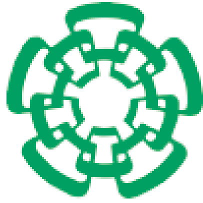


Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

RASupport: soporte flexible, bio-inspirado, auto-configurable y multi-agente para la agregación de recursos en sistemas P2P colaborativos

Tesis que presenta

Damian Isaid Arellanes Molina

para obtener el Grado de

Maestro en Ciencias

en Computación

Directores de Tesis

Dra. Sonia Guadalupe Mendoza Chapa

Dr. Dominique Decouchant

México, D.F. Noviembre 2014

A mis asesores mi más profundo respeto y admiración...

Dra. Sonia Mendoza:
Gracias por todo el tiempo, esfuerzo, dedicación, paciencia y sacrificio que ha implicado la dirección de este trabajo de tesis. Le agradezco infinitamente que me haya apoyado en todas mis decisiones y en la elección de mi tema de tesis, solo de esta manera pude hacer un trabajo que no solo me gustara, sino que me apasionara. Pero sobre todo, gracias por confiar en mí aún en los momentos más difíciles y por dejar fluir mi creatividad en un universo multi-direccional. Más allá de considerarla una gran directora de tesis, la considero un gran ser humano y una excelente amiga.

Dr. Dominique Decouchant:
Gracias por los consejos y las observaciones que mejoraron la calidad de mi trabajo. Le agradezco que me haya conducido por el camino de la simplicidad para lograr el entendimiento general de mi trabajo desde cualquier área del saber humano.

Gracias al Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional por permitirme un lugar, de esta manera me pude dar cuenta de que la ciencia es mi verdadera pasión y, por lo tanto, estaré inmerso en este hermoso ámbito el resto de mi vida.

Gracias al Consejo Nacional de Ciencia y Tecnología por brindarme apoyo económico durante la Maestría para poder lograr una meta más en mi vida.

Gracias a mi familia y amigos por las enseñanzas que he obtenido a partir de la experiencia. Como decía Aristóteles: "La experiencia es conocimiento de lo particular y el arte de lo general"...

Mamá:
Te agradezco por haberme traído al mundo y por estar siempre cuando te necesito. Estoy infinitamente agradecido por todo el apoyo que me has brindado y por esas pláticas en donde siempre desbordamos creatividad. Gracias a ti me he dado cuenta de que los límites no existen, el único límite que tenemos es nuestra imaginación.

Papá:
Gracias por enseñarme que la vida no es fácil y que siempre hay cosas que trascienden más allá de nuestros límites humanos. Los retos que me has impuesto toda la vida me han servido para no quedarme con lo necesario y siempre buscar más allá. Gracias por hacerme ver que la vida en blanco y negro no es suficiente para desplazarnos en este universo multi-direccional, en donde las matices de colores son más variadas de lo que nuestra mente puede percibir. Finalmente, gracias por confiar en mí y hacer posible mi acercamiento con el arte, de esta manera me pude dar cuenta de la estrecha relación que existe entre el arte y la ciencia.

Amigos:
Les agradezco cada momento que he vivido con ustedes, solo así me he dado cuenta de que no todo en la vida es estudiar en alguna institución académica, de que hay cosas que trascienden más allá y de que únicamente nos podemos percatar de ellas cuando miramos el mundo desde una perspectiva pragmática.

Resumen

En los últimos años, se han propuesto alternativas de uso de los sistemas P2P en donde los nodos proporcionan recursos y usan recursos de los demás de una manera colaborativa, con la finalidad de lograr tareas de gran escala. A tales sistemas se les denomina *sistemas P2P colaborativos*, los cuales tienen un importante rol en la agregación de recursos: anunciamiento, descubrimiento, selección, empatamiento y enlace. Actualmente, no existen herramientas de software (e.g., *toolkits*) que permitan a los desarrolladores incorporar soporte para llevar a cabo la agregación de recursos en sistemas P2P colaborativos, empleando algún lenguaje de programación específico. Además, la mayoría de los protocolos P2P actuales que permiten llevar a cabo la agregación de recursos únicamente se encuentran especificados teóricamente, no contemplan los requerimientos y las restricciones de una aplicación y no proporcionan cohesión entre las fases clave de la agregación de recursos: selección, empatamiento y enlace. La principal contribución de este trabajo de tesis es el diseño e implementación en lenguaje Java de un soporte de software denominado *RASupport*, el cual es flexible, auto-configurable, bio-inspirado y multi-agente. Este soporte permite a los desarrolladores de sistemas P2P colaborativos agregar recursos heterogéneos, distribuidos, multi-atributo, dinámicos y estáticos en una arquitectura basada en *super-peers*, la cual está inspirada en el crecimiento de los micelios y es mantenida a través del protocolo Myconet. En esta arquitectura, los *super-peers* seleccionan, empatan y enlazan recursos en nombre de sus *peers normales*, reduciendo así el tráfico en la red. *RASupport* ofrece alta cohesión y bajo acoplamiento debido a que utiliza técnicas de programación generativa y de desarrollo basado en componentes, por lo que fácilmente se pueden agregar, modificar y eliminar componentes sin necesidad de afectar otras partes del diseño. Además, su comportamiento varía en función de los requerimientos y las restricciones de una aplicación cliente, facilitando así el desarrollo de sistemas P2P colaborativos. Para validar el uso y el funcionamiento de *RASupport*, se desarrolló un sistema P2P colaborativo que permite la generación del conjunto de Mandelbrot, haciendo uso de las fases de la colaboración de recursos.

Palabras clave: sistemas P2P, *super-peers*, colaboración de recursos, agregación de recursos, *toolkit*, programación generativa, desarrollo de software basado en componentes, cómputo orgánico.

Abstract

In the last years, usage alternatives of P2P systems have been proposed wherein nodes provide resources and use resources of others in a collaborative fashion, in order to achieve high scale tasks. These systems are called *collaborative P2P systems* and they have an important role in resource aggregation: advertisement, discovery, selection, matching, and binding phases. Nowadays, there are no software tools (e.g., toolkits) allowing developers to incorporate support for resource aggregation in collaborative P2P systems using a specific programming language. Furthermore, most P2P protocols for resource aggregation are only specified theoretically and they neither achieve cohesion among the key phases of resource aggregation (i.e., selection, matching, and binding) nor contemplate restrictions and requirements of applications. The main contribution of this thesis work is the design and implementation of *RASupport* which is a flexible, auto-configurable, bio-inspired, multi-agent, and Java-based support that allows developers to aggregate heterogeneous, distributed, multi-attribute, dynamic, and static resources into collaborative P2P systems. This support uses Myconet protocol to mantain an unstructured topology based on super-peers, which is inspired in the growth of Mycelium. In such a topology, super-peers select, match, and bind on behalf of their normal peers, thereby reducing the network traffic. *RASupport* provides high cohesion and low coupling, since it uses techniques from both generative programming and component-based software development; thus, modules can be added, modified, or removed in an easy way without affecting other parts of the design. Additionally, the behaviour of *RASupport* changes depending on requirements and restrictions of client applications, thus facilitating the development of collaborative P2P systems. To validate both the use and the functionality of *RASupport*, a collaborative P2P system has been developed, which allows the generation of the Mandelbrot set by performing the phases of resource collaboration.

Keywords: P2P systems, super-peers, resource collaboration, resource aggregation, toolkit, generative programming, component-based software development, organic computing.

Índice general

Índice de figuras X

Índice de tablas XV

1. Introducción 1
- 1.1. Contexto de investigación . . . 1
- 1.2. Planteamiento del problema . . . 3
- 1.3. Objetivos del proyecto . . . 5
- 1.4. Organización del documento . . . 6

2. Marco teórico 9
- 2.1. Colaboración de recursos . . . 9
  - 2.1.1. Agregación de recursos . . . 13
  - 2.1.2. Sistemas *Peer-to-Peer* colaborativos . . . 14
- 2.2. Arquitecturas P2P . . . 24
  - 2.2.1. Arquitecturas estructuradas . . . 24
  - 2.2.2. Arquitecturas no estructuradas . . . 30

3. Trabajo relacionado 37
- 3.1. GENI (*Global Environment for Network Innovations*) . . . 37
- 3.2. SWORD . . . 43

4. Análisis y diseño de *RASupport* 53
- 4.1. Análisis de arquitecturas P2P . . . 54
  - 4.1.1. Ventajas y desventajas de las arquitecturas estructuradas . . . 54
  - 4.1.2. Ventajas y desventajas de las arquitecturas no estructuradas . . . 56
- 4.2. Arquitectura bio-inspirada basada en *super-peers* . . . 59
- 4.3. Descripción de recursos . . . 65
- 4.4. Fase de anunciamiento . . . 69
- 4.5. Fase de selección . . . 70
  - 4.5.1. Inundación con agentes de consulta . . . 72
  - 4.5.2. Paseos aleatorios inteligentes (iRandomWalks) . . . 73
- 4.6. Fase de empatamiento . . . 80
  - 4.6.1. Etapa 1: empatamiento de *peers* candidatos . . . 83

4.6.2. Etapa 2: empatamiento de grupos candidatos 86
4.7. Fase de enlace 87
4.8. *Toolkit* de agregación de recursos en sistemas P2P colaborativos: *RAToolkit* 89

5. Implementación de *RASupport* 95
5.1. Módulo principal de *RASupport*: *RASupportMain* 96
5.2. Módulo de configuración y modelado de recursos: *RASupportConfig* 98
5.3. Módulo de simulación de recursos heterogéneos caracterizados por atributos estáticos y dinámicos: ResourceSim 106
5.4. *Toolkit* para la agregación de recursos en sistemas P2P colaborativos: *RA-Toolkit* 109
5.4.1. Implementación y diseño de la base de datos SQLite 117
5.4.2. Soporte Myconet 119
5.4.3. API de anunciamiento de recursos: AdvertisementAPI 122
5.4.4. API de selección de recursos: SelectionAPI 127
5.4.5. API de empatamiento de recursos: MatchingAPI 132
5.4.6. API de enlace de recursos: Binding API 136

6. Resultados y caso de estudio 141
6.1. Resultados de la fase de anunciamiento de recursos 143
6.2. Resultados de la fase de selección de recursos 147
6.3. Resultados de la fase de empatamiento de recursos 154
6.4. Resultados de la fase de enlace de recursos 155
6.5. Caso de estudio: sistema P2P colaborativo para la generación del Conjunto de Mandelbrot 158
6.5.1. Análisis del sistema P2P colaborativo de validación 159
6.5.2. Diseño e implementación del sistema P2P colaborativo de validación 162
6.5.3. Resultados del caso de estudio 168

7. Conclusiones y trabajo a futuro 179
7.1. Recapitulación del planteamiento del problema 179
7.2. Conclusiones 180
7.3. Trabajo a futuro 183
A.1. Glosario 187
A.2. Siglas 192

Índice de figuras

1.1. Contexto de investigación. 2
1.2. Áreas relacionadas. 2
1.3. Organización del documento. 7

2.1. Colaboración de recursos para hacer una simulación con la ecuación de Schrodinger. 10
2.2. Atributos que caracterizan a un recurso que define la capacidad de almacenamiento de un nodo. 11
2.3. Fases de la colaboración de recursos. 13
2.4. Fases de la agregación de recursos. 14
2.5. Nube P2P de compartición de recursos. 18
2.6. Red de sensores para la detección de la carga de tráfico en una autopista. 19
2.7. Radar de alto poder y amplio rango. 20
2.8. Rastreo de 3 km por debajo de atmósfera usando una red de radares. 21
2.9. Localización del cajero automático más cercano a través de una red social móvil. 23
2.10. Un grupo de amigos compartiendo recursos en una cafetería. 23
2.11. Estructura de anillo en Chord. 26
2.12. *Torus* de d dimensiones en CAN. 27
2.13. Diseños estructurados basados en anillos que permiten la agregación de recursos: (a) Multi-anillo – se crea un anillo por cada tipo de atributo; (b) anillo particionado – el anillo se particiona en regiones y a cada región se le asigna un tipo de atributo; (c) anillos sobrepuestos – se colocan múltiples anillos sobre un mismo anillo y cada anillo sobrepuesto corresponde a un tipo de atributo. 28
2.14. Arquitecturas P2P no estructuradas deterministas: (a) Napster y (b) BitTorrent. 32
2.15. Arquitectura basada en *superpeers*. 34

3.1. *Framework* de agregación de recursos de GENI. 38
3.2. Selección de recursos en GENI. 39
3.3. Uso del API GENI AM. 40
3.4. Descubrimiento de recursos en GENI. 41

3.5. Flujo de trabajo en GENI. 43
3.6. Agregación de recursos en SWORD. 44
3.7. Anunciamiento de recursos en SWORD (Memoria=4GB ∧ SO="Linux"): (a) *MultiQuery*; (b) *Fixed*; (c) *SingleQuery* e *Index* 45
3.8. Lenguajes de consultas de recursos en SWORD: (a) SWORD XML; (b) EBNF. 46
3.9. Selección de recursos en SWORD: (a) *MultiQuery*; (b) *Fixed*; (c) *SingleQuery*; (d) *Index*. 47
3.10. Empatamiento de recursos en SWORD. 49
3.11. Flujo de trabajo en SWORD. 51

4.1. Crecimiento de un micelio[1]. 61
4.2. Transiciones de estado del protocolo Myconet [40]. Existen cuatro posibles estados (biomasa, extendiendo, ramificando e inmóvil) por los que puede transitar un *peer*, los cuales dependen de las condiciones de la red y de los valores de fuerza de dichos *peers*. 63
4.3. Diagrama de componentes de *RASupport*. 65
4.4. Estructura de los documentos XML de especificación de recursos dentro de *RASupport*. 67
4.5. Anunciamiento de recursos en *RASupport*. 70
4.6. Protocolo de inundación con agentes de consulta. 73
4.7. Protocolo de paseos aleatorios inteligentes. Los parámetros n (número máximo de *super-peers* que un agente de consulta puede visitar a partir de $SP_{initiator}$), $maxExclusionElements$ (número máximo de elementos que puede tener la lista de exclusión) y *ttl* (valor del TTL del agente de anunciamiento) son configurables dentro de *RASupport*. 75
4.8. Protocolo tradicional de paseos aleatorios. 76
4.9. Regiones de los valores que puede tomar la penalización de un *peer* para un atributo a de cualquier tipo en un grupo g. 82
4.10. Etapas del agente de empatamiento. 83
4.11. Empatamiento entre dos grupos: G1 y G2. 87
4.12. Fase de enlace de recursos dentro de *RASupport* para un conjunto compuesto por dos grupos de *peers* proveedores: $B_q = \{b_c, b_d\}$. 89
4.13. Arquitectura de *RAToolkit*. 90
4.14. Relación entre los agentes y el ambiente. 91
4.15. Comportamiento de los agentes dentro de *RAToolkit*. 92
4.16. Flujo de trabajo de *RAToolkit*. 94

5.1. Diagrama de paquetes de *RASupport*. 96
5.2. Diagrama de clases del submódulo *modulesmanagement*. 97
5.3. Diagrama de paquetes del módulo *RASupportConfig*. 100
5.4. Diagrama de clases del submódulo *resourcesmodel*. 102

5.5. Ejemplo de adición de un nuevo atributo dinámico (espacio libre en disco: *free_hdisk*) en *RASupport*, en el cual se especifica el tipo de atributo (flotante), el recurso al que caracteriza (recurso de almacenamiento: *STORAGE*) y el intervalo ([0,1000000]) utilizado por *ResourceSim* para generar valores aleatorios para dicho atributo. 103
5.6. Diagrama de clases del submódulo *resourcesim*. 107
5.7. Diagrama de clases del submódulo *rsimmonitor*. 108
5.8. Diagrama de paquetes de *RAToolkit*. 110
5.9. Capas de *RASupport*. 112
5.10. Diagrama de secuencia de *RASupport*. 112
5.11. Diagrama de clases del submódulo *ApisManagement*. 113
5.12. Diagrama de clases del submódulo *Common*. 114
5.13. Diagrama de clases del submódulo *DatabasesManagement*. 115
5.14. Diagrama de clases del submódulo *TransportLayer*. 116
5.15. Diagrama relacional de la base de datos de *RAToolkit*. 118
5.16. Diagrama de clases en donde se muestra que *RASupport* se encuentra desacoplado del soporte Myconet. 120
5.17. Ejemplo de una red P2P generada con el soporte Myconet. 121
5.18. Diagrama de componentes que muestra la relación entre un sistema P2P colaborativo, *RASupport*, *RAToolkit* y el soporte Myconet. 121
5.19. Diagrama de paquetes de *AdvertisementAPI*. 122
5.20. Diagrama de clases del módulo *agents*. 123
5.21. Diagrama de clases del módulo *rs*. 123
5.22. Diagrama de estados de los agentes de anunciamiento: (a) agente de anunciamiento inicial y (b) agente de anunciamiento de actualización. 124
5.23. Diagrama de comunicación para el anunciamiento inicial de recursos. . . . 126
5.24. Diagrama de comunicación para la actualización de un recurso dinámico. . 127
5.25. Diagrama de paquetes de *SelectionAPI*. 128
5.26. Diagrama de clases del módulo *agents*. 129
5.27. Diagrama de clases del módulo *protocols*. 130
5.28. Diagrama de estados de un agente de consulta. 130
5.29. Diagrama de comunicación para la selección de recursos. 131
5.30. Diagrama de paquetes de *MatchingAPI*. 133
5.31. Diagrama de clases del módulo *agents*. 133
5.32. Diagrama de clases del módulo *common*. 134
5.33. Diagrama de estados de (a) un agente de empatamiento y (b) un agente de subempatamiento. 135
5.34. Diagrama de comunicación para el empatamiento de recursos. 136
5.35. Diagrama de paquetes de *BindingAPI*. 137
5.36. Diagrama de clases del módulo *Agents*. 138
5.37. Diagrama de estados de un agente de enlace. 138
5.38. Diagrama de comunicación para el enlace de recursos. 139

6.1. Análisis de los algoritmos propuestos para la actualización de recursos dinámicos. . . . 143
6.2. Número de agentes de anunciamiento en un experimento de 50 ciclos con *churn* del 2 %. . . . 145
6.3. Agentes de anunciamiento de actualización en ambientes dinámicos y semi-dinámicos. . . . 146
6.4. Agentes de anunciamiento vs el tamaño de la red. . . . 147
6.5. Análisis de rendimiento de los algoritmos que permiten determinar la similitud que existe entre dos grupos de atributos requeridos. . . . 148
6.6. Correctitud del algoritmo propuesto que permite determinar la similitud entre grupos de atributos requeridos. . . . 150
6.7. Red P2P inicial del experimento que valida el funcionamiento de *SelectionAPI*. . . . 151
6.8. Interfaz gráfica para emitir una consulta de recursos en el simulador de *RASupport*. . . . 152
6.9. Mensajes obtenidos por *RASupport* durante la ejecución del protocolo de paseos aleatorios inteligentes (*iRandomWalks*). . . . 153
6.10. Resultados de la primera ejecución del protocolo de paseos aleatorios inteligentes (*iRandomWalks*). . . . 154
6.11. Resultados de la segunda ejecución del protocolo de paseos aleatorios inteligentes (*iRandomWalks*). . . . 154
6.12. Resultados del experimento que valida el funcionamiento de *MatchingAPI*. 156
6.13. Estado inicial de la red P2P del experimento que valida el funcionamiento de *BindingAPI*. . . . 157
6.14. Resultados del experimento que valida el funcionamiento de *BindingAPI*. . 158
6.15. Colaboración de recursos en el sistema P2P colaborativo de validación que permite la generación del conjunto de Mandelbrot. . . . 162
6.16. Diagrama de componentes del sistema P2P colaborativo de validación. . . 163
6.17. Diagrama de clases del módulo *Messagesmanager*. . . . 166
6.18. Diagrama de clases del módulo *PeerBehaviour*. . . . 167
6.19. Diagrama de comunicación del sistema P2P colaborativo de validación. . . 168
6.20. Estado inicial de la red P2P durante el primer experimento que valida el funcionamiento y uso de *RASupport*. . . . 169
6.21. Resultados obtenidos por *RASupport* en la fase de selección de recursos. . . 172
6.22. Listas iniciales para los grupos *Processing* ($I_{Processing}$) y *Display* ($I_{Display}$) y lista de referencia de *peers* que pueden trabajar entre sí en el grupo *Display* ($K_{Display}$). . . . 173
6.23. Resultados obtenidos por *RASupport* en la fase de empatamiento de recursos.173
6.24. Lista de referencia de *peers* que no pueden trabajar entre sí en el grupo *Display* ($M_{Display}$). . . . 174
6.25. Grupos candidatos para *Display* ($L_{Display}$). . . . 174
6.26. Resultados obtenidos por *RASupport* en la fase de enlace de recursos. . . . 175

6.27. Mensajes enviados en el sistema P2P colaborativo de validación, durante la generación del conjunto de Mandelbrot. 176
6.28. *Frame* con una resolución de 800×600, en donde se despliega el conjunto de Mandelbrot como resultado de la ejecución del sistema P2P colaborativo de validación. 176
6.29. Estado de la red P2P durante el segundo experimento que valida el funcionamiento y uso de *RASupport*. 177
6.30. Mensajes enviados en el segundo experimento durante la generación del conjunto de Mandelbrot. 178

Índice de tablas

2.1. Protocolos P2P estructurados de agregación de recursos. 31
2.2. Protocolos P2P no estructurados de agregación de recursos. 35

4.1. Ventajas y desventajas de las arquitecturas P2P estructuradas 55
4.2. Ventajas y desventajas de las arquitecturas P2P no estructuradas 57
4.3. Arquitecturas estructuradas vs arquitecturas no estructuradas 59
4.4. Notación para la agregación de recursos en *RASupport*. 65
4.5. Atributos mantenidos por *RASupport*. 68
4.6. Restricciones de uso de recursos mantenidas por *RASupport*. 69
4.7. Casos particulares en los que se envían agentes de anunciamiento inicial. . 70
4.8. Restricciones entre *peers* y entre grupos que se pueden especificar en consultas de recursos dentro de *RASupport*. 85
4.9. Agentes involucrados en *RAToolkit*. 92

5.1. Parámetros configurables de *RASupport*. 99
5.2. Tabla comparativa entre StAX, SAX y DOM. 104
5.3. Reglas para la creación de consultas de recursos en *RASupport*. 105
5.4. Reglas para la lectura de documentos XML de consultas de recursos en *RASupport*. 106
5.5. Modificaciones realizadas en el código del soporte Myconet para implementar la fase de anunciamiento de recursos. 125
5.6. Parámetros configurables de *RAToolkit* que definen el comportamiento de *SelectionAPI*. 131

Capítulo 1

Introducción

La *colaboración de recursos* es el proceso en el que los recursos de diversos nodos de un sistema distribuido, son compartidos e interactúan entre sí para lograr grandes y complejas tareas, difíciles de conseguir utilizando un solo nodo. Este proceso comprende siete fases: anunciamiento, descubrimiento, selección, empatamiento, enlace, uso y liberación. Por otro lado, la *agregación de recursos* es un subconjunto del proceso antes mencionado, el cual comprende las fases de anunciamiento, descubrimiento, selección, empatamiento y enlace. Las fases de uso y liberación dependen del propósito específico de las aplicaciones.

