al total}
      F = F + e × u(x_i − x_j)
    end for
    {Mover en la dirección de la fuerza}
    x_i = x_i + t × F
  end for
end while
```

Algoritmo distribuido

En la versión centralizada mostramos el cálculo de las coordenadas de cada nodo iterativamente. La idea detrás de la versión distribuida es que cada nodo computa su propia coordenada y simula su propio movimiento en el sistema contínuamente. Cada nodo obtiene la coordenada y el RTT de los nodos contra los que se conecta, y utiliza esa información para actualizar la suya.

Sean x_i la coordenada del nodo actual, x_j la del nodo contra la que se está actualizando y rtt la latencia medida entre i y j, podemos derivar del algoritmo centralizado la siguiente ecuación para actualizarla:

$$x_i = x_i + t \times (rtt - \|x_i - x_j\|) \times u(x_i - x_j)$$

Mencionamos en la sección 1.4.2 que es importante para el correcto funcionamiento del algoritmo elegir un valor adecuado para el parámetro temporal t. Como primera aproximación, elegiremos un valor constante δ para mostrar el algoritmo distribuido, pero analizaremos a continuación los problemas que presenta esta elección y veremos cómo los podemos solucionar. Podemos ver el pseudocódigo en el algoritmo 2.

El paso temporal indica cuánto se debe mover la coordenada en la dirección calculada por la iteración actual. Su magnitud condiciona la velocidad de convergencia del algoritmo: con un

Algoritmo 2 Vivaldi distribuido (timestep constante δ)

{Input: rtt (latencia medida entre el nodo actual i y el nodo j)}
{Input: x_i y x_j (coordenadas actuales de los nodos i y j)}
{Output: coordenada actual x_i más precisa}
$e = rtt - \|x_i - x_j\|$
$x_i = x_i + \delta \times e \times u(x_i - x_j)$

paso muy pequeño puede tardar mucho en converger, mientras que con un paso muy grande podría oscilar. Es evidente que necesitamos variar este valor para facilitar la convergencia.

Adicionalmente, tenemos el problema de que si la coordenada remota contra la que estamos actualizando tiene un error muy elevado, puede influir negativamente sobre la calidad de la estimación.

1.4.3. Paso temporal adaptativo

En una segunda aproximación podemos tomar un valor de δ que se acerque rápidamente cuando el error relativo es elevado, y que cuando "falte" menos para alcanzar la convergencia reduzca la velocidad con la que se acerca.

Una versión adaptativa sencilla podría consistir en utilizar una fracción constante $c_c < 1$ del error relativo local calculado e_l:

$$\delta = c_c \times e_l$$

El problema de esta elección es que no toma en cuenta el error de la coordenada remota. Esto es problemático ya que puede hacer que se mueva mucho hacia una dirección poco confiable.

Teniendo los errores relativos local y remoto podemos derivar una ecuación para δ que tenga en cuenta más la predicción remota cuanto más precisa sea de la siguiente forma:

$$\delta = c_c \times \frac{e_l}{e_l + e_r}$$

donde e_r es el error de la coordenada remota.

Finalmente, reescribimos el algoritmo de Vivaldi distribuido para tomar en cuenta la nueva definición de δ en el algoritmo 3.

1.4.4. Espacio métrico

Hasta ahora no hemos hecho ninguna mención sobre el espacio de las coordenadas. El algoritmo de Vivaldi puede funcionar con cualquier espacio métrico, mientras las coordenadas puedan ser sumadas, multiplicadas por un escalar y exista una noción de distancia. En el trabajo de Lucángeli Obes [10] se utilizó un espacio euclidiano de 3 dimensiones con vectores de altura. La utilización de los vectores de altura mostró captar apropiadamente una componente de la latencia denominada *last mile*, propiedad que es especialmente útil para los casos en los que se trata de un alto porcentaje de la latencia total, como ser el tráfico residencial tipo Cablemodem. Nosotros utilizaremos el mismo espacio métrico para nuestras coordenadas.

Algoritmo 3 Vivaldi distribuido (timestep adaptativo)

{Input: rtt (latencia medida entre el nodo actual i y el nodo j)}
{Input: x_i y x_j (coordenadas actuales de los nodos i y j)}
{Input: e_i y e_j (errores relativos de las coordenadas actuales de los nodos i y j)}
{c_c y c_e son parámetros ajustables que varían entre 0 y 1}
{Output: coordenada actual x_i más precisa}
$w = \frac{e_i}{e_i+e_j}$
{Calculo el error de la muestra actual}
$e_s = \frac{|\|x_i-x_j\|-rtt|}{rtt}$
{Calculo el error de la coordenada como el promedio pesado por c_e entre el error que ya tengo y el de la muestra actual}
$e_i = e_s \times c_e \times w + e_i \times (1 - c_e \times w)$
{Calculo el paso temporal δ}
$\delta = c_c \times w$
{Actualizo mi coordenada con el paso temporal calculado}
$x_i = x_i + \delta \times (rtt - \|x_i - x_j\|) \times u(x_i - x_j)$

Mostramos la definición de las operaciones básicas para el caso más sencillo de 2 dimensiones con altura (los casos con más dimensiones son análogos en la parte euclidiana, mostramos esta versión por simplicidad):

- Resta:
$$(a_1, a_2, a_h) - (b_1, b_2, b_h) = (a_1 - b_1, a_2 - b_2, |a_h + b_h|)$$
- Producto por un escalar:
$$\alpha \cdot (a_1, a_2, a_h) = (\alpha \cdot a_1, \alpha \cdot a_2, \alpha \cdot a_h)$$
- Norma:
$$\|(x_1, x_2, x_h)\| = \sqrt{x_1^2 + x_2^2} + x_h$$

1.4.5. Mejoras implementadas

Además del algoritmo básico describimos algunas mejoras que aparecen en el trabajo de Ledlie et al. [8] y que aplicamos a nuestra implementación. Las primeras dos además fueron evaluadas en simulaciones en el trabajo de Lucángeli Obes [10]. Estas son:

- Filtro de mediana: en las pruebas iniciales notamos que las mediciones de latencia tenían una variación considerable. Esta característica era esperable, pero influye negativamente en la calidad de las coordenadas calculadas. Para mitigar este problema incorporamos esta mejora, que consiste en guardar un historial de las últimas n mediciones de latencia (el valor de n es configurable desde el programa) y quedarnos con la mediana en lugar de la última medición realizada.
- Extensión del conjunto de vecinos (neighbor decay): un problema que acarrea el hecho de utilizar los mensajes ya existentes de BitTorrent para actualizar las coordenadas

es que las actualizaciones siguen el patrón de tráfico de BitTorrent. Este patrón no necesariamente es el mismo en la versión P2P de Vivaldi una vez que lo implementemos sobre BitTorrent, ya que a qué peers se conecta cada integrante del swarm en este último caso depende de parámetros del tracker y del algoritmo de choking que corre cada peer, entre otras cosas.

Este problema puede derivar en que exista una segmentación de las coordenadas de los peers, si, por ejemplo, cierto grupo se conectó únicamente entre sí recientemente y no contra el resto. Esto tiene efectos poco deseados, ya que si calculamos la distancia entre dos nodos que se encuentran en particiones diferentes, el valor obtenido no será necesariamente representativo de la distancia real, incluso si los valores por separado tienen poco error.

Para mitigar este problema [8] propone una modificación al algoritmo de actualización de coordenadas, donde cambiamos la fuerza incidente F_i en cada nodo i por la siguiente expresión:

$$F_i = F_i + \sum_{j \in rn(i)} F_j \times \frac{a_{max} - a_j}{\sum_k a_{max} - a_k}$$

En la ecuación anterior, F_i indica la fuerza ejercida sobre la coordenada actual, F_j el desplazamiento producido por el vecino j, a_j la edad de la información sobre el desplazamiento F_j, a_{max} la mayor edad de entre la información de desplazamiento de todos los vecinos y $rn(i)$ el conjunto de los vecinos recientes de i. Mostramos el pseudocódigo del algoritmo resultante en el algoritmo 4.

De esta manera, utilizamos la información de todas las coordenadas obtenidas en cada iteración del algoritmo, priorizando las más actuales.

- Gravity drift: un problema que tiene el algoritmo de Vivaldi es que a medida que transcurre el tiempo, el centro de las coordenadas se traslada hacia alguna dirección. Esto tiene dos consecuencias no son deseables en nuestro sistema. La primera es que crece el valor absoluto de la parte entera de cada una de las componentes de la coordenada, lo cual puede ocasionar errores numéricos, ya que perdemos precisión en la parte decimal. Por otro lado, un desplazamiento constante impide que el sistema "converja", y esto hace que las posiciones calculadas pierdan validez rápidamente. Para mitigar este problema, [8] propone una solución que consiste en "atraer" las coordenadas hacia el origen, simulando el comportamiento de un campo atractivo gravitatorio.

$$\overrightarrow{G} = \left(\frac{\|\overrightarrow{x_i}\|}{\rho} \right) \times u(\overrightarrow{x_i})$$

En la expresión anterior, $\overrightarrow{G}$ es un vector que indica cuánto se atrae a $\overrightarrow{x_i}$ hacia el centro del sistema de coordenadas, y ρ es una constante cuyo valor se determinó empíricamente en 2^8 [8].

Una vez obtenido el vector $\overrightarrow{G}$, se lo resta al valor calculado en cada actualización de la coordenada para obtener el valor final "atraído" hacia el centro del sistema de coordenadas, tal como lo indica la siguiente expresión:

Algoritmo 4 Vivaldi distribuido (timestep adaptativo, neighbor decay)

{Input: $rtt_j \forall j \in rn(i)$ (latencia medida entre el nodo actual i y cada nodo j que se conectó recientemente con el nodo i)}
{Input: $x_j \forall j \in rn(i)$ (coordenadas de los nodos j que se conectaron recientemente con el nodo i)}
{Input: $a_j \forall j \in rn(i)$ (edades de las últimas coordenadas obtenidas para los nodos j que se conectaron recientemente con el nodo i)}
{Input: $e_j \forall j \in rn(i)$ (errores relativos de las coordenadas actuales de los nodos j tales que se conectaron recientemente con el nodo i)}
{c_c y c_e son parámetros ajustables que varían entre 0 y 1}
{Output: coordenada actual x_i más precisa}
{a_{max} es la edad máxima de entre todos los vecinos recientes de i}
$a_{max} = max(a_j / j \in rn(i))$
{$total\_time$ es el tiempo total de los datos de los vecinos recientes}
$total\_time = \sum_k a_{max} - a_k$
for all j **in** rn(i) **do**
  $w = \frac{e_i}{e_i + e_j}$
  {Calculo el error de la muestra actual}
  $e_s = \frac{|\|x_i - x_j\| - rtt|}{rtt}$
  {Calculo el error de la coordenada como el promedio pesado por c_e entre el error que ya tengo y el de la muestra actual}
  $e_i = e_s \times c_e \times w + e_i \times (1 - c_e \times w)$
  {Calculo el paso temporal δ}
  $\delta = c_c \times w$
  {Calculo el peso del vecino actual en función de la edad de su coordenada}
  $nw = \frac{a_{max} - a_j}{total\_time}$
  {Actualizo mi coordenada con el paso temporal calculado}
  $x_i = x_i + nw \times \delta \times (rtt - \|x_i - x_j\|) \times u(x_i - x_j)$
end for

$$\vec{x_i} = \vec{x_i} - \vec{G}$$

1.5 BitTorrent

1.5.1. Descripción

BitTorrent [4, 5] es una red P2P que se utiliza para la distribución de archivos. Una red BitTorrent está compuesta básicamente por los hosts participantes, denominados peers[10], y un tracker, que es un servidor web especializado al que los peers se conectan inicialmente. Esta arquitectura hace que BitTorrent sea una red P2P *híbrida*, según nuestra definición de la sección 1.3.1. Al conjunto de peers participantes de un torrent se lo denomina *swarm*. La información que necesitan los peers para iniciar la transferencia se encuentra en archivos llamados *torrents*, usualmente con extensión *.torrent*.

El archivo compartido se encuentra dividido en partes de igual tamaño, a las que se las denomina *pieces*. El tamaño de cada una es configurable por torrent. Cada piece tiene además información asociada de, entre otras cosas, el hash[11] de la misma, de modo de poder comprobar si la descarga se realizó correctamente sin necesidad de descargar el archivo en su totalidad. Las pieces mal descargadas son inmediatamente descartadas y pedidas nuevamente, asegurando de esta manera la consistencia final de los datos.

Mencionamos a continuación a modo de reseña distintas extensiones que aparecieron implementadas en los clientes a lo largo de la vida de BitTorrent, si bien en nuestro trabajo nos concentraremos en la versión más elemental (ver Cohen [5]). La lista no pretende ser exhaustiva, simplemente ofrece una perspectiva de qué rumbos tomó el desarrollo de BitTorrent.

- Fast Peers Extension [25]: modifica la semántica de los mensajes de control del protocolo e incorpora los mensajes adicionales **have all**, **have none**, **suggest piece**, **reject request** y **allowed fast** que permiten un intercambio más eficiente entre peers.

- DHT, Peer Exchange Protocol, torrents *trackerless*: estas 3 extensiones forman de alguna manera una única funcionalidad. DHT se utiliza para rutear tráfico entre distintos nodos que no se conocen directamente, el Peer Exchange Protocol cumple la función de intercambiar entre peers sus propias listas de peers conocidos. Todo esto se combina para poder implementar torrents trackerless, es decir, donde no hay ningún nodo distinguido en la red al que todos se conectan para iniciar la descarga.

- Cifrado del protocolo: el cifrado del protocolo surgió originalmente como mecanismo para poder saltear límites que imponían usualmente los ISPs residenciales al ancho de banda asignado a P2P. Además suponía una ganancia en cuanto a privacidad al no transmitir los datos en texto plano por la red. Estas ventajas ya no son tan claras, ya que existen trabajos que identifican el tráfico BitTorrent según características del flujo y no de su contenido [26], y se puede obtener la lista de torrents que un nodo determinado está descargando utilizando un cliente instrumentado que simule ser un peer más y vaya pidiendo las listas de peers a un tracker hasta que encuentra a su "víctima".

[10] Denominaremos peers, clientes o nodos a los hosts participantes indistintamente.

[11] Identificador generado por alguna función de hashing, utilizado a menudo para indexar elementos en una estructura o para verificar la consistencia de un conjunto de datos.

- Uso de μTP como protocolo de transporte: el protocolo de transporte usado en la versión original de BitTorrent, TCP, tiene algunos problemas manejando la congestión cuando hay muchas conexiones simultáneas. Para aliviar este problema se diseñó un protocolo de capa de aplicación, montado sobre UDP, que implementa un algoritmo de control de congestión pensado específicamente para este tipo de cargas. Este protocolo apareció implementado inicialmente en el cliente oficial de BitTorrent, el μTorrent (de ahí su nombre).

1.5.2. Los archivos *torrent*

Un *torrent* es como se conoce generalmente a archivos con información sobre el archivo[12] compartido (por eso se los denomina archivos de *metadata*).

El contenido de estos archivos es un diccionario con los valores necesarios para que el peer pueda descargar o compartir el archivo en cuestión.

1.5.3. Los peers

Se denomina *peer* a cualquier nodo participante. Entre los *peers*, distinguimos dos clases:

- *Seeder*: cualquier peer que provee los datos del archivo completo para su descarga. Inicialmente es el que comparte el archivo, pero todos los que finalicen su descarga se convierten automáticamente en *seeders*.
- *Leecher*: cualquier peer que se encuentre descargando parte del archivo compartido.

1.5.4. El tracker

El *tracker* es un software que funciona como servidor web especializado. Se lo utiliza para que los peers puedan avisar cambios de estado (conexión, desconexión, etc) a todos los participantes de un torrent, y además es el punto de partida por el que los peers se pueden informar acerca de quiénes pueden bajar el archivo en el que se encuentran interesados.

1.5.5. El proceso de compartir archivos

El proceso por el cual se comparten archivos en BitTorrent se inicia con la creación de un archivo *torrent* con la *metadata* del archivo que queremos compartir. Una vez generado, debe ser almacenado en algún sitio desde donde sea posible acceder remotamente, o bien distribuido por algún medio digital. A su vez, el o los peers interesados en compartir el archivo deben anunciarse al tracker, donde notificarán su intención de participar de ese torrent. Los peers que se unan luego, una vez en posesión del torrent, se anunciarán a su vez al tracker, y pasarán a formar parte del swarm correspondiente a ese archivo. El tracker, por su lado, informará a cada miembro del swarm en el momento de la reasociación de quiénes puede bajarse el archivo, información que utilizarán internamente para realizar las transferencias. Cada cliente es libre de utilizar cualquier algoritmo para controlar el flujo de transferencias, pero idealmente el protocolo sugiere utilizar un esquema "tit-for-tat", que impida que un peer malintencionado

[12] Hablaremos del caso de un archivo compartido por torrent, aunque es posible compartir batches, o múltiples archivos, en uno sólo.

se aproveche de los que están compartiendo el archivo y pueda conseguirlo sin ofrecer nada a cambio.

Una vez que alguno de los leechers obtiene el archivo en su totalidad, se convierte en seeder, y continúa ofreciendo el archivo a los nuevos leechers hasta que todos lo tienen. Este proceso puede durar indefinidamente, ofreciendo el archivo a nuevos leechers, mientras el peer no decida cerrar las conexiones y desasociarse del tracker. Una vez que todos los peers cierran sus conexiones, se considera que el torrent se encuentra "inactivo".

Cada cierta cantidad de tiempo configurable, el peer debe reasociarse al tracker. Esto permite que varíen los peers de los que se realiza la descarga, de forma de poder obtener todas las pieces disponibles en el swarm. Para descargar una piece, envía su **bitfield** al cliente al que se conectó, quien a su vez le informa el suyo. Sabiendo qué pieces tiene cada uno, utiliza algún criterio de selección para pedir la piece. Este criterio no se encuentra definido en el standard documentado por Cohen [5]: cada cliente es libre de tomar sus propias decisiones. En general se utiliza el criterio *random first* en las primeras pieces transferidas, seguido por *rarest first* para el resto de la misma [4], aunque podrían utilizarse otros.

Cada peer debe guardar sobre los demás una tabla con los siguientes estados:

- **choked**: 1 si el peer remoto tiene al local en estado *choked*. Esto quiere decir que el peer local no va a recibir pieces del remoto mientras siga en ese estado. En el establecimiento de la conexión la entrada correspondiente tiene el valor 1.

- **interested**: 1 si el peer está interesado en alguna de las pieces que tiene el otro, 0 en caso contrario. En el establecimiento de la conexión la entrada correspondiente en este caso tiene el valor 0.

El funcionamiento de los dos estados anteriores se articula por medio del *algoritmo de choking*. Este algoritmo es de suma importancia para el correcto funcionamiento del sistema. Como ya mencionamos, las conexiones inicialmente se establecen como *not interested* y *choked*. Luego, en función del **bitfield** y los mensajes **have** que reciba del peer remoto, cada peer podrá cambiar el estado de **interested** a 1 (si tiene alguna piece que le interese). De esta forma, recibirá la piece que le pida si el peer remoto lo sacó previamente de la tabla de **choked**.

1.5.6. El algoritmo de choking

El algoritmo de choking (ver algoritmo 5) es una parte fundamental del funcionamiento de una red BitTorrent. Hay dos motivos por los que es sumamente necesario [5]:

- Problemas con el algoritmo de control de congestión de TCP: este algoritmo no está pensado para funcionar óptimamente en escenarios con una gran cantidad de conexiones. El algoritmo de choking permite mitigar este problema, pero existe trabajo ligado a atacarlo por medio del uso de un protocolo de transporte diferente (como ya mencionamos, por μTP [27], un protocolo implementado sobre UDP).

- Asegurar el funcionamiento equilibrado "*tit-for-tat*" [4] de la red: cada peer que no es seeder envía datos en la medida en que recibe de otros.

Algoritmo 5 Algoritmo de choking

```
{mejorTasaChoking(p) define si el peer p se encuentra dentro de los 4 peers con mejor tasa
de subida (si es seeder) o de bajada (si es leecher)}
{choking(p) define si el peer p está en estado choking}
{choke(p) setea al peer p en estado choking}
{unchoke(p) setea al peer p en estado not choking}
{elegirRandom(p in list) elige un elemento p de forma aleatoria de la lista list}
for all p in peers do
  if mejorTasaDeSubida(p) and choking(p) then
    unchoke(p)
  end if
  if not mejorTasaDeSubida(p) and not choking(p) then
    choke(p)
  end if
end for
unchoke(elegirRandom(p in peers))
```

1.5.7. Análisis de los clientes

Nuestro objetivo es construir un cliente de BitTorrent que maximice el tráfico de los clientes dentro de un mismo ISP. Esto requiere que construyamos primero una implementación de BitTorrent y luego que montemos sobre ella nuestra solución. Habiendo tantos clientes disponibles, creemos que no tiene sentido construir una nueva implementación que no ha sido probada en la práctica para construir sobre ella, por lo que decidimos utilizar uno de los muchos clientes libres[13] disponibles a través de Internet.

Nuestra intención fue buscar el cliente cuyo diseño fuese de baja complejidad y que estuviera implementado en un lenguaje de fácil prototipado. Para los clientes que implementasen una GUI nos interesó ver si era posible deshabilitarla o ejecutarlo de forma batch en múltiples máquinas simultáneamente, condición necesaria para poder hacer corridas de tamaño interesante automáticamente. Además, queríamos que fuese similar en funcionalidad a la implementación de referencia del protocolo original, para de esta manera poder analizar el funcionamiento de la extensión más aisladamente.

De todos los clientes analizados, decidimos utilizar **BitTornado**. Su código fuente está disponible bajo la licencia MIT[14], por lo que puede ser analizado, modificado y redistribuido.

BitTornado implementa una versión básica del protocolo de BitTorrent, sin extensiones (ver la documentación de Bram Cohen [5]).

[13] Los clientes cuyo código fuente no está disponible o no puede ser modificado y redistribuido obviamente no nos ofrecen una alternativa viable para la realización de este trabajo, por lo que fueron descartados sin mayor análisis.

[14] Licencia del X Consortium del Massachusetts Institute of Technology (MIT).

2 Implementación y metodología

2.1 Simulador de Vivaldi

2.1.1. Motivación

Como primer acercamiento implementamos el algoritmo distribuido de Vivaldi pero simulando su ejecución en un conjunto de nodos. En lugar de utilizar alguno de los simuladores de red existentes decidimos hacer nuestra propia implementación ad-hoc. Esto nos permitió utilizar el simulador como plataforma de prueba del algoritmo y además funcionó como punto de partida para la implementación real. El código correspondiente a Vivaldi es común a todas las implementaciones realizadas.

2.1.2. Descripción y uso

Para utilizar el simulador, en primer lugar necesitamos obtener datos sobre las latencias entre cada par de nodos simulados. Para hacer pruebas más realistas optamos por capturar información sobre las latencias promedio entre nodos de nuestra red de prueba. Construimos un script que utiliza `ICMP echo` y captura los valores de RTT. Esa información se guarda en un formato que reconoce el simulador. Una vez que tenemos las latencias le tenemos que informar al simulador cuántos nodos queremos simular (obviamente, la cantidad de nodos tiene que ser tal que haya suficientes datos de latencia).

Con todos estos datos cargados, el simulador inicia un thread por cada nodo simulado con un valor aleatorio de coordenada, y cada uno de ellos se "conecta" con otro elegido aleatoriamente una vez por segundo. La conexión es simplemente un llamado al método de otro thread donde el emisor pide el valor de la coordenada del receptor. Este valor está protegido por un lock (*mutex*) para evitar que accesos concurrentes corrompan los datos. Utilizando ese valor y la latencia fija parámetro de la simulación se actualiza la coordenada del emisor según el algoritmo 3.

Además de simular el comportamiento del sistema nos permite obtener una salida equivalente a la resultante de la ejecución de Vivaldi en la versión P2P o sobre BitTorrent, que veremos en las secciones subsiguientes. Esto nos permite analizar la evolución de las coordenadas de los nodos y comparar el funcionamiento del algoritmo entre utilizar valores teóricos fijos de latencia y los reales, con sus fluctuaciones correspondientes.

Para obtener detalles acerca del uso de este software, referirse al apéndice A.1.

2.2 Vivaldi versión P2P

2.2.1. Arquitectura

Una vez que obtuvimos una implementación inicial de Vivaldi, el siguiente paso fue construir un software P2P donde el sistema verdaderamente se ejecute de forma distribuida, tomando mediciones reales de latencia. Cada peer se inicializa con un conjunto de nodos fijo y una posición inicial aleatoria. Luego escucha conexiones por el puerto TCP 65432.

En el ciclo principal se conecta contra algún peer elegido aleatoriamente entre su lista de vecinos y recibe su coordenada. Con esta información y la latencia calculada, actualiza la suya y termina la conexión. Luego espera 1 segundo antes de continuar. Finaliza o bien cuando pasa una cantidad de segundos configurable por línea de comandos o bien cuando es terminado explícitamente.

Además de iniciar conexiones, también atiende pedidos de otros peers en el puerto TCP antes mencionado. En este caso, lo que hace es enviar su coordenada al peer que inició la conexión y finalizarla. Para no interrumpir el ciclo principal, cada conexión es manejada por un thread separado. Para asegurar la consistencia de los datos protegemos el acceso a la coordenada local con un lock (*mutex*).

2.2.2. Protocolo

El protocolo de Vivaldi tiene un único comando donde se envía la coordenada junto con el error calculado y el timestamp de envío. Según la cantidad de dimensiones, tiene los siguientes componentes, en el orden en el que aparecen a continuación:

- `x`: tipo `Float`, componente x de la posición del cliente que envía el mensaje.
- `y`: tipo `Float`, componente y.
- `z`: tipo `Float`, componente z. Sólo en 3D + H.
- `w`: tipo `Float`, componente w. Sólo en 3D + H y 4D + H.
- `h`: tipo `Float`, componente h. Esta componente representa una altura, independiente de la posición determinada por las otras componentes.
- `error`: tipo `Float`, representa el error calculado de la coordenada enviada.
- `tx`: timestamp (tipo `Double`) de envío del mensaje.

Sobre el timestamp debemos hacer una mención especial: este parámetro se utiliza para medir explícitamente la latencia entre los dos *peers* que se están comunicando. El receptor del mensaje sólo debe restar el valor del timestamp de recepción con el de emisión, y de esa forma obtiene el tiempo que demoró el mensaje en llegar a destino. Podemos ver claramente que este método tiene un problema: si los relojes no están sincronizados entre el emisor y el receptor, la medición va a ser incorrecta. Esta precondición es bastante fuerte, pero la podemos cumplir de forma relativamente sencilla sincronizando la hora de cada *host* contra un servidor NTP[1]

[1] Protocolo para sincronizar la hora entre distintos equipos (ver RFC 1305).

confiable. La mayoría de los sistemas operativos modernos incluye un cliente NTP *out-of-the-box* [28–30], y además ya existen otros servicios que requieren sincronización de relojes para funcionar correctamente (NFS [31], Globus, etc.).

Para más detalles sobre la codificación de los campos del mensaje en la implementación de Vivaldi, referirse al apéndice B.1.

Analizamos además otros métodos para medir la latencia explícitamente, que mencionamos a continuación:

- **SYNACK/ACK**: este método aparece descripto en el trabajo de Szymaniak et al. en [32]. Tiene la ventaja de que no requiere tráfico adicional ni otro tipo de sincronización entre los peers, pero funciona midiendo tiempos en el establecimiento de la conexión TCP entre los mismos. Este método fue implementado con una modificación en nuestra implementación de Vivaldi P2P, pero no puede ser trasladado directamente a la versión sobre BitTorrent, ya que este último mantiene las conexiones abiertas entre mensajes. Nuestra versión de SYNACK/ACK mide del lado del cliente, ya que es el iniciador de la conexión el que necesita medir la latencia para poder actualizar su coordenada. Podemos ver la comparación esquematizada en la figura 2.1.

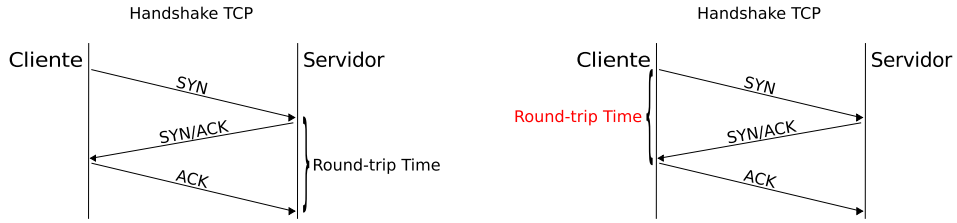


Figura 2.1: Comparación entre la versión original del método SYNACK/ACK [32] para medir RTT pasivamente del lado del servidor y la modificación propuesta, que mide activamente del lado del cliente.

- **ICMP echo**: este método consiste en enviar un mensaje ICMP echo, y tomar el tiempo que demora en llegar la respuesta con un ICMP reply (así es como funciona el programa **ping**), pero en muchas redes se encuentra filtrado, por lo que no es una buena idea utilizarlo en una implementación genérica del sistema.

El método SYNACK/ACK fue utilizado en nuestra implementación de Vivaldi P2P para corroborar la correctitud de las latencias medidas utilizando la resta de timestamps.

Por más detalles acerca del uso de este software, referirse al apéndice A.2.

2.3 Vivaldi sobre BitTorrent

2.3.1. Protocolo de BitTorrent extendido

Para poder incorporar Vivaldi sobre BitTorrent, extendimos algunos mensajes del protocolo para que incluyan los datos de coordenadas de Vivaldi. De la misma manera que la implementación de Vivaldi sin BitTorrent de la que hablamos en la sección 2.2, esta información se utiliza para corregir las coordenadas de cada cliente de manera distribuida.

Además de los mensajes, modificamos el campo **reserved** del mensaje **handshake** para que active el bit 20 si desea utilizar el protocolo extendido con el cliente remoto. Si este último también lo activa, se lo utilizará en la comunicación. En caso contrario, sigue utilizándose el protocolo normal. Decidimos cuál de los bits de **reserved** utilizar para señalar el uso del protocolo extendido en función de que no estuviera reservado para un fin diferente por otras implementaciones, según documentación de la comunidad [33].

Elegimos modificar todos los mensajes de control presentes en el protocolo, con la excepción de **cancel**, que se utiliza únicamente en el final de una transferencia.

Listamos a continuación todos los mensajes modificados con una breve descripción de su uso:

- **bitfield**: mensaje opcional, enviado luego del handshake, pero sólo una vez por conexión. Indica al peer remoto con una lista de bits qué pieces del archivo posee.
- **choke**: informa al peer remoto que se encuentra en estado **choked** en la tabla de estados del peer local, por lo que no va a aceptar pedidos de pieces del primero.
- **unchoke**: informa al peer remoto que se encuentra en estado **unchoked** en la tabla de estados del peer local. Si se encuentra en este estado, este último va a aceptar pedidos de pieces del primero.
- **interested**: informa al peer remoto que se encuentra en estado **interested** en la tabla de estados del peer local, por lo que se encuentra interesado en alguna *piece* que posee el primero.
- **not interested**: informa al peer remoto que se encuentra en estado **not interested** en la tabla de estados del peer local, por lo que no se encuentra interesado en ninguna piece que posee el primero.
- **have**: informa al peer remoto que obtuvo una piece determinada y que ahora está disponible para su envío a otros peers.

En cada uno de los mensajes modificados, agregamos la coordenada de Vivaldi al final de su contenido original, y además actualizamos el campo inicial, que indica de qué tamaño es el mensaje, para que se corresponda con la nueva longitud. Un cliente que se encuentra utilizando el protocolo extendido cierra la conexión si el cliente remoto le envía alguno de los mensajes modificados con un valor en el campo de longitud que no sea el correcto.

2.3.2. Dirección del tráfico utilizando Vivaldi

Luego de calcular las coordenadas y actualizarlas necesitamos utilizarlas para dirigir el tráfico de la red. Nuestra solución implementa una modificación al algoritmo de *choking* (ver algoritmo 5, pág. 24), donde en lugar de utilizar el rate de subida o bajada (se utiliza el de bajada si el cliente es un leecher o el de subida si es un seeder) para ordenar la preferencia de conexión con clientes remotos utilizamos el *upload appeal* [10], que es calculado de la siguiente forma:

$$appeal_i = \frac{{rate_i}^2}{dist_i}$$

donde

- $appeal_i$: *upload appeal* correspondiente al cliente i.
- $rate_i$: tasa de subida o bajada (según corresponda) del cliente i.
- $dist_i$: latencia estimada (distancia) al cliente i.

Mostramos la versión modificada del algoritmo de choking en el algoritmo 6.

Algoritmo 6 Algoritmo de choking modificado