En este capítulo se discute el contexto de investigación en el que se sitúa la presente tesis (ver sección 1.1). Posteriormente, se plantea el problema a resolver, el cual está relacionado con la agregación de recursos (ver sección 1.2). De igual forma, se detalla el objetivo general y los objetivos específicos que se pretenden cumplir en la presente tesis (ver sección 1.3). Finalmente se presenta la estructura del presente documento, el cual se encuentra dividido en siete capítulos (ver sección 1.4)

1.1. Contexto de investigación

De acuerdo con la clasificación proporcionada por CSC (*Computing Classification System*)[1] de ACM (*Association for Computing Machinery*), la presente tesis se inscribe en el área de investigación denominada "Bibliotecas de *software* y repositorios" que pertenece al área "Notaciones de software y herramientas" la cual, a su vez, se integra en el área "Software y su ingeniería" (ver Figura 1.1). En el área de investigación denominada "Bibliotecas de *software y repositorios*" se proponen, diseñan e implementan bibliotecas de software relacionadas entre sí (e.g., *toolkits*), con la finalidad de facilitar el desarrollo de aplicaciones pertenecientes a un dominio en particular.

Un *toolkit* es un conjunto de clases reusables y relacionadas, diseñadas para proporcionar funcionalidad de propósito específico y útil, e.g., un conjunto de clases que permite

[1]Disponible en http://www.acm.org/about/class/class/2012.

Figura 1.1: Contexto de investigación.

construir listas, tablas asociativas, pilas y otras estructuras similares. Un *toolkit* no impone un diseño sobre las aplicaciones que lo utilizan, simplemente proporciona funcionalidad que puede facilitar el desarrollo de dichas aplicaciones y evita tener que recodificar funcionalidad común. Debido a lo anterior, se dice que los *toolkits* se especializan en el reuso de código. El diseño de un *toolkit* conlleva un cierto grado de dificultad en comparación con el diseño de una aplicación, debido a que dicho *toolkit* debe trabajar en muchas aplicaciones para ser útil y el programador del *toolkit* no conoce el propósito específico de estas aplicaciones [14].

Además, la presente tesis también está asociada al área de investigación denominada "Arquitecturas *Peer-to-Peer*", la cual es una subclasificación de "Arquitecturas distribuidas" que, a su vez, pertenece al área "Arquitecturas". Finalmente, esta última se inscribe en el área denominada "Organización de sistemas de cómputo" (ver Figura 1.2) de acuerdo con la clasificación proporcionada por CSC de ACM. En esta área se proponen protocolos con el objetivo de optimizar redes P2P (*Peer-to-Peer*) y de solucionar problemas asociados al cómputo P2P.

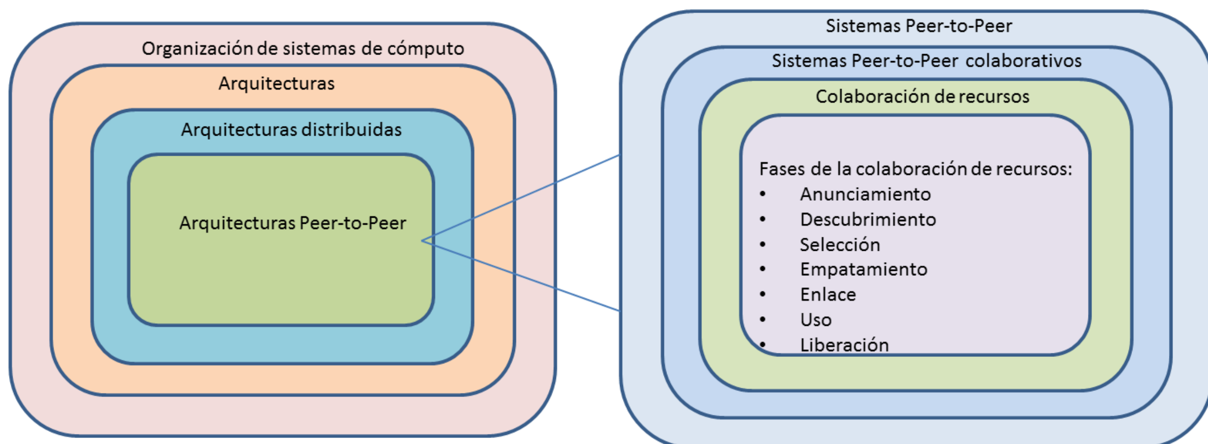


Figura 1.2: Áreas relacionadas.

Actualmente, las redes P2P se clasifican según su grado de centralización, su topología, la generación a la que pertenecen y el grado de protección de la identidad de los nodos que la componen. De acuerdo a su grado de centralización (i.e., la manera en que los recursos de los nodos son indexados) se clasifican en: redes P2P centralizadas, redes P2P híbridas y redes P2P puras. Según su topología (i.e., la manera en que está estructurada la red P2P) se clasifican en: redes P2P no estructuradas y redes P2P estructuradas. Respecto a la generación[2] a la que pertenecen, se clasifican en: primera, segunda y tercera generación. Finalmente, según el grado de protección de la identidad (i.e., de acuerdo a los mecanismos de seguridad que se utilizan para garantizar la privacidad de los nodos) se clasifican en: redes P2P sin características de anonimato, redes P2P basadas en pseudónimos, redes P2P privadas y redes P2P amigo-a-amigo. En particular, la presente tesis se enfocará en redes clasificadas según su topología, debido a que los protocolos actuales que permiten la agregación de recursos en sistemas P2P están basados en este tipo de redes [1]. La agregación de recursos no es propia de los sistemas P2P, ya que también se puede llevar a cabo en otro tipo de sistemas (e.g., plataformas centralizadas), tal y como se discute en el capítulo 3.

1.2. Planteamiento del problema

Actualmente, la mayoría de los protocolos P2P que permiten llevar a cabo la agregación de recursos solo implementan un subconjunto de las fases. Además, no es trivial descubrir, agregar y utilizar recursos dinámicos y heterogéneos que están distribuidos, por lo que es necesario crear soluciones eficientes para direccionar esta problemática [1]. Aunado a lo anterior, actualmente no existen herramientas que faciliten el desarrollo de sistemas P2P colaborativos. En particular, no existen *toolkits* que permitan llevar a cabo la agregación de recursos en este tipo de sistemas, usando algún lenguaje de programación específico.

Aunque se han propuesto protocolos para anunciar, descubrir y seleccionar recursos individuales en una variedad de sistemas P2P, estos proporcionan una forma primitiva de agregación de recursos debido a que no capturan correctamente las relaciones entre recursos y, por ende, son incapaces de seleccionar el mejor grupo de recursos en base a los requerimientos y restricciones de una aplicación. Ninguno de estos protocolos soporta de manera escalable y eficiente los requerimientos de sistemas P2P colaborativos del mundo real. Por lo tanto, se necesitan soluciones de agregación de recursos que sean más eficientes y que se adapten a las necesidades del mundo real [18].

Los protocolos P2P no estructurados de agregación de recursos presentan un alto costo en la fase de anunciamiento y en muchos de éstos ni siquiera hay garantía de encontrar recursos. Todos los protocolos P2P de agregación de recursos existentes tienden a presentar

[2] Cada generación de las redes P2P posee características distintas, las cuales han ido evolucionando a través del tiempo. Actualmente las redes P2P se encuentran en la tercera generación.

fallas al gestionar recursos dinámicos y al momento de realizar consultas multi-atributo; así mismo, la mayoría de estos protocolos únicamente se encuentran especificados teóricamente. En particular, en el esquema de inundación (*flooding*), el empatamiento de recursos solo se lleva a cabo si se inunda con RSs[3] y, en los protocolos *gossiping* y *random walk*, la fase de enlace de recursos solo se realiza si se usan agentes de consulta [1].

En el trabajo de Bandara y Jayasumana [4] se llevaron a cabo evaluaciones de siete arquitecturas de descubrimiento de recursos usando PlanetLab[4], en donde se comprobó que todos los protocolos P2P actuales de agregación de recursos presentan un bajo desempeño en cargas de trabajo reales. El costo del descubrimiento de recursos de estos protocolos fue significativamente alto en comparación con el costo establecido teóricamente en los artículos originales, ya que éstos se basaban en suposiciones convencionales. Por otro lado, todos los protocolos P2P existentes de agregación de recursos solo son aplicables bajo condiciones específicas y no brindan cohesión entre las fases de selección, empatamiento y enlace [1].

La tasa de cambio en atributos dinámicos (e.g., CPU libre, memoria y ancho de banda) es diferente en todos los nodos de un sistema P2P colaborativo, por lo cual algunos atributos cambian frecuentemente. Estos factores contribuyen a que el costo del anunciamiento de recursos sea elevado. Además, no es trivial determinar el tiempo que tardan en cambiar los atributos dinámicos o los RSs. De acuerdo con Bandara y Jayasumana [1], la mayoría de las consultas del *mundo real* solo especifican atributos dinámicos, siendo los más populares la memoria disponible, la carga de CPU y la tasa de transmisión de datos. Las consultas por lo regular solo especifican de uno a siete atributos (e.g., el 80 % de las consultas en PlanetLab especifican únicamente dos atributos [4]), un rango elevado de valores de atributos (e.g., espacio libre en disco de 5-1000GB) y solicitan un gran número de recursos (en las evaluaciones llevadas a cabo por Bandara y Jayasumana, el 42 % de las consultas solicitaron al menos cincuenta recursos [5]). De igual forma, es necesario mencionar que la mayoría de los protocolos P2P de agregación de recursos solo permiten consultas de un solo atributo [18].

El costo del descubrimiento de recursos en protocolos P2P estructurados de agregación de recursos basados en anillo es de $O(N)$, en donde N es el número de nodos; por lo tanto, el costo del anunciamiento de atributos dinámicos es directamente proporcional al número de atributos existentes. Además, los protocolos P2P estructurados de agregación de recursos únicamente se pueden aplicar bajo condiciones específicas y su desempeño es bajo en cargas de trabajo reales [18].

Los recursos disponibles en un sistema distribuido deben ser administrados dinámica-

[3]La especificación de recursos o RS (*Resource Specification*, por sus siglas en inglés) es el documento mediante el cual un nodo anuncia los recursos que posee, detallando sus respectivos atributos y restricciones de uso.

[4]PlanetLab se encuentra disponible de forma gratuita en www.planet-lab.org.

mente para satisfacer los requerimientos de una aplicación con una probabilidad aceptablemente alta. Debido a esto, el descubrimiento y combinación óptima de un conjunto de recursos son actividades extremadamente complejas. Una arquitectura P2P proporciona muchas de las características necesarias para llevar a cabo la compartición de recursos; sin embargo, un simple enfoque de agregación de recursos no es suficiente en aplicaciones sensibles a la latencia, en donde los datos necesitan ser fusionados de tal manera que se satisfagan los requerimientos de tiempo real específicos de la aplicación. Actualmente, existe la necesidad de contar con herramientas que permitan agregar oportunamente conjuntos de recursos en sistemas DCAS de misión crítica (e.g., CASA) [18].

De acuerdo con Bandara y Jayasumana [1], actualmente se requiere una propuesta para agregar grupos de recursos heterogéneos, distribuidos, dinámicos y multi-atributo. Aunque se han propuesto muchas soluciones basadas en P2P para llevar a cabo el descubrimiento de recursos multi-atributo, aún existe la necesidad de contar con una solución que permita llevar a cabo las fases clave de la agregación de recursos (i.e., selección, empatamiento y enlace). Se debe tener cuidado al combinar estas fases ya que, de otra manera, las optimizaciones en una fase podrían provocar conflictos en las siguientes fases. Debido a lo anterior, se propone desarrollar un soporte eficiente que abarque las fases de la agregación de recursos, proporcionando cohesión entre las fases clave. A través de este soporte se podrán establecer consultas que especifiquen múltiples atributos (*multi-attribute queries*) dinámicos y estáticos, un solo atributo (*single-attribute queries*) o simplemente el número de recursos que se requieren (*zero-attribute queries*), sin necesidad de especificar atributos. Por lo tanto, el soporte propuesto facilitará el desarrollo de sistemas P2P colaborativos, utilizando algún lenguaje de programación específico.

> La hipótesis que se origina a partir de la presente tesis es la siguiente: *“es posible desarrollar un soporte que facilite el desarrollo de sistemas P2P colaborativos, empleando algún lenguaje de programación específico. Dicho soporte podrá satisfacer los requerimientos y las restricciones de una aplicación cliente agregando recursos distribuidos, multi-atributo, de un solo atributo, estáticos, dinámicos y heterogéneos. Además, este soporte mantendrá cohesión entre las fases clave de la agregación de recursos: selección, empatamiento y enlace”.*

1.3. Objetivos del proyecto

El objetivo general de la presente tesis es desarrollar un soporte basado en Java que facilite el desarrollo de sistemas P2P colaborativos y que permita agregar recursos distribuidos, multi-atributo, de un solo atributo, estáticos, dinámicos y heterogéneos, manteniendo cohesión entre las fases clave de la agregación de recursos.

Los objetivos específicos de la presente tesis son:

1. Estudiar arquitecturas P2P estructuradas y no estructuradas, con el fin de seleccionar la más adecuada para el soporte propuesto.

2. Implementar un mecanismo destinado a mantener la arquitectura P2P seleccionada.

3. Estudiar sistemas de agregación de recursos, a fin de identificar sus ventajas, sus desventajas y la manera en que efectúan las fases de la agregación de recursos.

4. Identificar y estudiar mecanismos existentes que permitan llevar a cabo las fases de la agregación de recursos, para entender el funcionamiento de cada una de estas fases.

5. Diseñar un *toolkit* que permita la agregación de recursos en sistemas P2P colaborativos y que facilite el desarrollo de este tipo de sistemas.

6. Diseñar e implementar mecanismos P2P que permitan llevar a cabo la agregación de recursos con la finalidad de integrarlos en un único *toolkit*, manteniendo cohesión entre las fases clave de la agregación de recursos.

7. Diseñar un mecanismo que permita describir recursos, con la finalidad de que una aplicación cliente pueda especificar sus requerimientos y sus restricciones a través de una abstracción de alto nivel.

8. Diseñar una notación que facilite la descripción de las fases de la agregación de recursos.

9. Desarrollar un sistema P2P colaborativo para validar el uso y el funcionamiento del soporte propuesto. Dicho sistema permitirá la generación del conjunto de Mandelbrot, mediante la colaboración de recursos heterogéneos y distribuidos en una red P2P.

1.4. Organización del documento

Este documento está estructurado en siete capítulos (ver Figura 1.3). Una vez que se ha presentado el contexto de investigación en donde se ubica la presente tesis, que se ha planteado el problema a resolver y que se han definido los objetivos por cumplir, en el capítulo 2 se expone el marco teórico, en el cual se describen los conceptos esenciales para comprender la presente tesis. Posteriormente, en el capítulo 3 se presenta el trabajo relacionado, en donde se analizan las características, las ventajas y las desventajas de algunos sistemas de software actuales que permiten llevar a cabo la agregación de recursos.

En los capítulos 4, 5 y 6 se presenta la contribución de este trabajo de investigación. En particular, en el capítulo 4 se detalla el análisis y el diseño del soporte propuesto (denominado *RASupport*), el cual permite agregar recursos en sistemas P2P colaborativos. Posteriormente, en el capítulo 5 se presenta la implementación de *RASupport*. En el

capítulo 6 se muestran los resultados de la presente tesis y se presenta un caso de estudio que valida el uso y el funcionamiento de *RASupport*. Finalmente, en el capítulo 7 se presentan las conclusiones y el trabajo a futuro de este trabajo de tesis.

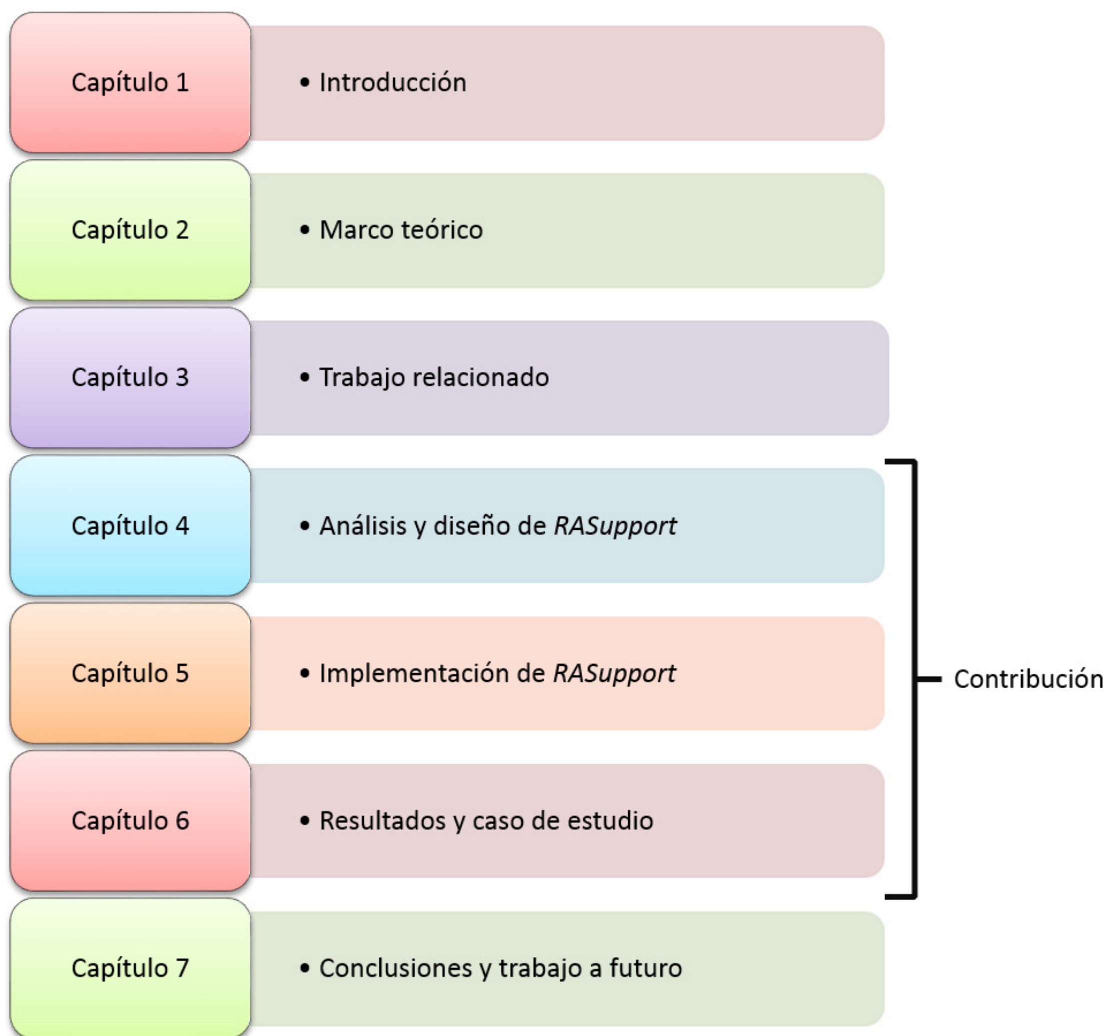


Figura 1.3: Organización del documento.

Capítulo 2

Marco teórico

En este capítulo se presenta el marco teórico asociado a este trabajo de tesis. En primer lugar, se define el concepto de *colaboración de recursos* (sección 2.1), el cual consta de sietes fases (anunciamiento, descubrimiento, selección, empatamiento, enlace, uso y liberación) en el que los recursos de diversos nodos de un sistema distribuido son compartidos e interactúan entre sí para lograr grandes y complejas tareas, difíciles de conseguir utilizando un solo nodo. Además, se describe el concepto de *agregación de recursos*, el cual es un subconjunto de la colaboración de recursos que comprende únicamente las fases de anunciamiento, descubrimiento, selección, empatamiento y enlace. Las fases de uso y liberación dependen del propósito específico de cada aplicación.

La agregación de recursos se puede llevar a cabo en sistemas *Peer-to-Peer* (P2P), aunque no es propia de estos. Los sistemas P2P son sistemas distribuidos sin un control central, cuyos nodos son autónomos y equivalentes en funcionalidad. Los *Sistemas P2P colaborativos* son un subconjunto de los sistemas P2P, cuyos dominios de aplicación son: cómputo en *grid*, cómputo en la nube (nubes P2P), cómputo oportunista, redes de sensores, Internet de las cosas, redes sociales móviles y gestión de emergencias (*emergency management*) [18].

Los protocolos P2P actuales, que permiten llevar a cabo la agregación de recursos, están basados en arquitecturas *overlay*. En la sección 2.2 se presenta una descripción de dichas arquitecturas de acuerdo a su clasificación: estructuradas y no estructuradas.

2.1. Colaboración de recursos

La *colaboración de recursos* es el proceso en el que los recursos de diversos nodos de un sistema distribuido son compartidos e interactúan entre sí para lograr grandes y complejas tareas, difíciles de conseguir utilizando un solo nodo. Un recurso puede ser un ciclo de procesador, capacidad de almacenamiento, ancho de banda de la red, un sensor/actuador, hardware especial, un *middleware*, un algoritmo científico, software de aplicación o

un servicio (e.g., Web y de una nube). La colaboración de recursos es fundamental en sistemas en los que se requiere realizar ciertas tareas que resultan difíciles de llevar a cabo en una sola computadora (e.g., tareas que involucran alto procesamiento de cómputo o que requieren gran espacio en disco) [1].

La colaboración de recursos puede ser útil en dispositivos móviles que carecen de poder de procesamiento o de almacenamiento. Un dispositivo de escasos recursos es aquel que posee poco poder de procesamiento, almacenamiento y/o comunicación; mientras que un dispositivo de altos recursos es aquel que posee alto poder de procesamiento, almacenamiento y comunicación. En la figura 2.1 se muestra a un usuario con un dispositivo móvil de escasos recursos que forma parte de una red P2P. Este usuario desea llevar a cabo una simulación utilizando la ecuación de Schrodinger; para lograrlo, hace uso de los recursos de los nodos que satisfacen los requerimientos (e.g., velocidad de CPU = [2-4]GHz) y restricciones (e.g., latencia de 10ms entre los nodos solicitados) para procesar dicha ecuación.

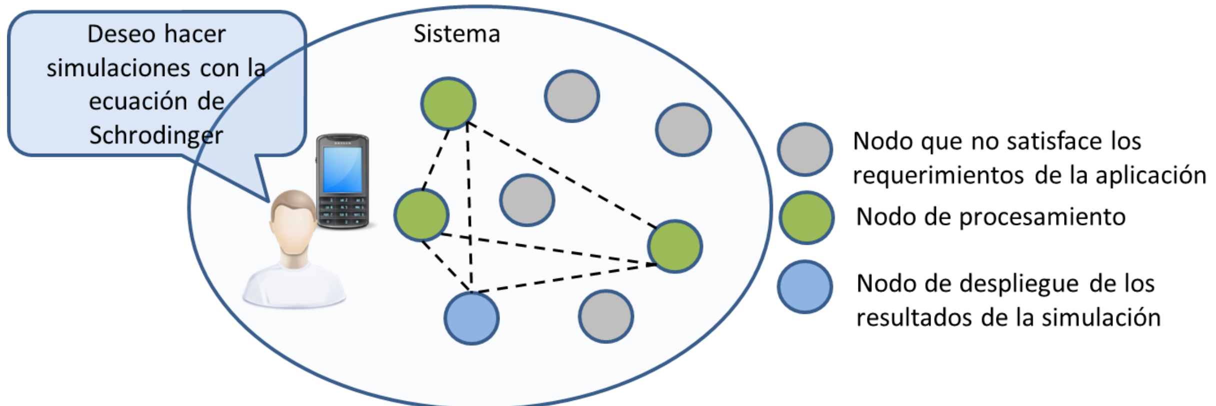


Figura 2.1: Colaboración de recursos para hacer una simulación con la ecuación de Schrodinger.

El poder de cómputo y el espacio en disco a nivel mundial no se encuentran concentrados en algún centro de supercómputo; por el contrario, estos se encuentran distribuidos alrededor de miles de millones de computadoras que pertenecen al público en general. La colaboración de recursos puede hacer uso de estos recursos para llevar a cabo supercómputo científico o simplemente para lograr tareas que serían díficiles de alcanzar utilizando un ambiente controlado (e.g., un clúster). Si la colaboración de recursos se utiliza para llevar a cabo supercómputo científico puede [6]:

1. Fomentar la conciencia pública de la investigación científica actual.

2. Englobar comunidades con intereses en común.

3. Proporcionar a la población (público en general) información sobre el progreso de la ciencia.

Los recursos de un nodo se caracterizan por tener ciertos atributos. Por ejemplo, el espacio disponible en disco duro, el espacio total del disco duro, el tiempo promedio de acceso al disco duro, el tipo de disco duro, el tiempo de lectura/escritura y la tasa de transferencia de datos podrían caracterizar a un nodo de almacenamiento (ver figura 2.2). Entonces, un *atributo* es aquel que caracteriza a un recurso y que tiene asociado un rango de valores; mientras que un *recurso* es un componente físico o virtual que está caracterizado por un conjunto de atributos.

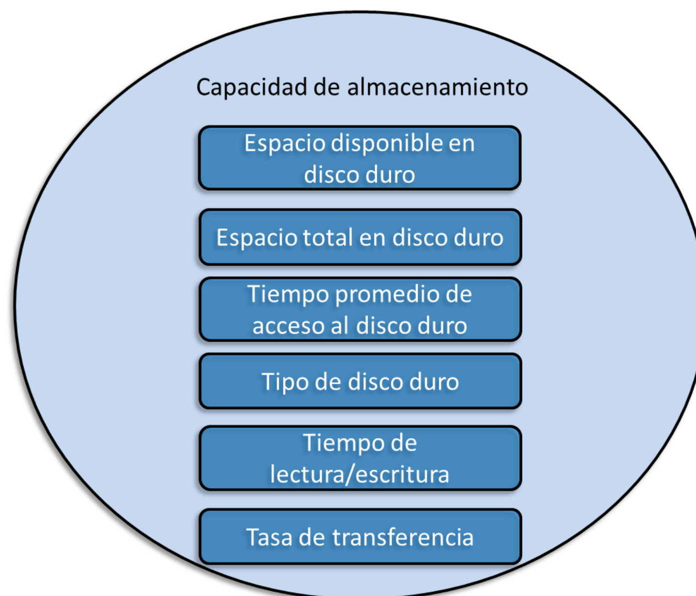


Figura 2.2: Atributos que caracterizan a un recurso que define la capacidad de almacenamiento de un nodo.

Existen recursos caracterizados por un único atributo (i.e., recursos de un solo atributo) y recursos caracterizados por múltiples atributos (i.e., recursos multi-atributo). Los recursos multi-atributo necesitan ser combinados para garantizar requerimientos de desempeño y de calidad de servicio (QoS). Además, algunos recursos son volátiles y voluntarios (e.g., recursos en nubes P2P) mientras que otros son estables y dedicados (e.g., recursos en CASA [9]) [18]. Los recursos dinámicos son aquellos que cambian constantemente (e.g., el ancho de banda de la red), mientras que los estáticos son aquellos que nunca o muy difícilmente cambian (e.g., el número de núcleos de un procesador).

Para localizar recursos en un sistema P2P es necesario realizar una *consulta de recursos*, la cual es el método que se utiliza para obtener un conjunto de recursos que satisfacen los requerimientos y las restricciones de una aplicación en particular. Para llevar a cabo la consulta de recursos, es necesario especificar el conjunto de atributos y sus respectivos rangos de valores, así como expresiones booleanas que sirvan como restricciones de búsqueda. Existen consultas que especifican un solo atributo (*single-attribute query*) y consultas que

especifican múltiples atributos (*multi-attribute query*). De igual forma, existen consultas que especifican únicamente el número de recursos requeridos sin especificar atributos (*zero-attribute queries*); este tipo de consultas son las que se llevan a cabo en *grids*, nubes y nubes P2P, en donde los usuarios están interesados en encontrar un conjunto de nodos independientemente de sus capacidades. En la práctica, las consultas se pueden especificar usando *Extended Backus–Naur Form* (EBNF), *Virtual Grid Description Language* (vgDL) o documentos XML [1].

La colaboración de recursos es un proceso que comprende siete fases: anunciamiento, descubrimiento, selección, empatamiento, enlace, uso y liberación [1], tal y como se puede observar en la figura 2.3. Una descripción más detallada de las fases de la colaboración de recursos puede ser encontrada en el trabajo de Bandara y Jayasumana [1]. En la colaboración de recursos hay dos tipos de nodos implicados: *nodos consumidores* y *nodos proveedores*. Un *nodo consumidor* es aquel que solicita y utiliza un recurso o un conjunto de recursos en particular, mientras que un *nodo proveedor* es aquel que proporciona los recursos que otro nodo necesita. En esta sección únicamente se mencionan a grandes rasgos las siete fases que comprende la colaboración de recursos:

1. **Anunciamiento.** Cada nodo anuncia sus recursos y sus capacidades, usando una *Especificación de recursos* (RSs, por sus siglas en inglés) o más.

2. **Descubrimiento.** Los nodos pueden enviar mensajes de sondeo para descubrir y construir un repositorio local de RSs con la finalidad de acelerar la resolución de una consulta, ya que dicho repositorio puede ser utilizado para dar seguimiento a las relaciones entre los recursos (e.g., latencia, ancho de banda y confianza).

3. **Selección.** Esta fase consiste en seleccionar un grupo(s) de recursos que satisfacen los requerimientos de una aplicación en particular. Dichos requerimientos son típicamente especificados usando consultas que muestran uno o más atributos y el rango de valores de estos atributos.

4. **Empatamiento.** No todas las combinaciones de recursos que satisfacen la consulta de recursos son adecuadas o capaces de trabajar juntas. Es importante tomar en cuenta cómo se relacionan e interactúan los recursos entre sí (e.g., ancho de banda o latencia entre dos nodos) para asegurarse de que pueden satisfacer las restricciones de la aplicación y del recurso.

5. **Enlace.** Una vez que ha sido identificado el subconjunto de recursos que cumplen con los requerimientos y las restricciones de la aplicación, es necesario asegurarse de que los recursos seleccionados están disponibles para ser utilizados. Para ello, se establece un enlace entre los recursos y el nodo que intenta usarlos. Esta fase es importante para satisfacer requerimientos de tiempo real y lograr la calidad de servicio (QoS) deseada.

6. **Uso.** Consiste en utilizar el subconjunto de recursos disponibles que satisfacen los requerimientos y las restricciones de la aplicación para llevar a cabo tareas que necesitan de los recursos adquiridos. Esta fase depende del propósito específico de la aplicación.

7. **Liberación.** Los recursos se liberan cuando la demanda de la aplicación disminuye, la tarea se ha completado o el enlace ha expirado (cualquiera que ocurra primero). Una aplicación puede liberar un recurso por completo (e.g., removiendo el recurso de la aplicación o finalizando la conexión) o puede reducir el uso de un recurso de manera que esta pueda colaborar con otra aplicación (e.g., reduciendo la carga de CPU y el consumo de memoria, pero sin abandonar el recurso que se está utilizando).

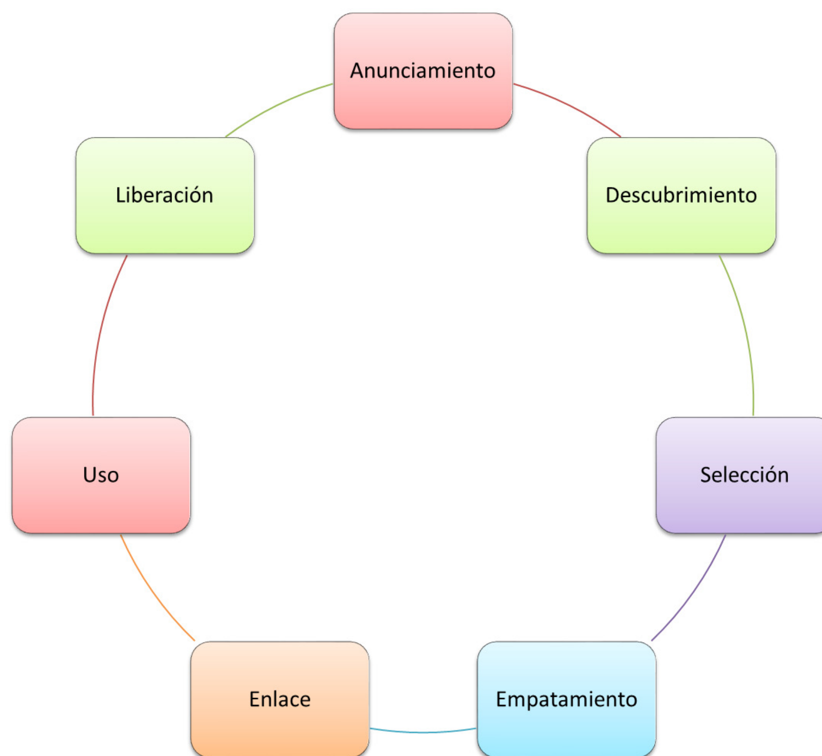


Figura 2.3: Fases de la colaboración de recursos.

2.1.1. Agregación de recursos

La *agregación de recursos* es un subconjunto de la *colaboración de recursos* y comprende únicamente las fases de anunciamiento, descubrimiento, selección, empatamiento y enlace de recursos (ver figura 2.4). En general, la agregación de recursos es el proceso que emplea mecanismos para incorporar y encontrar recursos de cómputo en un determinado sistema, el cual no necesariamente es un sistema P2P, ya que la agregación de recursos también se

puede llevar a cabo en el cómputo en *grid* y en plataformas centralizadas (e.g., GENI [19]).

Los sistemas P2P colaborativos poseen una naturaleza caracterizada por recursos dinámicos, distribuidos, heterogéneos y multi-atributo. Por lo tanto, la agregación de recursos se considera satisfactoria solo si proporciona una alta Calidad de Servicio (QoS) y permite agregar recursos heterogéneos, multi-atributo, de un solo atributo, dinámicos, estáticos y distribuidos en un sistema [1]. Además, una solución eficiente de agregación de recursos debe anunciar todos los recursos y sus estados actuales, descubrir recursos potencialmente útiles, seleccionar recursos que satisfagan los requerimientos de una aplicación, empatar recursos y aplicaciones de acuerdo a sus restricciones, así como enlazar recursos y aplicaciones para asegurar un servicio [18].

En la figura 2.4, se ilustran las fases clave de la agregación de recursos (i.e., selección, empatamiento y enlace), las cuales deben mantener cohesión para que su efectividad sea alta. Por el contrario, las fases de anunciamiento y descubrimiento se pueden llevar a cabo de manera independiente, i.e., las fases de la agregación de recursos no necesariamente siguen un orden secuencial (a excepción de las fases clave).

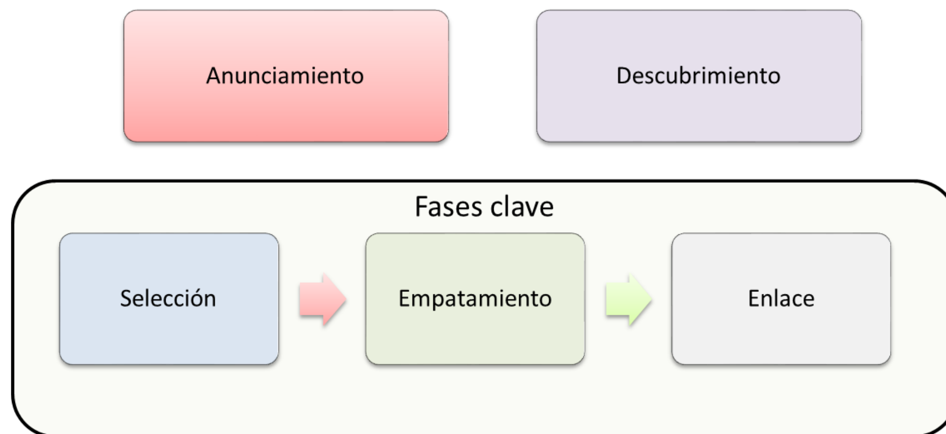


Figura 2.4: Fases de la agregación de recursos.

2.1.2. Sistemas *Peer-to-Peer* colaborativos

Los *sistemas P2P* son sistemas distribuidos sin un control central, cuyos nodos son autónomos y equivalentes en funcionalidad. Un nodo puede ser cliente y servidor simultáneamente. Estos sistemas pueden tener millones de nodos globalmente distribuidos cuyas conexiones pueden ser concurrentes [12]. Además, tienen muchos dominios de aplicación tales como compartición de archivos (e.g., BitTorrent, GNutella, KaZaA, Napster

y BearShare), servicio de televisión basado en IP (e.g., PPLive, CoolStreaming y SopCast), voz sobre IP (e.g., Skype), compartición de ciclos de CPU (e.g., SETI [6]), etc. Actualmente, *BitTorrent* es el sistema P2P de compartición de archivos más popular [17].

En un sistema P2P, los nodos pueden comunicarse directamente si estos se encuentran conectados de manera directa; en caso contrario, pueden usar mecanismos de ruteo a través de otros nodos [12]. Cada uno de los nodos determina sus propias capacidades basándose en sus recursos y toma la decisión sobre cuándo unirse o abandonar el sistema [18]. Una propiedad esencial de los sistemas P2P es ofrecer el rendimiento que presentan las arquitecturas convencionales cliente-servidor [20].

Podemos hacer la analogía de un sistema P2P con un grupo de mentes que tienen un propósito en común, pero que poseen capacidades diferentes. Un sistema P2P se puede definir como "inteligencia colectiva" en donde cada una de las mentes/nodos, que conforman al sistema, realiza funciones específicas para cumplir con una tarea en común, basándose en sus capacidades.

Un factor importante que se debe considerar es la *colaboración entre nodos*. Por ejemplo, si un nodo desea realizar una tarea, que está más allá de sus capacidades, puede lograrlo gracias a la colaboración/ayuda de otro nodo que se encuentre en el mismo sistema P2P [1]. Retomando la analogía del grupo de mentes, si una mente es incapaz de realizar una tarea en particular puede solicitar a otra mente que realice la tarea por completo o que simplemente le auxilie para lograr dicha tarea. De acuerdo a lo anterior, podemos afirmar que un sistema P2P se basa en lo que alguna vez afirmó Julio César en sus formas *divide ut regnes*: "divide y vencerás", i.e., divide un sistema en partes más pequeñas y cumplirás tus objetivos.

Los sistemas P2P han tenido un gran impacto y, por ende, han ocupado un gran procentaje del tráfico total de Internet (en 2008 ocuparon el 50 % y en 2009 el 39 %). Actualmente, el porcentaje de este tráfico se continua incrementando. En el año 2009, estos sistemas ocuparon 3.5 Exabytes por mes y se estima que a finales del presente año habrán ocupado 4 Exabytes por mes. En el año 2010, ocuparon el 20 % del tráfico total de datos móviles. [1]

La mayoría de los sistemas P2P actuales están destinados al intercambio de archivos; sin embargo, en los últimos años se han propuesto alternativas de uso en donde cada uno de los nodos que componen al sistema brinda sus recursos y, a su vez, utiliza recursos de los demás nodos, i.e., existe una colaboración entre los recursos de los nodos que componen al sistema P2P.

Los *sistemas P2P colaborativos* son aquellos que agregan un grupo de diversos recur-

[1] Datos obtenidos de *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012–2017.*

sos (e.g., hardware, software, servicios y datos) para lograr una tarea de gran escala. Tal colaboración involucra la relación entre los recursos específicos de cada uno de los nodos del sistema P2P, sin perder la calidad de servicio (QoS); para llevar a cabo esta colaboración de una forma eficiente, es necesario emplear una arquitectura P2P apropiada. Los sistemas P2P colaborativos comparten una variedad de recursos y datos que consumen, generan, modifican y administran una variedad de contenidos [1]. Además, son aplicables en una amplia variedad de contextos tales como cómputo en *grid*, cómputo en la nube (nubes P2P), cómputo oportunista, redes de sensores, Internet de las cosas, redes sociales móviles y gestión de emergencias (*emergency management*) [18].

Los dispositivos de cómputo ricos en recursos, el decremento en los costos de comunicación y las tecnologías Web 2.0, están cambiando la manera en que los sistemas distribuidos se comunican y colaboran entre sí. Con estos cambios, se prevé que los sistemas P2P jueguen un rol importante en los sistemas P2P colaborativos. En un futuro, dichos sistemas tendrán un conjunto de *peers* que proporcionarán capacidades únicas a una comunidad virtual, permitiéndoles cumplir tareas difíciles o imposibles de alcanzar por un único *peer* [18]. Además, se visualiza un gran conjunto de sistemas P2P colaborativos unificados en un *framework*, en donde los *peers* podrán contribuir y utilizar recursos para fines altruistas y comerciales. Por ejemplo, un proveedor de servicios en la nube podría contribuir con sus ciclos de procesador a la comunidad P2P, esperando obtener beneficios económicos si es posible; y durante períodos de menor demanda, podría proporcionar servicios similares o de menor calidad (e.g., poca capacidad de almacenamiento) para obtener beneficios no monetarios (e.g., para mostrar su alta disponibilidad o para ganar reputación). Tal *framework* también podría permitir que los usuarios, con dispositivos ricos en recursos, obtengan monedas virtuales por sus contribuciones, que más tarde pueden usarse para acceder a otros servicios del sistema [18]. A continuación se describen algunos dominios de aplicación de los sistemas P2P colaborativos.

Cómputo en la nube

El cómputo en la nube está cambiando la manera en que hospedamos y ejecutamos aplicaciones, debido a su escalabilidad y a su modelo económico de pago denominado *pay-as-you-go*. Una nube está compuesta por el hardware de un centro de datos (*datacenter*) y diversos recursos de software, mientras que el cómputo en la nube es el proceso de entrega de servicios por medio de Internet, en el que se utiliza dicho hardware y software; de esta manera, el cómputo en la nube combina hardware, software y recursos de red. Los servicios en la nube se clasifican de la siguiente manera [18]:

1. Infraestructura como un servicio (IaaS, por sus siglas en inglés). Estos servicios proporcionan acceso al hardware a través de Máquinas Virtuales (VMs, por sus siglas en inglés) previamente configuradas. Comúnmente, estos servicios se ofrecen como paquetes de instancias de recursos con diferentes capacidades. Por ejemplo, Amazon EC2 clasifica sus recursos en micro, estándar, cómputo en clúster, *high-CPU* y *high-memory*.

2. Plataforma como un servicio (PaaS, por sus siglas en inglés). Los proveedores de la nube ofrecen servicios *middleware*, lo cual permite a los usuarios desarrollar aplicaciones rápidamente. La complejidad de estas plataformas de servicios pueden ir de lo relativamente simple (e.g., el lenguaje común .NET en Microsoft Azure) a algo relativamente avanzado (e.g., Google App Engine).

3. Software como un servicio (SaaS, por sus siglas en inglés). En esta clasificación, una aplicación o un conjunto de aplicaciones son expuestas como un servicio (e.g., Google Docs y Office Live).

Con el incremento de los niveles de integración, los sistemas de cómputo en la nube exhiben atributos de sistemas distribuidos, en donde los grupos de recursos tales como nodos de procesamiento, almacenamiento, ancho de banda y hardware especial (e.g., GPUs y FPGAs) pueden ser agrupados para ejecutar complejas aplicaciones colaborativas. Estas aplicaciones necesitan establecer redes virtuales dentro del centro de datos para aislar el tráfico y proporcionar un ancho de banda razonable. Por ejemplo, las plataformas Amazon EC2 y Microsoft Azure ejecutan múltiples VMs sobre un mismo nodo físico. Aunque las VMs pueden ser configuradas para no exceder el uso de CPU, memoria y almacenamiento, su desempeño de entrada/salida no puede ser controlado, por lo que el desempeño general de una aplicación ejecutada en una VM depende del comportamiento de otras VMs sobre el mismo *host*. Lo anterior podría provocar un desempeño y comportamientos no predecibles y no deseables, particularmente en aplicaciones que requieren constantes entradas y salidas (e.g, la fusión de datos llevada a cabo en CASA) [18].

Actualmente, el cómputo en la nube intenta proporcionar acuerdos de grano fino a nivel de servicios (SLAs), debido a la inherente complejidad de describir las capacidades de recursos, relaciones entre recursos y requerimientos de una aplicación. Los sistemas de cómputo en la nube son escalables y responden rápidamente a la creciente demanda de las aplicaciones, mediante la compra *al vuelo* de recursos adicionales (e.g., Amazon Auto Scaling). Sin embargo, sus tiempos de respuesta son en minutos, ya que toman tiempo en transferir una imagen de un sistema operativo a un nodo y en arrancar una nueva VM. Debido a lo anterior, estos sistemas no proporcionan la escalabilidad y adaptabilidad que sistemas como CASA requieren, los cuales tienen picos de demanda significativamente altos que duran varios minutos. Sin embargo, se espera que con los avances en sistemas operativos y hardware, los sistemas de cómputo en la nube sean capaces de soportar aplicaciones sensibles a la latencia y sistemas de misión crítica. Actualmente existen algunas críticas con respecto al cómputo en la nube [18]:

- Su modelo de aplicación propietaria y sus datos centralizados contradicen al movimiento FOSS (*Free and Open Source Software*, por sus siglas en inglés) por lo que es considerado como una amenaza por aquellos usuarios que desean tener control sobre sus datos y aplicaciones. Aunque las aplicaciones FOSS pueden ser portadas para ejecutarse sobre la nube, la falta de centros de datos abiertos y libres sigue siendo

una problemática. Esta limitación se podría solucionar haciendo que una comunidad FOSS se vuelva un centro de datos distribuido.

- Representa un desperdicio de recursos en computadoras de escritorio modernas, las cuales se ven forzadas a actuar como clientes ligeros, mientras que el centro de datos lleva a cabo todo el trabajo pesado.

Podemos combinar el poder de los sistemas P2P y las demandas de la comunidad FOSS para construir una infraestructura P2P de cómputo en la nube, en donde los usuarios contribuyan con sus recursos a la nube manteniendo el control sobre sus datos y sus aplicaciones. Tal sistema de cómputo en la nube basado en recursos de cómputo no utilizados en hogares y negocios (cómputo voluntario) puede superar problemas inherentes a los datos centralizados, privacidad, aplicaciones propietarias y fallas en cascada en nubes modernas. Un sistema P2P colaborativo es el núcleo de dicho sistema, el cual permite a los usuarios contribuir con sus recursos no utilizados, mientras se lidia efectivamente con fluctuaciones de recursos y una mejor escalabilidad. A tales sistemas se les denomina *nubes P2P* [18]. La idea es contar con una nube descentralizada, i.e., distribuida entre varios *peers*, en donde cada uno de estos contribuya con sus recursos no utilizados, tal y como se realiza en el cómputo voluntario (ver figura 2.5). Además, es posible contar con una arquitectura P2P que interconecte diferentes nubes para solucionar los problemas de contar con un único punto de fallo y situaciones de cuello de botella, debido a la creciente demanda de recursos en las nubes.