```
{mejorUploadAppeal(p) define si el peer p está dentro de los 4 peers con mejor upload appeal}
{choking(p) define si el peer p está en estado choking}
{choke(p) setea al peer p en estado choking}
{unchoke(p) setea al peer p en estado not choking}
{elegirRandom(p in list) elige un elemento p de forma aleatoria de la lista list}
for all p in peers do
  if mejorUploadAppeal(p) and choking(p) then
    unchoke(p)
  end if
  if not mejorUploadAppeal(p) and not choking(p) then
    choke(p)
  end if
end for
unchoke(elegirRandom(p in peers))
```

2.3.3. Implementación sobre BitTornado

La funcionalidad nueva o modificada para la implementación de Vivaldi sobre BitTornado es la siguiente:

- Una estructura de representación para las coordenadas. Se implementa en los módulos `BitTornado.BT1.HeightCoordinate` y `BitTornado.BT1.VivaldiCoordinate`. Además, como debe poder ser transmitida por la red, es necesario incorporar los módulos `BitTornado.BT1.DoubleEncoder` (para el *timestamp*) y `BitTornado.BT1.FloatEncoder` (para el resto de los campos) para codificar las componentes de las coordenadas.
- Modificar el *parsing* de los mensajes del protocolo. El módulo que se encarga de esta tarea es `BitTornado.BT1.Connecter`.
- Para compatibilizar con otros clientes BitTorrent, un mecanismo para decidir si se debe utilizar el protocolo extendido o no. El mismo se detalla en las modificaciones al módulo `BitTornado.BT1.Encrypter`.
- Un mecanismo para recolectar datos de los clientes. El módulo `BitTornado.statslogger` se encarga de esta tarea.

La documentación detallada acerca de los módulos nuevos o modificados, con las funciones, clases y métodos agregados se encuentra en el apéndice C.

2.4 Metodología experimental

Un gran problema a la hora de analizar el funcionamiento de un protocolo de red en Internet es que es sumamente complicado caracterizar el funcionamiento de un sistema en condiciones "reales". A pesar de esto, resulta clara la necesidad de analizar aplicaciones en redes globales antes de ponerlas en funcionamiento, por lo que buscamos una alternativa que nos permitiese obtener conclusiones que nos ofrezcan una buena intuición acerca de si es razonable su implementación.

Trabajos anteriores centraban sus pruebas en simuladores de red o en simulaciones con trazas de latencia obtenidas experimentalmente en una red real [8,10,15,34,35]. Los simuladores son herramientas muy útiles para el diseño e implementación de aplicaciones que funcionan sobre Internet, pero el software debe ser preparado para correr en el mismo. Esto lleva a que existan diferencias entre la implementación utilizada en las simulaciones y la versión final que pueden introducir variables que no son tenidas en cuenta en el análisis. En nuestro caso, nos propusimos obtener implementaciones que funcionasen *out-of-the-box* en redes reales.

Para poner a prueba nuestra implementación utilizamos una plataforma de prueba (*testbed*) llamada Emulab [36]. El aspecto más importante que consideramos a favor de Emulab es el hecho de que podíamos configurar las redes de prueba con topologías y condiciones de red fijas. Este punto nos otorga repetibilidad en la experimentación, que es sumamente importante para poder contrastar los resultados experimentales, y en Internet en general no la podemos tener.

2.4.1. Características de la plataforma de prueba

Las condiciones de red que nos permite configurar Emulab son:

- Latencia de los links entre nodos.
- Latencia entre nodos de una misma LAN.
- Porcentaje de pérdida de paquetes.
- Reordenamiento de paquetes con cierta probabilidad.

En nuestras pruebas utilizamos la capacidad de configurar las latencias entre nodos y entre una misma LAN, ya que Vivaldi actúa construyendo un modelo que posiciona los nodos según su latencia.

En cuanto a la implementación de Emulab, internamente cada nodo simulado es representado por un único nodo real. Cada link también, con la diferencia de que en este caso el nodo real funciona como *bridge* entre los nodos que une, enviando paquetes de un lado al otro. La latencia se simula en los links configurando un retardo en el envío de paquetes. El manejo de colas tiene varias políticas configurables, si bien en nuestros experimentos utilizamos siempre la misma. Las LANs son construidas configurando VLANs[2], una por LAN, donde se incluye a los nodos correspondientes. Para poder automatizar esta tarea, los switches utilizados tienen que permitir la configuración de VLANs utilizando SNMP.

Todo el proceso anterior, incluyendo la decisión de cuáles recursos serán utilizados y la posterior inicialización del sistema operativo, es realizado de forma automática por el sistema de control de Emulab en el momento de instanciación (denominado *swapin*) de la red.

[2] Ver IEEE 802.1Q.

Para la adquisición de datos implementamos un servidor de logging centralizado, localizado en nuestro nodo central (`node0`). Cada uno de los clientes toma sus propias mediciones y envía actualizaciones cada 10 segundos al servidor de logging. Este último guarda la fecha y hora, dirección IP de origen del mensaje, y cada uno de los datos recibidos. Estos datos son finalmente procesados y agrupados en intervalos de tiempo de duración constante para su posterior visualización. Como ningún nodo envía actualizaciones cada menos de 10 segundos, tomamos ese valor como longitud del intervalo temporal.

El sistema operativo utilizado en todos los nodos fue la versión de 32 bits (i386) de Ubuntu 10.04 LTS, que incluye el intérprete de Python versión 2.6.3.

2.4.2. Validación de Vivaldi

Para analizar la performance de Vivaldi, observamos la topología generada por las posiciones dadas por el algoritmo para cada uno de los nodos. Las redes utilizadas fueron construidas de forma que todos los nodos pertenecientes a la misma LAN tengan latencias que sean un orden de magnitud menor que las de los links que se utilizan para la interconexión, por lo que esperamos que nuestro algoritmo agrupe nodos de una misma LAN con poca distancia entre sí, y que en cambio haya una distancia mucho mayor entre ese grupo y cualquier otro. En la figura 2.2 mostramos a modo de ejemplo una comparación entre la representación de la red pequeña **fourlan** como grafo y la visualización 3D de las coordenadas generadas en una corrida de Vivaldi para esa misma red. Allí podemos comparar la posición asignada a cada nodo con su correspondencia en la red. Como ayuda visual, para facilitar la ubicación de los nodos cada uno fue coloreado de la misma manera en ambos gráficos.

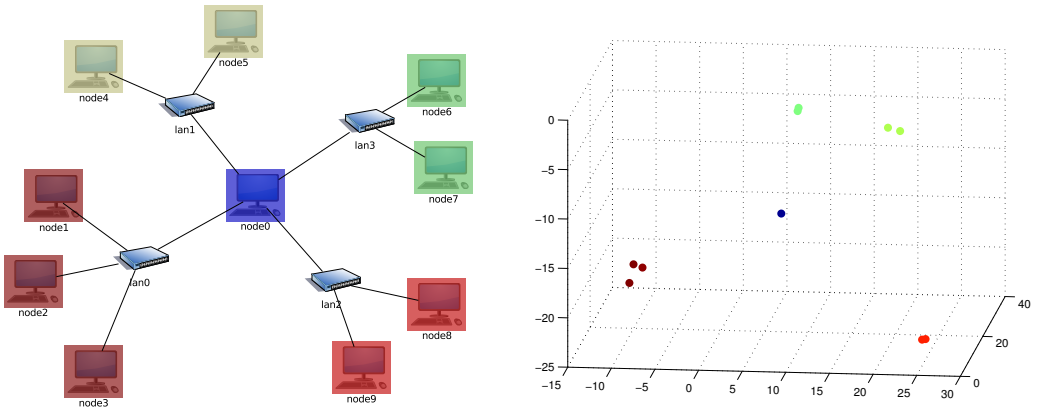


Figura 2.2: Comparación entre la representación como grafo de la red **fourlan** y la vista 3D de las coordenadas generadas por el algoritmo de Vivaldi para una corrida en la misma red.

En nuestro trabajo evaluamos la validez de Vivaldi como mecanismo para modelar latencias entre nodos.

Nuestro análisis se basa en comparar los grupos de nodos construidos utilizando el algoritmo de clustering *k-means* a partir de los datos generados por Vivaldi con la agrupación por latencias de los mismos nodos. K-means toma como entrada las posiciones de los nodos y cuántos (k) clusters queremos obtener y calcula las posiciones de los centroides y las pertenencias de nodos a clusters según algún criterio de minimización de distancias. En nuestro

caso, utilizamos el criterio de minimización del cuadrado de la distancia euclidiana. El valor del número de clusters es conocido, ya que sabemos cuántas LANs construimos en cada una de las redes. Omitimos el nodo central de los datos de entrada para evitar agregar mayores incertezas en la clusterización de las LANs, ya que en principio, como no está conectado a ningún nodo con un link de muy baja latencia, éste podría pertenecer a cualquiera de sus LANs vecinas. El resultado esperado tras realizar este proceso es que los clusters obtenidos no sean muy grandes (es decir, que no se agrupen nodos muy distantes entre sí, ya que los nodos de una misma LAN tienen muy poca latencia entre ellos), que se encuentren separados entre sí (esta separación describiría los links de mayor latencia que conectan LANs), y que además cada uno de los miembros pertenezca a una misma LAN.

A modo de ejemplo mostramos en la figura 2.3 cuatro gráficos donde podemos observar la evolución de las posiciones de las coordenadas generadas por Vivaldi para la red **fourlan** en su representación 3D, para cuatro instantes de tiempo diferentes, ordenados cronológicamente de izquierda a derecha y de arriba a abajo. Además de las coordenadas graficamos los clusters generados por *k-means* como una esfera semitransparente gris con centro en el centroide del cluster y el radio calculado como la distancia máxima entre el centroide y cada uno de los nodos del cluster. Todos los nodos forman parte de la clusterización, con la excepción del central, que se encuentra coloreado en azul. Los casos en los que no se observa la esfera gris indican que el radio de ese cluster es 0, es decir, que ningún otro nodo forma parte de ese cluster. Esta figura nos ofrece una noción visual de las métricas utilizadas.

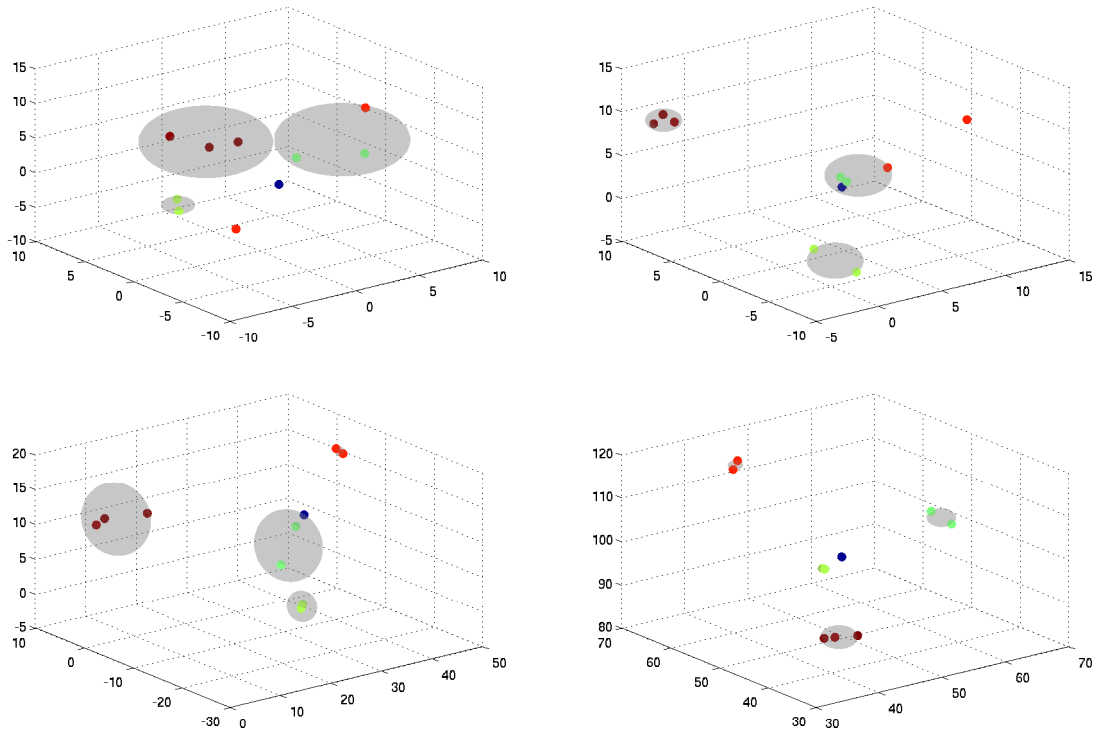


Figura 2.3: Representación en 3D de las coordenadas de los nodos en una misma corrida en la red **fourlan** junto con los clusters calculados en los instantes $t = 10$, $t = 50$, $t = 110$ y $t = 3920$ segundos

Realizamos dos análisis de la clusterización de Vivaldi para validar a este último como mecanismo para modelar latencias. Los describiremos en detalle a continuación.

Evolución de los tamaños de los clusters

El primer análisis consistió en observar el tamaño de los clusters para los distintos valores de las constantes c_c y c_e. Graficamos la evolución temporal de los radios máximo, medio y mínimo de entre todos los clusters en cada intervalo de tiempo. El radio mínimo indica cuál es el radio del cluster más chico en cada instante de tiempo, mientras que el radio máximo indica cuál es el radio del cluster más grande. El radio medio se calcula como el valor promedio de los radios de todos los clusters. Cada una de las curvas resultantes fue suavizada para mejorar su legibilidad, utilizando para cada punto x_i de la misma la siguiente relación:

$$\begin{aligned} x_{1_{\text{final}}} &= x_1 \\ x_{2_{\text{final}}} &= \frac{x_1 + x_2 + x_3}{3} \\ x_{3_{\text{final}}} &= \frac{x_1 + x_2 + x_3 + x_4 + x_5}{5} \\ i \geq 4, \quad x_{i_{\text{final}}} &= \frac{x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3}}{7} \end{aligned}$$

Este análisis nos habla acerca de la forma de los clusters obtenidos, pero nada dice sobre su composición. Resulta claro que, aunque la dispersión de los clusters sea grande y su radio pequeño, si los nodos que lo componen corresponden a diferentes redes, las coordenadas no nos permiten segmentar correctamente. Este punto será evaluado a continuación.

Correctitud de la clusterización

Para analizar qué tan buenos son los resultados de la clusterización propuesta, introducimos una métrica a partir de la comparación con otro método de clusterización basado en las latencias. Una clusterización puede pensarse como una separación de los nodos en clases de equivalencia. La idea detrás de la métrica propuesta es contar cuántos nodos quedaron en clases de equivalencia distintas, para luego normalizar utilizando el tamaño de la red. De esta manera, podemos escribir el error como la siguiente expresión:

$$error = \frac{nodos\_incorrectos}{nodos\_clusterizados}$$

donde *nodos_incorrectos* indica la cantidad de nodos que pertenecen a una clase de equivalencia distinta a la indicada por la clusterización por k-means, y *nodos_clusterizados* son todos los nodos que forman parte de ambas clusterizaciones.

Para construir la clusterización por latencias, adquirimos los datos de latencia de cada una de las redes entre cada nodo y el nodo central (desde donde se tomaron las mediciones). Estos valores fueron cuantizados tomados como colores en escala de grises. De esta manera obtuvimos un valor de color para cada nodo en una misma clase de equivalencia. Para la adquisición de las latencias enviamos (sin tráfico adicional en la red) una secuencia de 50 paquetes ICMP echo (ping), espaciados un segundo entre sí, y almacenamos el valor medio de toda la secuencia.

En base a los colores calculados podemos contar cuántos quedaron en la misma clase de equivalencia en ambas clusterizaciones. Para realizar esto contamos cuántos nodos en la

clusterización por k-means tienen un color distinto al mayoritario para ese cluster. Si un cluster tiene nodos del mismo color pero no se encuentran todos los que deberían estar, el error será contado en el cluster donde sí se lo incluyó, ya que ese color no se corresponderá con el utilizado en el otro cluster.

Para más información sobre el cálculo de *nodos_incorrectos*, en el algoritmo 7 se puede encontrar su definición algorítmica detallada.

Algoritmo 7 Cálculo de la cantidad de nodos incorrectamente clusterizados

{Input: peers (todos los nodos de la red)}
{Output: cantidad de nodos mal clusterizados por k-means}
{cuantizar(`color`, `maximoscolores`): cuantiza el color dado que los resultantes no sean más de `maximoscolores`}
{kmeans(`nodos`, `posiciones`): obtiene la clusterización por k-means de `nodos`, dadas las posiciones `posiciones`}
{contarcolordiferente(`nodos`, `colores`, `color`): cuenta cuántos nodos en `nodos`, cuyos colores corresponden a `colores`, tienen color diferente a `color`}
{maxlatencia(`nodos`, `nodoinicial`): calcula la latencia máxima entre los nodos en `nodos` y `nodoinicial`}
for all p **in** peers **do**
  colores(p) = cuantizar($\frac{\text{latencia}(p,\texttt{node0})}{maxlatencia(\text{peers},\texttt{node0})}$, numclusters)
end for
clusters = kmeans(peers - {`node0`}, posiciones)
E = 0
for all cluster **in** clusters **do**
  colormayoritario = max(colores(p) **in** colores)
  $\text{error}_{\text{cluster}}$ = contarcolordiferente(nodos(cluster), colores, colormayoritario))
  E = E + $\text{error}_{\text{cluster}}$
end for
return E

3 Experimentación

3.1 Entorno utilizado

Definimos para nuestros experimentos 3 redes diferentes: una red pequeña, que fuese sencilla de analizar, una tan grande como pudimos obtener con los recursos disponibles, y una mediana, de tamaño intermedio entre ambas.

Nuestra red pequeña cuenta con 12 nodos organizados en una topología de estrella, con un nodo central donde se realizaron todas las mediciones, y 4 LANs interconectadas por medio del nodo central. Cada LAN tiene dos a tres nodos, los cuales tienen una latencia entre ellos notablemente más baja que hacia cualquier otro nodo fuera de la misma. Podemos ver una esquematización de esta red, que llamamos **fourlan**, en la figura 3.1. Para más detalles, en el apéndice D.1 podemos encontrar el código completo de esta red en el formato que interpreta Emulab.

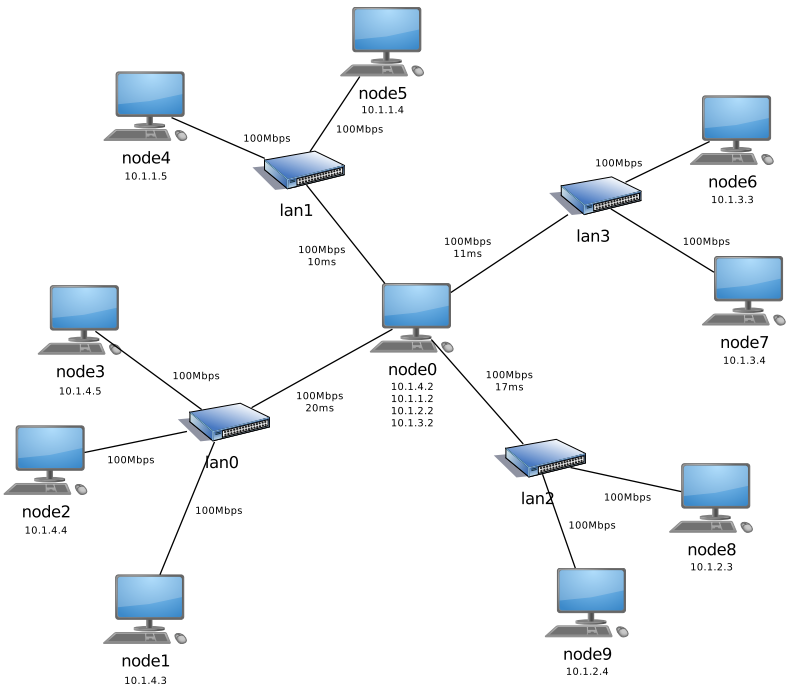


Figura 3.1: Red de prueba pequeña de 4 LANs interconectadas (fourlan).

La red mediana tiene 33 nodos dispuestos en 10 LANs, y fue construida extendiendo la topología de la red pequeña de forma similar a cómo se conectan las redes en Internet. En esta red encontramos un camino de nodos centrales que debemos recorrer para acceder de una LAN que se encuentra en un extremo a otra en el opuesto. Esto es similar a los caminos de routers que podemos encontrar en un camino entre dos redes en Internet. Además, por una limitación de la implementación de Emulab, como ningún nodo real subyacente tiene más de cinco interfaces de red físicas, sólo podemos conectarle un máximo de 4 links (una de las interfaces es utilizada exclusivamente por la red de control de Emulab). Esto nos impide armar topologías en estrella donde un único nodo central conecte más de 4 LANs (tal como teníamos en la red pequeña). La esquematización de esta red, que llamamos **tenlan**, se encuentra en la figura 3.2. Para más detalles sobre la red mediana, referirse al apéndice D.2.

El tamaño de la red grande estuvo limitado por el acceso a los recursos de Emulab. Decidimos utilizar 64 nodos dispuestos en 20 LANs, cuya topología también fue construida a partir de extender la red mediana con caminos entre nodos de mayor longitud y nuevas LANs conectadas a los mismos. En esta red tenemos caminos cuya longitud en cantidad de saltos y en latencia son del orden de los que encontramos en links internacionales. La esquematización de esta red, que llamamos **biglans**, se encuentra en la figura 3.3. La intención de la figura no es mostrar todos los detalles de esta red, sino simplemente dar una idea de sus proporciones y distribución de los nodos. Para más detalles referirse al apéndice D.3, donde se encuentran las definiciones de cada nodo, junto con las latencias de los links y las composiciones de las LANs.

3.2 Vivaldi P2P

Los datos utilizados fueron adquiridos realizando un total de 54 corridas por tamaño de red con diferentes combinaciones de valores de parámetros y limitando la duración de cada una a aproximadamente 4000 segundos. Este límite de tiempo fue elegido considerando que nos interesa embeber Vivaldi para dirigir el tráfico de un protocolo P2P. Este es el tiempo que toma descargar directamente un archivo de 125MB de un servidor a 32KB/seg, valor que podemos establecer como una cota superior del tiempo que puede tomar la descarga de un archivo de este tamaño en una red P2P. En el trabajo de Lucángeli Obes [10], que fue tomado como punto de partida, se utiliza este tiempo en simulaciones de BitTorrent. Además, en la literatura se suelen utilizar archivos de prueba de entre 100or ejemplo en el trabajo de Huang *et al.* [37].

A continuación describimos los valores de parámetros utilizados en las corridas. **ND** indica con 1 si se utilizó la mejora de extensión del conjunto de vecinos (*Neighbor Decay*), 0 en caso contrario. **NDT** es el tiempo, expresado en segundos, entre cada actualización de coordenadas con *Neighbor Decay* habilitado. Elegimos dos valores para esta variable, $\mathbf{NDT} = 1$ para que el tiempo de actualización se corresponda con al de Vivaldi sin esta mejora, y $\mathbf{NDT} = 5$, que es el valor utilizado por otra implementación [10]. Para los casos con $\mathbf{ND} = 0$ corrimos con un único valor de **NDT** ya que este último no influía en el resultado.

- **ND**: 0 y 1.
- **NDT**: 1,0 y 5,0 segundos.
- c_c: 0,005; 0,01; 0,1; 0,25; 0,5; 1,0.

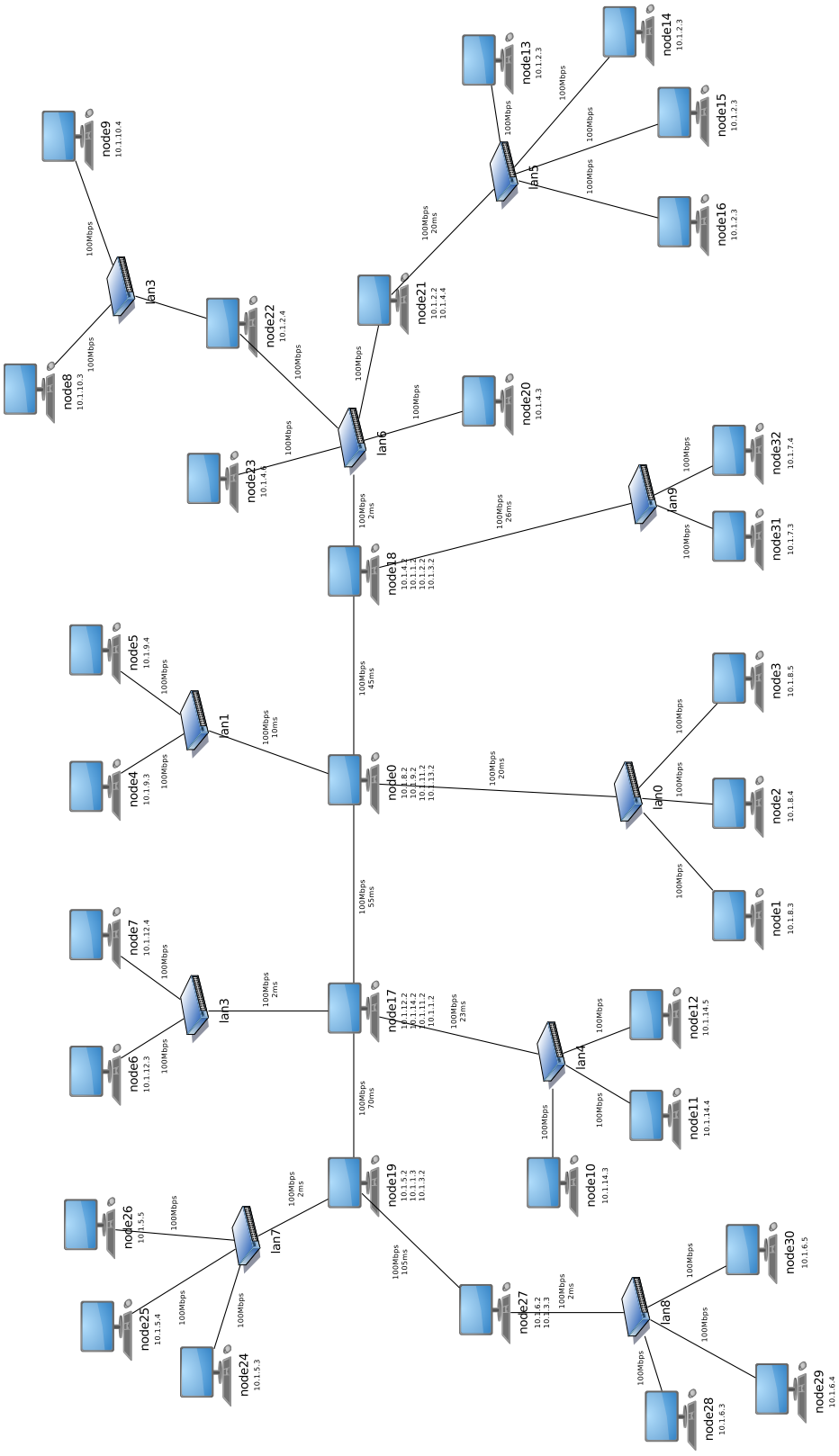


Figura 3.2: Red de prueba mediana de 10 LANs interconectadas (tenlan).

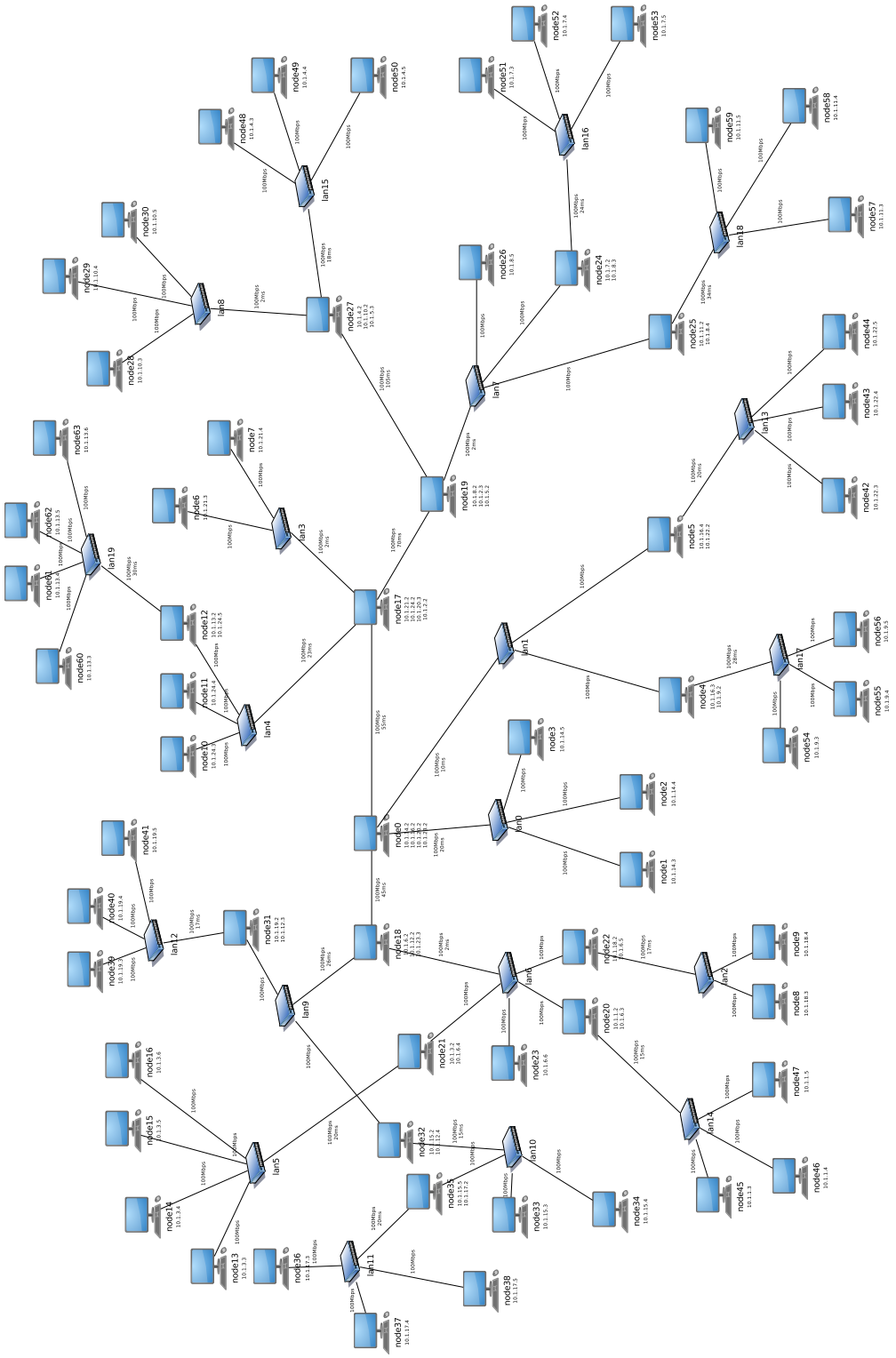


Figura 3.3: Red de prueba grande de 20 LANs interconectadas (biglans).

- c_e: 0,1; 0,25; 1,0.

Nuestro trabajo consistió en realizar dos análisis diferentes del comportamiento del algoritmo de Vivaldi para la combinación de los diferentes valores de las constantes c_c y c_e.

Empezaremos analizando la influencia de estos parámetros en el algoritmo de Vivaldi. Para esto, recordamos la definición del algoritmo adaptativo de Vivaldi (ver algoritmo 3, pág. 18) para la posición y el error locales (x_i y e_i respectivamente). La misma dice que:

$$\begin{aligned} w &= \frac{e_i}{e_i + e_j} \\ e_s &= \frac{|\|x_i - x_j\| - rtt|}{rtt} \\ e_i &= e_s \times c_e \times w + e_i \times (1 - c_e \times w) \\ x_i &= x_i + c_c \times w \times (rtt - \|x_i - x_j\|) \times u(x_i - x_j) \end{aligned}$$

En el caso de c_c podemos ver que influye en qué porcentaje del desplazamiento calculado para la iteración actual del algoritmo se moverá efectivamente el punto x_i. Un valor pequeño de c_c nos indica que el algoritmo es más conservador: se mueve un poco en la dirección calculada, aislando fuertemente las variaciones. Si la misma cambiase, nos estaría indicando que previamente estaba moviéndose en una dirección incorrecta, y en este caso habría evitado saltar hacia una posición con mayor error. Si el valor de c_c crece, sucede lo contrario, pero esto podría llevar a oscilaciones.

Por otra parte, c_e varía qué porcentaje del error calculado es atribuible a la coordenada local (x_i) y qué porcentaje a la coordenada remota (x_j). El error relativo local está dado por c_e, mientras que el remoto está dado por $1 - c_e$ ($0 \leq c_e \leq 1$).

En el trabajo de Elser et al. [35] se comparan estos aspectos, recomendando utilizar los valores $c_c = 0{,}005$ y $c_e = 0{,}1$, que son los que a su criterio mejor garantizan precisión y estabilidad. El trabajo de Ledlie et al. [8], por otro lado, utiliza valores mucho más elevados: $c_c = c_e = 0{,}25$.

A continuación analizaremos los resultados obtenidos en cada uno de nuestros análisis para los distintos tamaños de redes utilizados. En todos los casos, si no se especifica, mostramos únicamente los resultados para $\mathbf{ND} = 0$.

3.2.1. Fourlan

En la figura 3.4 podemos observar las curvas correspondientes a la evolución temporal de los radios mínimo, medio y máximo de los clusters generados por nuestra clusterización (ver la definición en la sección 2.4.2, pág. 33, párrafo **Evolución de los tamaños de los clusters**), para seis valores diferentes de la constante c_c y un valor fijo de $c_e = 0{,}25$.

La red pequeña no presenta mayores variaciones entre los distintos valores de c_c. Podemos ver un crecimiento lento pero progresivo del radio máximo para $c_c = 0{,}005$ y $c_c = 0{,}01$, mientras que para $c_c = 0{,}1$, $c_c = 0{,}25$ y $c_c = 0{,}5$ observamos un crecimiento del máximo en la primera parte de la corrida (de mayor duración cuanto menor el c_c), seguido de una caída y posterior estabilidad en valores muy bajos (exceptuando algunas irregularidades dadas por unos pocos picos pequeños, que atribuimos al error intrínseco de las mediciones). Para $c_c = 1$ apreciamos un comportamiento estable pero con una varianza más grande, y ya no se puede observar la curva creciente del comienzo que teníamos con valores menores.

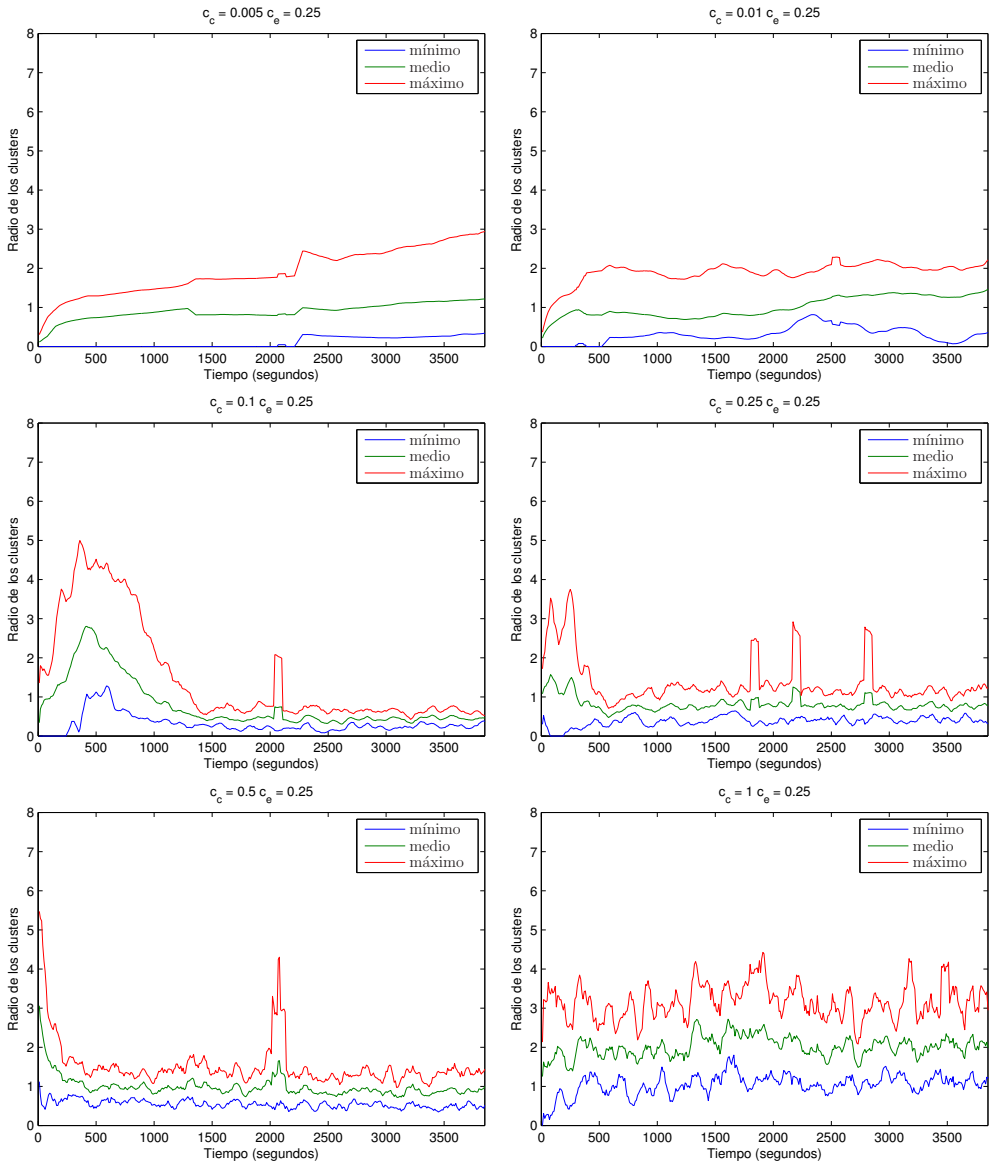


Figura 3.4: Evolución de los radios mínimo, medio y máximo de los clusters para distintos valores de c_c, con $c_e = 0{,}25$ y **ND** $= 0$ (red **fourlan**)

Analizamos además cómo afecta el valor de c_e los radios de los clusters para un c_c fijo, y lo que observamos es que este parámetro afecta principalmente los valores de los máximos. Cuanto menor el c_e, más grande y prolongado es el pico inicial. Esto nos indicaría que el tiempo de estabilización de los tamaños de los clusters se reduce incrementando el c_e.

En la figura 3.5 mostramos los resultados correspondientes al análisis del error de la clusterización propuesta (ver la definición en la sección 2.4.2, pág. 33, párrafo **Correctitud de la clusterización**). Podemos observar que, si bien con ninguno de los valores mostrados tenemos un porcentaje de error elevado, el mismo decrece conforme incrementamos el valor de c_c. Agregamos, a modo de comparación, los valores propuestos por Elser ($c_c = 0{,}005$ y $c_e = 0{,}1$) a los utilizados en el análisis anterior.

En cuanto a la comparación con diferentes valores de c_e para un mismo c_c, observamos que, en general, cuanto mayor el c_e, más tiempo toma en reducirse el error.

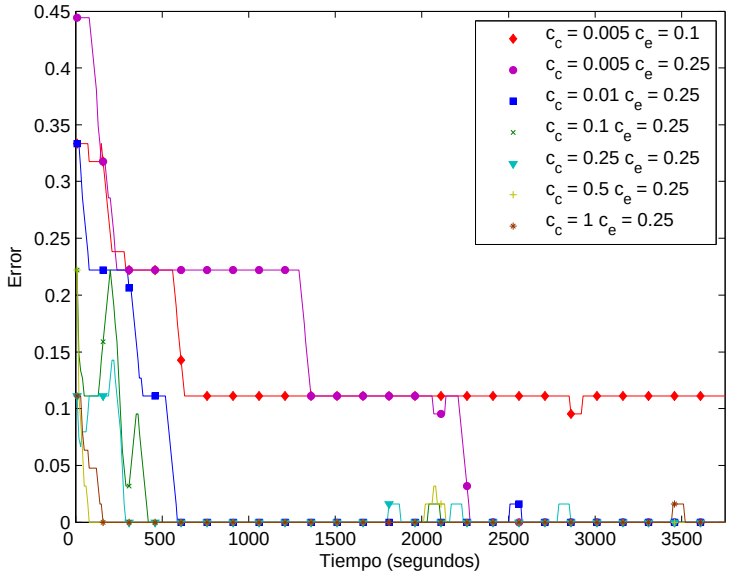


Figura 3.5: Cantidad de nodos incorrectamente clusterizados sobre el total en función del tiempo de corrida para distintos valores de c_c y c_e, con **ND** $= 0$ (red **fourlan**)

3.2.2. Tenlan

Veamos qué es lo que ocurre en la red mediana **tenlan**. Podemos encontrar en la figura 3.6 los gráficos correspondientes a la evolución de los radios de los clusters para los mismos valores de c_c y c_e que mostramos para la red pequeña. Modificamos la escala del eje X con respecto a la figura 3.4 para poder visualizar todo el espectro de valores.

En esta red encontramos el mismo patrón que en **fourlan** pero con radios mucho más grandes. Para $c_c = 0{,}005$ observamos en los máximos un crecimiento estable, que empieza a verse más errático hacia el final de la corrida, mientras que para $c_c = 0{,}01$ el comportamiento es muy similar, pero se mantiene "suave" durante menos tiempo. Para $c_c = 0{,}1$ y $c_c = 0{,}25$ hay un fuerte crecimiento inicial tanto en los valores máximos como promedio, que es luego acompañado por un descenso con posteriores oscilaciones. Es notable que si bien hay oscilaciones grandes en los valores máximos, los valores promedio se mantienen estables a lo largo del resto de la corrida. Finalmente, para $c_c = 0{,}5$ y $c_c = 1$ vemos lo mismo que en el caso anterior,

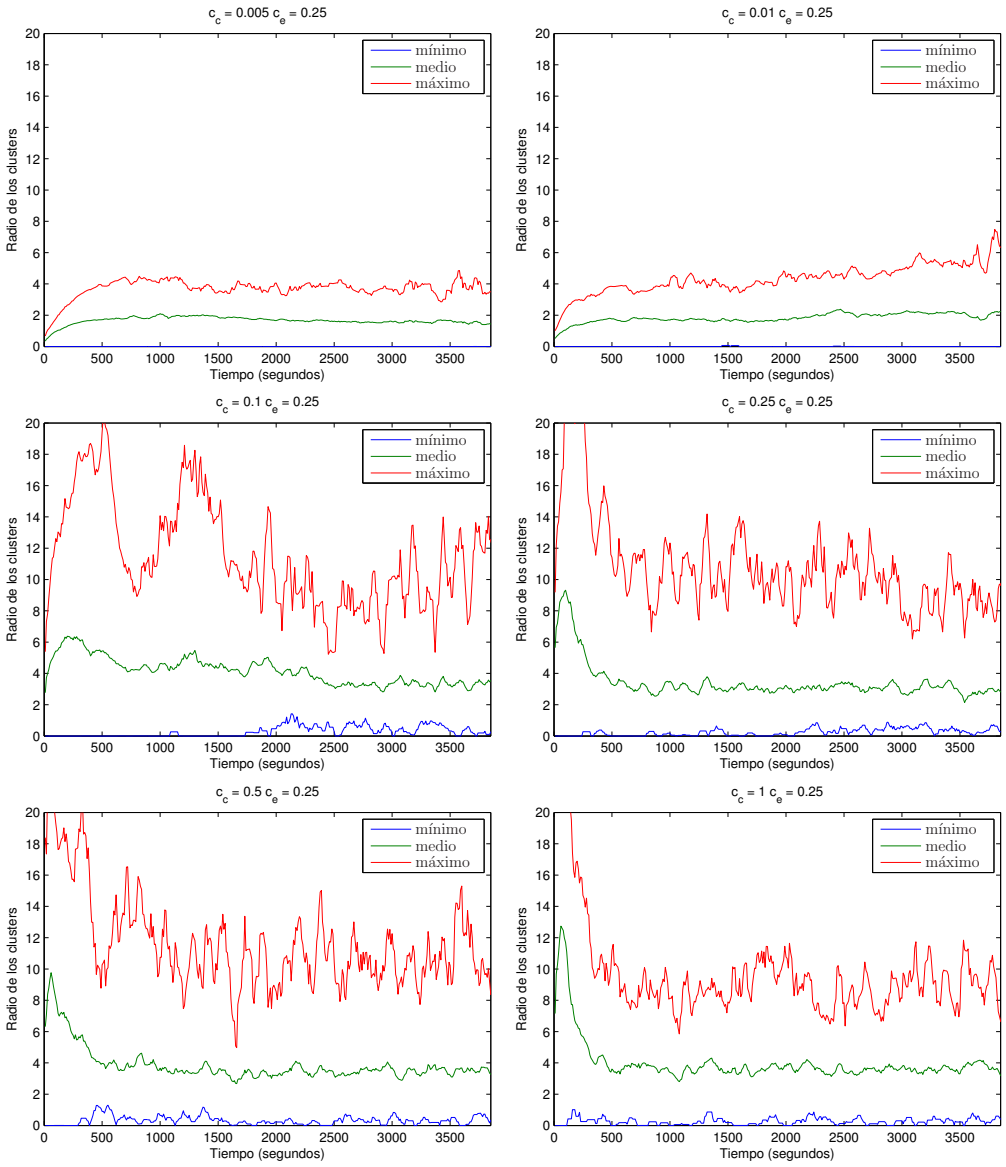


Figura 3.6: Evolución de los radios mínimo, medio y máximo de los clusters para distintos valores de c_c, con $c_e = 0{,}25$ y **ND** $= 0$ (red **tenlan**)

pero con los picos iniciales corridos mucho más cerca del comienzo de la corrida. El resultado para $c_c = 1$ es diferente al de **fourlan**, donde solamente podíamos observar oscilaciones, sin picos iniciales. Además, con el valor más elevado las oscilaciones de los radios promedio son más fuertes que en el resto de las corridas, pero aún así tienen una varianza inferior a 2. Los valores mínimos de los radios se mantienen entre 0 y 1, con pequeñas oscilaciones.

En la figura 3.7 observamos el gráfico del error de la clusterización correspondiente a esta red. Podemos observar un patrón muy similar al que encontramos en **fourlan**, con la diferencia de que los valores absolutos de error son mayores en todas las corridas. El motivo por el que creemos que no se puede alcanzar un valor de error de 0 en esta red tal como ocurría en **fourlan** tiene que ver con la estructura misma de la red. Aquí tenemos LANs que se encuentran cerca (con un link de menos de 20 ms de distancia entre sí), que desde el punto de vista de un nodo lejano podrían parecer juntas con sólo agregar un pequeño error en las mediciones. Aún así, estos errores no son significativos ya que no representan más del 10 % del tamaño de la red (alrededor de 3 nodos) en la mayor parte del tiempo de corrida.

En esta red es mucho más notoria la diferencia entre el comportamiento en los valores más bajos de c_c y los más elevados: con $c_c = 0{,}005$ no logramos conseguir un error menor al 20 %.

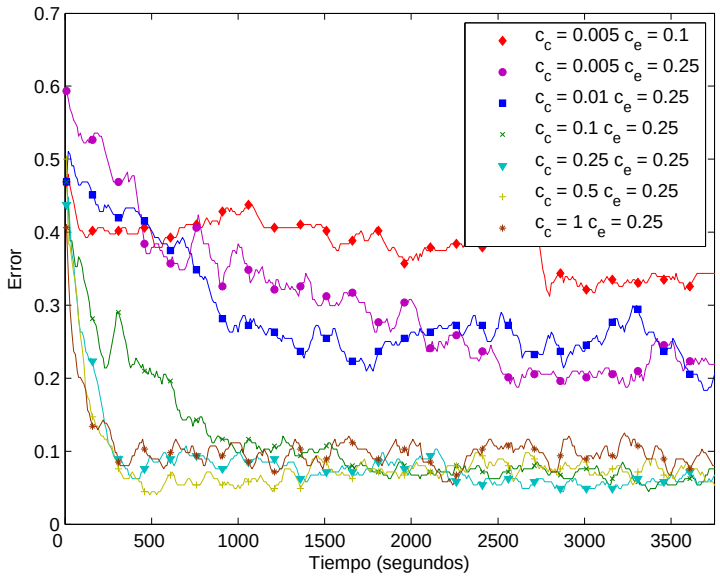


Figura 3.7: Cantidad de nodos incorrectamente clusterizados sobre el total en función del tiempo de corrida para distintos valores de c_c y c_e, con **ND** $= 0$ (red **tenlan**)

Finalmente debemos notar que el error para $c_c = 1$ es superior al obtenido para $c_c = 0{,}5$, y además se mantiene virtualmente constante durante casi la totalidad de la corrida. Esto nos indicaría que, si bien los valores más pequeños de c_c, como pudimos ver en **fourlan** y en este caso, hacen que sea mucho más lenta la estabilización del algoritmo, las coordenadas calculadas con $c_c = 1$ son muy fácilmente afectadas por errores de medición y no logran converger a valores de error tan bajos como con $c_c = 0{,}5$. No creemos que sea significativa la pequeña diferencia de tiempo extra que toma a $c_c = 0{,}5$ alcanzar los valores de error de $c_c = 1$, en contraposición con las ganancias obtenidas globalmente.

3.2.3. Biglans

Veamos ahora qué es lo que ocurre en la red **biglans**. Los gráficos correspondientes se encuentran en la figura 3.8. En esta última figura observamos patrones idénticos a los obtenidos para la red **tenlan**. La única diferencia que encontramos es que los picos iniciales en los radios máximos son más elevados que en la red mediana, y los mínimos tienen oscilaciones más pequeñas, con valores más proximos a 0.

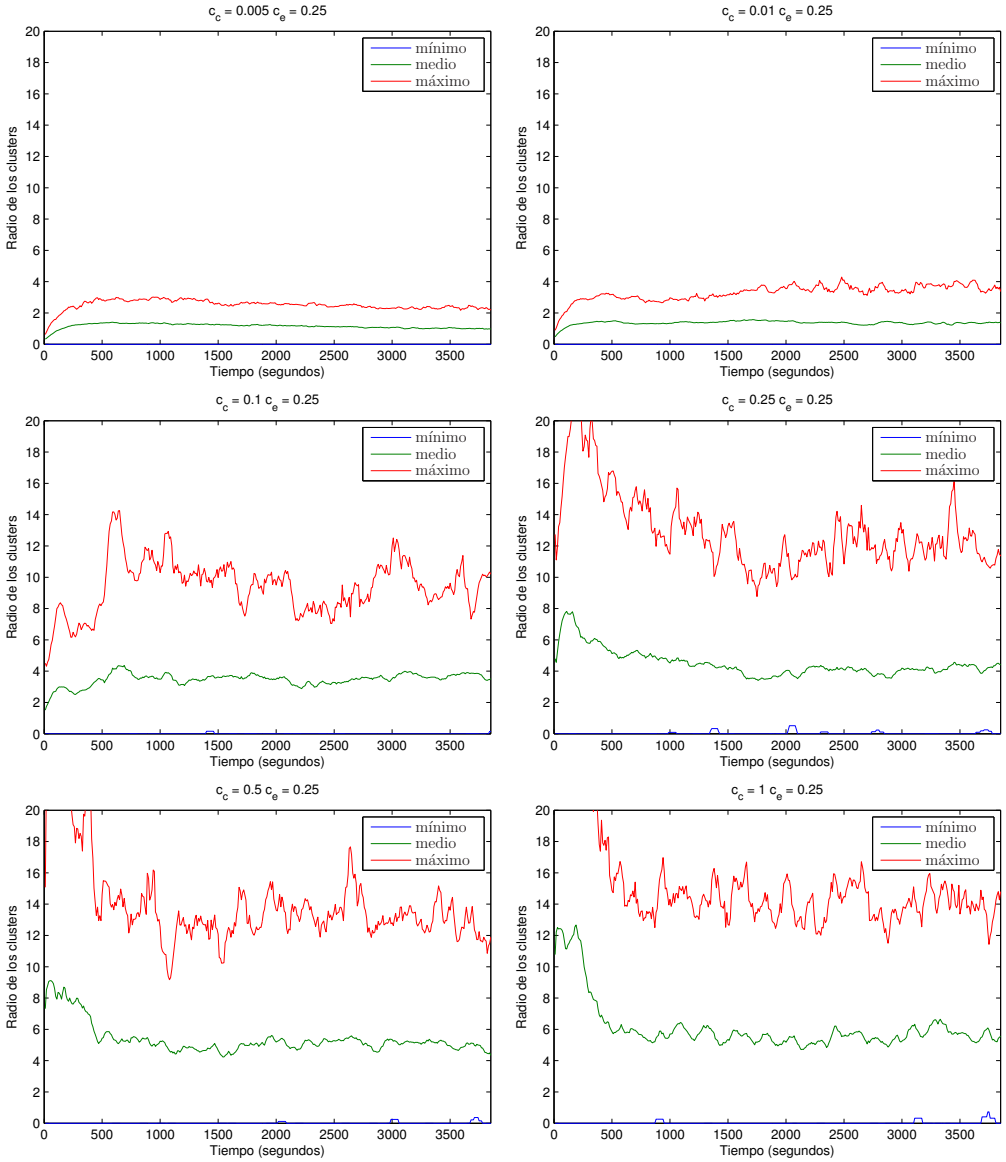


Figura 3.8: Evolución de los radios mínimo, medio y máximo de los clusters para distintos valores de c_c, con $c_e = 0{,}25$ y **ND** $= 0$ (red **biglans**)