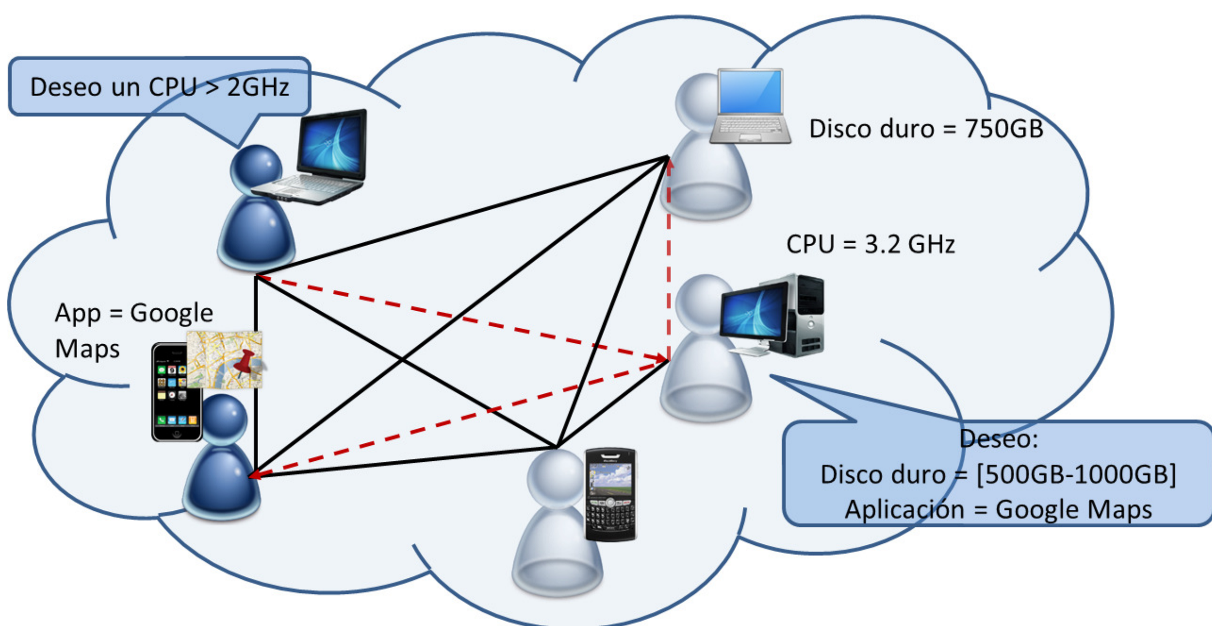


Figura 2.5: Nube P2P de compartición de recursos.

Redes de sensores

Una red de sensores es aquella que está compuesta por cientos o miles de pequeños nodos, cada uno de ellos equipado con un sensor. La mayoría de estas redes utilizan una comunicación inalámbrica y los nodos cuentan comúnmente con recursos y capacidades de comunicación limitadas. En la mayoría de los casos, estas redes son usadas para llevar a cabo procesamiento de información [12].

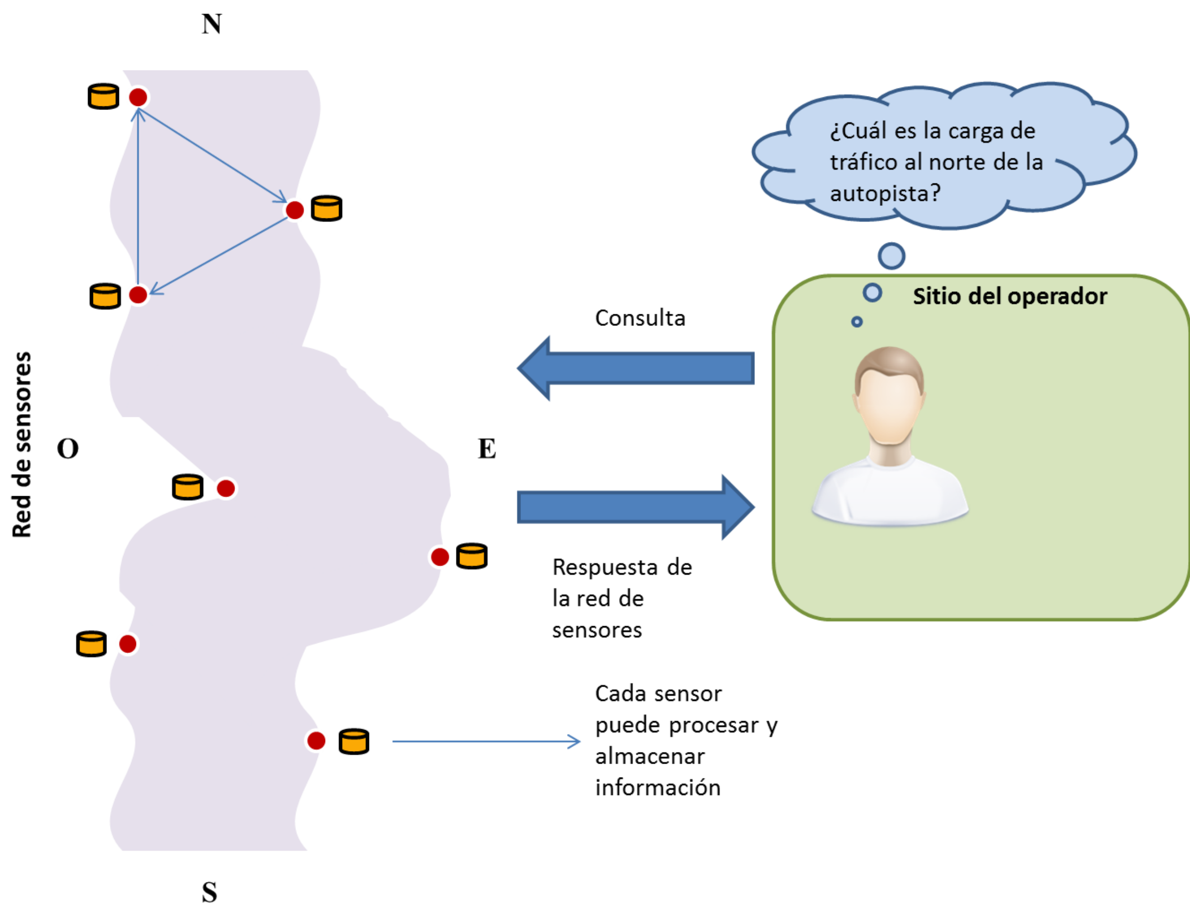


Figura 2.6: Red de sensores para la detección de la carga de tráfico en una autopista.

La figura 2.6 muestra una red de sensores que se encuentran distribuidos a lo largo de una autopista, cuyo objetivo es detectar la carga de tráfico de automóviles. Cada uno de los sensores se encarga de detectar el tráfico en una cierta zona de cobertura. Adicionalmente, un operador puede extraer información de (o parte de) la red haciendo consultas como:

1. ¿Cuál es la carga de tráfico al norte de la autopista?
2. ¿Cuál es la parte de la autopista que posee menos carga de tráfico?

3. Considerando la carga de tráfico actual, ¿Cuál es el tiempo estimado de llegada a un determinado punto?

Para responder a estas preguntas, algunos sensores colaboran entre sí, mientras que otros permanecen intactos.

Las redes de sensores se pueden aplicar en Sistemas de Detección Adaptativa Colaborativa, también conocidos como sistemas DCAS (*Distributed Collaborative Adaptive Sensing*) cuyo objetivo es sensar constantemente el mundo físico utilizando sensores, actuadores y nodos de procesamiento. Los datos generados son procesados utilizando grupos de recursos de procesamiento, almacenamiento y ancho de banda. En sistemas DCAS, los datos constantemente están siendo generados, procesados, almacenados y extraídos entre grupos de sensores, nodos de almacenamiento y procesamiento. Una característica esencial de estos sistemas es la obtención de datos, que después de ser procesados son mostrados a los usuarios finales. En este tipo de sistemas se necesita determinar cómo y qué grupo(s) de recursos son usados para generar y procesar los datos requeridos [1]. Por ende, los sistemas DCAS necesitan de la colaboración de recursos para llevar a cabo su cometido.

CASA [9] es un sistema DCAS que detecta y da seguimiento a fenómenos climáticos (e.g., tornados y tormentas) a través de una red de radares climáticos que colaboran en tiempo real y cuyo medio de comunicación es Internet (debido a su flexibilidad, accesibilidad global y bajo costo). Este sistema está revolucionando la manera en cómo observamos, evaluamos, entendemos y predecimos eventos del clima. En la figura 2.7, se puede observar que los principales eventos del clima suscitan 3 km por debajo de la atmósfera; sin embargo, CASA es el único sistema que puede detectar y dar seguimiento a dicha zona. La figura 2.8 ilustra la observación simultánea de un fenómeno climático por una red de radares CASA [18].

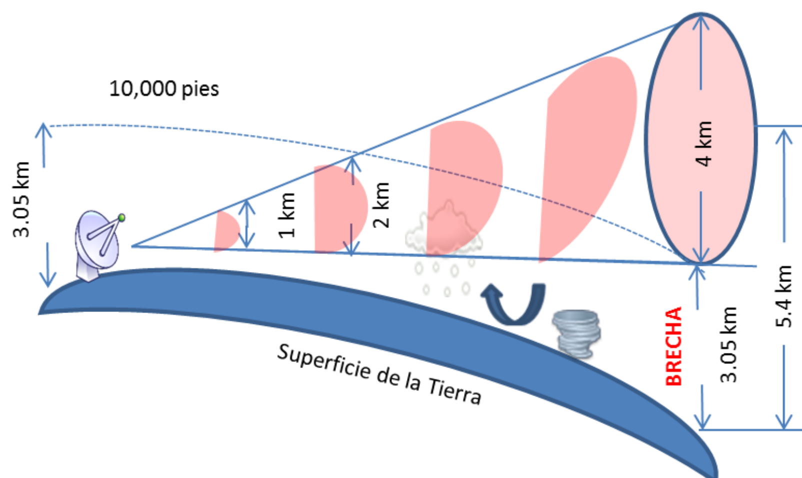


Figura 2.7: Radar de alto poder y amplio rango.

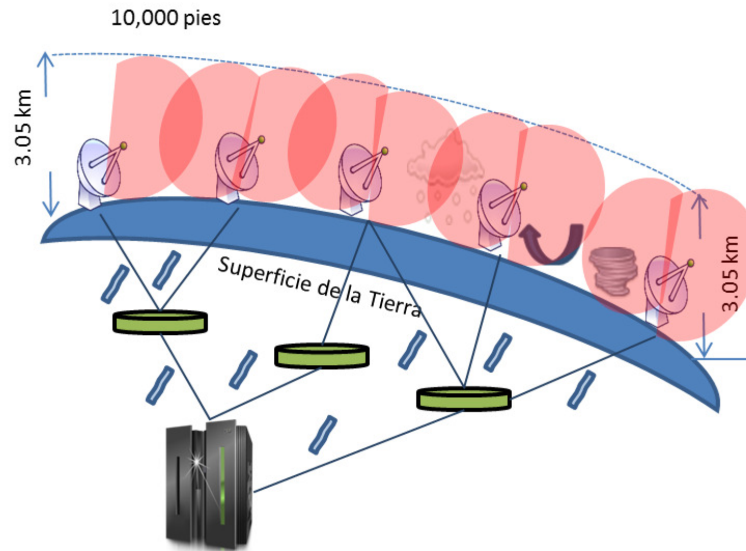


Figura 2.8: Rastreo de 3 km por debajo de atmósfera usando una red de radares.

Los radares, los nodos de procesamiento y los algoritmos de fusión de datos (DF) de CASA se comunican entre sí para ajustar la detección y las estrategias de procesamiento de datos, en respuesta directa a los eventos del clima y necesidades de los usuarios que están en constante cambio. CASA requiere que la re-asignación, transmisión, procesamiento y generación de datos se complete en 30 segundos. Sin embargo, la mayor parte de este tiempo es usada para generar datos, dejando menos tiempo para la transmisión y fusión de dichos datos [18].

CASA soporta diversos conjuntos de algoritmos metereológicos denominados aplicaciones. Cada una de estas aplicaciones obtiene datos de uno o más radares y requieren diferentes recursos de cómputo, almacenamiento y ancho de banda, ya que usan diferentes algoritmos metereológicos, tipos y volúmenes de datos. Los patrones del clima conocidos, la geografía, el costo y la disponibilidad de la infraestructura determinan en donde serán usadas las aplicaciones. Por ejemplo, las aplicaciones de seguimiento de tornados son usadas solo en áreas propensas a estos fenómenos físicos. Estas aplicaciones son accedidas por un conjunto de diversos usuarios finales, tales como NWS (*National Weather Service*), EMs (*Emergency Managers*), científicos, medios de comunicación y entidades comerciales [18].

CASA (en colaboración con el proyecto GENI) está desarrollando un ambiente de cómputo para investigadores denominado *Data-Intensive Cloud* (DICloud). Este ambiente permite a los usuarios especificar flujos de trabajo, indicando el subconjunto de sensores a utilizar, así como la forma en que se procesan y almacenan los datos generados por estos sensores, usando recursos de cómputo y almacenamiento en la nube. Actualmente, los usuarios de DICloud tienen acceso a radares, estaciones de clima y algunas cámaras

especiales. Los servicios de almacenamiento y procesamiento de datos son proporcionados por el Servicio Web de Amazon para utilizar recursos de almacenamiento y procesamiento [1].

Aunque una red densa en radares puede mejorar sustancialmente la detección, la predicción y el tiempo de advertencia, simultáneamente puede crear muchos retos debido al número de sensores involucrados, a la infraestructura de comunicación utilizada y a la heterogeneidad de la red, así como al volumen de datos generados. Por ello, se requieren nuevos protocolos de transporte de datos, soluciones de agregación de recursos y soluciones de fusión de datos multi-sensor para transmitir oportunamente y procesar grandes volúmenes de datos, lidiando con las condiciones dinámicas de la red. Además, ciertos eventos climáticos requieren aplicaciones específicas y grandes recursos de cómputo, almacenamiento y ancho de banda, por lo que es necesario contar con mecanismos que reúnan los requerimientos de tiempo real específicos de una aplicación, mientras se optimiza el uso de los recursos [18].

La fusión de datos multi-radar involucra la recolección de datos de múltiples radares remotos y se puede llevar a cabo de manera colaborativa y concurrente por diferentes nodos. Cada uno de estos nodos proporciona recursos de comunicación y cómputo en respuesta a las peticiones de los usuarios. Por lo tanto, un enfoque distribuido, dinámico y colaborativo basado en una arquitectura P2P es atractivo para agregar recursos (a través de la red) de múltiples sensores y nodos de procesamiento. Sin embargo, la red y la infraestructura de procesamiento pueden estar sujetas a condiciones adversas debido al clima, fallas en los recursos, degradación del enlace (entre recursos) y tráfico variable en enlaces alámbricos e inalámbricos. Actualmente, la arquitectura P2P de CASA no es lo suficientemente robusta, debido a que no funciona correctamente bajo tales condiciones adversas. Un escaso ancho de banda entre un nodo de procesamiento y un nodo de almacenamiento puede ser compensado por un procesamiento rápido de datos para ajustar el retardo inherente en la transmisión de datos hacia el nodo de almacenamiento [18].

Redes sociales móviles

Las redes sociales pueden ser mejoradas si se permite que los usuarios compartan recursos de sus dispositivos móviles. Por ejemplo, una persona con un teléfono móvil de escasos recursos (pertenciente a una red social móvil con una infraestructura de red *ad-hoc*) podría conectarse al teléfono inteligente de algún amigo (conectado a la misma red) con GPS, con la finalidad de localizar el cajero automático más cercano (ver figura 2.9). En la figura 2.10, se muestra a un grupo de amigos reunidos en una cafetería para compartir sus experiencias. Uno de ellos (persona A), desea mostrar imágenes a todos desde su teléfono celular, sin embargo este no cuenta con un proyector. Una persona B del mismo grupo de amigos cuenta con un teléfono inteligente con proyector. La persona A puede utilizar el teléfono de la persona B para proyectar sus imágenes o, si lo desea, puede mostrar los videos que se encuentran en el servidor de su hogar. Estas aplicaciones se benefician de la

diversidad de recursos existentes en una comunidad, en la que los miembros comparten recursos entre sí. Dependiendo de la situación, los usuarios pueden establecer conexiones transitorias para lograr una tarea común (e.g., durante una conferencia) [18].

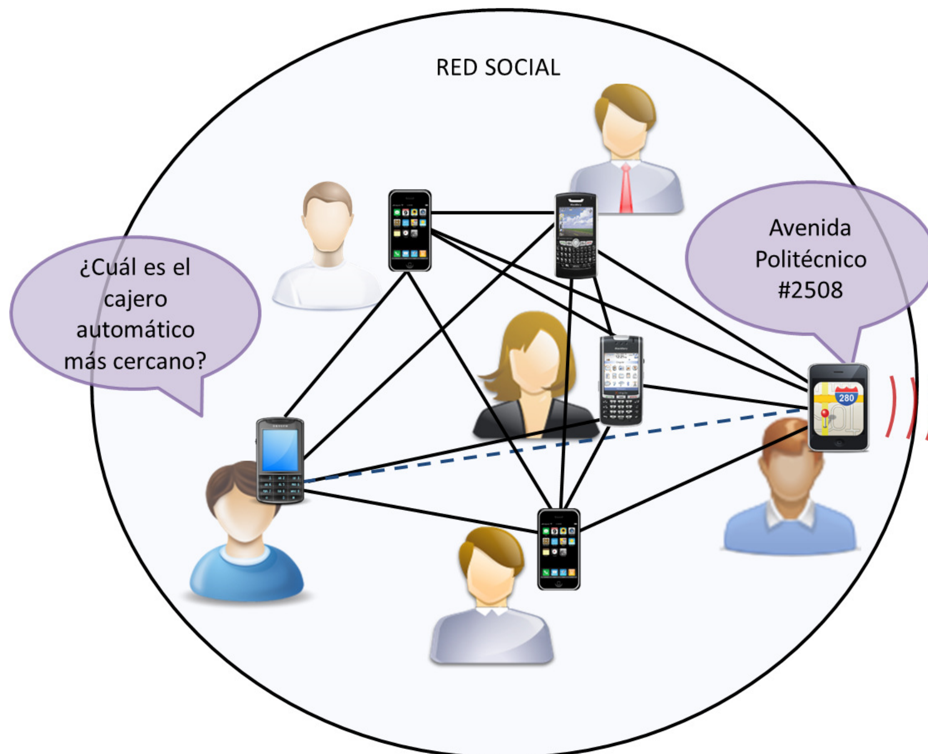


Figura 2.9: Localización del cajero automático más cercano a través de una red social móvil.

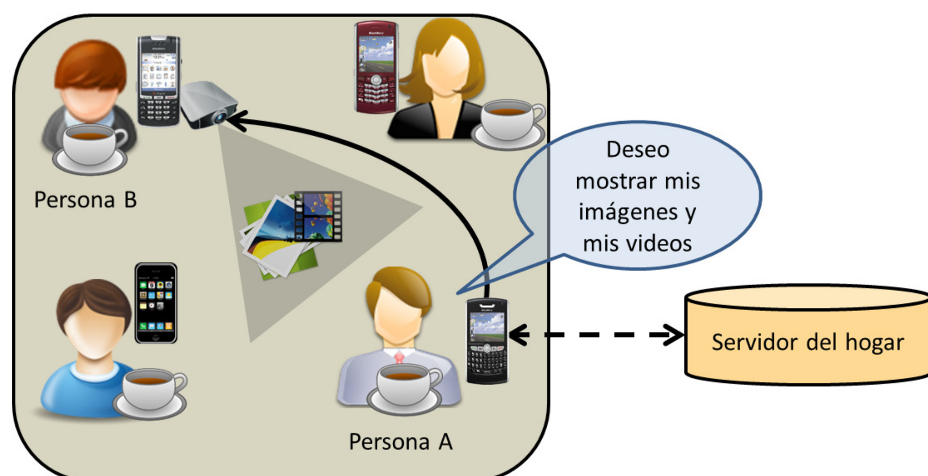


Figura 2.10: Un grupo de amigos compartiendo recursos en una cafetería.

En grandes reuniones sociales tales como carnavales, eventos deportivos o campañas políticas, los dispositivos móviles de los usuarios pueden ser usados para compartir ofertas especiales, comentarios, videos o votar por una cierta proposición. Para lograr esto, no es necesario depender de una infraestructura de red fija, en lugar de ello se puede construir una red *ad-hoc*. Estas aplicaciones han emergido bajo el dominio del cómputo oportunista. Por ende, es necesario tomar en cuenta las relaciones sociales entre usuarios para que estos se encuentren en disposición de compartir sus recursos [18].

Las redes sociales móviles son aquellas que necesitan comprobar la proximidad entre dos recursos, con la finalidad de evadir a un cierto proveedor de servicios, minimizar la latencia o reducir la pérdida de paquetes (i.e., es necesario que lleven a cabo la fase de empatamiento). Los sistemas P2P colaborativos son de vital importancia en redes sociales móviles, debido a su naturaleza distribuida, altamente dinámica y autónoma. Las fases de anunciamiento, descubrimiento y empatamiento de recursos, basados en las relaciones entre recursos y restricciones de usuarios, son requerimientos fundamentales en este tipo de redes [18].

2.2. Arquitecturas P2P

Las arquitecturas P2P permiten organizar nodos en una red *overlay* (una red superpuesta, i.e., una red sobre otra red). En estas arquitecturas, los nodos están compuestos por procesos y los enlaces entre ellos representan los posibles canales de comunicación (usualmente se utilizan conexiones TCP). Estos enlaces forman un grafo (a, b) y son elegidos dependiendo del algoritmo de búsqueda que se esté empleando. Para llevar a cabo la comunicación entre nodos se requiere el envío de mensajes a través de dichos canales [12].

Los protocolos P2P actuales que permiten llevar a cabo la agregación de recursos están basados en arquitecturas *overlay* [1]. Estas arquitecturas se clasifican en: estructuradas (sección 2.2.1) y no estructuradas (sección 2.2.2).

2.2.1. Arquitecturas estructuradas

Los esfuerzos por resolver el problema de tener un control centralizado dieron lugar al surgimiento de las arquitecturas estructuradas. Comúnmente, en este tipo de arquitecturas los nodos se organizan en jerarquías mediante Tablas Hash Distribuidas (DHT, por sus siglas en inglés), permitiendo que los nodos sean responsables de una parte específica de la red. En un DHT, se mantiene un índice de recursos, i.e., una colección de pares $(llave, valor)$. De esta manera, los datos tienen asignado un identificador único (de 128 o 160 bits) y los nodos siguen un mismo protocolo para llevar a cabo las consultas. Cuando se busca un elemento en particular, se regresa la dirección IP del nodo responsable de dicho elemento [12]. No todos los nodos necesitan ser parte del DHT y la complejidad del

ruteo de mensajes es $O(logN)$, en donde N es el número de nodos en el sistema [18].

En las arquitecturas estructuradas, cuando un nodo P contacta a un nodo Q y se percata de que este último ya no responde, simplemente lo elimina de su lista y selecciona otro nodo. Las peticiones deben recorrer toda la red P2P para encontrar datos. La principal desventaja de este tipo de arquitecturas es que las consultas de recursos no siempre pueden resolverse. Como se había mencionado anteriormente, la mayoría de los sistemas P2P se utilizan para la compartición de archivos y muchos de estos poseen una arquitectura estructurada [12]. Estas arquitecturas son apropiadas en sistemas de gran escala, debido a su alta escalabilidad y a que garantizan un buen desempeño [18].

Chord [2] es el protocolo, basado en una arquitectura estructurada, más conocido, flexible y robusto. Este protocolo mantiene un anillo (ver figura 2.11) y mapea los nodos y recursos en un espacio de llaves circular [21]. Un nodo es colocado en una posición aleatoria dentro del anillo, mientras que un recurso es indexado en el *sucesor* de su llave, i.e., el nodo más cercano en dirección de las manecillas del reloj. Cada nodo n mantiene un conjunto de punteros (*fingers*) a nodos que están en $(n + 2^{i-1})mod2^b$, en donde b es la longitud de la llave en bits y $1 \leq i \leq b$. Por ejemplo, el nodo E de la figura 2.11 mantiene *fingers* a los nodos F, H y J. La tabla de ruteo de un nodo es el conjunto de *fingers* y es llamada *tabla de fingers*, la cual se actualiza periódicamente. Dichos *fingers* son usados para enviar un mensaje recursivamente a una llave específica, e.g., el nodo E puede contactar al nodo L a través de la ruta $E \rightarrow J \rightarrow L$. Un nodo también puede reducir la latencia, e.g., si el nodo E conoce al nodo K, un mensaje puede seguir la ruta $E \rightarrow K \rightarrow L$. El costo de agregar o remover un nodo del anillo es de $O(log^2N)$. Para garantizar un ruteo de mensajes con una complejidad de $O(logN)$, es necesario utilizar un protocolo de estabilización para que los predecesores y sucesores de un nodo sean válidos. Debido a lo anterior, es costoso mantener el protocolo Chord en redes dinámicas [18]. Chord y otros protocolos estructurados (e.g., CAN [22] y Pastry [23]) están diseñados para indexar recursos de un solo atributo [1].

La red de contenido direccionable (CAN, por sus siglas en inglés) [22] está basada en un *torus* (i.e., un arreglo) de d dimensiones. La figura 2.12 ilustra un *torus* de dos dimensiones particionado en varias zonas. A los nodos se les asignan identificadores aleatorios y a los recursos se les asignan identificadores que son el resultado de aplicar una función *hash* a sus nombres únicos. El primer nodo se encarga de rastrear todo el *torus*. Cuando se agrega un nuevo nodo, se le asigna una llave aleatoria y se posiciona en la zona responsable de indexar su llave. La zona es dividida en dos zonas de volúmenes iguales y cada zona se asigna al nuevo nodo y al propietario de la zona anterior. Cuando los nodos se unen al sistema, las zonas se dividen y, por el contrario, cuando abandonan el sistema, las zonas se combinan. Cada nodo es responsable de dar seguimiento a los pares $(llave, valor)$ mapeados en su zona. Los recursos se localizan enviando mensajes de consulta a la zona encargada de indexar la llave especificada, utilizando ruteo *greedy*. Los nodos en CAN mantienen hasta $2d$ entradas de ruteo para las zonas de sus vecinos. Dichas entradas de ruteo son usadas para rutear mensajes en $O(dN^{1/d})$ saltos, en donde N es el número de

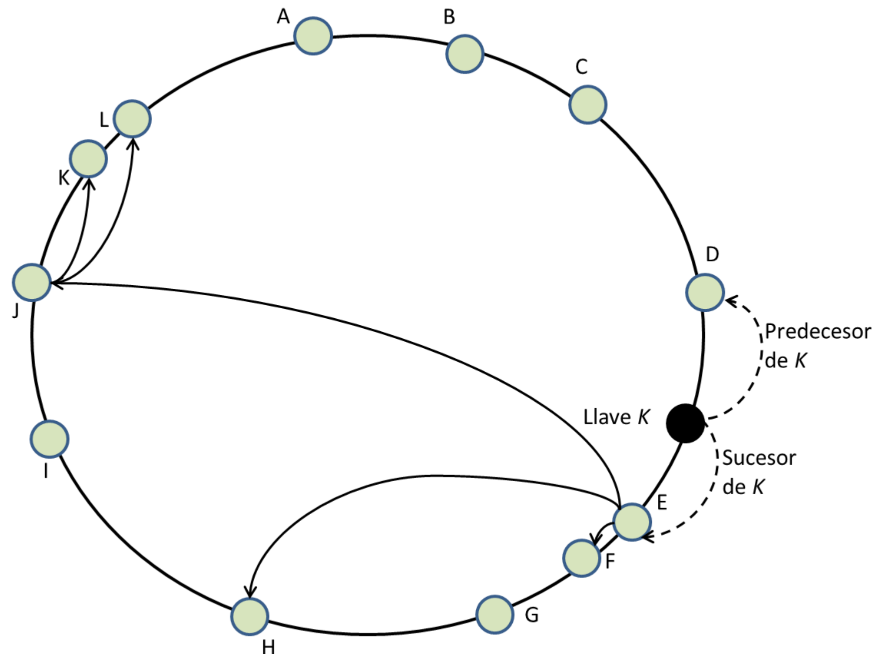


Figura 2.11: Estructura de anillo en Chord.

nodos en el sistema. El esquema de ruteo en CAN calcula la distancia de un identificador a la orilla de una zona, en lugar de calcularla a un punto específico. Además, un número menor de rutas alternativas y fallas de vecinos reduce la resiliencia[2] [18].

Pastry [23] es una solución que utiliza una arquitectura P2P estructurada basada en un hipercubo. En Pastry, las llaves se representan utilizando un arreglo de caracteres de dígitos, en donde cada dígito se encuentra en base 2^b. Además, rutea mensajes usando un mecanismo basado en prefijos, con la finalidad de alcanzar una llave dada o el nodo numéricamente más cercano (corrigiendo un dígito a la vez). Cada nodo mantiene una tabla de ruteo que posee log_{2b} registros, cada uno con $2^b - 1$ entradas, por lo que apuntan a nodos espaciados a diferentes distancias. Pastry rutea mensajes en $log_{2b}N$, a menos de que múltiples nodos con llaves adyacentes fallen simultáneamente. Para mejorar la resiliencia y el balance de cargas, se almacena un par $(llave, valor)$ en múltiples nodos cercanos a la llave dada. Las tablas de ruteo en Pastry son relativamente grandes y contienen $O(blog_bN)$ entradas por nodo [18].

Aunque los protocolos presentados anteriormente garantizan el descubrimiento de recursos, poseen muchas limitaciones. Un nodo no puede saber qué nodo es responsable de un elemento de datos en particular cuando se utilizan DHTs, ya que los datos están asociados con llaves pseudoaleatorias [20]. Además, requieren que todas las copias de la

[2]La resiliencia es la capacidad de proveer y mantener un nivel aceptable de servicio ante la presencia de fallos que pudiesen alterar la operación normal de una red.

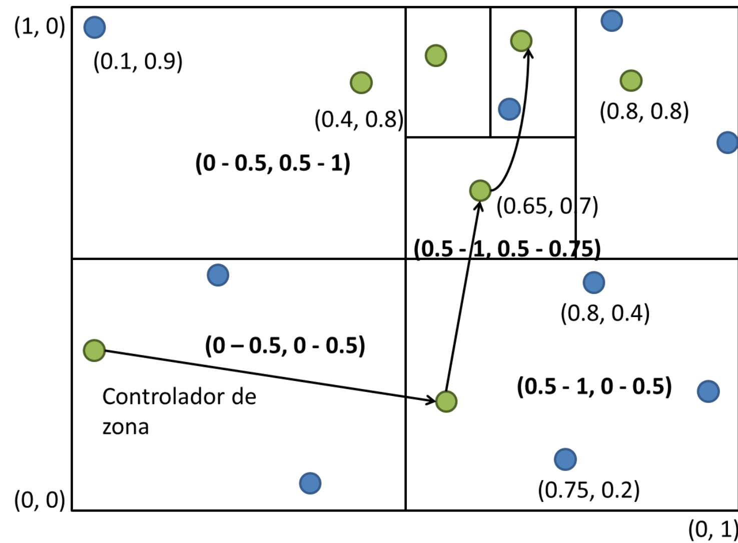


Figura 2.12: *Torus* de d dimensiones en CAN.

misma llave sean almacenadas en el mismo nodo o vecindario, lo que conduce a únicos puntos de fallo. Para solucionar este problema, muchos protocolos basados en arquitecturas estructuradas utilizan replicación. Sin embargo, la replicación podría provocar que los índices de recursos se vuelvan obsoletos cuando los recursos son altamente dinámicos o un nodo abandona el sistema. También presentan un problema de balance de cargas cuando se necesita revisar la consistencia de múltiples copias del par $(llave, valor)$. Se garantiza un buen desempeño solo si la red es consistente, sin embargo es costoso mantenerla de esta manera (excepto en Kademlia [29]). El desempeño promedio de estos protocolos es bueno en sistemas sensibles a la latencia, e.g., CASA y nubes P2P. Sin embargo, aún existe la necesidad de contar con un sistema P2P que combine las propiedades de las arquitecturas P2P estructuradas y no estructuradas [18].

Típicamente los DHTs indexan RSs, sin embargo están diseñados para soportar recursos de un solo atributo. Por lo tanto, dichos DHTs no permiten la selección simultánea de múltiples recursos multi-atributo, requeridos por sistemas P2P colaborativos. Los protocolos presentados anteriormente solo proporcionan el anunciamiento y selección de recursos individuales [18]. En la figura 2.13, se muestran tres decisiones de diseño alternativas basadas en anillos que soportan recursos multi-atributo.

Mercury [24] es un protocolo que mantiene un anillo separado para cada tipo de atributo (ver figura 2.13(a)). Cada nodo anuncia sus RSs a los anillos que tienen a su cargo los atributos de sus recursos, i.e., existe una relación atributo-anillo. Mercury utiliza consultas de un único atributo denominadas *Single Attribute Dominated Queries* (SADQ, por sus siglas en inglés), en donde dichas consultas se realizan únicamente sobre el anillo que corresponde al atributo más selectivo (i.e., el anillo con el mínimo costo de resolución

de consultas). Por ejemplo, en la figura 2.13(a) la consulta da tres saltos en los anillos *Velocidad de CPU* y *Ancho de banda*, y dos saltos en el anillo *Memoria*. De esta forma, es más eficiente resolver la consulta usando el anillo *Memoria*. Para llevar a cabo SADQ, cada nodo necesita rastrear todos los atributos de los RSs que indexa. Mercury utiliza un algoritmo basado en muestras aleatorias para estimar la selectividad de una consulta dentro de cada anillo. Una consulta se resuelve enviándola a una serie de nodos a través de sus sucesores. Como la consulta se propaga, se agregan aquellos recursos que satisfacen todos los atributos especificados en la consulta y el último nodo regresa el conjunto de dichos recursos al iniciador de la consulta. El algoritmo de muestras aleatorias también se utiliza para estimar el balance de cargas dentro de cada anillo y, basándose en la carga estimada, los nodos se reorganizan dentro de cada anillo para balancear la carga de la consulta. Es posible agregar nuevos anillos para soportar nuevos atributos y cada anillo puede ser personalizado para especificar las características de dichos atributos. Sin embargo, no se recomienda utilizar Mercury en sistemas P2P colaborativos, cuyos recursos están caracterizados por muchos atributos (e.g., CASA), debido al gran número de entradas de ruteo que se podría generar (lo cual lo hace ineficiente para este tipo de sistemas) [18]. Además, el costo del anunciamiento puede ser significativo cuando un sistema P2P colaborativo tiene un gran número de atributos dinámicos que cambian rápidamente [24].

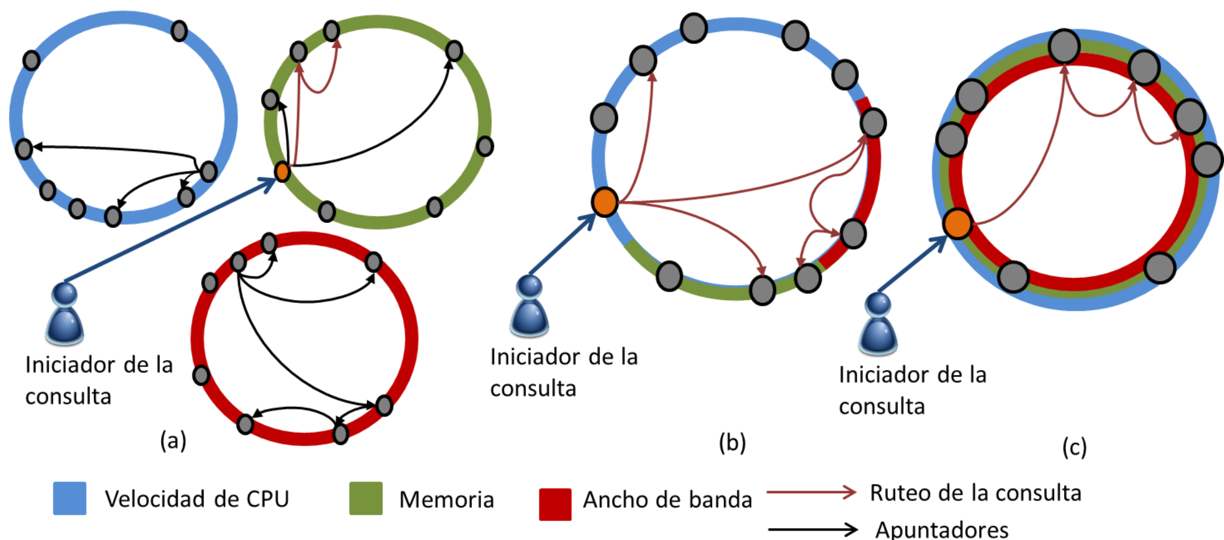


Figura 2.13: Diseños estructurados basados en anillos que permiten la agregación de recursos: (a) Multi-anillo – se crea un anillo por cada tipo de atributo; (b) anillo particionado – el anillo se particiona en regiones y a cada región se le asigna un tipo de atributo; (c) anillos sobrepuestos – se colocan múltiples anillos sobre un mismo anillo y cada anillo sobrepuesto corresponde a un tipo de atributo.

LORM (*Low-Overhead, Range-query and Multi-attribute*) [25] es una solución P2P estructurada, basada en Cycloid [41], que asigna un segmento diferente de un anillo a cada

tipo de atributo (ver figura 2.13(b)). Los valores de los atributos se representan como una cadena de bits, en donde los prefijos representan el segmento (tipo de atributo) y los sufijos representan la posición dentro del segmento (valor del atributo). Los bits sufijos se generan aplicando una función *hash* preservada localmente (LPH, por sus siglas en inglés) al valor del atributo. Las consultas de recursos se realizan mediante sub-consultas en cada segmento de acuerdo al conjunto de atributos requeridos. Finalmente, los resultados de dichas consultas se combinan utilizando una operación *join* como en bases de datos [18]. Los atributos populares en cargas de trabajo reales [4] fuerzan a LORM a utilizar muchos segmentos, mientras que otros rara vez son utilizados [1].

MAAN (*Multi-Attribute Addressable Network*) [26] es un protocolo basado en Chord que propone un mecanismo SADQ para un anillo (ver figura 2.13(c)). Este protocolo mapea los valores de los atributos de un recurso en un mismo anillo, usando una misma función LPH para cada tipo de atributo. Para preservar la localidad, las funciones LPH distribuyen uniformemente los valores *hash* en todo el anillo [18]. El uso de un anillo sobrepuesto reduce el ruteo y el costo de las consultas. Sin embargo, una función LPH no proporciona balance de cargas cuando existen muchos recursos idénticos o se presentan recursos populares en cargas de trabajo reales (e.g., CASA y *grids*) [5].

MADPastry (*Mobile AD-hoc Pastry*) [27] propone un esquema para redes móviles *ad-hoc* de gran escala, particionando un hipercubo Pastry de manera que se preserve la localidad física de los nodos (determinada utilizando un conjunto de puntos de referencia). Primero se envía una consulta a la partición local y, si los recursos requeridos no se encontraron localmente, la consulta se envía a otras particiones. Otra alternativa es construir un conjunto de hipercubos por cada región geográfica y luego conectarlos para formar un *backbone* [32]. Los nodos que se encuentran en un mismo vecindario forman un hipercubo, mientras que los *backbones* se forman mediante la interconexión de dichos hipercubos con nodos *gateway*. Los recursos se anuncian a los hipercubos locales y remotos a través del *backbone*. De manera similar a MADPastry, los recursos se consultan primero en el hipercubo local y los hipercubos remotos únicamente se consultan cuando los recursos no se encuentran localmente. El balance de cargas dinámico se logra ajustando las dimensiones del hipercubo de acuerdo a la carga de la consulta. Por ejemplo, cuando la carga promedio de la consulta de un hipercubo excede un umbral, las dimensiones del hipercubo se incrementan y se añaden más nodos al hipercubo para distribuir la carga. Este enfoque no es óptimo, ya que las consultas populares (especificadas en [4] y [5]) tendrán alta latencia debido al incremento de las dimensiones del hipercubo, lo cual ocasiona un incremento en la longitud de la ruta [18]. Además, MADPastry no puede ser utilizada en redes *ad-hoc* altamente dinámicas, debido a que su arquitectura estructurada puede ser fácilmente desconectada/particionada como consecuencia de la alta movilidad de los nodos [1].

LORM, MADPastry y el *backbone* de hipercubos mantienen un número más bajo de entradas de ruteo que Mercury. MADPastry y el *backbone* de hipercubos proporcionan la fase de empatamiento basándose en la latencia y el conteo de saltos, gracias a su capaci-

dad para localizar recursos locales. Sin embargo, el costo de búsqueda de recursos en los tres protocolos se incrementa cuando existen múltiples sub-consultas y una tendencia a regresar un gran número de recursos inutilizables [18].

MURK (*Multidimensional Rectangulation with Kd-trees*) [28] es una solución basada en CAN, en donde los RSs se mapean a puntos en un *torus* de d dimensiones. Cada dimensión del *torus* representa un tipo de atributo y cada nodo indexa en su zona todos los RSs que mapea. MURK organiza dichas zonas en un árbol de k dimensiones (*kd-tree*) con la finalidad de rastrearlas. El balance de cargas dinámico se logra separando y agregando zonas. Una consulta multi-atributo se encierra en un hiperrectángulo (i.e., un espacio contiguo de d dimensiones) en el *torus*. Las consultas se resuelven usando ruteo *greedy* y, debido a que la consulta se propaga, cada nodo reporta recursos coincidentes al iniciador. A pesar de que se agregan recursos potenciales conforme se propaga una consulta, se introduce un costo adicional debido a que los recursos próximos en el *torus* no están mapeados en regiones contiguas sobre el anillo [18].

En Mercury, MAAN, LORM, MADPastry, MURK y el *backbone* de hipercubos, la fase de descubrimiento no es aplicable debido a que los recursos se anuncian explícitamente [1]. Además, estos protocolos funcionan apropiadamente en ambientes semi-dinámicos, en donde los recursos no cambian rápidamente. Las DHTs son ineficientes e incapaces de mantener el estado en ambientes dinámicos (e.g., nubes P2P) [30]. Los protocolos que utilizan SADQ poseen un bajo costo de resolución de consultas y conducen a un desbalance de cargas en las consultas y a una representación imprecisa de recursos dinámicos, debido al gran número de réplicas [18].

Mercury, MAAN y LORM son escalables, ya que permiten añadir nuevos atributos de recursos sin requerir cambios significativos en la topología existente. A excepción de MADPastry y el *backbone* de hipercubos, los demás protocolos no proporcionan la fase de empatamiento de recursos y ningún protocolo analizado proporciona la fase de enlace [31]. Ciertas aplicaciones son capaces de compensar la falta de ancho de banda entre un nodo de procesamiento y un nodo de almacenamiento (e.g., CASA), las cuales generan consultas complejas y requieren coordinación entre las fases de selección y empatamiento. Dadas las limitaciones que poseen los protocolos estructurados, aún existe la necesidad de contar con una solución que sea cohesiva y pueda anunciar, descubrir, seleccionar, empatar y enlazar recursos de una forma eficiente. En la tabla 2.1 se muestra un resumen de los protocolos P2P de agregación de recursos analizados en esta sección [18].

2.2.2. Arquitecturas no estructuradas

En este tipo de arquitecturas, no existen jerarquías por lo que los nodos están conectados entre sí mediante enlaces arbitrarios. Un nodo no tiene un lugar específico en la red

Esquema	**Diseño**	**Anunciamiento**	**Descubrimiento***	**Selección**	**Empatamiento**	**Enlace***
Mercury [24]	Múltiples anillos	Sí	N/A	Garantizada	No	No
LORM [25]	Anillo particionado	Sí	N/A	Garantizada	No	No
MADPastry [27]	Anillo particionado	Sí, a particiones locales y vecinas	N/A	Garantizada	Sí, latencia y conteo de saltos	No
Backbone de hipercubos [32]	*Backbone* basado en hipercubos	Sí, a hipercubos locales y vecinos	N/A	Garantizada	Sí, latencia y conteo de saltos	No
MAAN [26]	Anillos sobrepuestos	Sí	N/A	Garantizada	No	No
MURK [28]	*Torus* de d dimensiones	Sí	N/A	Garantizada	No	No

*N/A – No aplicable

Tabla 2.1: Protocolos P2P estructurados de agregación de recursos.

y puede unirse fácilmente a esta, formando sus propios enlaces a partir de los existentes. Además, cada nodo mantiene una lista de sus vecinos [12]. Las arquitecturas no estructuradas son altamente flexibles y robustas ante fallas aleatorias de nodos, debido a que los nodos no necesitan tener un conocimiento previo sobre la topología antes de unirse, simplemente se unen a la red de acuerdo a un conjunto de reglas predeterminadas. Las arquitecturas P2P no estructuradas se clasifican en deterministas y no deterministas [18].

En 1999 surgió Napster, el cual permitía a los usuarios la descarga de archivos. Una base de datos centralizada mantenía el *índice de un recurso* y daba seguimiento a la lista de archivos de un nodo, su dirección IP y su puerto (ver figura 2.14(a)). La búsqueda de recursos se llevaba a cabo consultando directamente dicha base de datos. Los nodos establecían y terminaban conexiones constantemente, formando un *overlay* no estructurado. A estos sistemas, se les denomina *sistemas P2P no estructurados deterministas*, debido a que garantizan que los recursos (indexados en la base de datos) serán encontrados. Sin embargo, una base de datos centralizada constituye un único punto de fallo y limita la escalabilidad [18].