En cuanto al análisis del error de la clusterización para esta red, podemos observar en la figura 3.9 un patrón sumamente similar al que encontramos en la red mediana. Las curvas son menos suaves que en los casos anteriores, pero se comportan de una manera muy similar. En este caso es incluso más notable que en la red mediana la escasa correctitud de las corridas

con los valores más pequeños de c_c. Para $c_c = 0{,}005$ no obtenemos errores menores al 35 % en ningún momento de la corrida. Esto nos indica que a medida que incrementamos el tamaño de la red, las corridas con los valores más bajos de c_c comienzan a mostrar mayor dificultad en obtener buenas soluciones en un tiempo razonable.

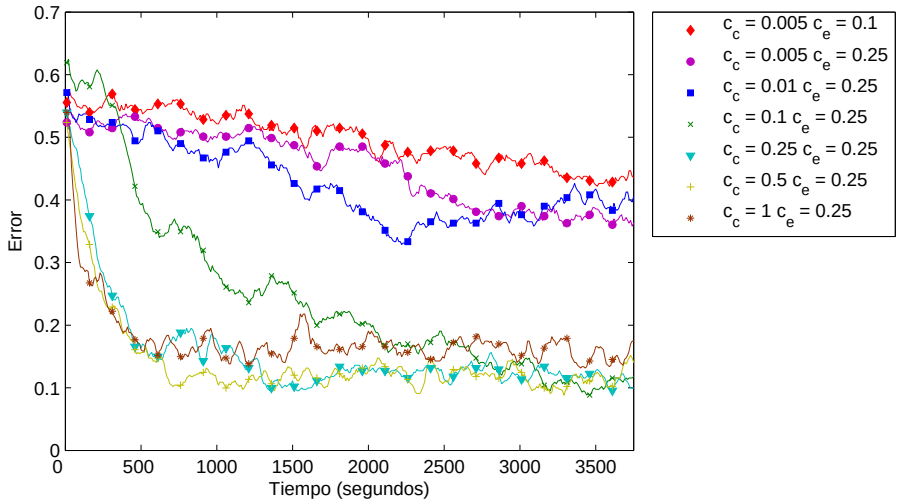


Figura 3.9: Cantidad de nodos incorrectamente clusterizados sobre el total en función del tiempo de corrida para distintos valores de c_c y c_e, con **ND** $= 0$ (red **biglans**)

Por otro lado, aquí también observamos una mejora entre $c_c = 0{,}5$ y $c_c = 1$, que es incluso más marcada que en el caso de **tenlan**. Despeja toda duda con respecto a la conveniencia de uno sobre el otro, ya que además demora menos tiempo en superar $c_c = 0{,}5$ a $c_c = 1$.

3.2.4. Conclusiones

En una primera lectura de nuestros resultados podría parecer que existe una contradicción entre los mismos y los obtenidos por Elser et al.

El punto central de su análisis es la estabilidad de las coordenadas a lo largo del tiempo. Nuestras corridas son de poco más de una hora de duración, contra los días enteros de trazas de latencia que se utilizaron en su trabajo [35]. Si observamos las curvas correspondientes a los errores para sus valores ($c_c = 0{,}005$ y $c_e = 0{,}1$), podemos apreciar que existe una leve tendencia decreciente de los mismos, por lo que no es incorrecto suponer que extendiendo nuestras corridas a períodos de tiempo mucho mayores podríamos obtener valores de error bajos. Sin embargo, como ya mencionamos anteriormente, no permite montar Vivaldi eficientemente sobre un sistema de transferencia de archivos como BitTorrent, ya que estamos tomando decisiones para dirigir el tráfico constantemente durante todo el proceso de estabilización de las coordenadas. Si la estabilización toma del orden de varias horas, estamos desperdiciando potencialmente una gran cantidad de tráfico que podría ser redirigido hacia nodos más cercanos.

Tomando en cuenta esto, desaconsejamos la utilización de valores muy bajos de c_c en Vivaldi si la intención de esas coordenadas es guiar el tráfico de un sistema distribuido como puede ser una red de distribución de archivos P2P. Entre los valores más elevados, observamos que las corridas con $c_c = 0{,}5$ se comportaron mejor en cuanto a velocidad de estabilización y correctitud en las redes mediana y grande que con $c_c = 1$.

Finalmente, hacemos una mención a los resultados obtenidos en las corridas con **ND** = 1. En todas las corridas realizadas obtuvimos resultados similares a los de **ND** = 0, con la diferencia de que los tiempos de estabilización observados fueron mayores en todos los casos. Entendemos por tiempo de estabilización el tiempo que toma el error inicial en reducirse hasta alcanzar, en promedio, los valores que conserva hacia el final de la corrida. Este tiempo fue observado comparando los gráficos obtenidos de los resultados para **ND** = 0, **ND** = 1 con **NDT** = 1, y **ND** = 1 con **NDT** = 5, para los mismos valores de c_c y c_e, y analizando la forma de las curvas resultantes. En reglas generales, en cuanto al análisis del error de la clusterización el patrón observado fue el siguiente:

$$te_{ND=0} < te_{ND=1,NDT=1} < te_{ND=1,NDT=5}$$

donde te es el tiempo de estabilización observado en cada una de las corridas con los mismos valores de c_c y c_e. Estos resultados son esperables, ya que neighbor decay es una mejora pensada para aprovechar las coordenadas de los vecinos recientes en todas las iteraciones del algoritmo, de forma de poder mitigar la falta de actualizaciones periódicas por parte de todos los peers, caso que no sucede en Vivaldi P2P. En cuanto a los radios de los clusters, el comportamiento observado también es similar al caso con **ND** = 0, con la diferencia de que los radios medio y máximo son mayores que en el primer caso.

3.3 BitTorrent con Vivaldi

Realizamos un análisis preliminar de Vivaldi en nuestra implementación de la versión sobre BitTorrent. Utilizamos las conclusiones parciales de las corridas de la sección anterior y reproducimos el mismo análisis.

En este caso, tomamos datos de 36 corridas en la red **fourlan** para diferentes valores de los parámetros de Vivaldi más una corrida extra sin las extensiones habilitadas. Esta última se corresponde a los resultados que obtendríamos en la ejecución del cliente de BitTorrent original, con la única diferencia de que se encuentra instrumentado el logging de parámetros. Nuestra versión de BitTornado sin las extensiones habilitadas no calcula Vivaldi, pero genera para cada cliente un valor de coordenada x_i aleatorio tal que $\|x_i\| = 1$ que es enviado al servidor de logging de la misma manera en la que lo hacen los clientes que sí calculan Vivaldi. Claramente, esta coordenada se mantiene constante a lo largo de la corrida, pero cumplirá el rol de servir de punto de comparación entre la calidad de los resultados obtenidos con y sin Vivaldi en cuanto a las clusterizaciones obtenidas.

Con respecto a las corridas que sí calculan Vivaldi, utilizamos la misma combinación de valores de parámetros que en el caso anterior, pero eliminamos de los valores elegidos para c_c los más pequeños (0,005 y 0,01) por las razones que ya fueron expuestas.

Los parámetros de BitTorrent nos ofrecen toda otra gama de posibles variantes a explorar en nuestro análisis. En este trabajo decidimos focalizarnos en la influencia de Vivaldi únicamente, por lo que utilizamos un conjunto de parámetros fijo para todas las corridas con BitTorrent. Esta fue la configuración utilizada:

- **Archivo transferido**: en nuestras pruebas utilizamos un archivo correspondiente a la imagen ISO[1] del instalador por red ("netinst") del sistema operativo Debian GNU/Linux

[1] Copia fiel del contenido de un CD-ROM. Ver ISO 9660:1988.

5.0.6 "lenny" para la arquitectura i386. Este archivo tiene un tamaño de 157.630.464 bytes.

- **Ancho de banda de subida**: limitamos el ancho de banda de subida a 32 KB/seg. Este valor se encuentra en el rango de lo que se puede contratar en conexiones de banda ancha residencial en la actualidad.
- **Ancho de banda de bajada**: el ancho de banda de bajada no fue limitado, por lo que se utiliza el máximo disponible.
- **Distribución de seeders**: el caso que analizamos es el que se presenta cuando un único individuo desea compartir un archivo a un conjunto de personas. De esta manera, inicialmente el único poseedor es el nodo que inicia el torrent (en nuestro caso, el nodo central `node0`), mientras que ninguno de los otros nodos participantes posee partes del mismo.
- **Distribución de los tiempos de asociación y desasociación de los peers**: tomamos el caso en el que todos los peers se conectan simultáneamente al torrent, y se mantienen conectados hasta el momento en el que finaliza la vida del mismo.

La duración de cada corrida fue de aproximadamente 6000 segundos. Este tiempo es el suficiente como para que podamos analizar la transferencia completa de nuestro archivo de prueba en todos los nodos de la red a la velocidad de subida configurada.

3.3.1. Radios de los clusters

En la figura 3.10 podemos observar los radios mínimo, medio y máximo de la clusterización de las coordenadas generadas por Vivaldi sobre BitTorrent para la red **fourlan**. Aquí podemos ver resultados sumamente diferentes a los de la figura 3.4. Con ninguna de las combinaciones de valores de los parámetros analizados obtenemos clusters muy estables, y además los radios son órdenes de magnitud más grandes que sin BitTorrent. En esta figura observamos un efecto que nos llamó mucho la atención: luego de la primera mitad de la corrida, los valores de los radios de los clusters siguen la forma de una exponencial decreciente hasta llegar a valores cercanos a 0. En los gráficos no se podía apreciar correctamente por la escala utilizada, por lo que agregamos a la derecha de cada uno versiones de los mismos con el eje X en escala logarítimica. Allí sí puede verse que los valores decrecen de esa manera hasta detenerse en radios del orden de 10^{-4} para $c_c = 0{,}1$ y 10^{-1} para $c_c = 0{,}25$ cuando finaliza la transmisión del archivo. El efecto de decrecimiento lo podemos atribuir al uso de la mejora gravity drift, que atrae en cada actualización las coordenadas hacia el origen con una fuerza equivalente una fracción constante del módulo de la distancia de la coordenada al origen, pero no logramos obtener una razón conclusiva acerca de por qué ocurre este fenómeno tan marcadamente en las corridas con $c_c = 0{,}1$ y $c_c = 0{,}25$, mientras que con los valores más grandes no sucede lo mismo.

En la figura 3.11 podemos ver lo que ocurre para los valores más grandes de c_c analizados.

Pasamos ahora a la figura 3.12, donde podemos observar el mismo gráfico de los radios de los clusters, con los mismos valores de los parámetros c_c y c_e, pero con $\mathbf{ND} = 1$ y $\mathbf{NDT} = 5$. En este caso, observamos diferencias con respecto a no utilizarla, ya que las curvas resultantes son mucho más suaves. Esto es esperable ya que estamos utilizando los datos de todos los peers en cada actualización. El efecto de la exponencial decreciente que mencionamos para la

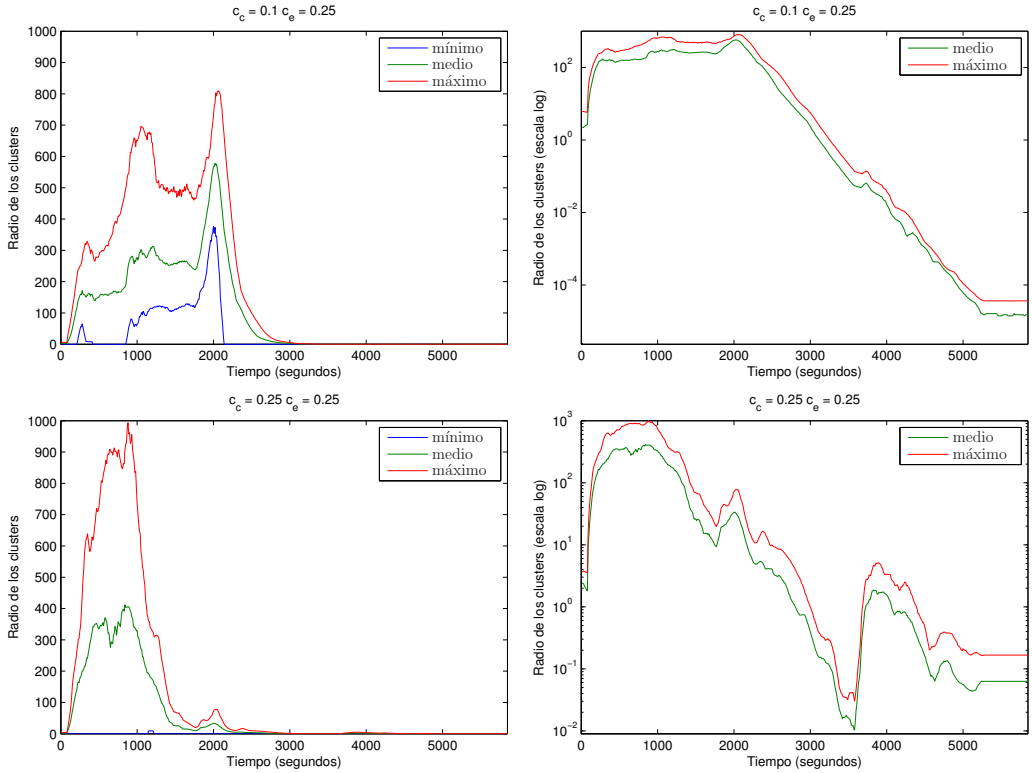


Figura 3.10: Evolución de los radios mínimo, medio y máximo de los clusters obtenidos por Vivaldi sobre BitTorrent utilizando los parámetros $c_c = 0{,}1$ y $c_c = 0{,}25$ con $c_e = 0{,}25$ en escalas lineal y logarítmica y **ND** $= 0$ (red **fourlan**)

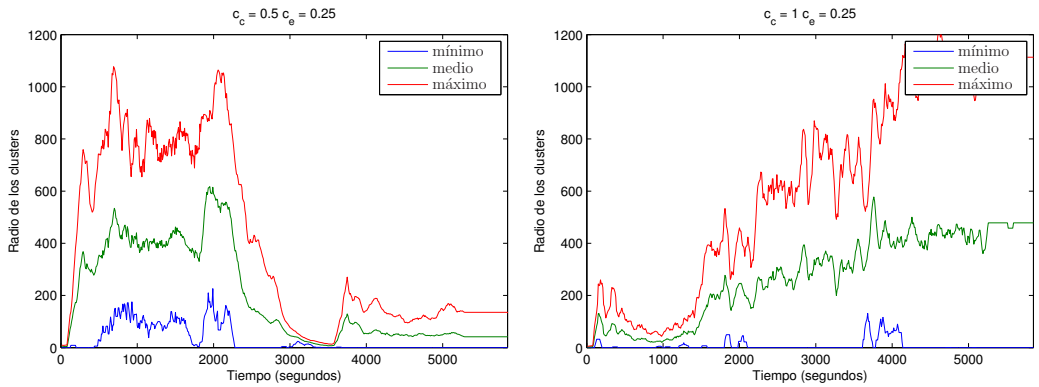


Figura 3.11: Evolución de los radios mínimo, medio y máximo de los clusters obtenidos por Vivaldi sobre BitTorrent, para los parámetros $c_c = 0{,}5$ y $c_c = 1$, con $c_e = 0{,}25$ y **ND** $= 0$ (red **fourlan**)

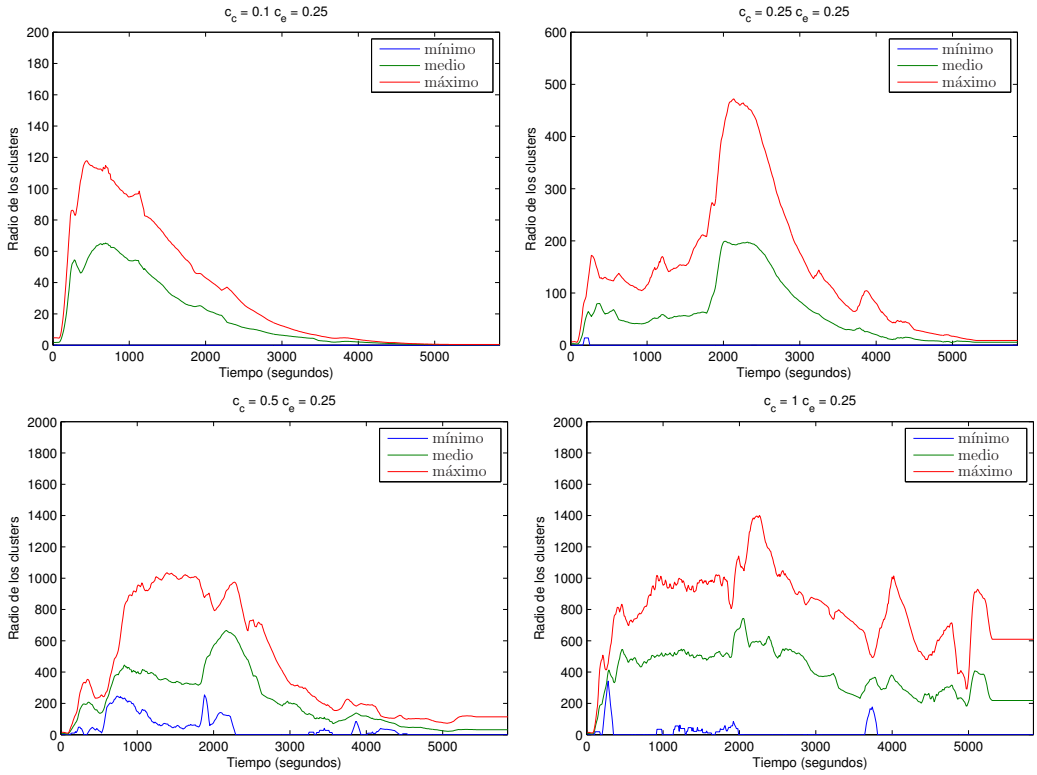


Figura 3.12: Evolución de los radios mínimo, medio y máximo de los clusters obtenidos por Vivaldi sobre BitTorrent, para distintos valores de c_c, con $c_e = 0{,}25$, **ND** $= 1$ y **NDT** $= 5$ (red **fourlan**)

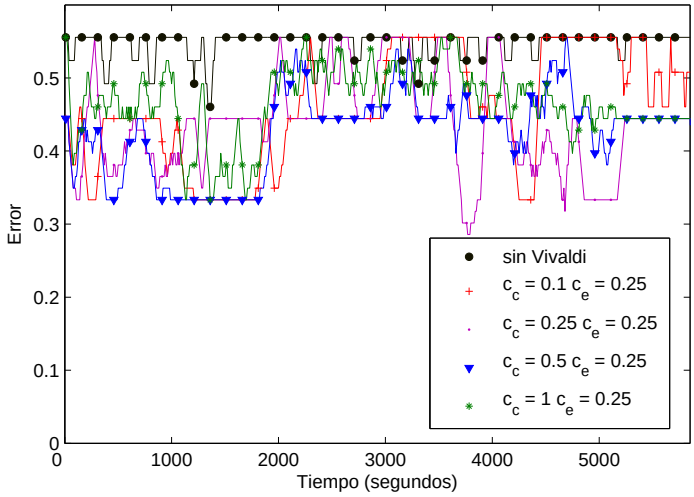


Figura 3.13: Cantidad de nodos incorrectamente clusterizados sobre el total en función del tiempo de corrida, comparando el uso de coordenadas aleatorias de norma 1 con el uso de Vivaldi sobre BitTorrent para distintos valores de c_c y $c_e = 0{,}25$, con **ND** $= 0$ (red **fourlan**)

segunda mitad de las corridas sin neighbor decay sigue presente en $c_c = 0{,}1$ y $c_c = 0{,}25$, pero es mucho menos pronunciado que en el caso anterior, por lo que no incluimos versiones de los mismos gráficos en escala logarítimica. Las curvas para $c_c = 0{,}5$ son muy parecidas entre utilizar neighbor decay y no utilizarlo, pero con $c_c = 1$ se comporta muy diferente, acotando un poco los valores de los radios máximos.

Este análisis no arroja resultados conclusivos, pero nos indica que las formas de los clusters son mucho más inestables que en el caso de Vivaldi sin BitTorrent, lo cual se confirma con el análisis de la calidad de la clusterización realizado a continuación.

3.3.2. Correctitud de la clusterización

En la figura 3.13 se encuentra el gráfico correspondiente al error de la clusterización en Vivaldi sobre BitTorrent en la red **fourlan**, para diferentes valores de c_c con $c_e = 0{,}25$. Agregamos además, a modo de comparación, el error de la clusterización correspondiente a no utilizar Vivaldi para actualizar los valores de las coordenadas. El cliente instrumentado inicializa un valor de coordenada sobre el borde de la esfera de radio unitario con centro en el origen, y son estas coordenadas las que utilizamos para calcular el error de la clusterización sin utilizar Vivaldi. En la figura podemos apreciar que no utilizar Vivaldi nos dá un error casi constante levemente superior al 50 %. Sin embargo, tal y como esperábamos, los resultados de Vivaldi sobre BitTorrent no son muy alentadores, ya que si bien representan una mejora con respecto a utilizar valores aleatorios de coordenadas, en el mejor caso analizado, $c_c = 0{,}5$, el error no se encuentra nunca por debajo del 30 %, con un valor promedio del 38 %. Por otro lado, las curvas de error ya no tienen la forma exponencial decreciente presente en el caso sin BitTorrent. Estos resultados son consistentes con los obtenidos por Lucángeli Obes [10], donde el análisis del tráfico intra-ISP ganado daba como resultado unas mejoras leves.

Finalmente, en la figura 3.14 analizamos el error de clusterización obtenida utilizando

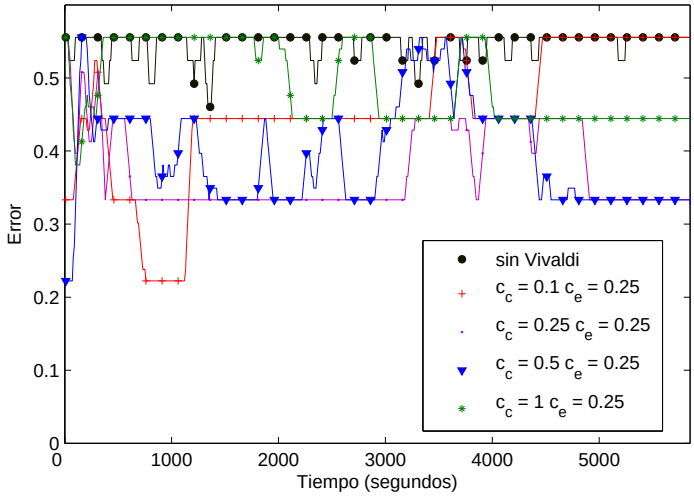


Figura 3.14: Cantidad de nodos incorrectamente clusterizados sobre el total en función del tiempo de corrida, comparando el uso de coordenadas aleatorias de norma 1 con el uso de Vivaldi sobre BitTorrent para distintos valores de c_c y $c_e = 0{,}25$, con **ND** $= 1$ y **NDT** $= 5$ (red **fourlan**)

neighbor decay. En este caso, observamos que los resultados para $c_c = 1$ son considerablemente peores que si no utilizamos la mejora, mientras que con $c_c = 0{,}5$ son muy similares al caso sin neighbor decay, con la diferencia de que la curva es más suave. En este último caso no observamos ninguna diferencia en los valores promedio de error entre utilizar neighbor decay y no utilizarlo (en ambos casos se encuentran en el 38 %). Por otro lado, sí observamos pequeñas mejoras en los casos $c_c = 0{,}25$ y $c_c = 0{,}1$, pero nuevamente no son significativas, por lo que no queda clara la ganancia de utilizar neighbor decay más allá de la mejora en la estabilidad de los resultados. También comparamos los resultados obtenidos con los dos valores de **NDT** analizados, donde tampoco obtuvimos diferencias significativas. Con **NDT** $= 5$ observamos valores mínimos de error menores a los obtenidos con **NDT** $= 1$, pero los valores promedio no se alteran significativamente.

3.3.3. Conclusiones

Las diferencias encontradas entre utilizar Vivaldi sin BitTorrent y hacerlo funcionar sobre el patrón de tráfico de este último son realmente muy importantes. En la red **fourlan** observamos una performance excelente del algoritmo de Vivaldi con los valores propuestos para los parámetros c_c y c_e. Esto cambia con los resultados obtenidos sobre BitTorrent, si bien obtenemos una mejora con respecto a utilizar un criterio random. Un breve análisis de los mismos resultados sobre **tenlan** nos muestra que el patrón se mantiene, y es incluso agravado. Por este motivo creemos que no es razonable continuar con las pruebas orientadas en esta dirección, y que, en cambio, deberíamos analizar cómo se degradan los resultados por la utilización de Vivaldi sobre el tráfico generado por BitTorrent.

Los gráficos con los resultados para **tenlan**, así como también los correspondientes al resto de las combinaciones de valores que no aparecen en la memoria de la tesis, fueron incluidos a

modo de referencia en el apéndice E.

4

Conclusiones y trabajo a futuro

En el transcurso de este trabajo construimos una implementación de Vivaldi que utilizamos en tres programas diferentes: un simulador, que toma valores constantes de latencia y simula la evolución temporal del sistema, un software P2P, que calcula los valores de las coordenadas utilizando mediciones reales de latencia obtenidas a través de la red, y finalmente una modificación a un cliente de BitTorrent que calcula las coordenadas generadas por Vivaldi y las utiliza para dirigir el tráfico de la red utilizando el criterio de *upload appeal*. Todas las implementaciones pueden ser utilizadas en redes reales sin modificaciones. Además del algoritmo de Vivaldi incorporamos algunas mejoras analizadas en la bibliografía: filtro de mediana para las latencias, gravity drift y neighor decay.

Luego analizamos la validez de Vivaldi como mecanismo para modelar latencias. Propusimos una metodología para abordar este problema que consistió en analizar el comportamiento del sistema en redes reales de distinto tamaño con una topología y latencias fijas, sobre las que realizamos varias corridas de la misma duración utilizando distintas combinaciones de los parámetros de Vivaldi. Con estos datos comparamos dos clusterizaciones, una obtenida a partir de mediciones explícitas de latencia, y otra a partir de las coordenadas calculadas por Vivaldi. Comparamos la calidad de los resultados para diferentes valores de los parámetros de Vivaldi y la influencia de neighbor decay en los mismos.

Obtuvimos muy buenos resultados para nuestro caso de uso de Vivaldi. Ajustando los parámetros de modo que el algoritmo no sea excesivamente conservador logramos obtener rápidamente estimaciones de latencia sumamente precisas en los 3 tamaños de red analizados, con errores que rondan apenas el 10 %. Por otro lado, utilizar neighbor decay no ofrece ninguna ventaja, ya que en general los tiempos observados de estabilización con los valores de los parámetros analizados fueron mayores que no utilizando esta mejora.

Sin embargo, los resultados obtenidos una vez que montamos Vivaldi sobre el tráfico de BitTorrent no son tan optimistas. Incluso en la red pequeña, donde Vivaldi tenía una performance excelente con errores del 0 % en prácticamente todas las corridas, no logramos errores menores al 40 %, apenas un 10 % menos que estimando aleatoriamente. Esta diferencia tan notable la atribuimos al patrón de tráfico de BitTorrent. Se puede observar la correlación en el hecho de que logramos mejorar levemente los resultados incorporando el uso de neighbor decay, cuya principal motivación es incorporar datos de todos los nodos en la actualización de las coordendas, para suplir la falta de actualizaciones regulares, como sucede al montar Vivaldi sobre BitTorrent. Aún así, estas pequeñas mejoras pueden significar un ahorro significativo para los ISPs, por lo que si bien sería deseable mejorar estos resultados, son moderadamente satisfactorios.

Los resultados de las corridas de Vivaldi sobre BitTorrent ameritan un análisis más exhaustivo de la influencia del patrón de tráfico de BitTorrent en la calidad de las coordenadas

calculadas por Vivaldi. Como primer paso es necesario analizar el comportamiento de la versión P2P de Vivaldi con patrones de tráfico diferentes sin la influencia de BitTorrent. Para esto proponemos modificar Vivaldi para que en lugar de conectarse al resto de los peers aleatoriamente lo haga sobre un subconjunto aleatorio de los mismos, tal como hace BitTorrent. Luego, un análisis separado de cada uno de los algoritmos que se utilizan en BitTorrent para dirigir el tráfico nos permitiría analizar qué influencia tiene cada uno en degradar la calidad de las coordenadas de Vivaldi y tal vez encontrar alguna mejora diferente a neighbor decay que funcione más eficientemente sobre el patrón de tráfico de BitTorrent. Creemos que un análisis de Vivaldi simulando un patrón de tráfico típico de BitTorrent pero sin estar montado sobre este último nos puede dar una respuesta conclusiva acerca de la factibilidad de calcular coordenadas de Vivaldi precisas con patrones de red que no sean los encontrados en la implementación original del algoritmo. Nuestra impresión es que es una mejor idea computar Vivaldi independientemente y exportar esas coordenadas a la aplicación que necesite realizar estimaciones de latencia.

Por otro lado, como mencionamos en la introducción, existen otros sistemas de coordenadas basados en Vivaldi que incorporan jerarquías, y que surgieron como modificaciones que hacen más robusto y preciso al algoritmo original. El mismo análisis expuesto en nuestro trabajo podría realizarse sobre las variantes de Vivaldi para descubrir qué tanto mejoran los resultados en nuestro caso problemático (si los mejoran). Esto implica hacer un análisis de factibilidad de la aplicación de alguna de las variantes sobre BitTorrent, y luego construir implementaciones de los mismos.

Finalmente, creemos que un siguiente paso una vez obtenidos resultados satisfactorios en las pruebas sobre topologías de red fijas debería ser la evaluación de la misma implementación sobre de Internet (por ejemplo, utilizando clientes en conexiones residenciales) como plataforma de prueba en lugar de un entorno controlado. Esta es una tarea sumamente compleja, ya que en Internet hay una multitud de factores que no se pueden controlar que podrían afectar el resultado de los experimentos, por lo que sería necesario en ese caso realizar un gran número de muestreos y utilizar herramientas de análisis estadístico, de modo de poder establecer tendencias y obtener conclusiones.

El uso de sistemas de coordenadas sintéticos para agregar una noción de vecindad al tráfico de red en sistemas distribuidos es aún una idea muy joven, pero se han obtenido resultados razonables en varios de los enfoques analizados que indicarían que se puede sacar provecho de su uso para distribuir más eficientemente el siempre creciente volumen de tráfico en Internet.

Lista de acrónimos

AS Autonomous System

BGP Border Gateway Protocol

CDN Content Distribution Network

DHT Distributed Hash Table

DNS Domain Name Service

FQDN Fully Qualified Domain Name

GNP Global Network Positioning

HTTP HyperText Transfer Protcol

ICMP Internet Control Message Protocol

IP Internet Protocol

IPv4 Internet Protocol versión 4

ISP Internet Service Provider

LAN Local Area Network

LANC Locality-aware Network Coding

NFS Network Filesystem

NTP Network Time Protocol

P2P Peer-to-Peer

P4P Provider Portal for P2P Applications

RFC Request For Comments

RON Resilient Overlay Networks

RTT Round-Trip Time

SNMP Simple Network Management Protocol

TCP Transmission Control Protocol

TIV Triangle Inequality Violation

UDP User Datagram Protocol

VLAN Virtual Local Area Network

WAN Wide Area Network

Bibliografía

[1] H. Schulze and K. Mochalski. *ipoque Internet Study 2008/2009* (2009).

[2] A. Righi. *SystemImager Benchmark (BitTorrent transport)* (2008).
URL `http://wiki.systemimager.org/index.php/BitTorrent#Benchmark`

[3] J. Alderman. *Sonic Boom: Napster, MP3, and the New Pioneers of Music* (2001).

[4] B. Cohen. *Incentives Build Robustness in BitTorrent.* Technical report, bittorrent.org (2003).
URL `http://jmvidal.cse.sc.edu/library/cohen03a.pdf`

[5] B. Cohen. *The BitTorrent Protocol Specification* (2008).
URL `http://bittorrent.org/beps/bep_0003.html`

[6] M. Winther. *Tier 1 isps: What they are and why they are important.* IDC White Paper (2006).

[7] F. Dabek *et al. Vivaldi: A Decentralized Network Coordinate System. SIGCOMM'04*, 15–26 (2004).

[8] J. Ledlie *et al. Network Coordinates in the Wild.* NSDI'07: 4th USENIX Symposium on Networked Systems Design & Implementation (2007).

[9] G. Wang and T. S. E. Ng. *Distributed Algorithms for Stable and Secure Network Coordinates* (2008).

[10] J. Lucángeli Obes. *Una aplicacion de coordenadas de red para aumentar el tráfico intra-ISP en BitTorrent.* Tesis de licenciatura, Facultad de Ciencias Exactas y Naturales, UBA (2009).

[11] C. Lumezanu and N. Spring. *Playing Vivaldi in Hyperbolic Space. In Proc. of SIGCOMM-IMC* (2006).

[12] P. Moravek *et al. Study and performance of localization methods in IP based networks: Vivaldi algorithm.* J. Network and Computer Applications **34** (2011) 351.

[13] D. Andersen *et al. Resilient overlay networks.* SIGOPS Oper. Syst. Rev. **35** (2001) 131.
URL `http://doi.acm.org/10.1145/502059.502048`

[14] P. Francis *et al. IDMaps: a global Internet host distance estimation service.* IEEE/ACM Transactions on Networking **9** (2001) 525.

[15] T. S. E. Ng and H. Zhang. *Predicting Internet Network Distance with Coordinates-Based Approaches. INFOCOM*, 170–179 (2001).

[16] M. A. Kaafar *et al. Towards a two-tier internet coordinate system to mitigate the impact of triangle inequality violations. Proceedings of the 7th international IFIP-TC6 networking conference on AdHoc and sensor networks, wireless networks, next generation internet*, NETWORKING'08, 397–408. Springer-Verlag, Berlin, Heidelberg (2008).
URL `http://portal.acm.org/citation.cfm?id=1792514.1792560`

[17] B. Elser *et al.* *Spring for Vivaldi – Orchestrating Hierarchical Network Coordinates. 2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, 1–4 (2010).

[18] Y. Chen *et al.* *Pharos: A Decentralized and Hierarchical Network Coordinate System for Internet Distance Prediction. Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, 421 –426 (2007).

[19] H. Xie *et al.* *P4P: Explicit Communications for Cooperative Control Between P2P and Network Providers.*

[20] G. Zhang *et al.* *LANC: Locality-aware network coding for better P2P traffic localization.* Computer Networks **In Press, Uncorrected Proof** (2010).

[21] H. V. Madhyastha *et al.* *A structural approach to latency prediction. Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, 99–104. ACM, New York, NY, USA (2006).
URL `http://doi.acm.org/10.1145/1177080.1177092`

[22] D. R. Choffnes and F. E. Bustamante. *Taming the torrent: a practical approach to reducing cross-isp traffic in peer-to-peer systems.* SIGCOMM Comput. Commun. Rev. **38** (2008) 363.
URL `http://doi.acm.org/10.1145/1402946.1403000`

[23] Akamai Technologies. *Akamai.*
URL `http://www.akamai.com/`

[24] H. Wang *et al.* *To Unify Structured and Unstructured P2P Systems. Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 104a (2005).

[25] D. Harrison. *Fast Extension* (2008).
URL `http://bittorrent.org/beps/bep_0006.html`

[26] E. Hjelmvik and W. John. *Breaking and Improving Protocol Obfuscation.* Technical report, Department of Computer Science and Engineering, Chalmers University of Technology (2010).

[27] A. Norberg. *uTorrent transport protocol* (2009).
URL `http://bittorrent.org/beps/bep_0029.html`

[28] Ubuntu Documentation Team. *UbuntuTime.*
URL `https://help.ubuntu.com/community/UbuntuTime`

[29] Microsoft Corp. *Windows Time Service Tools and Settings: Windows Time Service.*
URL `http://technet.microsoft.com/en-us/library/cc773263%28WS.10%29.aspx`

[30] D. E. Dilger. *Snow Leopard Server.* Wiley Publishing, Inc. (2010).

[31] H. Stern. *Managing NFS and NIS.* O'Reilly & Associates, Inc. (1991).

[32] M. Szymaniak *et al.* *Practical large-scale latency estimation.* Comput. Netw. **52** (2008) 1343.

[33] Varios autores. *Bittorrent Protocol Specification v1.0.*
URL `http://wiki.theory.org/BitTorrentSpecification`

[34] R. Bindal *et al. Improving Traffic Locality in BitTorrent via Biased Neighbor Selection. Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, 66 – 66 (2006).

[35] B. Elser *et al. Tuning Vivaldi: Achieving Increased Accuracy and Stability.* T. Spyropoulos and K. Hummel, eds., *Self-Organizing Systems*, volume 5918 of *Lecture Notes in Computer Science*, 174–184. Springer Berlin / Heidelberg (2009).

[36] E. Eide *et al. An Experimentation Workbench for Replayable Networking Research. Proceedings of the Symposium on Networked System Design and Implementation* (2007).

[37] K. Huang *et al. Optimizing the BitTorrent performance using an adaptive peer selection strategy.* Future Generation Computer Systems **24** (2008) 621 .
URL `http://www.sciencedirect.com/science/article/pii/S0167739X07001616`

A

Uso del software implementado

A.1 vivaldisimulator

Los parámetros se pasan por línea de comandos. La sintaxis es la siguiente:

```
vivaldisimulator [--simulate-output] num_nodes latency_list
```

- `num_nodes`: entero que indica la cantidad de nodos a simular.
- `latency_list`: nombre de un archivo de texto con la información sobre las latencias entre nodos. El formato del mismo es el siguiente:

  $host_1$
  ==========
  $host_2$: rtt min/avg/max/mdev = $val_{1,2}1/val_{1,2}2/val_{1,2}3/val_{1,2}4$ ms
  ⋮
  $host_n$: rtt min/avg/max/mdev = $val_{1,n}1/val_{1,n}2/val_{1,n}3/val_{1,n}4$ ms

  $host_2$
  ==========
  $host_1$: rtt min/avg/max/mdev = $val_{2,1}1/val_{2,1}2/val_{2,1}3/val_{2,1}4$ ms
  $host_3$: rtt min/avg/max/mdev = $val_{2,3}1/val_{2,3}2/val_{2,3}3/val_{2,3}4$ ms
  ⋮
  $host_n$: rtt min/avg/max/mdev = $val_{2,n}1/val_{2,n}2/val_{2,n}3/val_{2,n}4$ ms

  ⋮

  $host_n$
  ==========
  $host_1$: rtt min/avg/max/mdev = $val_{n,1}1/val_{n,1}2/val_{n,1}3/val_{n,1}4$ ms
  ⋮
  $host_{n-1}$: rtt min/avg/max/mdev = $val_{n,n-1}1/val_{n,n-1}2/val_{n,n-1}3/val_{n,n-1}4$ ms

  - $host_n$ indica el nombre del host número n (puede ser un nombre corto, el Fully Qualified Domain Name (FQDN) de un host o su dirección IP, no hay restricciones).

- $val_{n,m}l$ es el valor l de latencia (escrito como número decimal separado por punto) entre el host n y el host m. El número l es un entero tal que $1 \leq l \leq 4$ e indica cada uno de los distintos valores de RTT, en orden: el mínimo, el promedio, el máximo y el desvío promedio. El simulador siempre utiliza el valor promedio, pero es sencillo adaptarlo para que use el mínimo o el máximo.
- La información sobre las latencias se asume simétrica. Si ya se cargó la información de latencia para un par dado, el inverso se ignora (cuál de los dos de un par es utilizado depende del orden en el que se cargan los datos).

Notar que la información sobre las latencias tiene el mismo formato que la salida del comando `ping` en UNIX. Esto es intencional, ya que lo utilizamos para generar los valores.

- `simulate-output`: si este parámetro se encuentra presente, el simulador genera por pantalla una salida equivalente a la que se obtiene por medio de **syslog** en el servidor de logging central.

A.2 vivaldistandalone

Los parámetros se pasan por línea de comandos. La sintaxis es la siguiente:

```
vivaldistandalone host_list [time]
```

- `host_list`: nombre del archivo de texto con la lista de *hosts* participantes, uno por línea (es la misma para todos los nodos, no es necesario eliminarse a sí mismo ya que se encarga el programa de detectar su nombre de *host* y de eliminarse). Para detectar su IP, genera una estructura de conexión (**socket**) al *host* configurado como *logging server*. Esto implica que, según dónde se encuentre el server configurado, es posible que utilice la interfaz con conexión a Internet para comunicarse, y además deba tener una configuración de DNS que permita resolver direcciones externas para funcionar correctamente.
- `time`: cantidad de segundos que se ejecutará el programa. Si el valor es 0 o no está presente, seguirá ejecutándose hasta que reciba una señal de terminación (como Ctrl-C, SIGTERM, SIGKILL).

A.3 BitTornado

Este software tiene distintas implementaciones de interfaz de usuario. Según cuál de ellas utilicemos (headless, interfaz de consola, interfaz gráfica sobre X/Windows, etc.), las opciones se pasan por línea de comandos o por archivos de configuración. En nuestras pruebas siempre utilizamos la interfaz headless, `btdownloadheadless.py`, por lo que las opciones se pasan por línea de comandos. Mostramos a continuación la sintaxis de uso y las opciones agregadas, junto con sus respectivos valores predeterminados:

`btdownloadheadless.py` `[--opción`$_1$ `valor`$_1$`] [--opción`$_2$ `valor`$_2$`] [...] [--opción`$_n$ `valor`$_n$`]`

donde `opción`$_n$ puede ser una de las siguientes:

- `use_extended_protocol`: si vale 1, entonces se utiliza el protocolo extendido con coordenadas de Vivaldi que definimos en la sección 2.3.1. Valor predeterminado: 1.
- `upload_appeal_choker`: si vale 1, entonces se utiliza el algoritmo de choking modificado que definimos en la sección 2.3.3. Valor predeterminado: 1.
- `neighbor_decay`: si vale 1, entonces se utiliza la mejora de extensión del conjunto de vecinos (neighbor decay). Valor predeterminado: 0.
- `neighbor_decay_time`: su valor indica cada cuánto tiempo (en segundos) se actualiza la coordenada local utilizando neighbor decay. Sólo se tiene en cuenta este parámetro si `neighbor_decay` vale 1. Valor predeterminado: 5.
- `logger_host`: su valor indica a qué host debe conectarse el cliente para enviar su información de *logging*. Valor predeterminado: `tesis.dc.uba.ar`.
- `logger_url`: su valor indica a qué recurso del servidor indicado por la opción `logger_host` debe conectarse (utilizando HTTP) para enviar su información de logging. Valor predeterminado: `/`. Los valores default de esta opción y la anterior indican que, predeterminadamente, el programa se conecta a `http://tesis.dc.uba.ar/` para enviar sus *logs*.
- `statistics_report_time`: su valor indica el período de tiempo de espera medido en segundos entre una actualizacion al servidor de *logging* y la siguiente. Valor predeterminado: 10.
- `median_filter`: si vale 1, entonces se aplica un filtro de mediana en las mediciones de latencia que el algoritmo de Vivaldi utiliza como entrada. Valor predeterminado: 1.
- `median_size`: indica cuántas de las últimas mediciones de latencia almacena como máximo en la lista utilizada por el filtro de mediana. Valor predeterminado: 10.
- `vivaldi_cc`: indica el valor de la constante c_c, utilizada en el algoritmo de actualización de coordenadas de Vivaldi (ver algoritmo 3). Valor predeterminado: 1,0.
- `vivaldi_ce`: indica el valor de la constante c_e, utilizada en el algoritmo de actualización de coordenadas de Vivaldi (ver algoritmo 3). Valor predeterminado: 1,0.

B

Formatos de representación

B.1 Coordenadas Vivaldi

Para codificar las coordenadas Vivaldi diseñamos un formato sencillo que pueda ser enviado por la red y donde además se pueda acceder con facilidad a cada uno de los campos.

B.1.1. Floats

Los números de punto flotante presentes en las cordenadas son codificados utilizando `struct.pack` de Python. Cada `Float` codificado tiene 4 bytes de longitud ordenados como *big-endian*[1]. Además, no se utiliza la estructura de representación interna de la máquina que corre el programa sino IEEE 754 binary32.

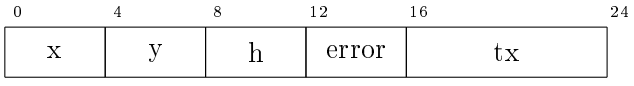
B.1.2. Doubles

Utilizamos `Double` para el campo del *timestamp* ya que la precisión de `Float` nos resulta insuficiente. La codificación es idéntica al caso de los `Floats`, con la diferencia de que un `Double` codificado tiene una longitud de 8 bytes en lugar de 4. La estructura de representación es IEEE 754 binary64.

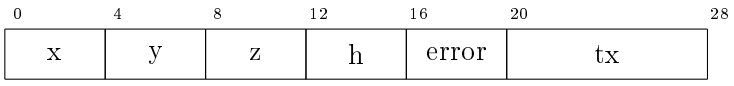
B.1.3. Coordenadas

Las coordenadas son simplemente cadenas de `Floats` y un `Double`, cada uno codificado como mencionamos anteriormente. Según la cantidad de dimensiones del espacio métrico utilizado tendremos diferentes formatos de coordenada:

- 2D + H

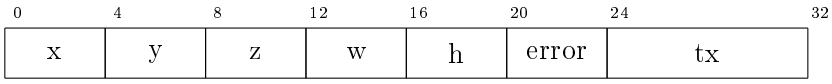


- 3D + H



[1] Se utiliza *big-endian* convencionalmente al transmitir valores numéricos por la red.

- 4D + H



Las coordenadas 2D + H tienen una longitud de 24 bytes, 3D + H 28 bytes, y las de mayor cantidad de dimensiones, 4D + H, tienen 32 bytes de longitud.

C

Documentación de las clases extendidas en BitTornado

Hablaremos a continuación de cada uno de los módulos de BitTornado en los que trabajamos y de cómo fueron modificados o implementados (en el caso de los nuevos). Utilizaremos la notación de Python para describir el objeto invocador (`self`), el constructor (método `__init__`), los métodos y los atributos de las clases. Cuando listamos los parámetros en la *signature* del método excluimos el parámetro implícito `self`.

C.1 Módulos en el namespace BitTornado

C.1.1. FloatEncoder

Este nuevo módulo implementa una clase del mismo nombre que permite codificar `Floats` de Python como cadenas de bytes para su transmisión por la red en un formato independiente de la plataforma. Para la codificación utiliza `struct.pack`, incluído en la biblioteca standard de Python. La interfaz exportada es la siguiente:

- `__init__(f)`: el valor default de `f` es 0. El mismo es utilizado para inicializar el codificador.
- `setdata(data)`: sobreescribe el valor codificado almacenado en el objeto por `data`. Se lo utiliza para decodificar un valor obtenido de otra fuente (por ejemplo, de la red).
- `decode()`: devuelve el valor almacenado, nuevamente convertido a `Float`.
- `dump()`: devuelve el valor codificado.

C.1.2. DoubleEncoder

Este módulo es idéntico a `BitTornado.FloatEncoder`, con la diferencia de que codifica los números de punto flotante en `Double` en lugar de en `Float`. Esta clase se utiliza para codificar el *timestamp* de envío de las coordenadas, ya que no alcanza la precisión para enviarla como `Float`. La interfaz exportada es también la misma.

C.1.3. Encrypter

Esta clase se encuentra un nivel por debajo de `BitTornado.Connection`. Entre otras cosas, se la utiliza como *wrapper* para cifrar y descifrar los mensajes transmitidos y para inicializar la conexión.

En el constructor hace el *handshake* con el *peer* remoto, e intercambia el valor del campo **reserved** del mensaje homónimo. Este campo se lo utiliza como un conjunto de *flags*, para que cada cliente pueda determinar si el otro soporta alguna *feature* o no. Nuestra implementación utiliza el bit 20 para indicar si el cliente soporta el protocolo extendido con coordenadas Vivaldi o no. Además, esta flag es configurable desde el programa con la opción `use_extended_protocol`.

Notar que el protocolo extendido se utiliza sólo si en el *handshake* ambos clientes informan que lo soportan.

C.1.4. statslogger

Esta clase instancia un objeto del módulo logging de Python que se conecta a las IP y URL en `http://LOGGING_HOST/LOGGING_URL` (ambos parámetros fijos en el módulo) y envía por POST de HTTP el mensaje especificado. Agregamos además las opciones `logger_host` y `logger_url` para poder configurar desde el programa ambos valores.

Los métodos que implementa esta clase son:

- `set_position(pos)`: setea `pos` como el valor de posición actual.
- `add_counter(counter, value)`: incrementa un contador de nombre `counter` en el valor especificado por `value` (lo crea si no existe) en el diccionario de contadores a guardar en el *log*.
- `add_counter_dictionary(dict, key, value)`: agrega un diccionario de contadores de nombre `dict`, dentro del cual define la clave `key` con el valor `value`.
- `remove_counter(counter)`: elimina el contador de nombre `counter`.
- `reset_logger(host, url)`: reinicia el *logger*, reconfigurando el host y URL donde se conecta para enviar los mensajes de *log*.
- `info(msg)`: envía `msg`, prefijándole el peer ID.
- `reset()`: borra todos los contadores almacenados.
- `log()`: envía el diccionario de contadores convertido en texto y luego llama a `self.reset`.

C.1.5. download_bt1

Este módulo es el loop principal de la aplicación.

Le agregamos un método, `statistics_reporter`, que envía cada cierta cantidad de segundos configurable datos que consideramos pertinente almacenar en el servidor central, utilizando el método `info` de `BitTornado.statslogger`.

La implementación del timer la realizamos utilizando el mecanismo para timers ya presente en BitTornado (BitTornado no usa *threads*). Su arquitectura es sencilla: registramos la función a ejecutar por medio del método `add_task`, a la que se le pasa la referencia a la función y la cantidad de segundos que debe esperar para ejecutarla. En cada ciclo de ejecución del cliente, itera por la lista de las *tasks* agregadas y si el tiempo transcurrido es mayor al tiempo de espera las ejecuta y elimina de la cola. Luego la misma función se encarga de agregarse nuevamente con `add_task`. Es posible que el timer no se ejecute en el momento exacto en el que

nosotros queremos ya que podría suceder que la ejecución del resto de las operaciones de cada ciclo demore una cantidad grande de tiempo (esto implica que puede haber expirado el timer pero aún no haber sido ejecutado, ya que primero deben terminar otros procesos), pero otros timeouts del orden de segundos funcionan de la misma manera, por lo que no consideramos que sea un problema.

C.2 Módulos en el namespace BitTornado.BT1

C.2.1. HeightCoordinate

Este nuevo módulo implementa una clase del mismo nombre que representa un punto en un sistema de coordenadas de hasta 4 dimensiones euclidianas con una componente adicional de altura. La clase exporta la siguiente interfaz:

- `__init__(dim, x, y, z, w, h)`: inicializa las coordenadas de hasta *dim* dimensiones (máximo 4: x, y, z, w) y h (de altura). El valor predeterminados de *dim* es 3, mientras que el todas las otras variables es 0.
- `randomUnity`: reinicializa las coordenadas en valores aleatorios con distancia 1.
- `add(another)`: suma las coordenadas `self` y `another` y devuelve un nuevo objeto con el resultado.
- `sub(another)`: resta las coordenadas `self` y `another` y devuelve un nuevo objeto con el resultado.
- `scale(factor)`: multiplica el vector de `self` por el escalar `factor`, devolviendo un nuevo objeto con el resultado.
- `measure()`: calcula la norma del vector y le suma a la altura `h`.
- `atOrigin()`: auxiliar que devuelve `True` si el vector representado es el vector nulo.
- `distance(another)`: devuelve la distancia entre `self` y `another`.
- `planeDistance(another)`: devuelve la distancia planar (es decir, sólo considerando las coordenadas del vector sin la altura) entre `self` y `another`.
- `unity()`: auxiliar que convierte a `self` en el mismo vector pero de norma unitaria.
- `copy(another)`: devuelve en `self` una copia atributo a atributo de `another`.
- `copynew(another)`: devuelve un objeto nuevo con una copia de `another`.
- `serialize()`: devuelve una versión serializada de `self` para su envío por algún medio (como la red). El formato es diferente en función en la cantidad de dimensiones de la coordenada. Primero se codifican las componentes de la coordenada, después `h`, `error` y `tx_timestamp`. Esto implica que para que los clientes puedan comunicarse correctamente deben usar el mismo formato de coordenadas. Notar que no se codifica `rx_timestamp`, ya que no es información que le pueda interesar al receptor del mensaje.

- `unserialize(s)`: realiza la operación inversa a `serialize`, toma de entrada una cadena `s` con el formato de salida de `serialize` y obtiene los valores de los atributos, sobreescribiendo con estos los de `self`.

La clase tiene los siguientes atributos:

- `x`: componente x
- `y`: componente y
- `z`: componente z
- `w`: componente w
- `dim`: cantidad de dimensiones del vector
- `h`: componente de altura
- `error`: error calculado de la coordenada
- `tx_timestamp`: tiempo (en ticks de `time.time()`[1] de Python) de transmisión de la coordenada
- `rx_timestamp`: tiempo (en ticks de `time.time()` de Python) de recepción de la coordenada

C.2.2. VivaldiCoordinate

Este módulo implementa dos clases, la homónima `VivaldiCoordinate`, que implementa coordenadas de Vivaldi, y `VivaldiNeighborDecayer`, que implementa un actualizador de coordenadas progresivo que otorga mayor importancia a los nodos cuya información es más reciente que a los más antiguos.

- `VivaldiCoordinate`: esta clase extiende `HeightCoordinate`, agregándole los siguientes atributos:
  - `decayer`: Instancia de la clase `VivaldiNeighborDecayer` (ver a continuación).
  - `logger`: Instancia de la clase `logging` de Python, utilizada para enviar mensajes de *log* en la actualización de las coordenadas.
  - `c_c`: Indica el valor de la constante c_c, utilizada en el algoritmo de actualización de coordenadas de Vivaldi (ver algoritmo 3). Su valor es configurable por medio de la opción `vivaldi_cc`.
  - `c_e`: Indica el valor de la constante c_e, utilizada en el algoritmo de actualización de coordenadas de Vivaldi (ver algoritmo 3). Su valor es configurable por medio de la opción `vivaldi_ce`.

[1] Cantidad de segundos desde el 1 de Enero de 1970 (tiempo UNIX). El valor devuelto es un número de punto flotante, por lo que la precisión puede variar según el SO y la implementación. En el caso de Linux, nuestra plataforma de prueba, la precisión es del orden de los *microsegundos*.

También implementa nuevos métodos relacionados con los atributos agregados y con la actualización de coordenadas Vivaldi. Los mismos se detallan a continuación:

- `set_logger(logger)`: *setter* de `logger`.
- `set_decayer(decayer)`: *setter* de `decayer`.
- `_inplace_add(another)`: este método suma `self` con `another`, almacenando el resultado en `self`. No devuelve nada.
- `updateCoordinate(peer_id, peer_coord, latency)`: aquí se implementa la lógica de actualización de coordenadas de Vivaldi. El parámetro `peer_id` se utiliza como identificador para almacenar el corrimiento que produce el *peer* remoto en la coordenada local. La información del *peer* remoto se encuentra en los otros dos parámetros: `peer_coord` es la coordenada remota, mientras que `latency` indica la latencia medida entre el *peer* remoto y el local.

- `VivaldiNeighborDecayer`: esta clase implementa un desplazador adaptativo para la coordenada del *peer* local basado en una mejora de ampliación del conjunto de vecinos que aparece en [8]. Su uso está condicionado al valor de la variable de configuración `neighbor_decay`

  El algoritmo de actualización de coordenadas es modificado para que almacene una lista con las coordenadas de los *peers* recientemente contactados, y además cuándo se realizó la última actualización. Esta lista se procesa una vez cada cierta cantidad de tiempo configurable desde la opción `neighbor_time`, y se actualiza la coordenada local utilizando la información de todos los vecinos modulada por la antigüedad de cada dato.

C.2.3. Connecter

Este módulo se encarga de manejar las conexiones, recibe los mensajes y los procesa. Aquí es donde aparecen las coordenadas. Detallamos a continuación las modificaciones realizadas a cada una de las dos clases implementadas en el mismo: `Connecter` y `Connection`.

- `Connecter`

  Esta clase representa las conexiones hechas por el cliente. Mantiene la coordenada del cliente en el atributo `coordinate`, de tipo `VivaldiCoordinate`. En el método `got_message` se comprueba que el `peer` remoto soporte nuestra extensión del protocolo, y en caso afirmativo, y si está activado el uso de la extensión, actualiza las coordenadas del cliente con los nuevos valores.

  Finalmente, llamamos al método `updateCoordinate()` de `VivaldiCoordinate`, que corrige la coordenada propia usando el valor de la obtenida del *peer* remoto y la latencia calculada. Con respecto al valor de latencia, según se encuentre habilitado el uso del filtro de mediana o no, se guarda un historial con las últimas mediciones de latencia y se toma la mediana entre ellas como parámetro de `updateCoordinate()` en lugar de la medición actual. La corrección de la coordenada local se computa según el algoritmo 3.

- `Connection`

  Esta clase representa cada conexión entre el cliente local y uno remoto. Se extendió el método `_send_message` para que detecte si el *peer* remoto soporta el protocolo extendido

(de la misma forma que en `Connecter`), y en caso afirmativo envía adosada al mensaje la información de las coordenadas locales.

C.2.4. Choker

Esta clase implementa el algoritmo de choking de BitTornado. Este algoritmo fue modificado para tener en cuenta la distancia en el proceso de decisión. El método cambiado es `_rechoke`, donde implementamos nuestra versión modificada del algoritmo de choking, que mostramos en el algoritmo 6.

Agregamos además el método estático `_get_upload_appeal`, utilizada para obtener el *upload appeal* de un cliente dado. Como puede suceder que el *peer* no tenga una distancia asociada (por ejemplo, porque no soporta nuestra extensión o porque no se encuentra deshabilitada), en ese caso estima una dentro de los valores esperables de distancia. En nuestra implementación elegimos un número aleatorio entre 0 y 1000, ya que eran valores aceptables dentro de los cuales podía encontrarse la distancia real, sin priorizar de esta manera ni los nodos sobre los que no tenemos datos reales de posición ni los que sí están utilizando Vivaldi.

También se puede elegir utilizar la versión original del *choker* desactivando la opción `upload_appeal_choker`.

D

Redes utilizadas en formato NS

Los códigos de las redes utilizadas se encuentran en Tcl, un lenguaje de scripting utilizado por el simulador de red NS-2 para describir las redes simuladas. Este formato es el mismo que acepta Emulab para generar las topologías. La sintaxis es la siguiente:

- `set` *obj* `[new` *class*`]`

  Crea el objeto *obj* como una nueva instancia de la clase *class*.

- $function\ param_1\ \ldots\ param_n$

  Invoca la función $function$ con los parámetros $param_1$ a $param_n$. No guarda el resultado.

- `set` obj_1 `[`$obj_2\ method\ param_1\ \ldots\ param_n$`]`

  Guarda el resultado de invocar el método *method* del objeto obj_2 con los parámetros $param_1$ a $param_n$ en obj_1.

Los comandos utilizados por nuestras redes tienen la siguiente semántica:

- `set ns [new Simulator]`

  Inicializa el objeto `ns` de tipo `Simulator`. Este objeto es utilizado para realizar todas las operaciones relacionadas con nodos en Emulab.

- `set node0 [$ns node]`

  Crea un nodo de nombre `node0`, invocando el método `node` del objeto `ns`.

- `tb-set-node-os $node0 UBUNTU10-STD`

  Configura `node0` para que cargue la imagen del sistema operativo UBUNTU10-STD.

- `set link0 [$ns duplex-link $node0 $node17 100000.0kb 55.0ms DropTail]`

  Establece un link de nombre `link0` entre `node0` y `node17` de 55 ms de latencia y un ancho de banda de 100Mbps. La política de la cola de paquetes asociada al link es `DropTail` (elimina paquetes de la cola cuando se llena).

- `set lan0 [$ns make-lan "$node0 $node1 $node2 $node3" 100000.0kb 2.0ms]`

  Crea una LAN llamada `lan0` con los nodos `node0`, `node1`, `node2` y `node3`, y establece 100Mbps de ancho de banda y 2 ms de latencia entre cada nodo (con excepción del punto de acceso, ver `tb-set-node-lan-delay`). Todos los nodos de la LAN se encuentran

unidos entre sí a través de un switch. Un nodo puede pertenecer a varias LANs, pero utiliza una interfaz de red por cada una, por lo que si se lo configura en más LANs que interfaces disponibles no se podrá instanciar la red.

- `tb-set-node-lan-delay $node0 $lan0 20.0ms`

  Establece un link entre `node0` y la LAN `lan0` de 20 ms de latencia. El nodo especificado es el punto de acceso a la LAN.

- `$ns rtproto Static`

  Utiliza ruteo estático entre los nodos. Calcula las tablas de ruteo en el momento de instanciar la red. El algoritmo utilizado es determinístico.

- `$ns run`

  Instancia la red. Esta línea es siempre la última.

D.1 Fourlan