En 2001, BitTorrent propuso un protocolo unificado para permitir la comunicación entre múltiples bases de datos distribuidas, permitiendo a los usuarios la búsqueda de cualquiera de los recursos indexados. En la figura 2.14(b), se muestra la topología de BitTorrent, la cual está compuesta por tres capas. Comúnmente, se accede a los índices de los recursos a través de un sitio Web conocido como motor de búsquedas *torrent*, sitio *torrent* o comunidad. Los recursos se anuncian utilizando un *archivo torrent*, el cual almacena el nombre del archivo, su longitud, el número de pedazos (i.e., segmentos de igual tamaño de un archivo) y una lista de valores hash SHA1 para cada pedazo. Este archivo también contiene una lista de rastreadores, i.e., el conjunto de nodos que mantienen una lista de los

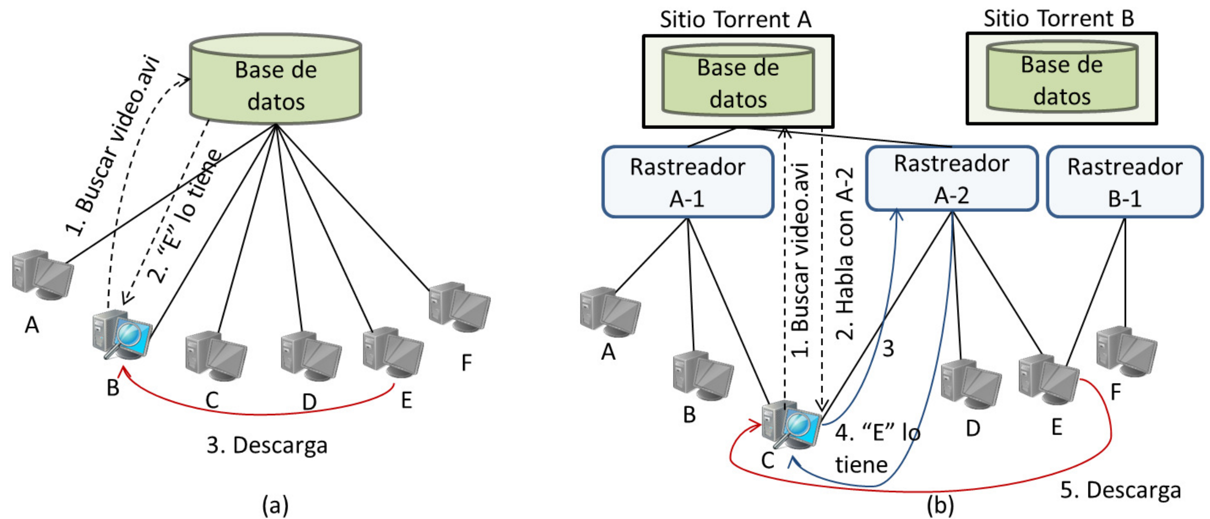


Figura 2.14: Arquitecturas P2P no estructuradas deterministas: (a) Napster y (b) BitTorrent.

nodos que están descargando/subiendo un mismo archivo. De manera similar a Napster, BitTorrent es un sistema P2P no estructurado determinista [18].

En las arquitecturas no estructuradas, se envían mensajes utilizando las técnicas de inundación [35] y paseos aleatorios [33] para anunciar, descubrir y/o seleccionar recursos. La técnica de inundación consiste en enviar mensajes a todos los nodos vecinos (los cuales actúan de manera similar) y el proceso es detenido por un Valor de Tiempo de Vida (TTL, por sus siglas en inglés) que evita el envío de peticiones. Esta técnica tiene la desventaja de causar tráfico en la red y de poseer una complejidad exponencial [20], además de ser adecuada únicamente en sistemas de pequeña escala y/o aplicaciones P2P móviles. La técnica de inundación puede ser usada para anunciar RSs o seleccionar recursos *al vuelo*, a través del envío de consultas multi-atributo. De manera similar, la técnica *goosiping* [34][38] permite anunciar RSs utilizando agentes de software que intercambian estas especificaciones entre dos nodos seleccionados aleatoriamente. Esta técnica utiliza múltiples agentes para acelerar el proceso y un nodo selecciona recursos consultando los RSs obtenidos. Un nodo puede llevar a cabo empatamiento simple ya que conoce múltiples recursos, e.g., *¿Los recursos r1 y r2 son de mis amigos?* Aunque este enfoque es simple de implementar, no garantiza que un nodo conozca cualquier recurso necesitado (sin importar que exista). Además, no se puede conocer el estado de los recursos (e.g., el estado de atributos dinámicos como CPU libre y memoria) debido a que la propagación en la técnica *goosiping* es lenta e impredecible [1].

La mayoría de las soluciones no estructuradas que permiten llevar a cabo la agregación de recursos están basadas en la técnica de paseos aleatorios, en donde se envían mensajes a través de nodos seleccionados aleatoriamente con la finalidad de llevar a cabo el anun-

ciamiento de RSs, descubrimiento y consultas de recursos multi-atributo. Esta técnica es similar a *goosiping*, ya que un agente se encarga de llevar un conjunto de RSs o consultas multi-atributo de un nodo a otro (seleccionado aleatoriamente). Sin embargo, es posible enviar múltiples agentes a diferentes regiones de la red para anunciar RSs a otros nodos y, al mismo tiempo, ir recolectando RSs de estos (i.e., llevar a cabo de manera simultánea las fases de anunciamiento y descubrimiento) [33]. También es posible realizar un empatamiento simple de recursos, debido a que los agentes recolectan RSs de múltiples nodos. Aunque los agentes anuncian y descubren recursos, no garantizan resultados en la fase de selección de recursos [18].

La técnica de paseos aleatorios puede resolver consultas de recursos utilizando un agente que *camina* de un nodo a otro, en busca de recursos que satisfagan una consulta multi-atributo. Si se encuentran recursos relevantes, el agente regresa directamente al iniciador de la consulta o vuelve a través de la ruta que trazó (para garantizar el anonimato entre el proveedor y consumidor) [1]. Estos agentes poseen un TTL limitado para prevenir la acumulación de agentes de consulta sin resolución dentro del sistema P2P (máximo número de saltos). Con la finalidad de incrementar la probabilidad de descubrir recursos, se envía un agente de consulta con un gran TTL o se envían muchos agentes; sin embargo, no es trivial determinar el TTL o el número de agentes de consulta requeridos. Por lo tanto, estas soluciones únicamente proporcionan resultados en la fase de selección de recursos, basándose en el mejor esfuerzo. Los agentes de consulta pueden identificar el estado actual de un recurso y proporcionar la fase de enlace, debido a que checan la disponibilidad de los recursos de los nodos que contactan. Además, es posible construir un sistema en donde un conjunto de agentes se encargue de anunciar RSs, mientras que otro conjunto se encargue de consultar recursos. A pesar de que esta solución acelera la resolución de una consulta, elimina la posibilidad de enlazar recursos ya que las consultas son resueltas por nodos intermedios [18].

Las arquitecturas no estructuradas se consideran completamente descentralizadas. Usualmente, el descubrimiento de contenido se lleva a cabo fuera de la red P2P y el desempeño (e.g., *delay* o QoS) no se puede predecir [17]. Además, los nodos tienden a mantener una lista de vecinos, cuyas entradas son reemplazadas únicamente si un nodo falla. La topología de la red crece de una manera arbitraria y no estructurada, por lo que es difícil garantizar la conectividad entre grupos de nodos y, por lo tanto, es difícil garantizar la fase de enlace. Esta situación impacta en el desempeño y confiabilidad, por lo que de manera no intencional algunos nodos se pueden convertir en cuellos de botella [20].

Las arquitecturas no estructuradas son usadas en sistemas como KaZaA, Gnutella, BearShare, LimeWire y McAfee. Gnutella es altamente resistente a fallas de nodos, ya que cada uno de los nodos mantiene una lista dinámica de vecinos [12], sin embargo, no garantiza el descubrimiento de contenido, por lo que es considerado como un *sistema no estructurado no determinista* [18].

Actualmente, Gnutella y KaZaA poseen una topología en donde los nodos ricos en recursos (llamados *superpeers*) forman un *overlay* separado y actúan como proxies para el resto de los nodos (ver figura 2.15). Un nodo (también conocido como *peer*) con una alta capacidad de ancho de banda, gran poder de procesamiento y gran capacidad de almacenamiento es el candidato perfecto para ser un *superpeer*, el cual da seguimiento a los RSs de los *peers* que se encuentran bajo su responsabilidad. Además anuncia, descubre, selecciona y/o enlaza recursos en nombre de sus *peers*, contactando a otros *superpeers* a través de las técnicas de inundación, *goosiping* o paseos aleatorios. Este enfoque permite la resolución de consultas multi-atributo, debido a que los *superpeers* están conscientes de múltiples RSs. Además, el costo de las fases de anunciamiento, descubrimiento y selección de recursos se reduce, ya que los *superpeers* son los únicos involucrados [1]. El costo de la fase de anunciamiento se puede reducir, anunciando únicamente la información de recursos indexados en un *superpeer*, e.g., el número total de núcleos de CPU o la capacidad de procesamiento total (medida en FLOPS) de todos sus *peers* [34]. Sin embargo, dichos valores de atributos son muy abstractos para sistemas sensibles a la latencia, en donde el desempeño está determinado por interacciones entre recursos dinámicos e individuales. Los *superpeers* pueden resolver consultas del mundo real con un costo moderado [4] y pueden actuar como *matchmakers* en caso de que los *peers* de un mismo vecindario o con diferentes recursos se encuentren asignados al mismo *superpeer* [1]. Actualmente, el uso de *superpeers* no garantiza el descubrimiento de recursos, debido a la imprevisibilidad de la técnica *goosiping* y al ámbito limitado de las técnicas de inundación y paseos aleatorios. Además su aplicabilidad está limitada, ya que se enfocan únicamente en *peers* individuales y asumen que los requerimientos de la aplicación y *peers* de procesamiento están caracterizados únicamente por los MIPS (*Million Instructions Per Second)* [18].

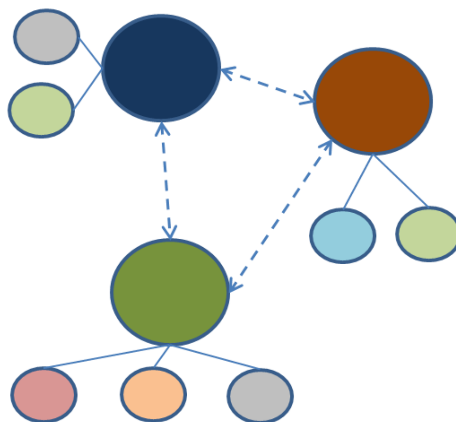


Figura 2.15: Arquitectura basada en *superpeers*.

Las técnicas mencionadas anteriormente proporcionan un servicio basado en el mejor esfuerzo (excepto cuando se utiliza la técnica de inundación) en las fases de descubrimiento, selección y enlace de recursos. Las arquitecturas P2P no estructuradas son aplicables en sistemas de pequeña a mediana escala y en ambientes altamente dinámicos, e.g., redes *ad-hoc* y redes sociales móviles. Además, poseen un buen balance de cargas debido a su

topología aleatoria. Es útil llevar a cabo consultas al vuelo (con múltiples atributos y un gran rango de valores) ya que se pueden resolver consultas complejas gracias a que se pueden contactar nodos de manera individual, por lo que además se simplifica la fase de enlace. Dependiendo de la implementación, es posible un empatamiento limitado de recursos; sin embargo, cuando un nodo no puede llevar cabo la fase de empatamiento satisfactoriamente, se informan todos los recursos obtenidos en la fase de selección a la aplicación que emitió la consulta. En este caso, la aplicación toma la decisión final sobre el empatamiento y enlace de recursos. Las arquitecturas no estructuradas dificultan el rastreo de las relaciones entre recursos, por lo que se obstaculiza la fase de empatamiento (excepto en *superpeers*). Por lo tanto, las relaciones entre recursos deben descubrirse al vuelo, e.g., después de seleccionar los recursos potenciales, se podrían solicitar los nodos proveedores para medir el ancho de banda y latencia entre ellos, verificar si se pueden alcanzar entre sí (e.g., enviando mensajes *ping*) o evaluar sus relaciones sociales. Sin embargo, el descubrimiento de tales relaciones al vuelo toma tiempo e incrementa el costo de la fase de empatamiento (e.g., se necesita un número de paquetes y tiempo para estimar el ancho de banda entre dos nodos). Los *superpeers* constituyen una alternativa viable, ya que cada uno de estos puede rastrear las relaciones entre recursos de los *peers* bajo su responsabilidad, por lo que facilitan la fase de empatamiento [18]. En la tabla 2.2 se muestra un resumen de los protocolos no estructurados que permiten la agregación de recursos.

Esquema	**Diseño**	**Anunciamiento**	**Descubrimiento***	**Selección**	**Empatamiento***	**Enlace***
Inundación [35]	Se envían mensajes recursivamente a todos los nodos vecinos	Sí	N/A	Garantizada	Sí, cuando se inunda con RSs	Sí, cuando se inunda con consultas
Goosiping [34][38]	Agentes que comparten los RSs que conocen	Sí	Sí	Sí, probabilidad moderada de éxito (mejor esfuerzo)	Sí, empatamiento simple	No
Paseos aleatorios [33]	Agentes de RSs y agentes de consulta	Sí	Sí	Sí, probabilidad moderada de éxito (mejor esfuerzo)	Sí, empatamiento simple	Sí, cuando existen agentes de consulta
Superpeers [39][34][38]	Dos capas	Sí	Sí, aunque no está garantizado	Garantizada	Sí, empatamiento simple	Sí

*N/A – No aplicable

Tabla 2.2: Protocolos P2P no estructurados de agregación de recursos.

Capítulo 3

Trabajo relacionado

En la actualidad, existen pocos sistemas que proporcionan soporte para la agregación de recursos y ninguno de ellos ofrece cohesión entre las fases clave de la agregación de recursos. Además, estos sistemas no pueden ser usados en el desarrollo de alguna aplicación o de otro sistema, utilizando algún lenguaje de programación específico.

En este capítulo se presentan y analizan los trabajos más importantes que están relacionados con *RASupport*. En primer lugar, se presenta GENI (*Global Environment for Network Innovations*) (sección 3.1), la cual es una plataforma exploratoria y colaborativa que permite realizar descubrimientos e innovación. Posteriormente, se presenta SWORD (sección 3.2), una herramienta de software que permite la agregación de recursos en sistemas distribuidos de área amplia. Para cada uno de los trabajos relacionados, se identifican, analizan y describen las fases de la agregación de recursos.

3.1. GENI (*Global Environment for Network Innovations*)

GENI (*Global Environment for Network Innovations*) [19] [1] es una plataforma exploratoria y colaborativa para realizar descubrimientos e innovación, patrocinada por la NSF (*National Science Foundation*). Se trata de un conjunto de infraestructuras de investigación, cuya finalidad es explorar el futuro de la Internet a gran escala. Esta plataforma permite a los usuarios realizar experimentos relacionados con el diseño y la evaluación de protocolos, la integración de redes sociales, la gestión de contenidos y el desarrollo de servicios de red. Para llevar a cabo estos experimentos, GENI permite agregar recursos (e.g., nodos de procesamiento, almacenamiento, redes, sensores y actuadores) a partir de múltiples dominios de administración. La figura 3.1 ilustra el *framework* de agregación de recursos de GENI, en donde un usuario solicita la creación de una rebanada (*slice*)[2];

[1] GENI se encuentra disponible de forma gratuita en http://www.geni.net.

[2] Un *slice* es un conjunto de recursos virtualizados que se asignan a un experimento en particular.

posteriormente, un repositorio central (*clearinghouse*) responde al usuario agregando un conjunto de recursos propagados a través de múltiples dominios de administración. En su diseño actual, un repositorio de GENI envía una lista de recursos potencialmente útiles al usuario esperando que este posea la habilidad de utilizarlos en un sistema en funcionamiento [18]. Actualmente, el portal del experimentador y GENI *clearinghouse* se encuentran en fase beta.

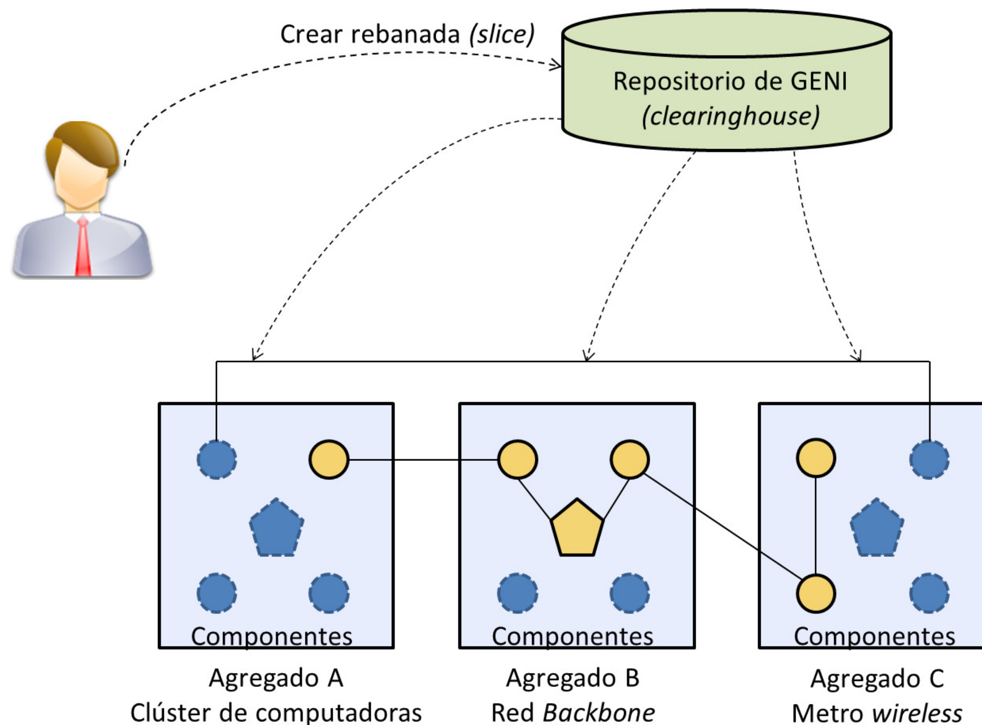


Figura 3.1: *Framework* de agregación de recursos de GENI.

Los experimentadores de GENI agregan recursos (e.g., recursos de cómputo y enlaces de red) a sus *slices* y ejecutan experimentos que usan estos recursos. Un agregado (*aggregate*) es una colección de recursos similares gestionados por un administrador que proporciona sus recursos a los experimentadores de GENI [19].

Los experimentadores solicitan recursos de un agregado usando herramientas específicas, las cuales utilizan el API GENI AM (GENI *Aggregate Manager* API). Esta API constituye una interfaz entre las herramientas de los experimentadores y los administradores de agregados (ver figura 3.3). El API GENI AM permite a los proveedores de recursos hacer disponibles (anunciar) sus capacidades en agregados, mientras que a los experimentadores les permite descubrir, seleccionar, usar y liberar recursos [19]. La última versión de esta API es la versión 3, la cual permite:

1. mostrar los recursos disponibles en un agregado (*Listresources*);

2. solicitar los recursos disponibles de un agregado para asignarlos a sus *slices* (*Allocate*);

3. consultar el estado de los recursos (asignados a sus *slices*) de un agregado (*Status*); y

4. eliminar recursos de sus *slices* (*Delete*).

El API GENI AM utiliza el protocolo XML-RPC con una conexión SSL (*Secure Sockets Layer*) sobre Internet para llevar a cabo la comunicación. Esta API utiliza documentos de especificación (conocidos como *RSpecs*), en un formato prescrito, para describir los recursos que poseen los administradores de agregados. Los experimentadores envían una *solicitud RSpec*, que describe los recursos que desean, a los agregados y obtienen un *manifiesto RSpec*, el cual describe los recursos que obtuvieron y se envía como copia al repositorio de GENI (ver figura 3.2). El *manifiesto RSpec* incluye información que los experimentadores necesitarán para usar estos recursos (e.g., nombres y direcciones IP de los recursos, cuentas de usuario creadas en los recursos y etiquetas VLAN asignadas a los enlaces de red). La mayoría de los experimentadores no necesita aprender detalles del API GENI AM o leer/escribir archivos *RSpec*, ya que las herramientas de los experimentadores ocultan gran parte de esta complejidad mediante interfaces de usuario (GUIs, por sus siglas en inglés) o líneas de comandos (CLIs, por sus siglas en inglés) [19]. Estas herramientas actúan como un agente por encima del API GENI AM, tal y como se observa en la Figura 3.3.

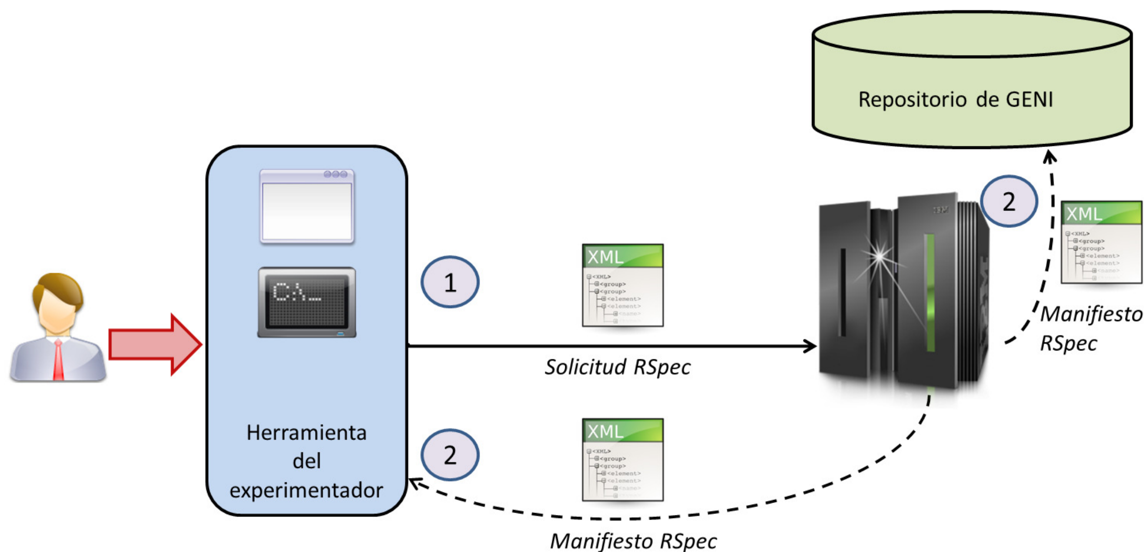


Figura 3.2: Selección de recursos en GENI.

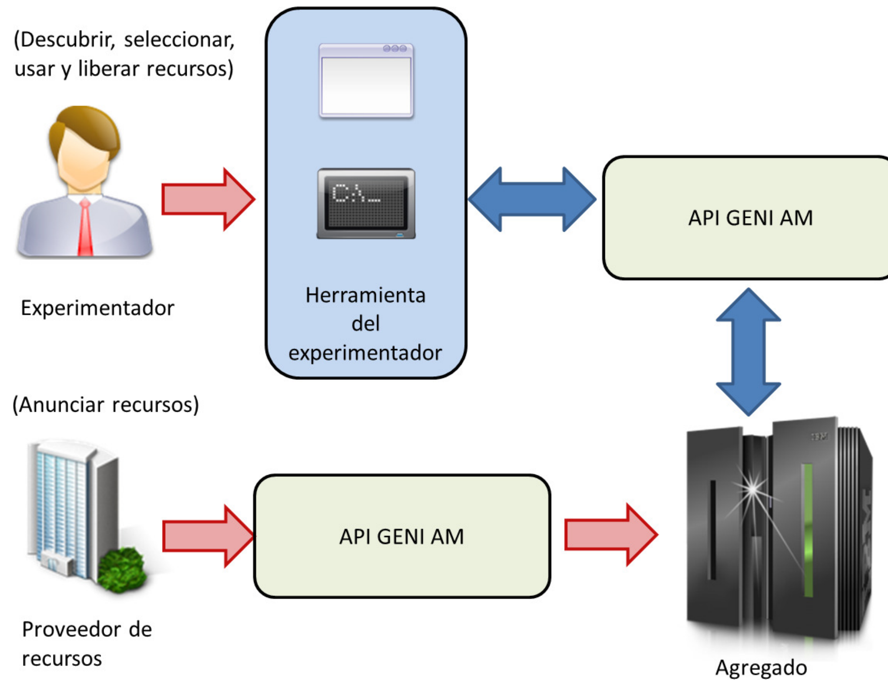


Figura 3.3: Uso del API GENI AM.