```
# Generated by NetlabClient

set ns [new Simulator]
source tb_compat.tcl

# Nodes
set node0 [$ns node]
tb-set-node-os $node0 UBUNTU10-STD
set node1 [$ns node]
tb-set-node-os $node1 UBUNTU10-STD
set node2 [$ns node]
tb-set-node-os $node2 UBUNTU10-STD
set node3 [$ns node]
tb-set-node-os $node3 UBUNTU10-STD
set node4 [$ns node]
tb-set-node-os $node4 UBUNTU10-STD
set node5 [$ns node]
tb-set-node-os $node5 UBUNTU10-STD
set node6 [$ns node]
tb-set-node-os $node6 UBUNTU10-STD
set node7 [$ns node]
tb-set-node-os $node7 UBUNTU10-STD
set node8 [$ns node]
tb-set-node-os $node8 UBUNTU10-STD
set node9 [$ns node]
tb-set-node-os $node9 UBUNTU10-STD

# Lans
set lan0 [$ns make-lan "$node0 $node1 $node2 $node3" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node0 $lan0 20.0ms
```

```
set lan1 [$ns make-lan "$node0 $node4 $node5" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node0 $lan1 10.0ms
set lan2 [$ns make-lan "$node0 $node8 $node9" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node0 $lan2 17.0ms
set lan3 [$ns make-lan "$node0 $node6 $node7" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node0 $lan3 11.0ms