Existe un tercer tipo de *RSpec* llamado *RSpec de anunciamiento* que es la especificación de recursos regresada por un agregado cuando el experimentador lista los recursos disponibles en este, utilizando el comando *Listresources* (ver figura 3.4). Esta especificación describe todos los recursos disponibles en el agregado. Existen diversas herramientas de experimentadores que permiten llevar a cabo el descubrimiento de recursos en GENI (e.g., Flack, omni y Myslice), el cual es un proceso que consta de dos pasos (ver figura 3.4) [19]:

1. Se consulta algún repositorio de GENI para identificar a los administradores de agregados que poseen recursos disponibles. El repositorio responde con una lista de administradores de agregados.

2. Se consultan los recursos disponibles de los administradores de agregados, utilizando el comando *Listresources*. Los administradores de agregados responden con un *RSpec de anunciamiento*.

En la práctica, un usuario encontrará que es extremadamente complejo desarrollar un sistema usando GENI debido a lo siguiente [18]:

1. El usuario tiende a solicitar un conjunto de recursos arbitrarios confiando en sus instintos o en el *principio del menor esfuerzo* (i.e., tiende a especificar pocos o los recursos que sean) en lugar de tomar en cuenta los requerimientos específicos de la aplicación. Por lo tanto, el usuario termina solicitando muchos o un número insuficiente de recursos.

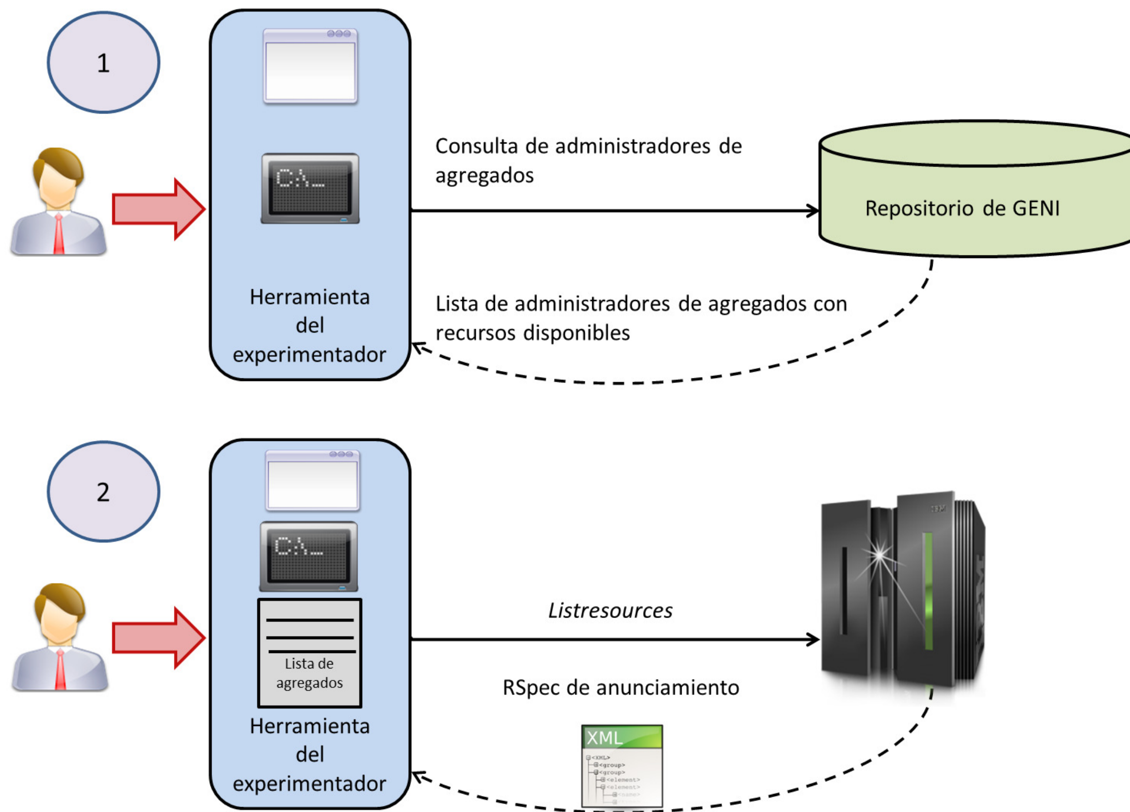


Figura 3.4: Descubrimiento de recursos en GENI.

2. Ya que el repositorio de GENI se enfoca únicamente en recursos individuales, no todas las combinaciones de recursos seleccionados pueden ser adecuadas o capaces de trabajar juntas (i.e., GENI no ofrece la fase de empatamiento), lo cual degrada la QoS de la aplicación del usuario o, en ciertos casos, el usuario puede no ser capaz de construir una solución que funcione por completo. Idealmente, un repositorio de GENI debe evitar estas complejidades para el usuario y debe ser capaz de agregar inteligentemente un conjunto óptimo de recursos, basándose en los requerimientos de la aplicación (i.e., debe llevar a cabo las fases de la agregación de recursos de una manera óptima). Para ello, se necesitan tomar en cuenta los recursos y las relaciones entre estos, por ende, es necesario considerar la fase de empatamiento.

3. Un repositorio centralizado será insuficiente, ya que GENI continúa creciendo y adquiriendo usuarios y recursos que están geográficamente distribuidos. Por lo tanto, GENI debería ser desarrollado como un sistema P2P colaborativo debido a su naturaleza distribuida, dinámica y colaborativa.

4. No ofrece soporte para crear consultas personalizadas de tipo *zero-attribute*, *single-attribute* o *multi-attribute*.

5. No soporta el desarrollo de aplicaciones que necesitan agregar recursos mediante

algún lenguaje de programación específico.

En la figura 3.5 se muestra el flujo de trabajo de un experimento en GENI. Este flujo de trabajo inicia una vez que un proveedor de recursos ha anunciado sus capacidades en un agregado, i.e., una vez que se ha llevado a cabo la fase de anunciamiento. En esta misma figura se observa que GENI lleva a cabo las fases de descubrimiento, selección, enlace, uso y liberación; sin embargo, no lleva a cabo la fase de empatamiento. A continuación, se presenta la manera en que se llevan a cabo fases de la colaboración de recursos en GENI:

1. **Fase de anunciamiento** (ver figura 3.3): el API GENI AM permite a los proveedores de recursos hacer disponibles (i.e., anunciar) sus capacidades en agregados a través de RSpecs [19]. Sin embargo, estos recursos no se anuncian explícitamente a los experimentadores.

2. **Fase de descubrimiento** (ver figura 3.4): se utilizan RSpecs para describir recursos. El experimentador tiene que solicitar dichos recursos utilizando alguna herramienta en particular, la cual utiliza el API GENI AM. Cuando tal herramienta lista los recursos existentes en un agregado (utilizando el comando *Listresources*), este último responde con una especificación de recursos (*RSpec de anunciamiento*) [19].

3. **Fase de selección** (ver figura 3.2): una vez que el experimentador ha listado los recursos disponibles en un agregado, selecciona los recursos que necesita (a través de la herramienta de experimentador) independientemente de los requerimientos específicos de la aplicación. El experimentador envía una *solicitud RSpec* (que describe los recursos que se desean) a los agregados y obtiene un *manifiesto RSpec* (que describe los recursos que se obtuvieron) [19]. El *manifiesto RSpec* es el resultado de una consulta de recursos, los cuales se asignan a un *slice* utilizando la llamada al comando *Allocate* del API GENI AM. El comando *Provision* permite instanciar recursos en un agregado.

4. **Fase de empatamiento**: no soportada (N/S).

5. **Fase de enlace**: a través de la herramienta del experimentador, se determina si los recursos seleccionados están disponibles para su uso. Para ello, se realiza una llamada al comando *Status* del API GENI AM.

6. **Fase de uso**: los recursos se pueden utilizar haciendo uso del comando *PerformOperationalAction* del API GENI AM. El comando *Renew* permite cambiar la expiración de un recurso, de manera que este se pueda continuar usando.

7. **Fase de liberación**: los recursos de un agregado se liberan llamando al comando *Delete*. Si el administrador de un *slice* desea detener todos los recursos de un experimento, se utiliza el comando *Shutdown*.

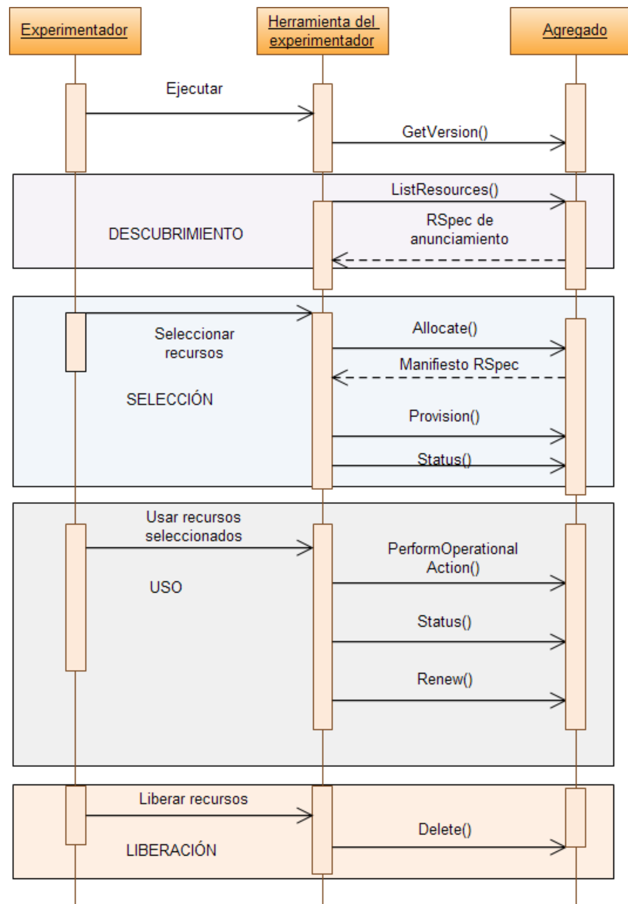


Figura 3.5: Flujo de trabajo en GENI.

3.2. SWORD

SWORD (*Scalable Wide-Area Resource Discovery*)[3] [36] es un servicio de agregación de recursos desarrollado en lenguaje Java para sistemas distribuidos de área amplia, el cual permite describir y agregar recursos utilizando una topología de grupos interconectados. SWORD recolecta reportes de recursos dinámicos y estáticos disponibles en nodos proveedores, además permite consultas de recursos en las que se pueden definir grupos de recursos requeridos, restricciones por nodo (e.g., memoria y espacio en disco disponibles), restricciones entre nodos (e.g., latencia entre nodos) y restricciones entre grupos (e.g., ancho de banda entre grupos), así como penalizaciones por no satisfacer dichos requerimientos. La figura 3.6 ilustra la manera en que se lleva a cabo la agregación de recursos en SWORD.

[3]Disponible de forma gratuita en http://sword.cs.williams.edu/.

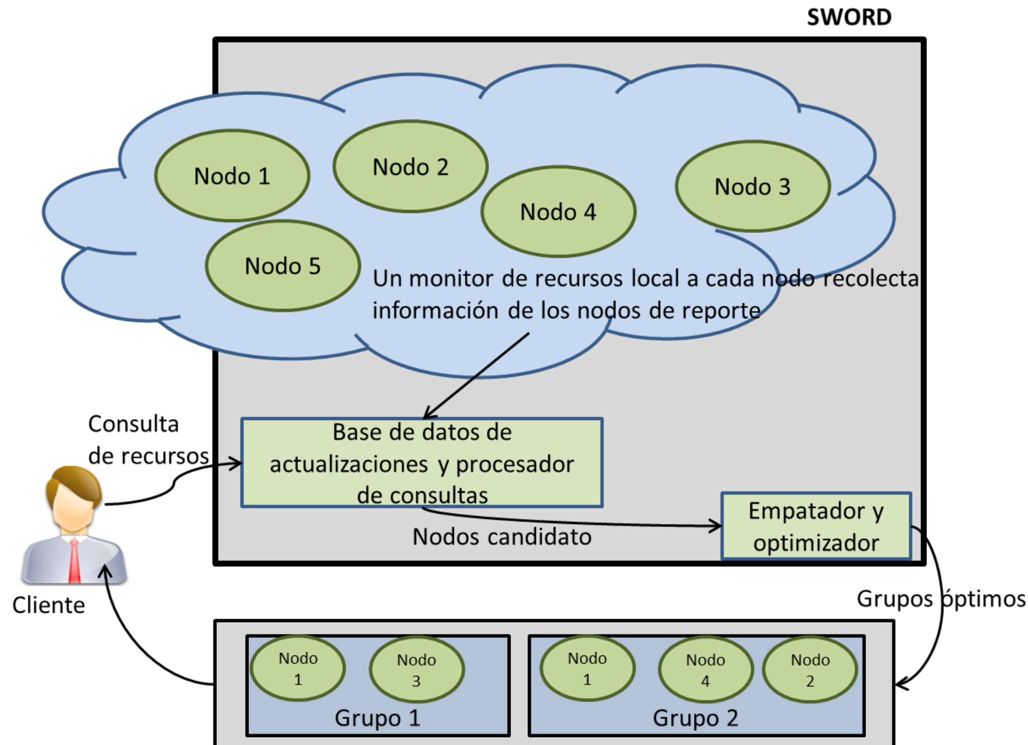


Figura 3.6: Agregación de recursos en SWORD.

Actualmente, SWORD es una herramienta de software que se puede integrar en PlanetLab[4] [42] (en donde los recursos disponibles son suministrados por CoTop[5]) y existen dos versiones: distribuida y centralizada. PlanetLab actualmente emplea la versión centralizada, sin embargo esta cuenta con un único punto de fallo. La versión distribuida solo se encuentra especificada teóricamente (i.e., no existe implementación alguna), presenta un buen desempeño y se beneficia de la resiliencia de las DHTs. Por lo tanto, la versión distribuida presenta una arquitectura estructurada y está basada en un anillo particionado, como el que se muestra en la figura 2.13 (b). Por ejemplo, puede existir un servidor gestor de actualizaciones de todos los nodos con cargas entre 1.0 y 2.0, etc; otro servidor para cargas entre 3.0 y 4.0; otro servidor para espacio libre en disco entre 0MB y 99MB; otro servidor para espacio libre en disco entre 100MB y 199MB, etc. SWORD únicamente soporta las fases de anunciamiento (explícita por parte de los nodos de reporte), selección (a través del procesador de consultas) y empatamiento (a través del optimizador) de recursos estáticos, dinámicos, de un solo atributo y multi-atributo.

Los *nodos de reporte* son aquellos que anuncian los atributos de sus recursos, utilizando un documento XML de especificación de recursos. Para llevar a cabo dicho anunciamiento, SWORD emplea uno de cuatro enfoques disponibles: *SingleQuery*, *MultiQuery*, *Fixed*

[4]PlanetLab es una red de investigación global que soporta el desarrollo de nuevos servicios de red.
[5]CoTop es una herramienta de software que permite monitorear los recursos disponibles en un nodo.

o *Index*. Los enfoques *SingleQuery*, *MultiQuery* e *Index* son estructurados, por lo cual, están basados en un DHT; por otro lado, *Fixed* es un enfoque centralizado. Cuando se utiliza *MultiQuery* (ver figura 3.7(a)), los nodos de reporte anuncian N atributos a N servidores, i.e., existe una correspondencia atributo-servidor. En el enfoque *Fixed* (ver figura 3.7(b)), los nodos de reporte siempre anuncian sus atributos a un mismo servidor que le fue asignado al unirse a la infraestructura de SWORD; la división de trabajo para los servidores es: $\frac{numNodosReporte}{numServidores}$. Finalmente, en los enfoques *SingleQuery* e *Index* (ver figura 3.7(c)), los nodos de reporte anuncian sus N atributos a N servidores, los cuales a su vez, indexan el rango de valores que corresponde a cada atributo.

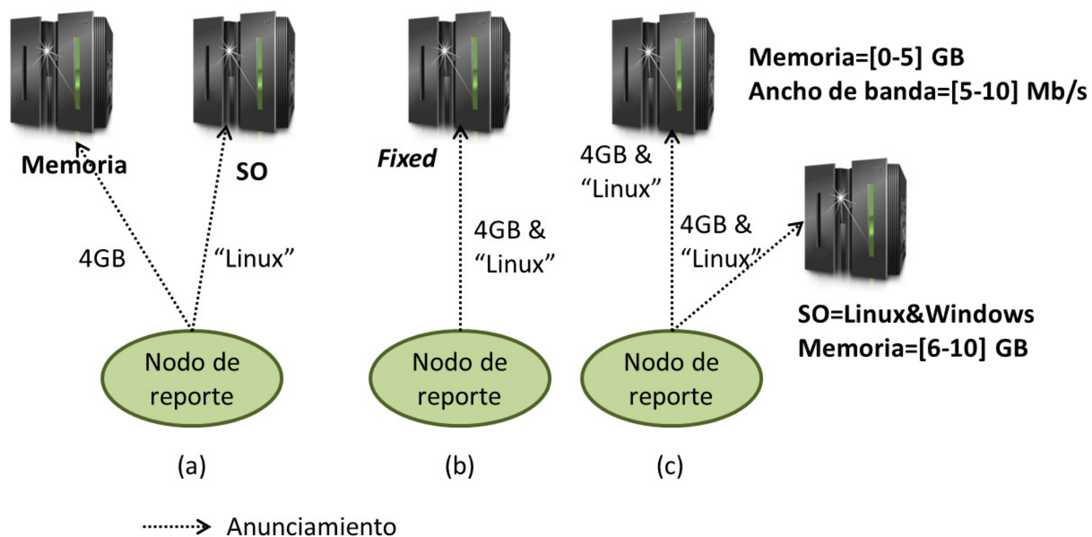


Figura 3.7: Anunciamiento de recursos en SWORD (Memoria=4GB ∧ SO="Linux"): (a) *MultiQuery*; (b) *Fixed*; (c) *SingleQuery* e *Index*

SWORD soporta dos lenguajes de consultas: SWORD XML (archivo XML modificado para mayor claridad) (ver figura 3.8(a)) y EBNF (ver figura 3.8(b)). Además, soporta Condor ClassAd[6] para tener compatibilidad con aplicaciones *grid* existentes, sin embargo este lenguaje no soporta restricciones entre nodos, por lo que únicamente permite restricciones por nodo y las clasificaciones de grupos se basan en dichas restricciones [36]. La fase de selección de recursos es llevada a cabo por el procesador de consultas, el cual genera una lista de los recursos que satisfacen los requerimientos de un usuario; posteriormente, dicha lista se ordena de acuerdo a los grupos que satisfacen mejor los requerimientos. Las consultas de recursos especifican el período de tiempo durante el cuál se usarán los recursos solicitados y los resultados se filtran para excluir a los nodos que no están disponibles durante dicho período de tiempo.

[6]Condor ClassAd es un lenguaje que sirve para representar las características y restricciones de máquinas y trabajos en un sistema Condor. A cada expresión se le denomina atributo, e.g., Arch="Intel".

```
Group NA
   NumMachines 4
   Required Load [0.0, 2.0]
   Preferred Load [0.0, 1.0], penalty 100.0
   Required Free Disk [500.0, MAX] (MB)
   Preferred Free Disk [100.0, MAX], penalty 100.0
   Required OS["Linux"]
   Required AllPairs Latency [0.0, 20.0] (ms)
   Preferred AllPairs Latency [0.0, 10.0] (ms), penalty 100.0
   Required AllPairs BW [0.5, MAX] (Mb/s)
   Required AllPairs BW [1.0, MAX] (Mb/s), penalty 2.0
   Required Location ["NorthAmerica", 0.0, 50.0] (ms)

Group Europe
   NumMachines 4
   Required Load [0.0, 2.0]
   Preferred Load [0.0, 1.0], penalty 100.0
   Required Free Disk [300.0, MAX] (MB)
   Preferred Free Disk [100.0, MAX], penalty 100.0
   Required OS["Linux"]
   Required AllPairs Latency [0.0, 20.0] (ms)
   Preferred AllPairs Latency [0.0, 10.0] (ms), penalty 100.0
   Required AllPairs BW [0.5, MAX] (Mb/s)
   Required AllPairs BW [1.0, MAX] (Mb/s), penalty 2.0
   Required Location ["Europe", 0.0, 50.0] (ms)

InterGroup
   Required OnePair BW NA Europe [3.0, MAX] (Mb/s)
   Preferred OnePair BW NA Europe [5.0, MAX] (Mb/s), penalty 0.5
```

a)

```
<QUERY> ::= <GRP>+ <INTERGRP>?
<GRP> ::= <GRPNAME> <NUMMACHINES>? (<REQ> <PREF>?)+
<GRPNAME> ::= 'Group' <GNAME>
<GNAME> ::= <STRING>
<NUMMACHINES> ::= 'NumMachines' <INTEGER>
<REQ> ::= <PERREQ> | <INTERREQ>
<PERREQ> ::= 'Required' <ATTRNAME> [<DOUBLE> | 'Min', <DOUBLE> | 'Max'] |
             'Required' <ATTRNAME> ['True'? 'False'?] |
             'Required' <ATTRNAME> [<STRING>+] |
             'Required' <ATTRNAME> [<REFPOINT>+]
<REFPOINT> ::= <LOCATION>, <DOUBLE> | 'Min', <DOUBLE> | 'Max'
<INTERREQ> ::= 'Required' 'OnePair' <ATTRNAME> [<DOUBLE> | 'Min', <DOUBLE> | 'Max'] |
               'Required' 'AllPairs' <ATTRNAME> [<DOUBLE> | 'Min', <DOUBLE> | 'Max']
<PREF> ::= <PERPREF> | <INTERPREF>
<PERPREF> ::= 'Preferred' <ATTRNAME> [<DOUBLE> | 'Min', <DOUBLE> | 'Max'],
                  'penalty' <PENALTY> |
              'Preferred' <ATTRNAME> [('True', <PENALTY>)? ('False', <PENALTY>)?] |
              'Preferred' <ATTRNAME> [(<STRING>+, <PENALTY>)+] |
              'Preferred' <ATTRNAME> [(<REFPOINT>+, <PENALTY>)+]
<INTERPREF> ::= 'Preferred' 'OnePair' <ATTRNAME> [<DOUBLE> |
                    'Min', <DOUBLE> | 'Max'], <PENALTY> |
                'Preferred' 'AllPairs' <ATTRNAME>
                    [<DOUBLE> | 'Min', <DOUBLE> | 'Max'], 'penalty' <PENALTY>
<INTERGRP> ::= (<IREQ> <IPREF>*)+
<IREQ> ::= 'Required' 'OnePair' <ATTRNAME> <GNAME> <GNAME>
               [<DOUBLE> | 'Min', <DOUBLE> | 'Max'] |
           'Required' 'AllPairs' <ATTRNAME> <GNAME> <GNAME>
               [<DOUBLE> | 'Min', <DOUBLE> | 'Max']
<IPREF> ::= 'Preferred' 'OnePair' <ATTRNAME> <GNAME> <GNAME>
               [<DOUBLE> | 'Min', <DOUBLE> | 'Max'], 'penalty' <PENALTY> |
            'Preferred' 'AllPairs' <ATTRNAME> <GNAME> <GNAME>
               [<DOUBLE> | 'Min', <DOUBLE> | 'Max'], 'penalty' <PENALTY>
<PENALTY> ::= <DOUBLE>
```

b)

Figura 3.8: Lenguajes de consultas de recursos en SWORD: (a) SWORD XML; (b) EBNF.