$ns rtproto Static
$ns run

# NetlabClient generated file ends here.
# Finished at: 12/4/10 8:01 PM
```

D.2 Tenlan

```
# Generated by NetlabClient

set ns [new Simulator]
source tb_compat.tcl

# Nodes
set node0 [$ns node]
tb-set-node-os $node0 UBUNTU10-STD
set node1 [$ns node]
tb-set-node-os $node1 UBUNTU10-STD
set node2 [$ns node]
tb-set-node-os $node2 UBUNTU10-STD
set node3 [$ns node]
tb-set-node-os $node3 UBUNTU10-STD
set node4 [$ns node]
tb-set-node-os $node4 UBUNTU10-STD
set node5 [$ns node]
tb-set-node-os $node5 UBUNTU10-STD
set node6 [$ns node]
tb-set-node-os $node6 UBUNTU10-STD
set node7 [$ns node]
tb-set-node-os $node7 UBUNTU10-STD
set node8 [$ns node]
tb-set-node-os $node8 UBUNTU10-STD
set node9 [$ns node]
tb-set-node-os $node9 UBUNTU10-STD
set node10 [$ns node]
tb-set-node-os $node10 UBUNTU10-STD
set node11 [$ns node]
tb-set-node-os $node11 UBUNTU10-STD
set node12 [$ns node]
```

```
tb-set-node-os $node12 UBUNTU10-STD
set node13 [$ns node]
tb-set-node-os $node13 UBUNTU10-STD
set node14 [$ns node]
tb-set-node-os $node14 UBUNTU10-STD
set node15 [$ns node]
tb-set-node-os $node15 UBUNTU10-STD
set node16 [$ns node]
tb-set-node-os $node16 UBUNTU10-STD
set node17 [$ns node]
tb-set-node-os $node17 UBUNTU10-STD
set node18 [$ns node]
tb-set-node-os $node18 UBUNTU10-STD
set node19 [$ns node]
tb-set-node-os $node19 UBUNTU10-STD
set node20 [$ns node]
tb-set-node-os $node20 UBUNTU10-STD
set node21 [$ns node]
tb-set-node-os $node21 UBUNTU10-STD
set node22 [$ns node]
tb-set-node-os $node22 UBUNTU10-STD
set node23 [$ns node]
tb-set-node-os $node23 UBUNTU10-STD
set node24 [$ns node]
tb-set-node-os $node24 UBUNTU10-STD
set node25 [$ns node]
tb-set-node-os $node25 UBUNTU10-STD
set node26 [$ns node]
tb-set-node-os $node26 UBUNTU10-STD
set node27 [$ns node]
tb-set-node-os $node27 UBUNTU10-STD
set node28 [$ns node]
tb-set-node-os $node28 UBUNTU10-STD
set node29 [$ns node]
tb-set-node-os $node29 UBUNTU10-STD
set node30 [$ns node]
tb-set-node-os $node30 UBUNTU10-STD
set node31 [$ns node]
tb-set-node-os $node31 UBUNTU10-STD
set node32 [$ns node]
tb-set-node-os $node32 UBUNTU10-STD

# Links
set link0 [$ns duplex-link $node0 $node17 100000.0kb 55.0ms DropTail]
set link1 [$ns duplex-link $node0 $node18 100000.0kb 45.0ms DropTail]
set link2 [$ns duplex-link $node17 $node19 100000.0kb 70.0ms DropTail]
set link3 [$ns duplex-link $node19 $node27 100000.0kb 105.0ms DropTail]
```

```
# Lans
set lan0 [$ns make-lan "$node0 $node1 $node2 $node3" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node0 $lan0 20.0ms
set lan1 [$ns make-lan "$node0 $node4 $node5" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node0 $lan1 10.0ms
set lan2 [$ns make-lan "$node22 $node8 $node9" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node22 $lan2 17.0ms
set lan3 [$ns make-lan "$node17 $node6 $node7" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node17 $lan3 11.0ms
set lan4 [$ns make-lan "$node17 $node10 $node11 $node12" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node17 $lan4 23.0ms
set lan5 [$ns make-lan "$node21 $node13 $node14 $node15 $node16" 100000.0kb
2.0ms]
tb-set-node-lan-delay $node21 $lan5 20.0ms
set lan6 [$ns make-lan "$node18 $node20 $node21 $node22 $node23" 100000.0kb
2.0ms]
tb-set-node-lan-delay $node18 $lan6 10.0ms
set lan7 [$ns make-lan "$node19 $node24 $node25 $node26" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node19 $lan7 16.0ms
set lan8 [$ns make-lan "$node27 $node28 $node29 $node30" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node27 $lan8 10.0ms
set lan9 [$ns make-lan "$node18 $node31 $node32" 100000.0kb 2.0ms]
tb-set-node-lan-delay $node18 $lan9 26.0ms

$ns rtproto Static
$ns run

# NetlabClient generated file ends here.
# Finished at: 12/4/10 8:01 PM
```

D.3 Biglans

```
# Generated by NetlabClient

set ns [new Simulator]
source tb_compat.tcl

# Nodes
set node0 [$ns node]
tb-set-node-os $node0 UBUNTU10-STD
set node1 [$ns node]
tb-set-node-os $node1 UBUNTU10-STD
set node2 [$ns node]
tb-set-node-os $node2 UBUNTU10-STD
set node3 [$ns node]
```

```
tb-set-node-os $node3 UBUNTU10-STD
set node4 [$ns node]
tb-set-node-os $node4 UBUNTU10-STD
set node5 [$ns node]
tb-set-node-os $node5 UBUNTU10-STD
set node6 [$ns node]
tb-set-node-os $node6 UBUNTU10-STD
set node7 [$ns node]
tb-set-node-os $node7 UBUNTU10-STD
set node8 [$ns node]
tb-set-node-os $node8 UBUNTU10-STD
set node9 [$ns node]
tb-set-node-os $node9 UBUNTU10-STD
set node10 [$ns node]
tb-set-node-os $node10 UBUNTU10-STD
set node11 [$ns node]
tb-set-node-os $node11 UBUNTU10-STD
set node12 [$ns node]
tb-set-node-os $node12 UBUNTU10-STD
set node13 [$ns node]
tb-set-node-os $node13 UBUNTU10-STD
set node14 [$ns node]
tb-set-node-os $node14 UBUNTU10-STD
set node15 [$ns node]
tb-set-node-os $node15 UBUNTU10-STD
set node16 [$ns node]
tb-set-node-os $node16 UBUNTU10-STD
set node17 [$ns node]
tb-set-node-os $node17 UBUNTU10-STD
set node18 [$ns node]
tb-set-node-os $node18 UBUNTU10-STD
set node19 [$ns node]
tb-set-node-os $node19 UBUNTU10-STD
set node20 [$ns node]
tb-set-node-os $node20 UBUNTU10-STD
set node21 [$ns node]
tb-set-node-os $node21 UBUNTU10-STD
set node22 [$ns node]
tb-set-node-os $node22 UBUNTU10-STD
set node23 [$ns node]
tb-set-node-os $node23 UBUNTU10-STD
set node24 [$ns node]
tb-set-node-os $node24 UBUNTU10-STD
set node25 [$ns node]
tb-set-node-os $node25 UBUNTU10-STD
set node26 [$ns node]
tb-set-node-os $node26 UBUNTU10-STD
```

```
set node27 [$ns node]
tb-set-node-os $node27 UBUNTU10-STD
set node28 [$ns node]
tb-set-node-os $node28 UBUNTU10-STD
set node29 [$ns node]
tb-set-node-os $node29 UBUNTU10-STD
set node30 [$ns node]
tb-set-node-os $node30 UBUNTU10-STD
set node31 [$ns node]
tb-set-node-os $node31 UBUNTU10-STD
set node32 [$ns node]
tb-set-node-os $node32 UBUNTU10-STD
set node33 [$ns node]
tb-set-node-os $node33 UBUNTU10-STD
set node34 [$ns node]
tb-set-node-os $node34 UBUNTU10-STD
set node35 [$ns node]
tb-set-node-os $node35 UBUNTU10-STD
set node36 [$ns node]
tb-set-node-os $node36 UBUNTU10-STD
set node37 [$ns node]
tb-set-node-os $node37 UBUNTU10-STD
set node38 [$ns node]
tb-set-node-os $node38 UBUNTU10-STD
set node39 [$ns node]
tb-set-node-os $node39 UBUNTU10-STD
set node40 [$ns node]
tb-set-node-os $node40 UBUNTU10-STD
set node41 [$ns node]
tb-set-node-os $node41 UBUNTU10-STD
set node42 [$ns node]
tb-set-node-os $node42 UBUNTU10-STD
set node43 [$ns node]
tb-set-node-os $node43 UBUNTU10-STD
set node44 [$ns node]
tb-set-node-os $node44 UBUNTU10-STD
set node45 [$ns node]
tb-set-node-os $node45 UBUNTU10-STD
set node46 [$ns node]
tb-set-node-os $node46 UBUNTU10-STD
set node47 [$ns node]
tb-set-node-os $node47 UBUNTU10-STD
set node48 [$ns node]
tb-set-node-os $node48 UBUNTU10-STD
set node49 [$ns node]
tb-set-node-os $node49 UBUNTU10-STD
set node50 [$ns node]
```

```
tb-set-node-os $node50 UBUNTU10-STD
set node51 [$ns node]
tb-set-node-os $node51 UBUNTU10-STD
set node52 [$ns node]
tb-set-node-os $node52 UBUNTU10-STD
set node53 [$ns node]
tb-set-node-os $node53 UBUNTU10-STD
set node54 [$ns node]
tb-set-node-os $node54 UBUNTU10-STD
set node55 [$ns node]
tb-set-node-os $node55 UBUNTU10-STD
set node56 [$ns node]
tb-set-node-os $node56 UBUNTU10-STD
set node57 [$ns node]
tb-set-node-os $node57 UBUNTU10-STD
set node58 [$ns node]
tb-set-node-os $node58 UBUNTU10-STD
set node59 [$ns node]
tb-set-node-os $node59 UBUNTU10-STD
set node60 [$ns node]
tb-set-node-os $node60 UBUNTU10-STD
set node61 [$ns node]
tb-set-node-os $node61 UBUNTU10-STD
set node62 [$ns node]
tb-set-node-os $node62 UBUNTU10-STD
set node63 [$ns node]
tb-set-node-os $node63 UBUNTU10-STD

# Links
set link0 [$ns duplex-link $node0 $node17 100000.0kb 55.0ms DropTail]
set link1 [$ns duplex-link $node0 $node18 100000.0kb 45.0ms DropTail]
set link2 [$ns duplex-link $node17 $node19 100000.0kb 70.0ms DropTail]
set link3 [$ns duplex-link $node19 $node27 100000.0kb 105.0ms DropTail]

# Lans
set lan0 [$ns make-lan "$node0 $node1 $node2 $node3" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node0 $lan0 20.0ms
set lan1 [$ns make-lan "$node0 $node4 $node5" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node0 $lan1 10.0ms
set lan2 [$ns make-lan "$node22 $node8 $node9" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node22 $lan2 17.0ms
set lan3 [$ns make-lan "$node17 $node6 $node7" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node17 $lan3 11.0ms
set lan4 [$ns make-lan "$node17 $node10 $node11 $node12" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node17 $lan4 23.0ms
set lan5 [$ns make-lan "$node21 $node13 $node14 $node15 $node16" 100000.0kb
0.0ms]
```

```
tb-set-node-lan-delay $node21 $lan5 20.0ms
set lan6 [$ns make-lan "$node18 $node20 $node21 $node22 $node23" 100000.0kb
0.0ms]
tb-set-node-lan-delay $node18 $lan6 10.0ms
set lan7 [$ns make-lan "$node19 $node24 $node25 $node26" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node19 $lan7 16.0ms
set lan8 [$ns make-lan "$node27 $node28 $node29 $node30" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node27 $lan8 2.0ms
set lan9 [$ns make-lan "$node18 $node31 $node32" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node18 $lan9 26.0ms
set lan10 [$ns make-lan "$node32 $node33 $node34 $node35" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node32 $lan10 15.0ms
set lan11 [$ns make-lan "$node35 $node36 $node37 $node38" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node35 $lan11 10.0ms
set lan12 [$ns make-lan "$node31 $node39 $node40 $node41" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node31 $lan12 10.0ms
set lan13 [$ns make-lan "$node5 $node42 $node43 $node44" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node5 $lan13 10.0ms
set lan14 [$ns make-lan "$node20 $node45 $node46 $node47" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node20 $lan14 10.0ms
set lan15 [$ns make-lan "$node27 $node48 $node49 $node50" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node27 $lan15 10.0ms
set lan16 [$ns make-lan "$node24 $node51 $node52 $node53" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node24 $lan16 10.0ms
set lan17 [$ns make-lan "$node4 $node54 $node55 $node56" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node4 $lan17 10.0ms
set lan18 [$ns make-lan "$node25 $node57 $node58 $node59" 100000.0kb 0.0ms]
tb-set-node-lan-delay $node25 $lan18 10.0ms
set lan19 [$ns make-lan "$node12 $node60 $node61 $node62 $node63" 100000.0kb
0.0ms]
tb-set-node-lan-delay $node12 $lan19 10.0ms

$ns rtproto Static
$ns run

# NetlabClient generated file ends here.
# Finished at: 12/4/10 8:01 PM
```

E

Gráficos adicionales de las corridas

En las siguientes secciones presentamos los gráficos completos del análisis del radio de los clusters y de la comparación con la clusterización por latencias de todas las corridas realizadas en los tres tamaños de red.

Los gráficos se encuentran organizados primero por software, luego por tamaño de red y finalmente por tipo de análisis.

Para el caso de Vivaldi sobre BitTorrent, no hicimos corridas en la red **biglans**, por lo que sólo presentamos los resultados para las redes pequeña y mediana.

E.1 Vivaldi P2P

E.1.1. Fourlan

Radios de los clusters

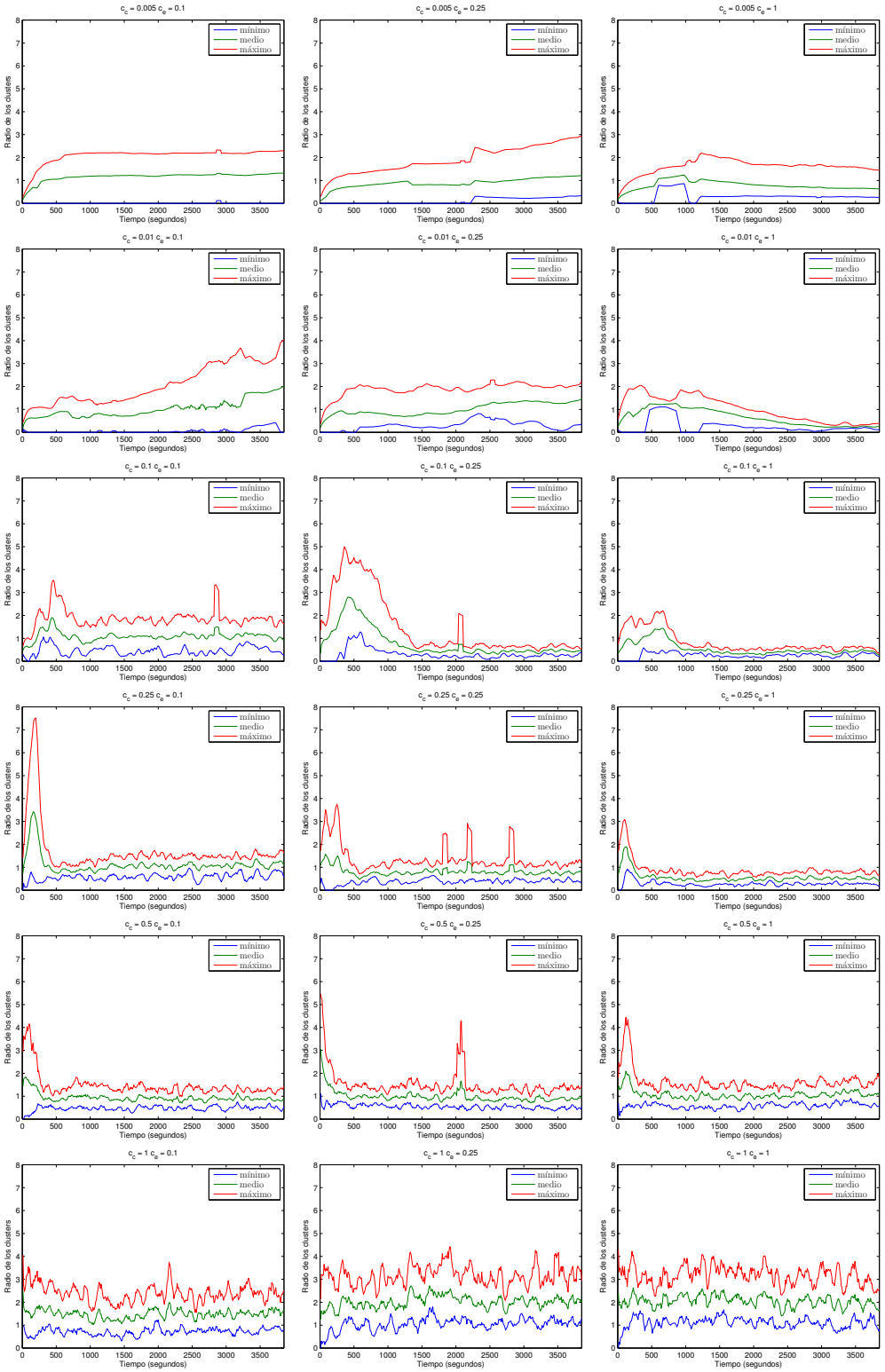


Figura E.1: Radios mínimo, medio y máximo para **ND** = 0 (red **fourlan**)

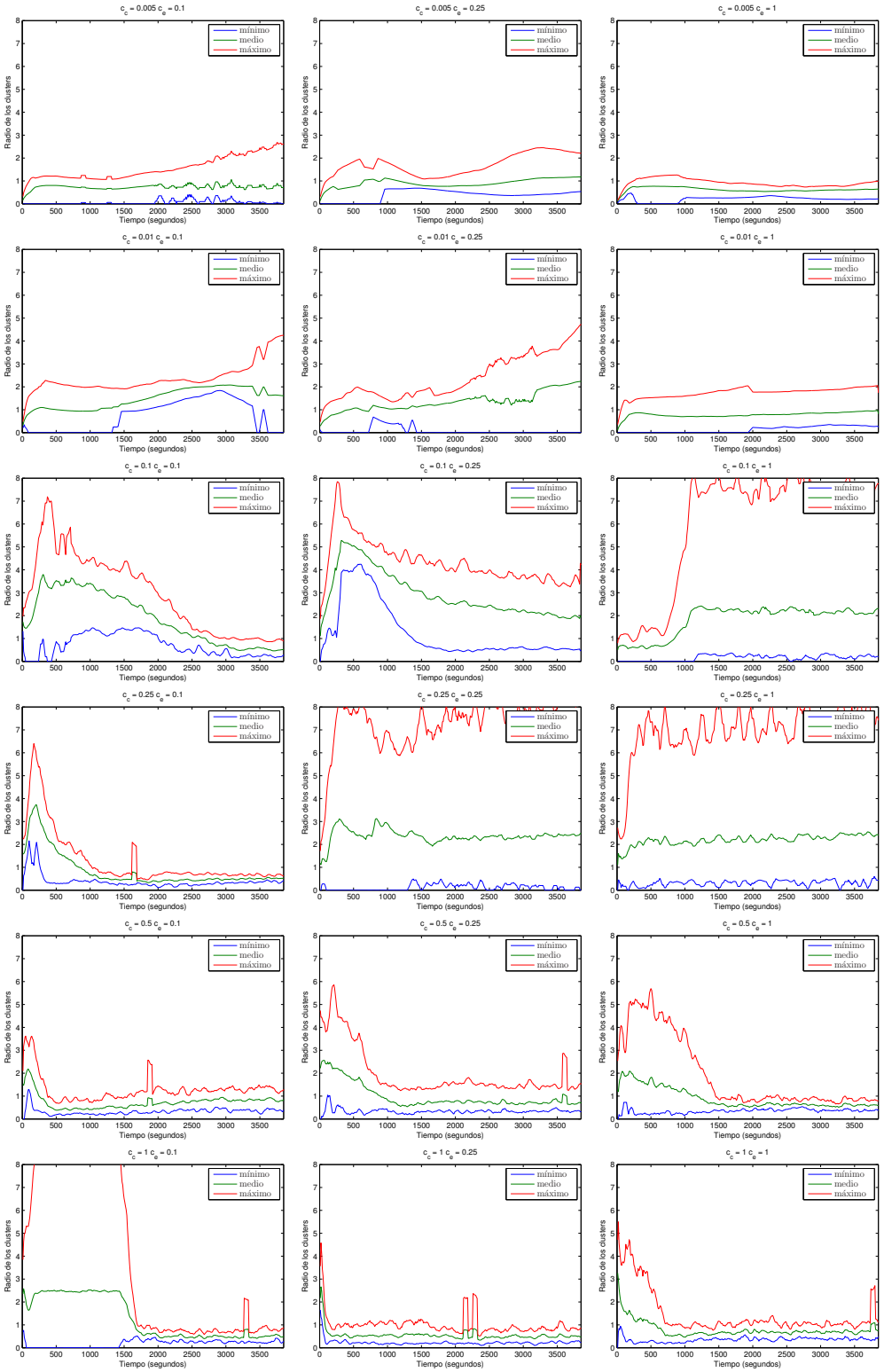


Figura E.2: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 1 (red **fourlan**)

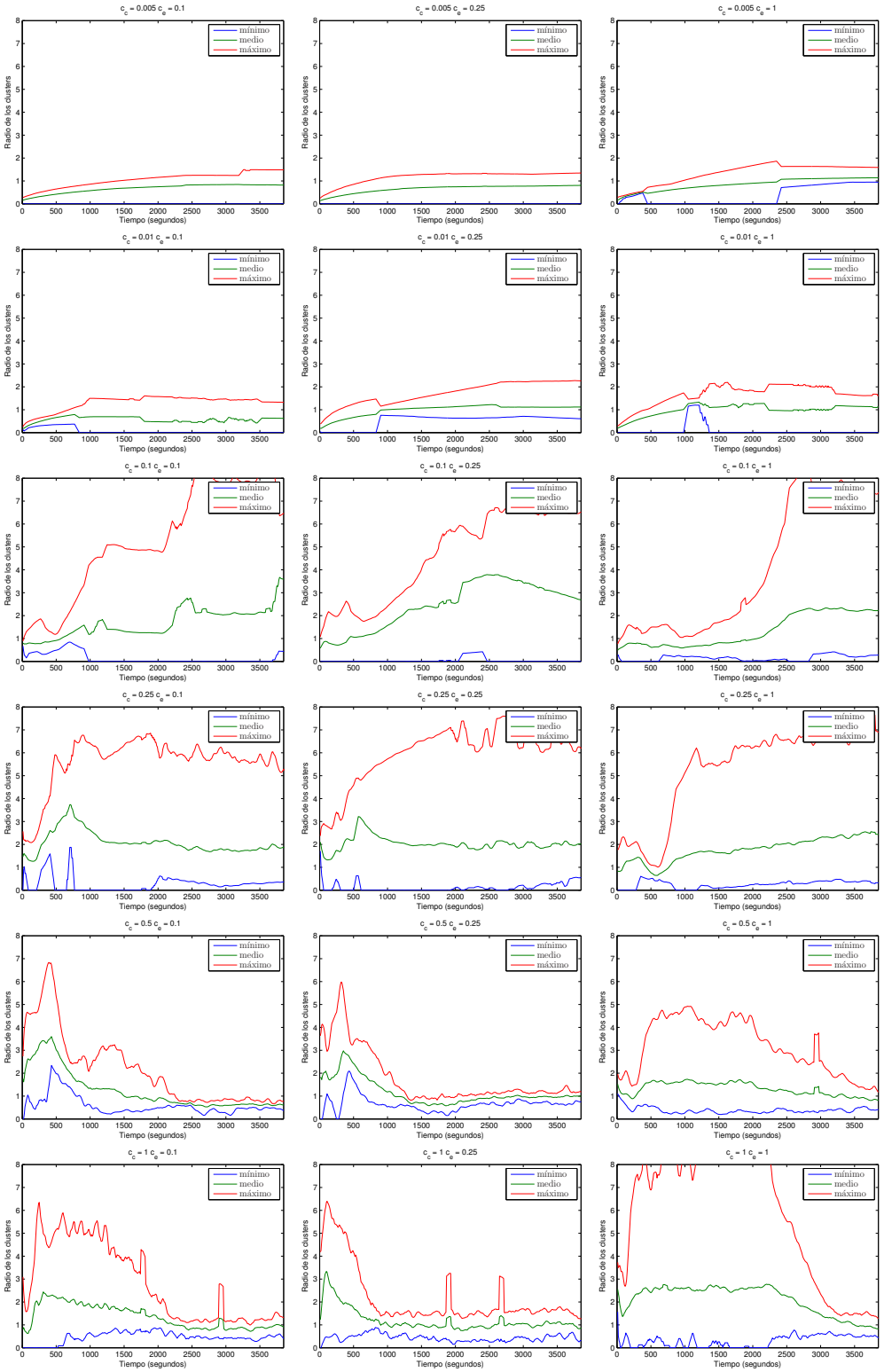


Figura E.3: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 5 (red **fourlan**)

Error en la clusterización

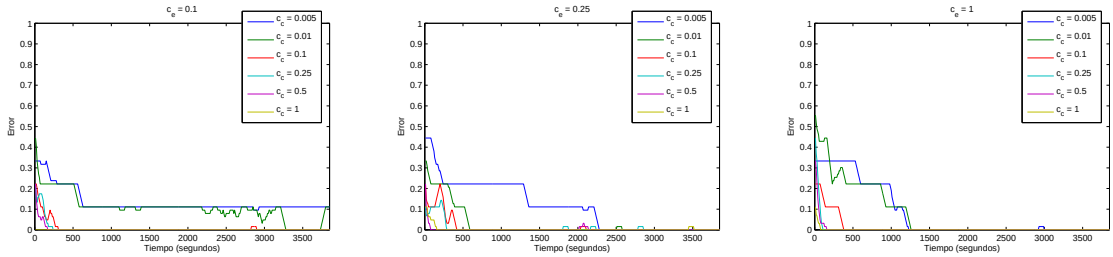


Figura E.4: Error en la clusterización para **ND** = 0 (red **fourlan**)

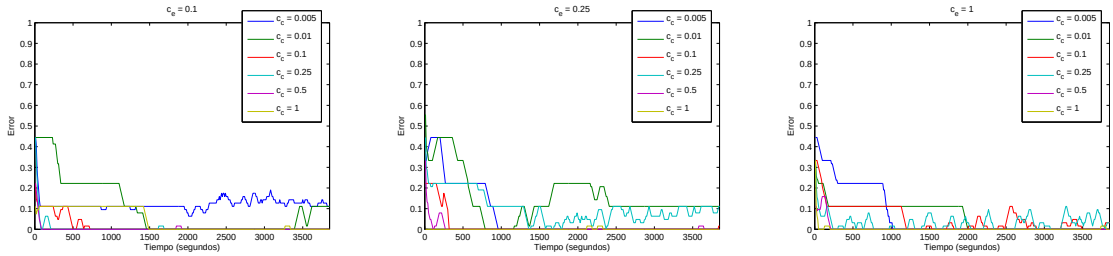


Figura E.5: Error en la clusterización para **ND** = 1, **NDT** = 1 (red **fourlan**)

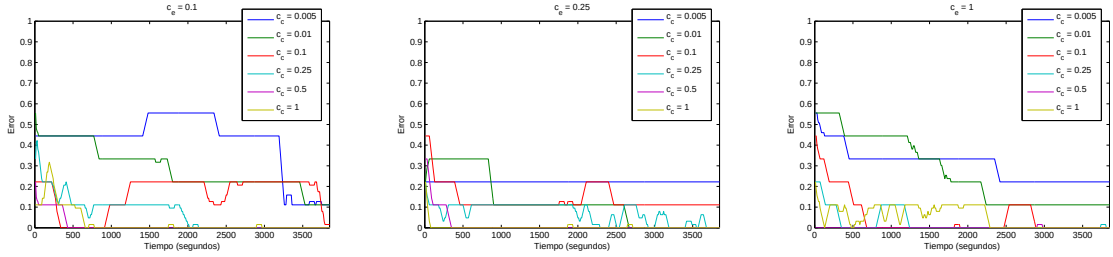


Figura E.6: Error en la clusterización para **ND** = 1, **NDT** = 5 (red **fourlan**)

E.1.2. Tenlan

Radios de los clusters

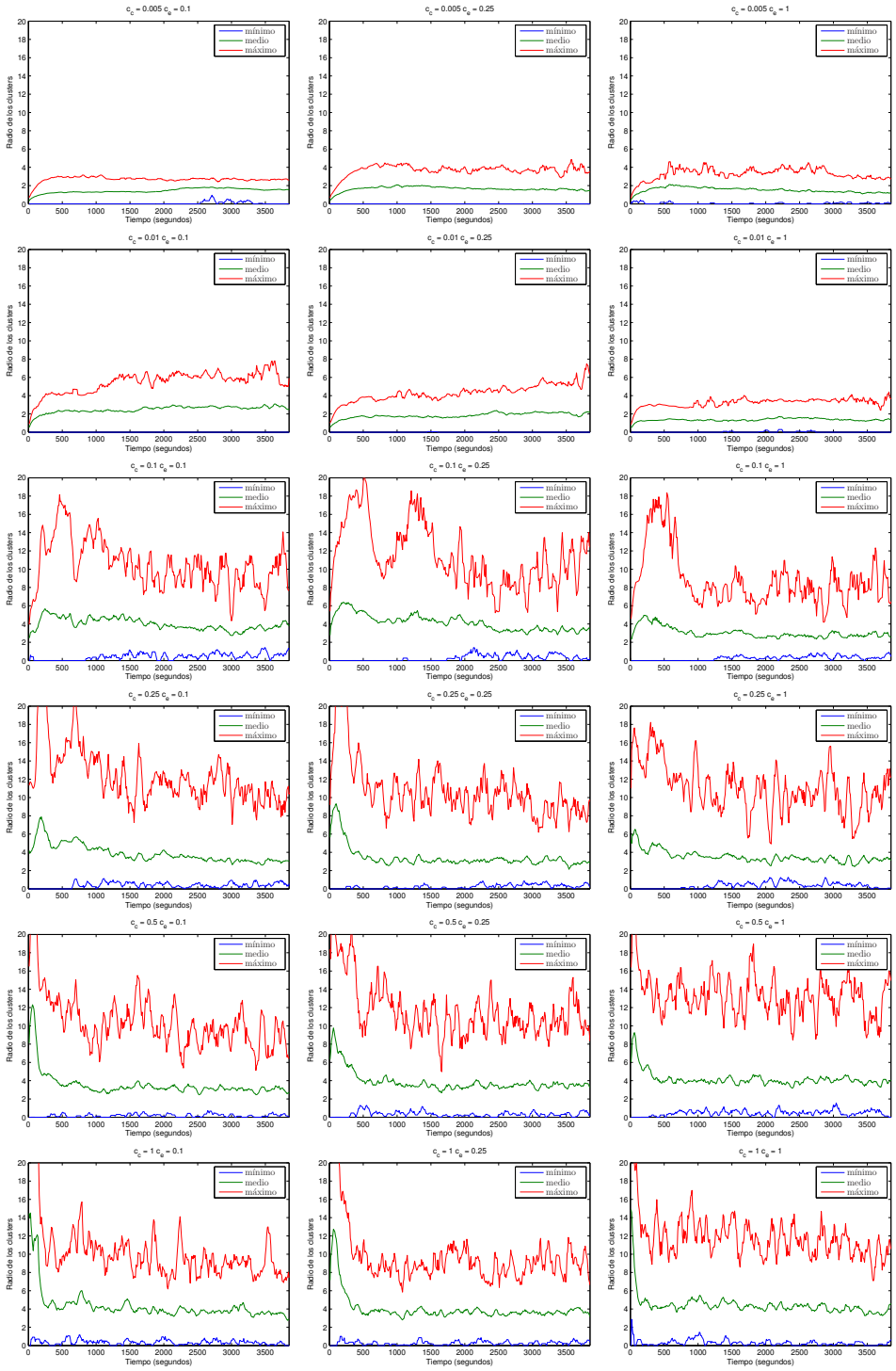


Figura E.7: Radios mínimo, medio y máximo para **ND** = 0 (red **tenlan**)

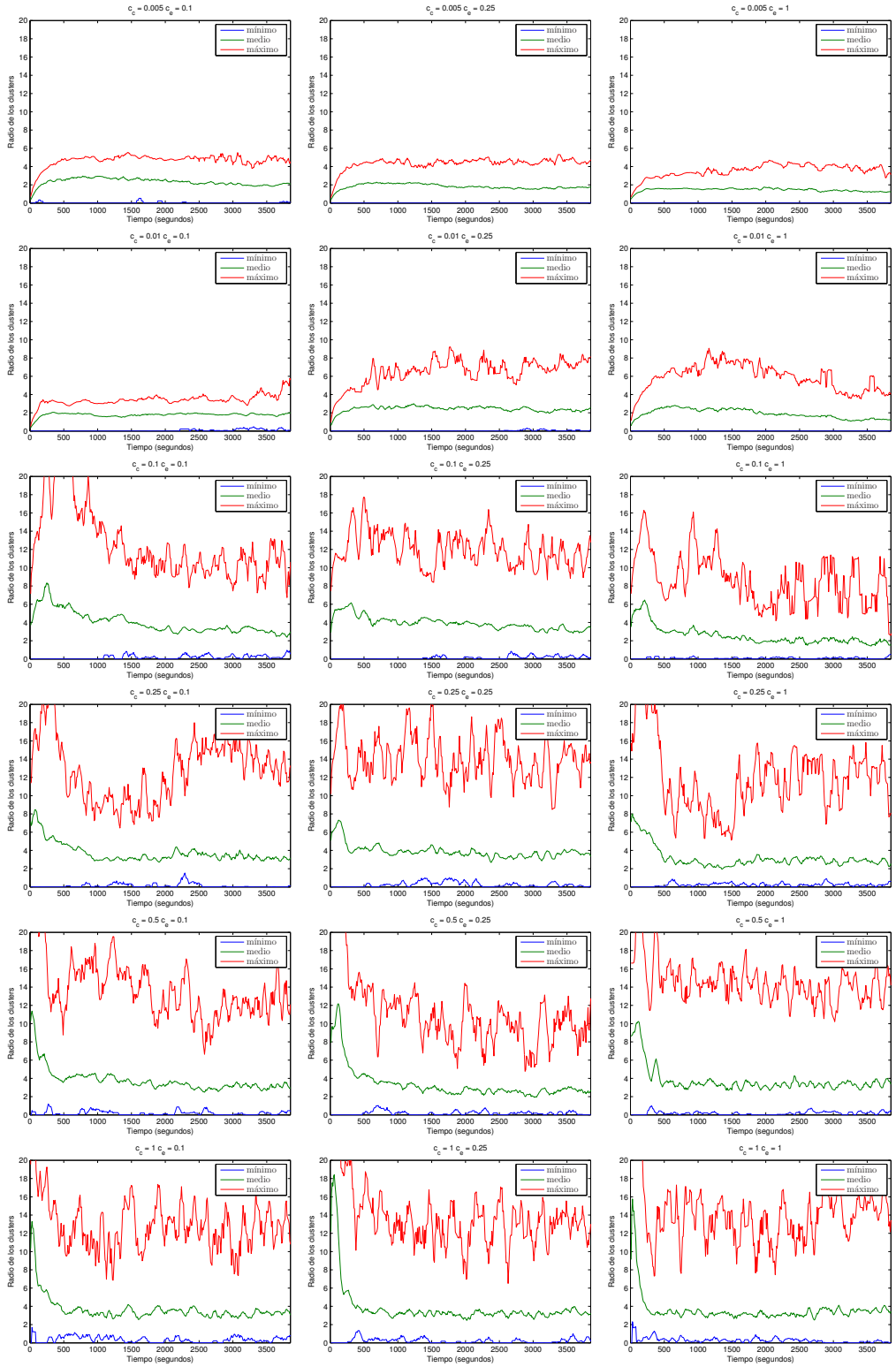


Figura E.8: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 1 (red **tenlan**)

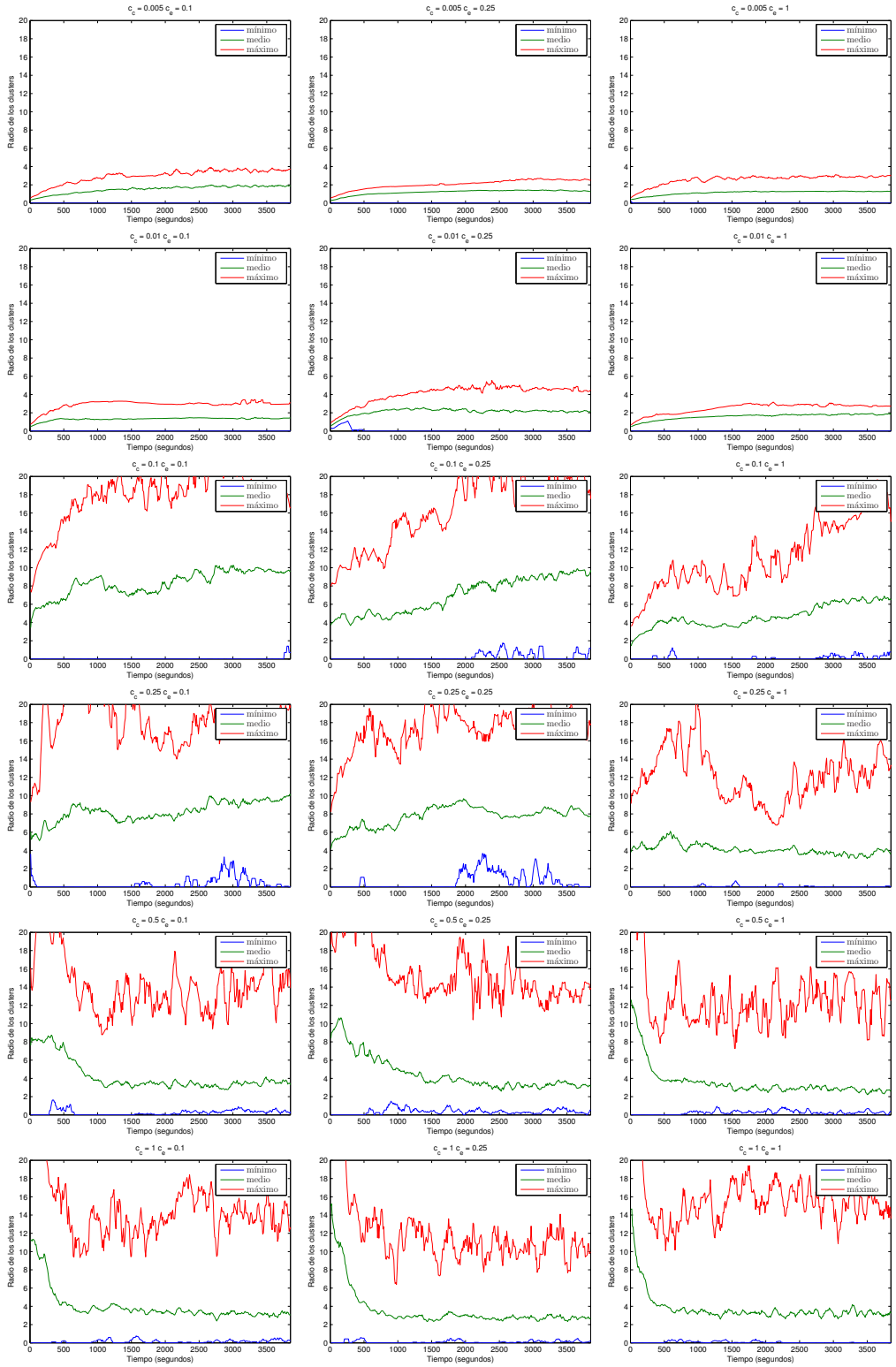


Figura E.9: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 5 (red **tenlan**)

Error en la clusterización

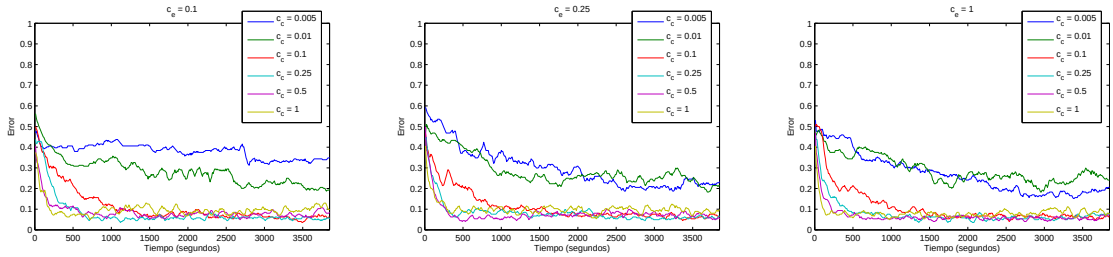


Figura E.10: Error en la clusterización para **ND** = 0 (red **tenlan**)

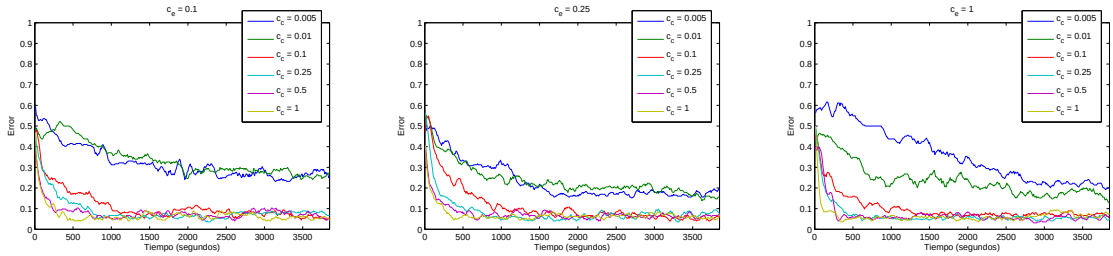


Figura E.11: Error en la clusterización para **ND** = 1, **NDT** = 1 (red **tenlan**)

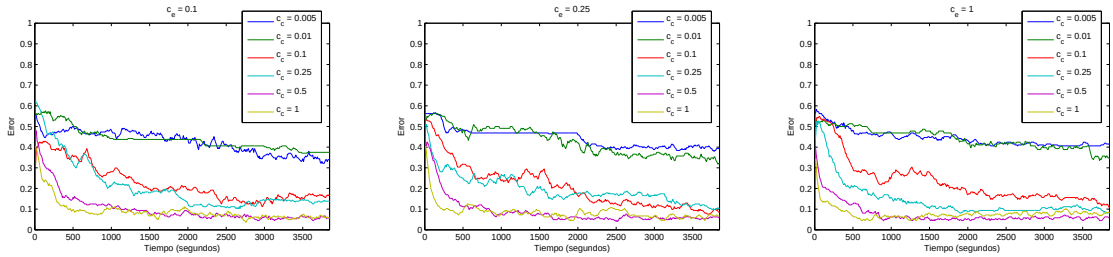


Figura E.12: Error en la clusterización para **ND** = 1, **NDT** = 5 (red **tenlan**)

E.1.3. Biglans

Radios de los clusters

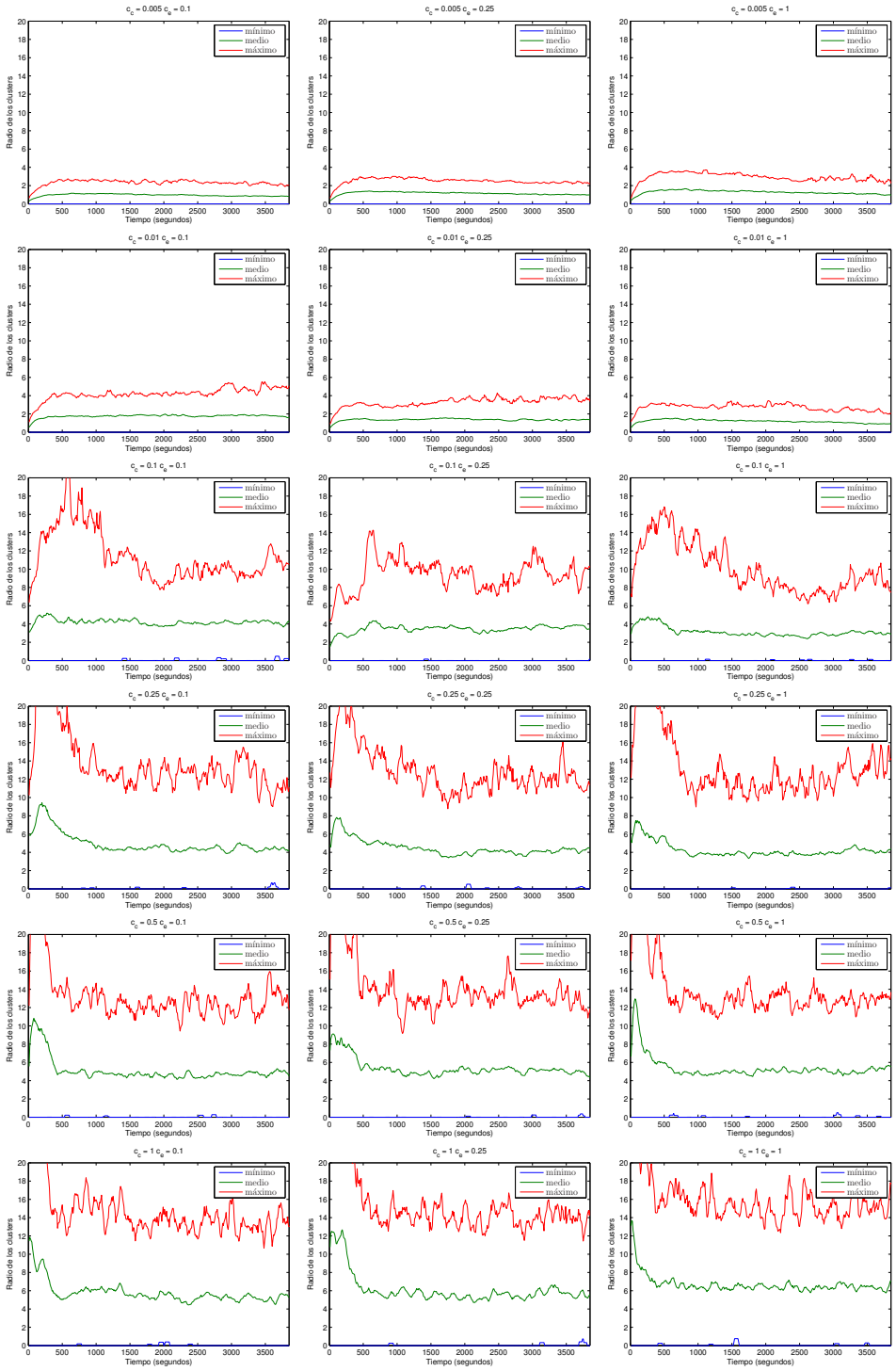


Figura E.13: Radios mínimo, medio y máximo para **ND** = 0 (red **biglans**)

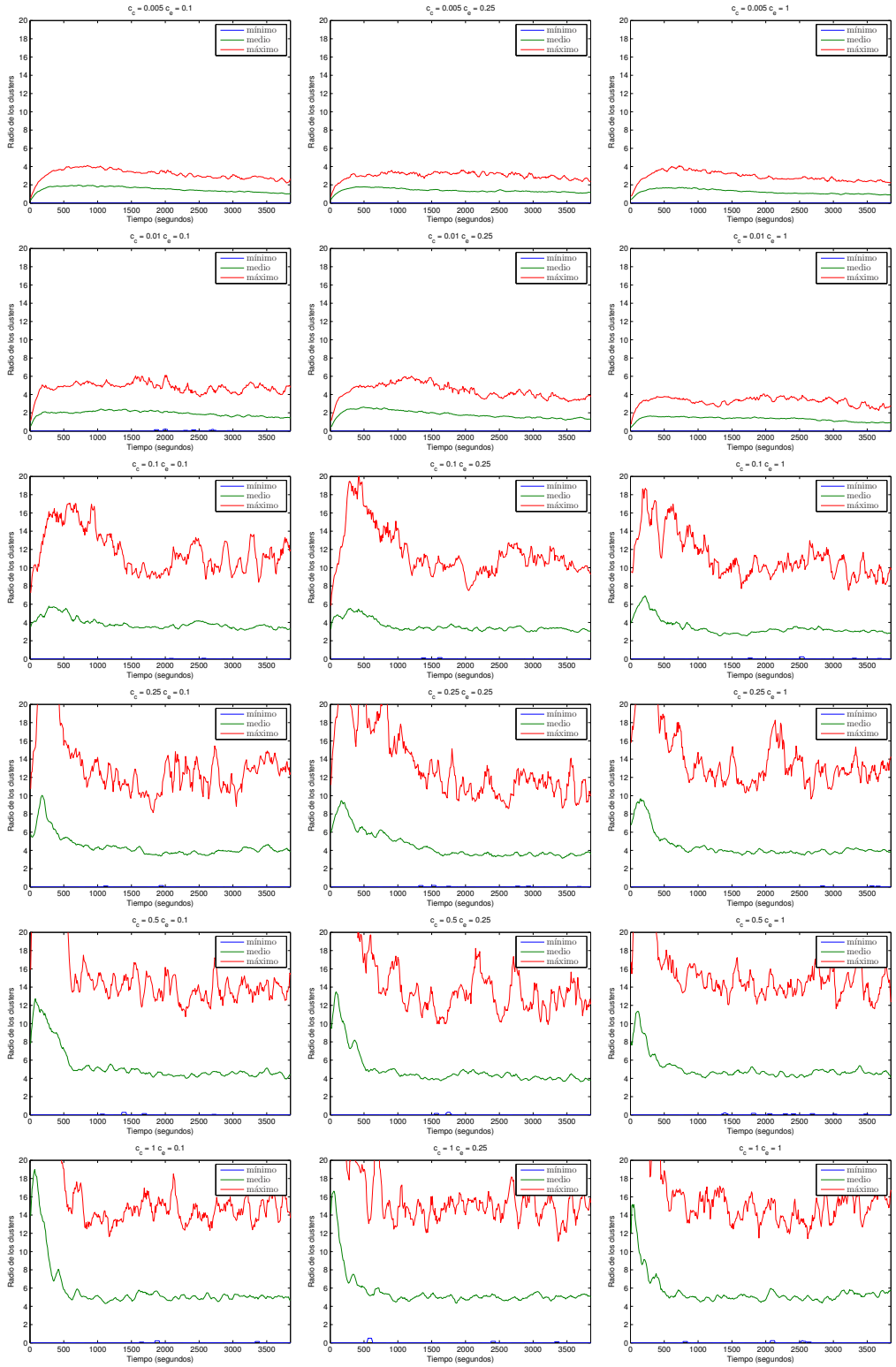


Figura E.14: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 1 (red **biglans**)

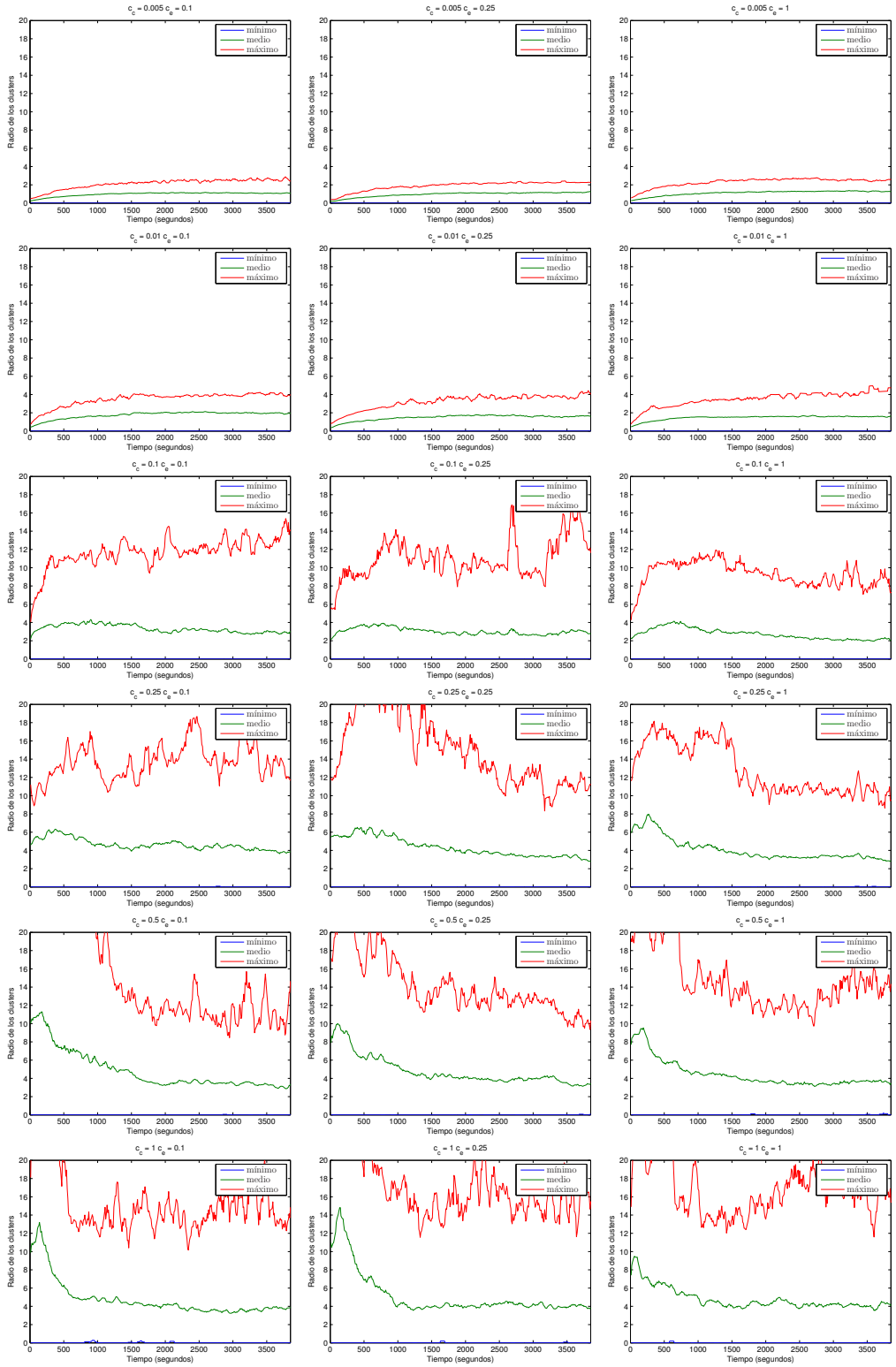


Figura E.15: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 5 (red **biglans**)

Error en la clusterización

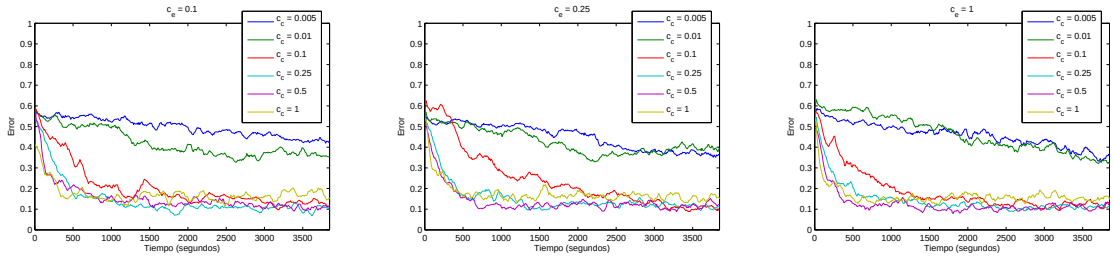


Figura E.16: Error en la clusterización para **ND** = 0 (red **biglans**)

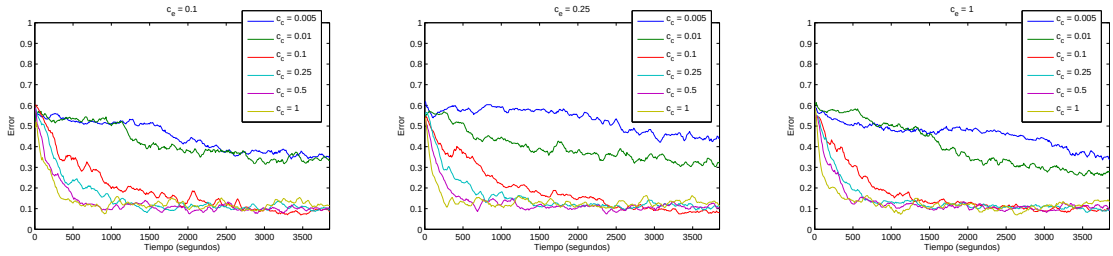


Figura E.17: Error en la clusterización para **ND** = 1, **NDT** = 1 (red **biglans**)

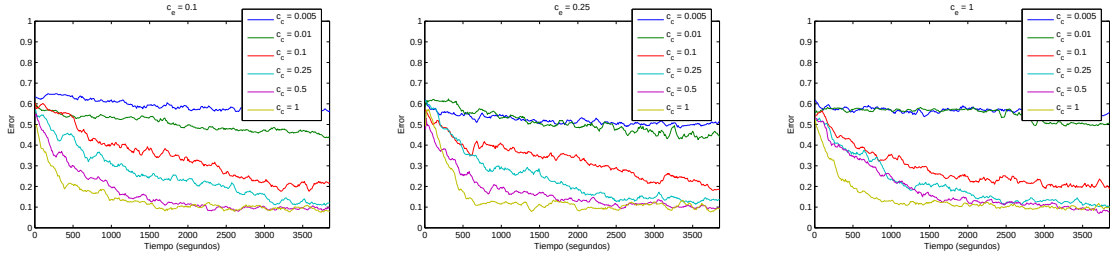


Figura E.18: Error en la clusterización para **ND** = 1, **NDT** = 5 (red **biglans**)

E.2 Vivaldi sobre BitTorrent

E.2.1. Fourlan

Radios de los clusters

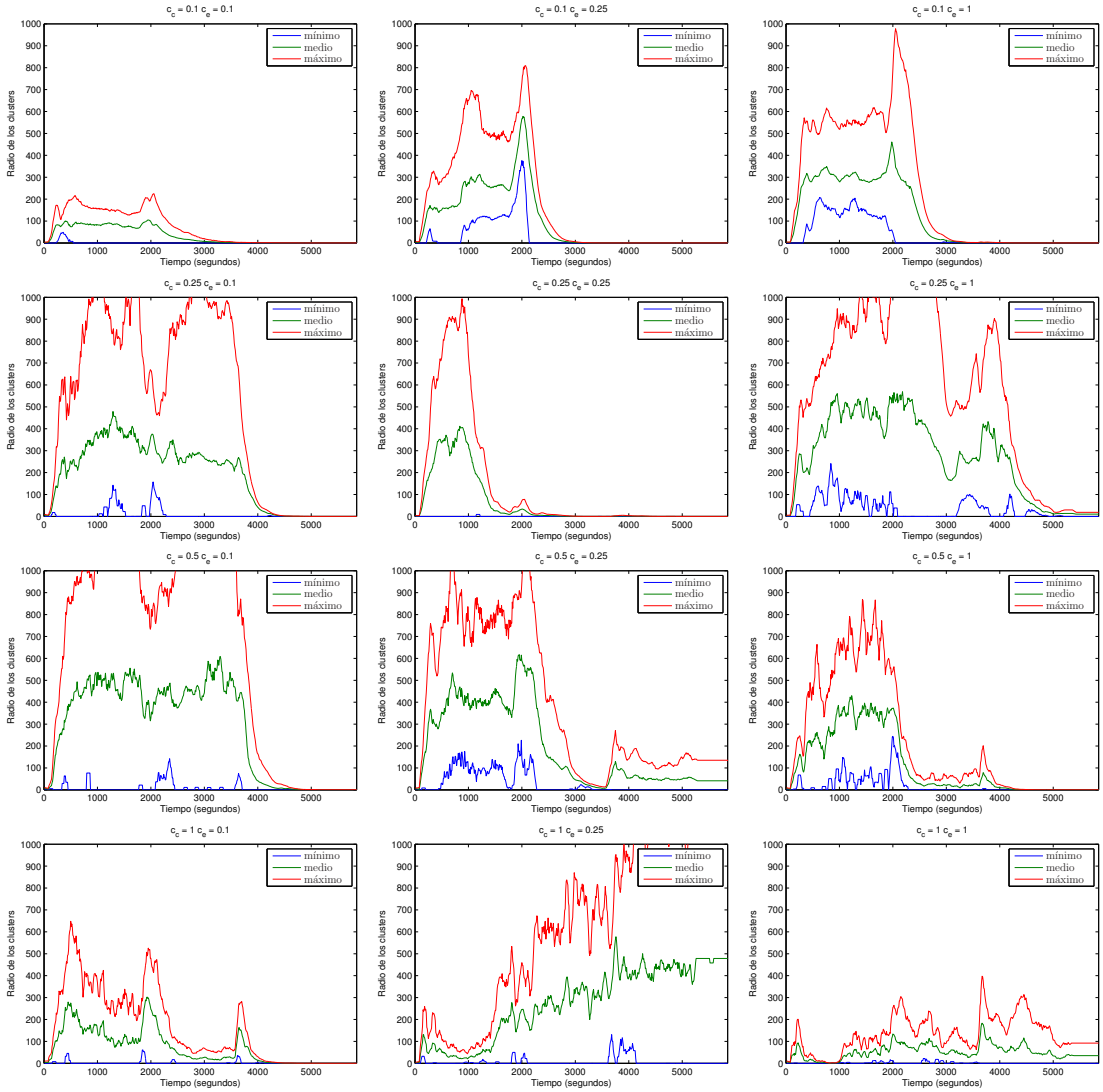


Figura E.19: Radios mínimo, medio y máximo para **ND** = 0 (red **fourlan**)

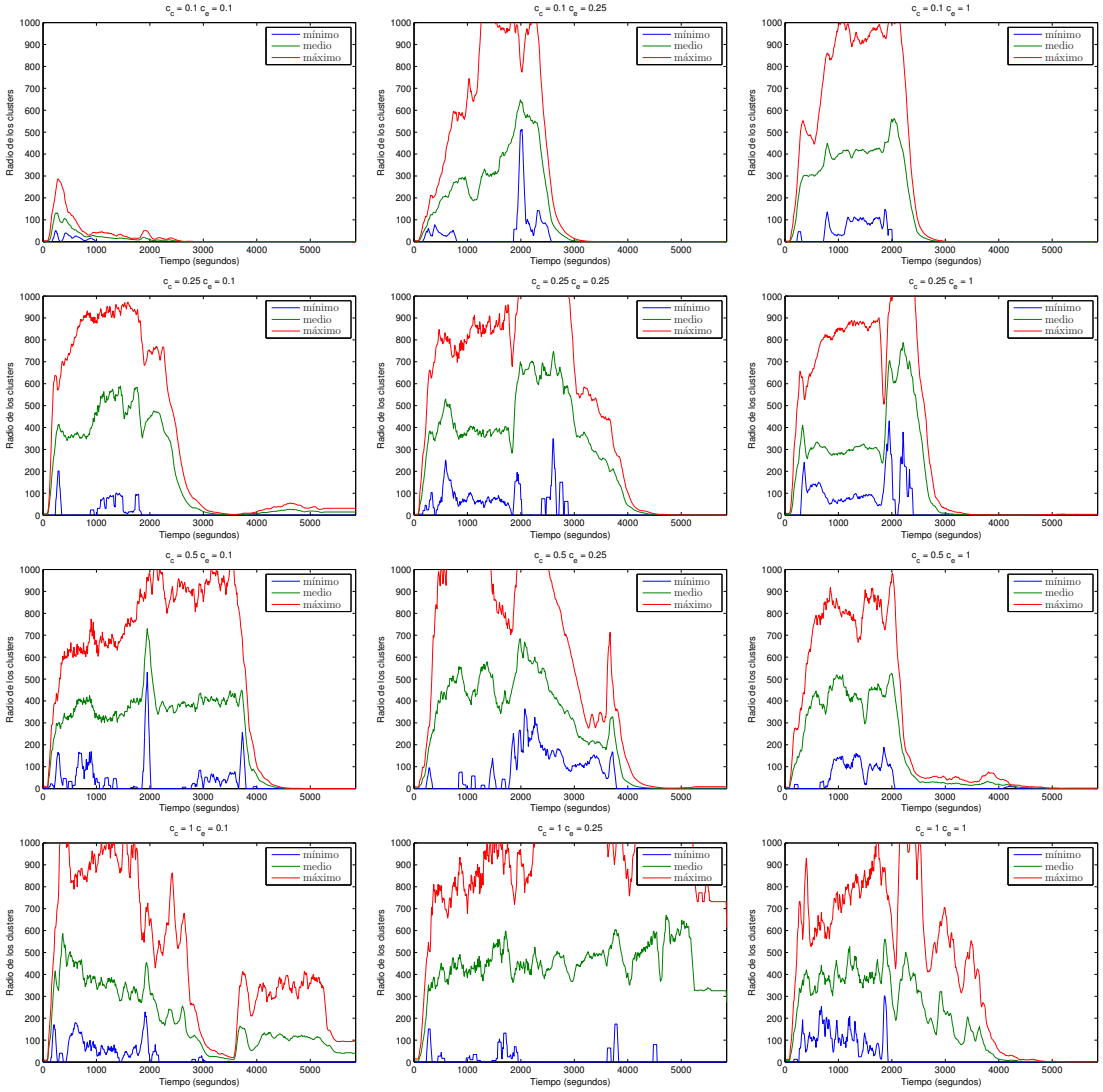


Figura E.20: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 1 (red **fourlan**)

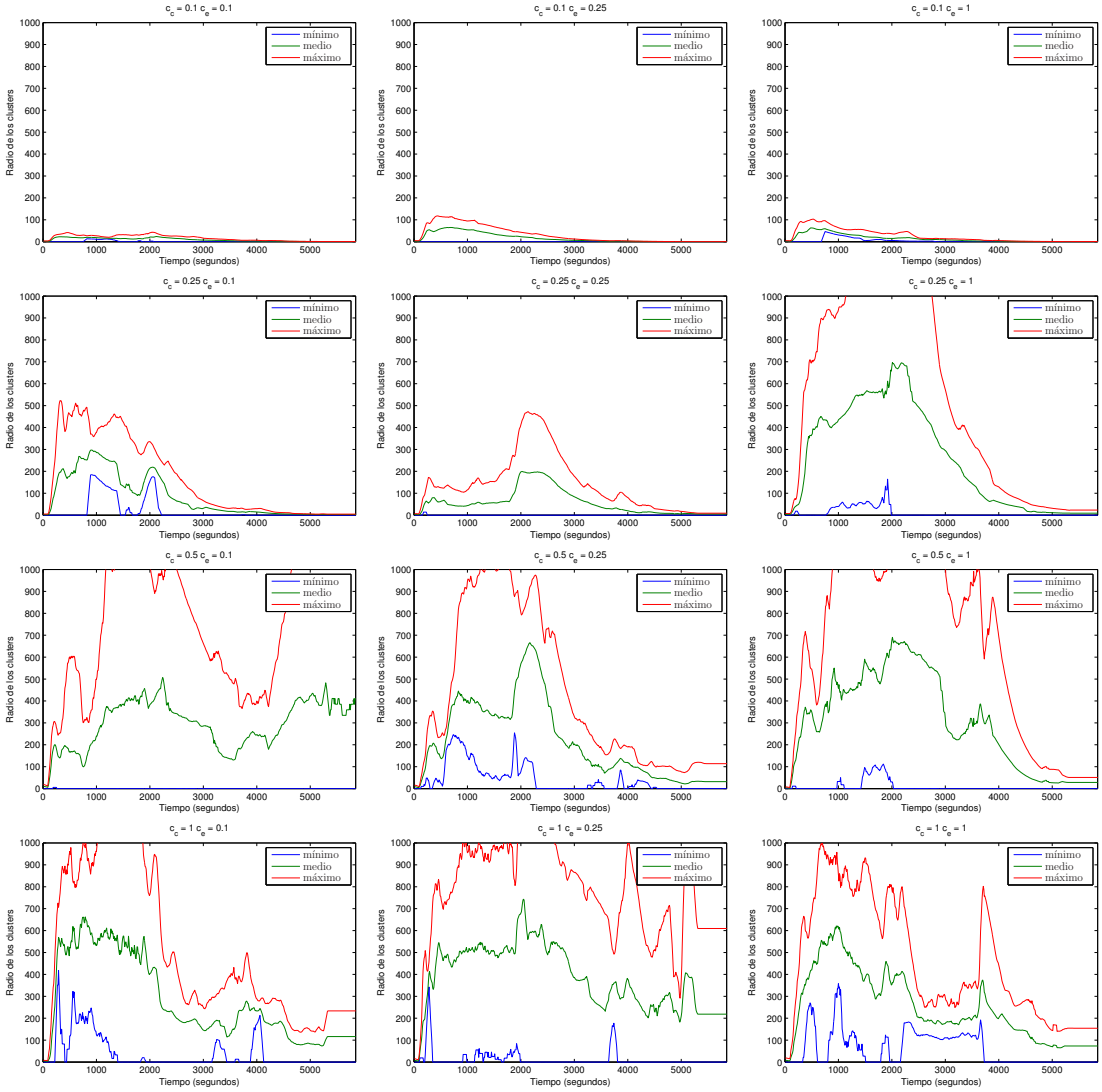


Figura E.21: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 5 (red **fourlan**)

Error en la clusterización

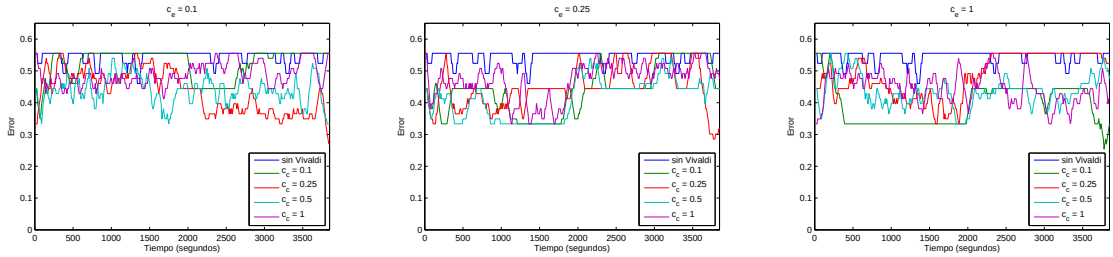


Figura E.22: Error en la clusterización para **ND** = 0 (red **fourlan**)

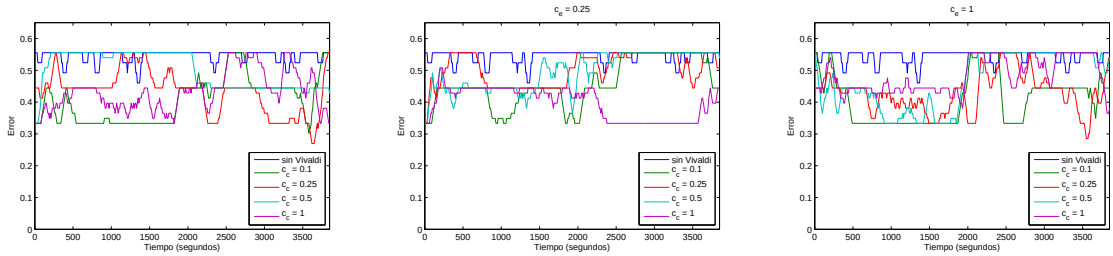


Figura E.23: Error en la clusterización para **ND** = 1, **NDT** = 1 (red **fourlan**)

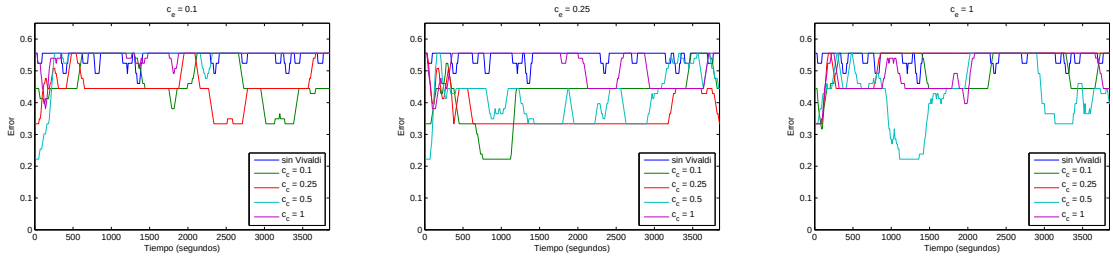


Figura E.24: Error en la clusterización para **ND** = 1, **NDT** = 5 (red **fourlan**)

E.2.2. Tenlan

Radios de los clusters

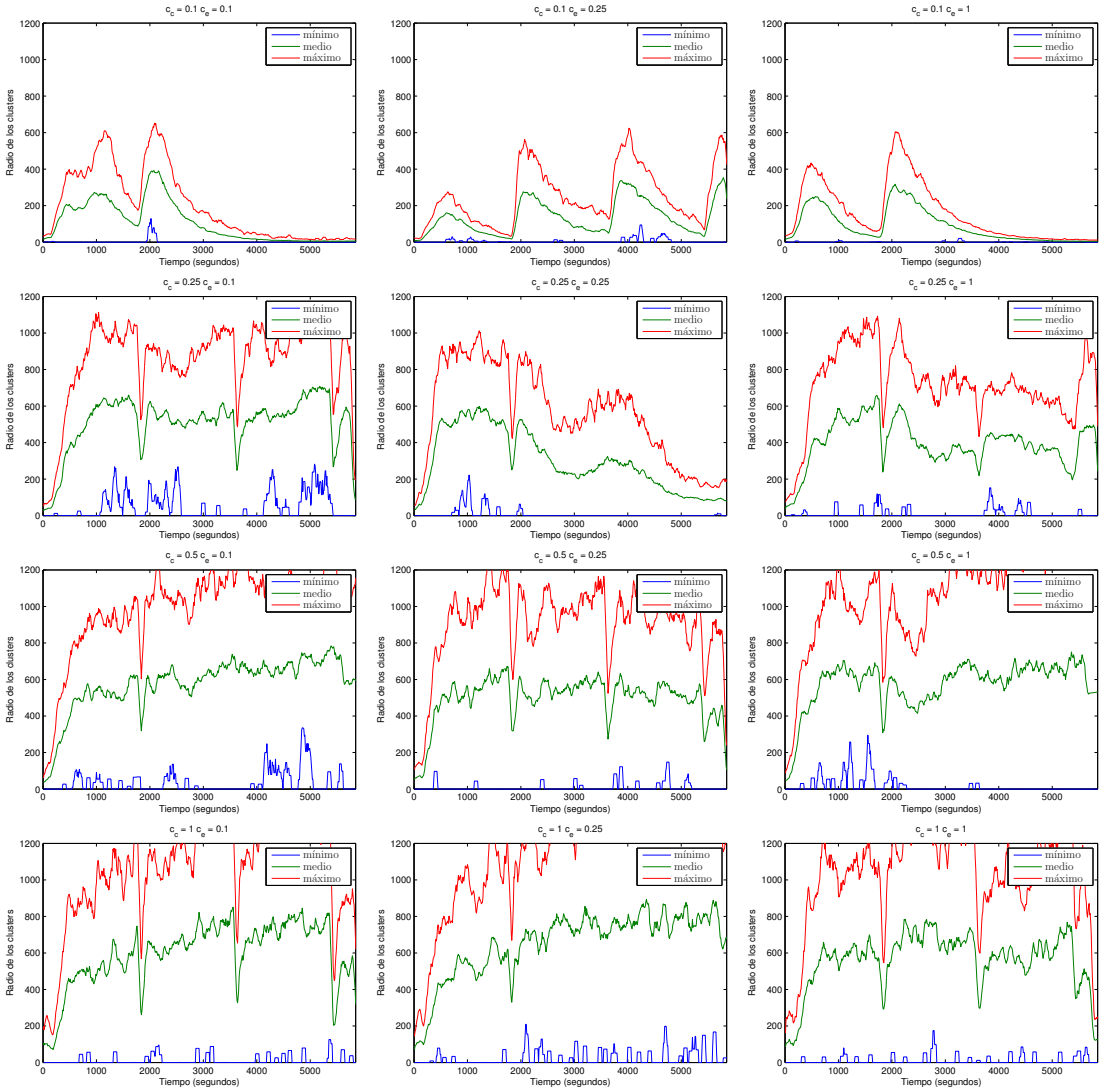


Figura E.25: Radios mínimo, medio y máximo para **ND** = 0 (red **tenlan**)

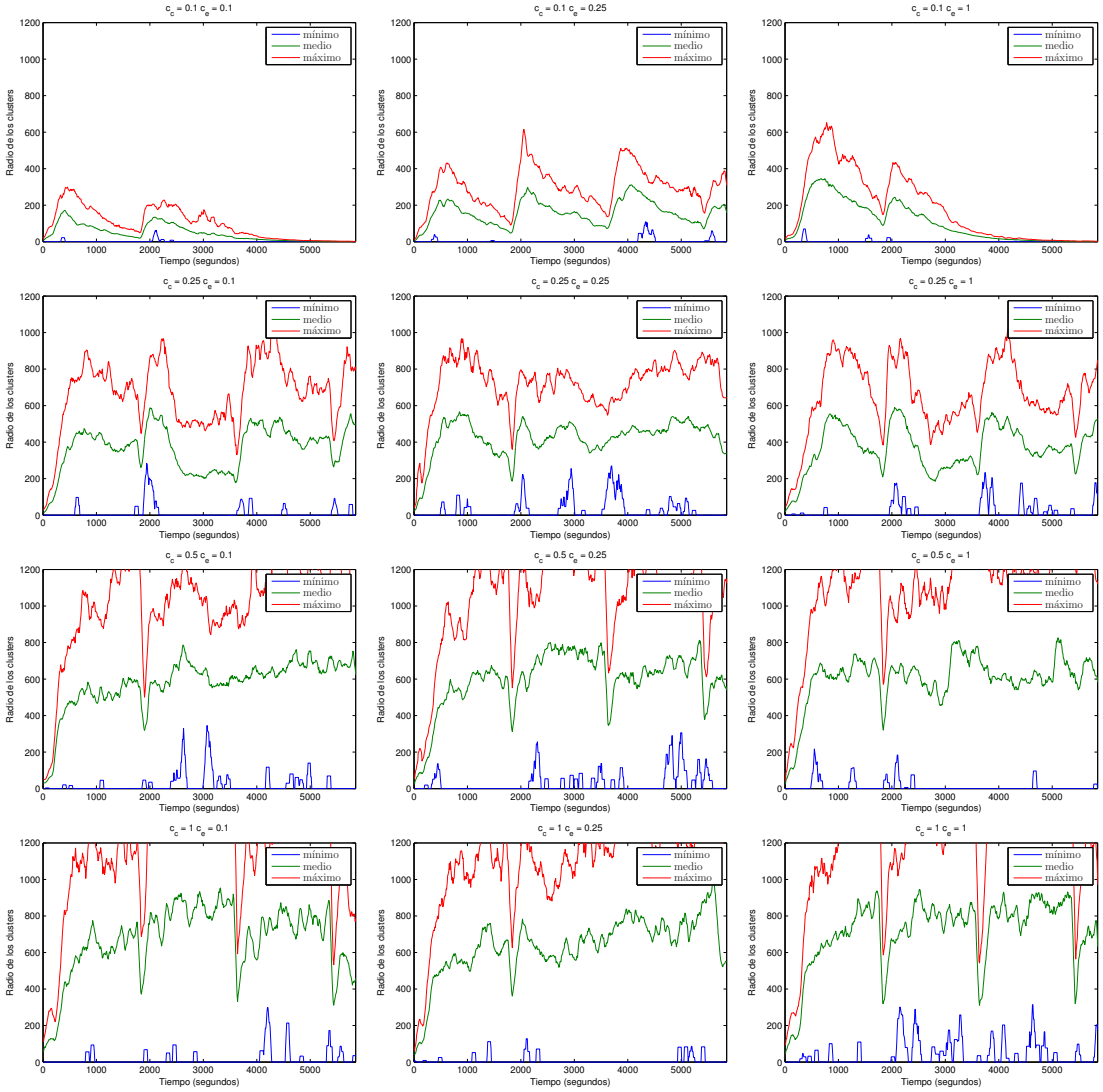


Figura E.26: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 1 (red **tenlan**)

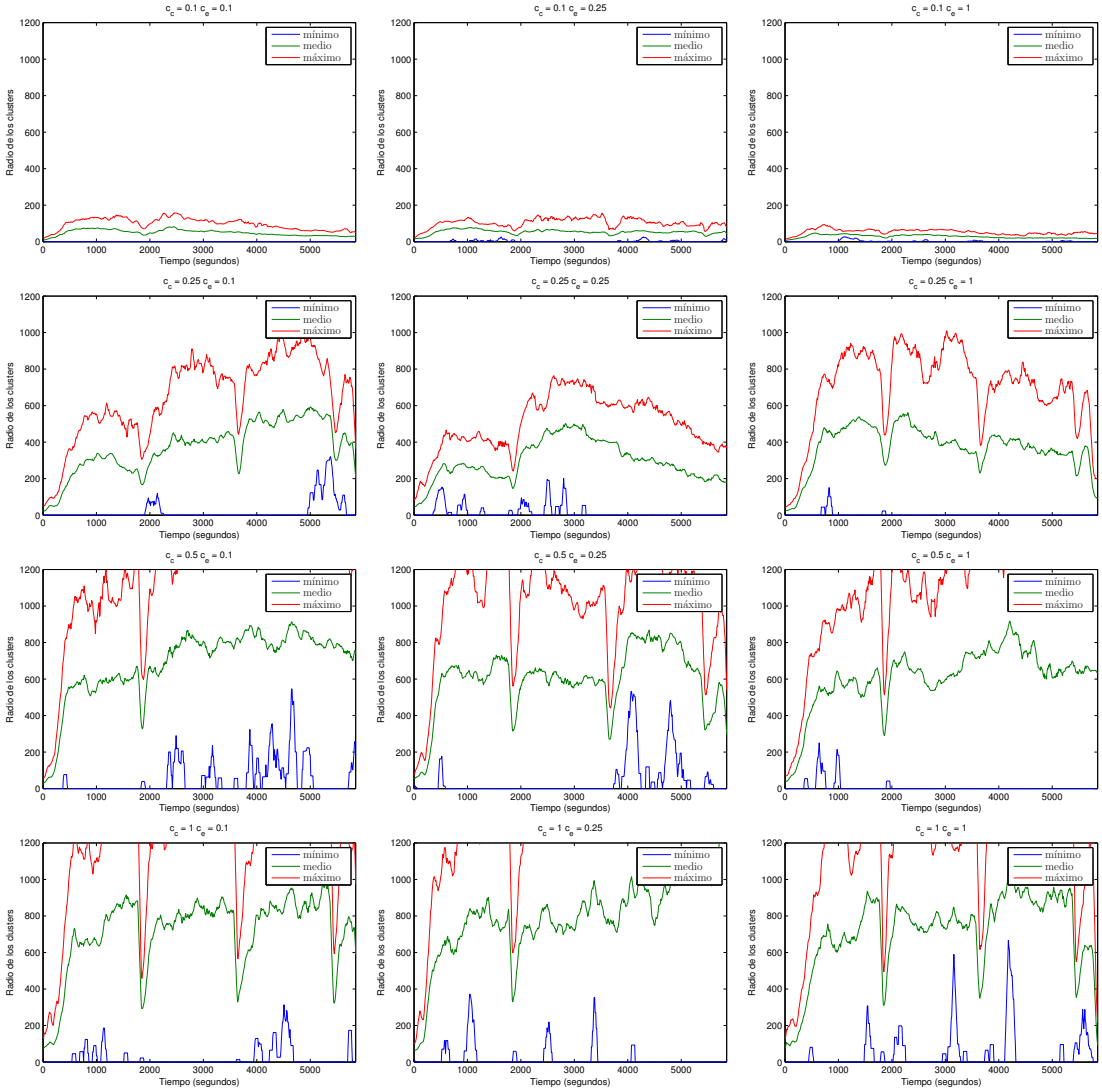


Figura E.27: Radios mínimo, medio y máximo para **ND** = 1, **NDT** = 5 (red **tenlan**)

Error en la clusterización

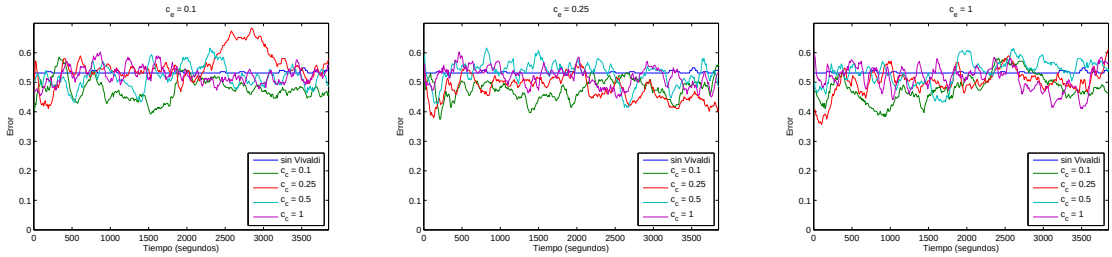


Figura E.28: Error en la clusterización para **ND** = 0 (red **tenlan**)

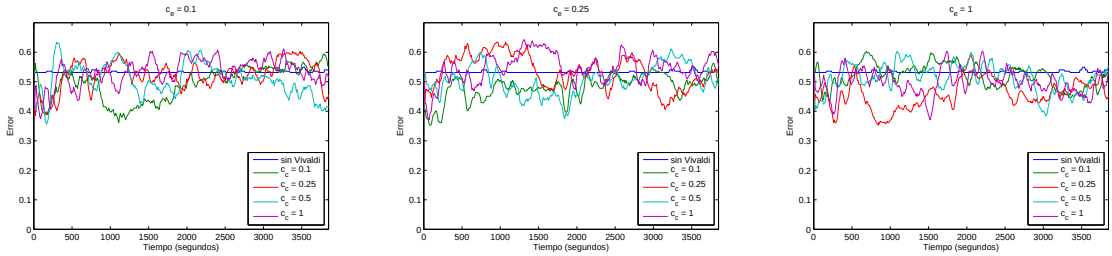


Figura E.29: Error en la clusterización para **ND** = 1, **NDT** = 1 (red **tenlan**)

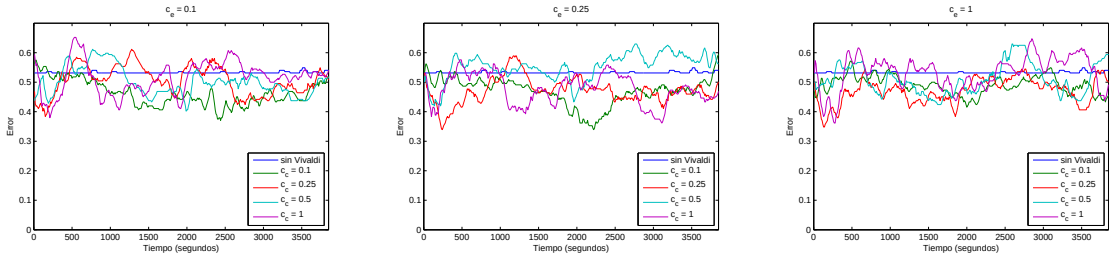


Figura E.30: Error en la clusterización para **ND** = 1, **NDT** = 5 (red **tenlan**)

Agradecimientos

a Esteban Mocskos, por todos los “te lo dije”.

a Claudio Righetti, por permitirme hacer una tesis en este tema después de mucha insistencia.

a los jurados, José Ignacio Álvarez-Hamelin y Agustín Gravano, por la veloz lectura y los comentarios.

a Alejandro Sztrajman, porque cualquier paralelo encontrado en el desarrollo de nuestras tesis puede no haber sido coincidencia.

a Alejandro Burzyn, porque cualquier paralelo encontrado en nuestras tesis puede no haber sido coincidencia.

a Andrés De Nichilo, por jugar tan pero tan bien al Tony Hawk.

a Alex Weil, por aquél gran grupo de TP de dos años y medio de duración.

a Florencia Savoretti, por haberse quedado escuchando más de una hora la defensa cuando todavía duraba más de una hora.

al resto de mis viejos compañeros de cursada: Gabi, Damián (B), Jona, Martiniano, Mati, Nati, Damián (M), Cele.

al resto de mis amigos “de la vida”: Pablo, Ine, Lk, Surak, Janus, Guido, Fede, Marie, Ciro, Alan, Charly.

a David y Nahuel, a quienes molesté en reiteradas oportunidades con preguntas sobre la tesis y que me dieron una mano para mejorar la calidad de la presentación.

a los compañeros de la Secretaría Técnica del DC: Damián, Fer, Andrés, Mateo, Guille.

a toda la gente de Secretaría Administrativa del DC, que me ayudó a que la parte gastronómica de la defensa saliera bien.

a la gente del DF y aledaños que conocí a lo largo de todos estos años de mucho andar por la facultad: Iván, María Emilia, Darío, Nico, Adrián, Mati, Liz, Yago, Marto, Vane.

a las tías Salem, que siempre me preguntaban cómo iba con la tesis.

a Brenda Schneider.

a mis viejos.