Para llevar a cabo las consultas de recursos, SWORD cuenta con un nodo que actúa como *proxy* dentro de la infraestructura. A este nodo se le conoce como nodo de consultas y envía sub-consultas a servidores remotos empleando *SingleQuery*, *MultiQuery*, *Fixed* o *Index*. Sin importar el enfoque que se utilice, el nodo de consultas regresa una lista con los nodos candidatos (i.e., aquellos que satisfacen los requerimientos). *MultiQuery* es similar a las consultas de recursos de XenoSearch[7] [43] y es un enfoque en el que el nodo de consultas envía una copia de la consulta en paralelo a cada grupo de servidores que indexan un atributo en particular; posteriormente, intersecta la lista de nodos regresados para encontrar aquellos que satisfacen todos los atributos de la consulta (ver figura 3.9(a)). En el enfoque *Fixed*, el nodo de consultas envía la consulta a cualquier servidor y este, a su vez, la envía a los demás servidores, recolecta resultados y los regresa al nodo de consultas (ver figura 3.9(b)). *SingleQuery* funciona de manera similar a Mercury [24] y las consultas multi-atributo solo se envían al(los) servidor(es) responsable(s) del rango de algún atributo solicitado; dicha consulta es reenviada a través de los nodos sucesores del DHT (ver figura 3.9(c)). Finalmente, cuando se utiliza *Index*, las consultas se realizan en tres saltos: primero se envía al(los) servidor(es) índice, después al(los) servidor(es) DHT que almacena(n) actualizaciones de nodos de reporte y, finalmente, se regresa el resultado al nodo de consultas (ver figura 3.9(d)).

[7] XenoSearch es un mecanismo de consulta de recursos desarrollado para la plataforma abierta denominada XenoServer.

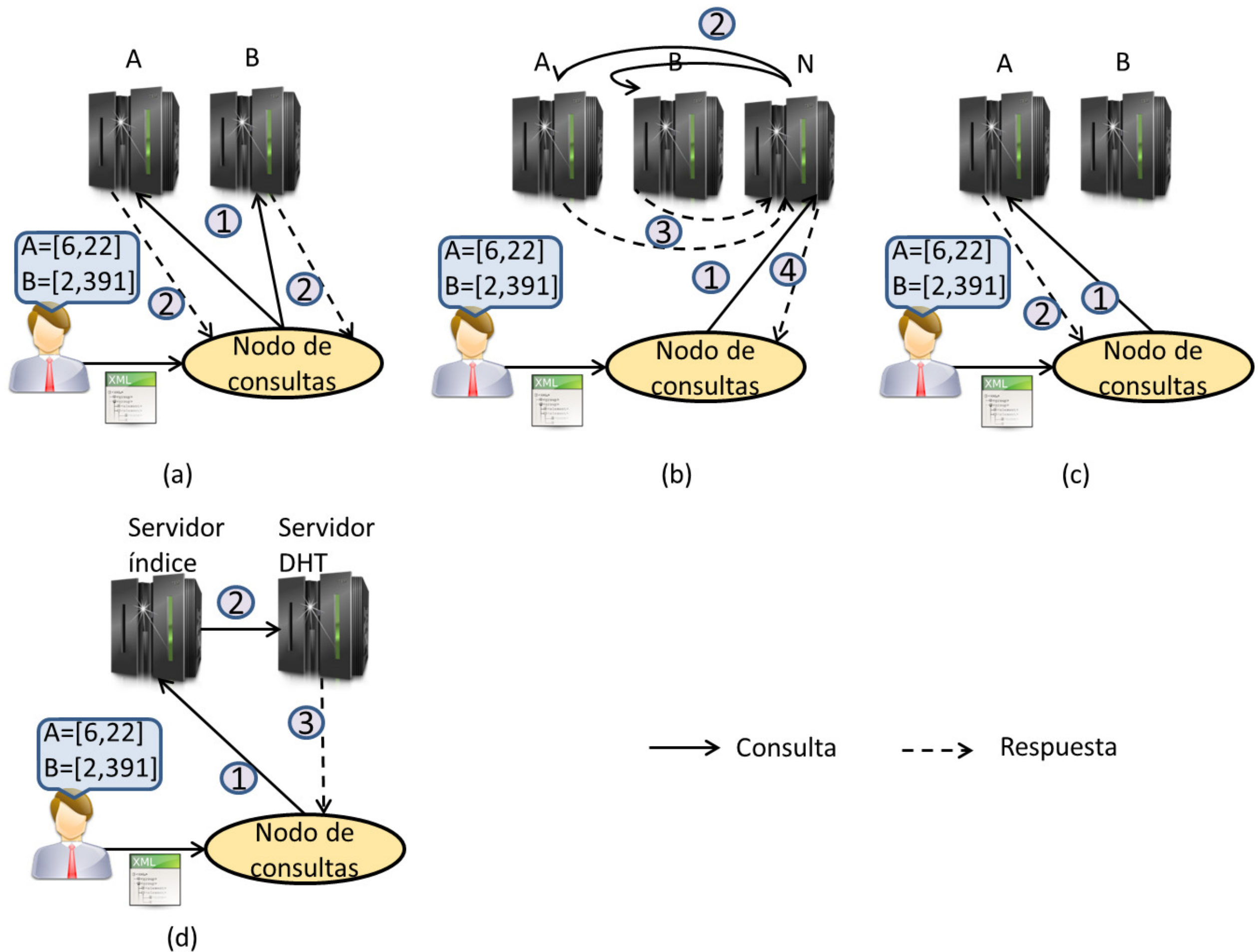


Figura 3.9: Selección de recursos en SWORD: (a) *MultiQuery*; (b) *Fixed*; (c) *SingleQuery*; (d) *Index*.

Una vez que se ha completado la fase de selección, el nodo optimizador lleva a cabo la fase de empatamiento de recursos, encontrando la penalización de todas las posibles combinaciones de nodos y dando preferencia a la búsqueda exponencial de grupos que probablemente tienen una baja penalización, por lo que regresa un conjunto de nodos con la menor penalización para grupos especificados en la consulta. La tarea de crear grupos de un tamaño específico que satisfagan las restricciones entre nodos y entre grupos es NP-difícil. Debido a que es impráctico rastrear todas las relaciones entre recursos, SWORD agrupa nodos en clases de equivalencia basándose en un sistema autónomo (SA). Se designa a un nodo representante para que rastree las relaciones de latencia y ancho de banda entre dos SA. El empatamiento de recursos se encuentra dividido en dos fases (ver figura 3.10):

1. El optimizador genera un conjunto de *grupos candidatos* que reúnen los requeri-

mientos por nodo y entre nodos para cada grupo.

1.1 Para cada grupo G en la consulta, el optimizador muestra todas las combinaciones de nodos que reúnen los requerimientos por nodo.

1.2 Para cada combinación, se calcula la penalización total por nodo.

1.3 Para cada combinación, se calcula la penalización total entre nodos (la cual es infinita si el conjunto no reúne los requerimientos). Las combinaciones de nodos que no tengan penalización infinita, se vuelven grupos candidato. SWORD permite definir un tiempo límite para esta fase y evalúa cada combinación dando prioridad a los grupos con menor penalización por nodo (i.e., de menor a mayor penalización). De esta manera, si el tiempo expira antes de que el optimizador haya evaluado todas las combinaciones, por lo menos habrá evaluado los conjuntos de nodos con la menor penalización total por nodo. Dichos conjuntos generalmente también tienen la menor penalización total por grupo, la cual se determina una vez que se ha calculado la penalización entre nodos por grupo.

2. El optimizador prueba y clasifica los grupos candidatos basándose en las preferencias y los requerimientos entre grupos.

   2.1 El optimizador se comporta de manera similar a la etapa anterior, pero en lugar de buscar grupos a partir de nodos, busca grupos a partir de grupos (i.e., los mejores grupos) comenzando con los grupos candidatos de menor penalización.

El algoritmo de búsqueda exponencial para encontrar la menor penalización puede tomar mucho tiempo, por lo que se implementaron una serie de heurísticas simples para detener la búsqueda más rápido:

- **Timeout de tres segundos**: detiene la búsqueda después de tres segundos. Se divide el tiempo asignado en partes iguales entre las dos fases del optimizador y el tiempo de la primera fase se sub-divide en partes iguales entre todos los grupos G. Por ejemplo, si se tienen 3 segundos como *timeout*, se asignan 1.5s para la primera fase y otros 1.5s para la segunda fase. Si se tienen 3 grupos en la consulta, entonces en la primera fase se asignan $\frac{1.5}{3}$ s para cada grupo, i.e., 0.5s para cada grupo.

- **Mitad del top de candidatos**: busca únicamente en la mitad de los grupos candidatos con menor penalización.

- **Top 5 de candidatos**: elimina todos los grupos candidatos, excepto los primeros 5 con la menor penalización.

- **Primera respuesta**: una vez que los grupos candidatos con la menor penalización se han procesado, la primera respuesta puede ser insuficiente. Esta heurística detiene el proceso una vez que se encuentra la primera solución válida.

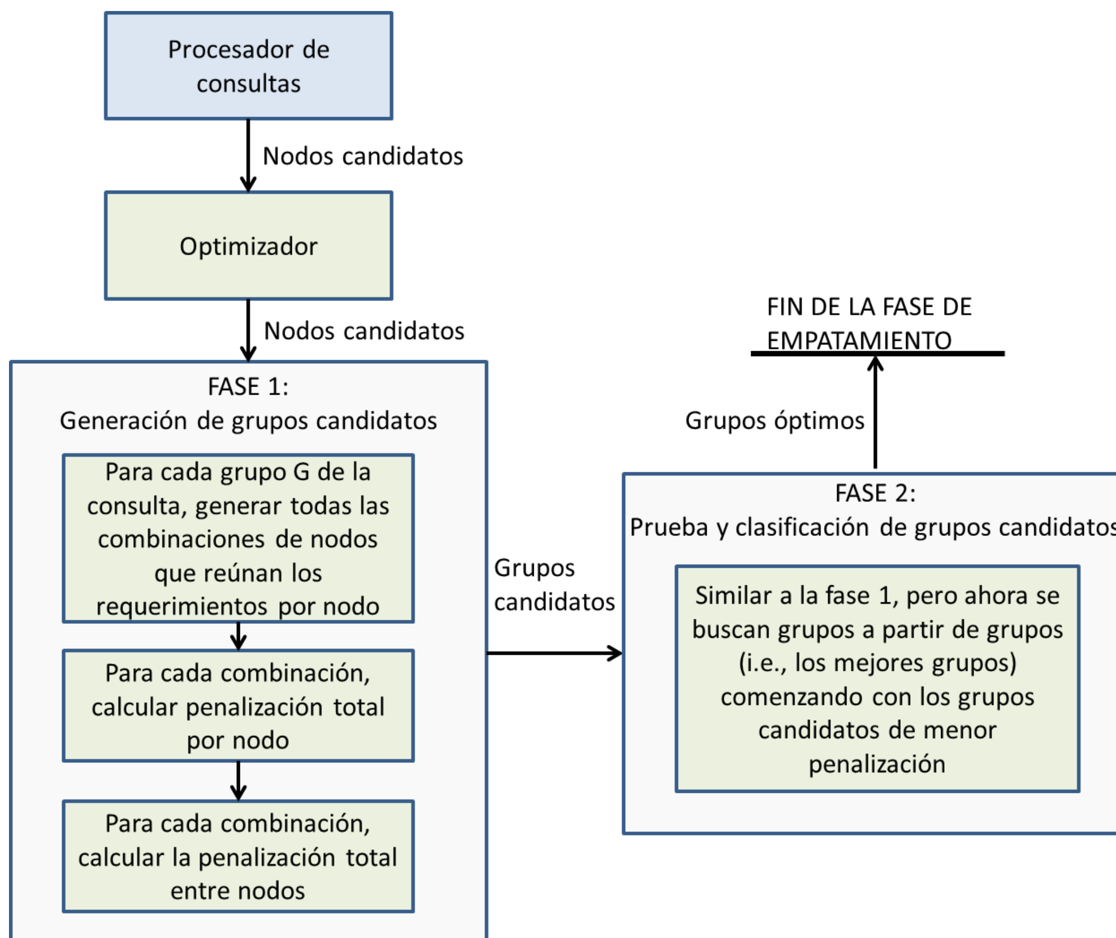


Figura 3.10: Empatamiento de recursos en SWORD.

Los recursos se seleccionan utilizando SADQ (similar a MAAN [26]) o múltiples subconsultas (similar a LORM [25]). Los recursos seleccionados se utilizan para formar un conjunto de grupos candidatos, con la finalidad de satisfacer los requerimientos de la aplicación. Posteriormente, los recursos en cada grupo candidato se empatan basándose en las relaciones entre SAs. Los grupos se ordenan de acuerdo a su capacidad para satisfacer las restricciones de la aplicación y, finalmente, se envían a dicha aplicación con la finalidad de que esta tome la decisión final sobre qué grupo(s) de recursos utilizar [18].

En la práctica, un usuario encontrará que SWORD no es útil para desarrollar aplicaciones P2P colaborativas debido a lo siguiente:

- Las consultas populares en cargas de trabajos reales solicitan recursos dinámicos con un amplio rango de valores de atributos [4], sin embargo SWORD solo es aplicable en ambientes semi-dinámicos (e.g., cómputo en *grid*) en donde los recursos no cambian rápidamente.
- La versión distribuida de SWORD cuenta con un único punto de fallo: el servidor

DHT.

- No proporciona la fase de enlace requerida por sistemas P2P colaborativos para garantizar la disponibilidad de los recursos. Por lo tanto, no proporciona cohesión entre las fases clave de la agregación de recursos.
- En la versión distribuida de SWORD, los nodos proveedores envían actualizaciones de atributos cada dos segundos. Sin embargo, este enfoque no es útil cuando se necesita dar soporte a atributos que cambian rápidamente.
- Actualmente, PlanetLab implementa la versión centralizada de SWORD. La versión distribuida únicamente se encuentra especificada teóricamente.
- En la versión distribuida de SWORD, los atributos populares en cargas de trabajo reales [4] lo forzan a utilizar muchos segmentos, mientras que otros rara vez son utilizados [1].
- Actualmente, no hay soporte para requerimientos entre nodos y entre grupos, debido a que los sensores de latencia y ancho de banda no están disponibles en PlanetLab.
- Si el servidor Web que hospeda a SWORD no responde, entonces no se pueden utilizar los servicios que ofrece dicha plataforma [8].
- Aunque las medidas basadas en SAs proporcionan una estimación razonable de latencia, no determinan adecuadamente el ancho de banda y otras complejas relaciones entre recursos [18].
- No existe algún soporte para integrar la infraestructura de SWORD, empleando algún lenguaje de programación específico.

En la figura 3.11 se muestra el flujo de trabajo de SWORD, en donde se observa que únicamente se llevan a cabo las fases de anunciamiento, selección y empatamiento. La fase de enlace no se lleva a cabo y la fase de descubrimiento no es aplicable (debido a que los recursos se anuncian explícitamente). A continuación, se describe la manera en que SWORD lleva a cabo de la agregación de recursos:

1. **Fase de anunciamiento** (ver figura 3.7): los nodos de reporte anuncian explícitamente sus recursos empleando *SingleQuery*, *MultiQuery*, *Fixed* o *Index*.
2. **Fase de descubrimiento**: no aplicable (N/A), debido a que los recursos se anuncian explícitamente.
3. **Fase de selección** (ver figura 3.9): los recursos se seleccionan a través de un procesador de consultas que emplea *SingleQuery*, *MultiQuery*, *Fixed* o *Index*.

[8]Durante el período de pruebas, SWORD no estuvo disponible debido a que el servidor Web que lo hospeda no respondía.

4. **Fase de empatamiento** (ver figura 3.10): los recursos se empatan a través de un optimizador que toma como entrada un conjunto de nodos candidato y da como salida un conjunto de grupos óptimos. Esta fase se encuentra dividida en dos etapas: generación de grupos candidatos y prueba y clasificación de grupos candidatos.

5. **Fase de enlace**: no soportada (N/S).

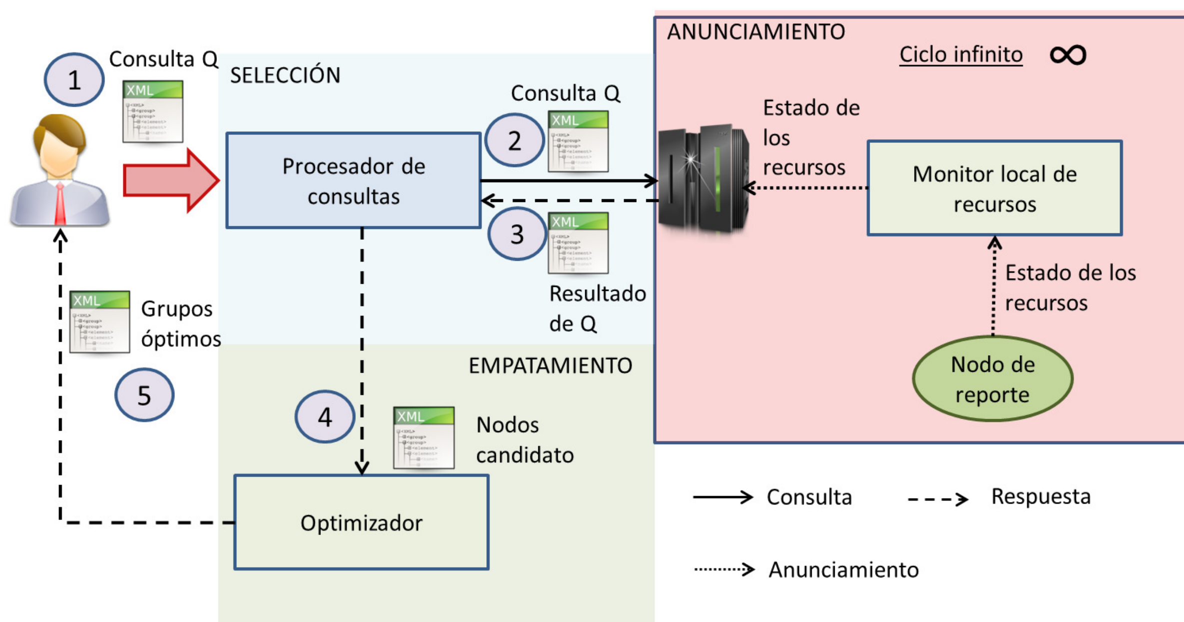


Figura 3.11: Flujo de trabajo en SWORD.

Capítulo 4

Análisis y diseño de *RASupport*

Frecuentemente, una aplicación incorpora clases de una o más bibliotecas predefinidas llamadas *toolkits*. Un *toolkit* es un conjunto de clases reusables y relacionadas, diseñadas para proporcionar funcionalidad de propósito específico y útil (e.g., un conjunto de clases para construir listas, tablas asociativas, pilas y otras estructuras similares). Los *toolkits* no imponen un diseño particular sobre la aplicación, simplemente proporcionan funcionalidad que puede ayudar a dicha aplicación a llevar a cabo su trabajo. En general, estos *toolkits* evitan tener que recodificar funcionalidad común [14].

El diseño de *toolkits* es más difícil que el diseño de aplicaciones, ya que estos deben integrarse en muchas aplicaciones para ser realmente útiles. El programador no sabe qué aplicaciones utilizarán su toolkit. Los *toolkits* poseen principalmente las siguientes características:

- Proporcionan funcionalidad de propósito específico, permitiendo el reuso de código,
- Pueden ser utilizados por cualquier aplicación (perteneciente al dominio para el que fueron diseñados) que los necesite,
- Permiten que el desarrollador tome la decisión sobre el diseño de la aplicación, i.e., no dictan el diseño de las aplicaciones que los utilizan,
- Permiten invocar subclases de bibliotecas directamente, debido a que comúnmente ofrecen fachadas para acceder a su funcionalidad.

El objetivo de este capítulo es describir el análisis y el diseño del soporte propuesto (denominado *RASupport*) que permitirá la agregación de recursos en sistemas P2P colaborativos. *RASupport* está constituído por un *toolkit* (denominado *RAToolkit*), el cual permite el reuso de código y proporciona la funcionalidad necesaria para el desarrollo de sistemas P2P colaborativos, de manera que el programador se puede concentrar en el propósito específico (fases de uso y liberación) de la aplicación. *RAToolkit* ofrece principalmente cuatro APIs (*Application Programming Interfaces*) que permiten llevar a cabo las fases de anunciamiento, selección, empatamiento y enlace (la fase de descubrimiento

se excluye, debido a que los recursos se anuncian explícitamente).

En particular, en este capítulo se describe el análisis correspondiente para seleccionar la arquitectura P2P apropiada para *RASupport* (sección 4.1). Por lo tanto, *RAToolkit* cuenta con un componente para mantener dicha arquitectura, cuyo diseño está basado en *super-peers* (sección 4.2). Además, en este capítulo se presenta la descripción de los recursos soportados por *RASupport* (sección 4.3), así como el análisis y diseño de los protocolos que permiten llevar a cabo las fases de la agregación de recursos: anunciamiento (sección 4.4), selección (sección 4.5), empatamiento (sección 4.6) y enlace (sección 4.7). Finalmente, se detalla el diseño de *RAToolkit*, el cual está basado en un sistema multi-agente, en donde múltiples agentes colaboran entre sí con la finalidad de llevar a cabo las fases previamente mencionadas (sección 4.8).

4.1. Análisis de arquitecturas P2P

Actualmente existe la necesidad de contar con un sistema P2P que combine las propiedades de las arquitecturas estructuradas y no estructuradas [18]. Así mismo, se tiene el requerimiento de contar con una solución que sea cohesiva y pueda anunciar, descubrir, seleccionar, empatar y enlazar recursos de una forma eficiente. Sin embargo, para la presente propuesta es necesario seleccionar alguna arquitectura existente, debido a la limitación del tiempo. La elección de una arquitectura apropiada depende de la cantidad de *peers* que se soportarán, la disponibilidad requerida de recursos, el nivel de carga esperado y los niveles requeridos de desempeño y disponibilidad [36].

En esta sección, se presentan las ventajas y desventajas que poseen las arquitecturas estructuradas (sección 4.1.1) y no estructuradas (sección 4.1.2), respectivamente. En particular, se seleccionó una arquitectura no estructurada, debido a que estas poseen una mayor resiliencia, se desempeñan mejor en sistemas altamente dinámicos y poseen una mayor flexibilidad en comparación con las arquitecturas estructuradas. Además, los protocolos no estructurados, que permiten llevar a cabo la agregación de recursos, soportan un mayor número de fases que los protocolos estructurados. En particular, se seleccionó una arquitectura no estructurada basada en *super-peers* debido a que estos anuncian, descubren, seleccionan, empatan y enlazan recursos en nombre de sus *peers*, con lo cual se ahorra significativamente ancho de banda de la red. La tabla 4.3 sintetiza una comparación entre las arquitecturas estructuradas y no estructuradas [18].

4.1.1. Ventajas y desventajas de las arquitecturas estructuradas

La mayoría de los sistemas P2P actuales se utilizan para la compartición de archivos y muchos de estos poseen una arquitectura estructurada [12]. Estas arquitecturas son apropiadas en sistemas de gran escala, en sistemas sensibles a la latencia (e.g., CASA y nubes

P2P) y en ambientes semi-dinámicos (en donde los recursos no cambian constantemente) [18]. En la tabla 4.1 se presentan las ventajas y desventajas que poseen estas arquitecturas.

Ventajas	**Desventajas**
Buena adaptación a la infraestructura de Internet	Ofrecen una cantidad finita de identificadores de recursos, ya que utilizan cadenas de bits de longitud fija
Alta escalabilidad, ya que es fácil agregar nuevos *peers*	No existe una relación explícita entre un recurso y su identificador
Buen desempeño en sistemas sensibles a la latencia	Se garantiza un buen desempeño solo si la red es consistente, sin embargo es costoso mantenerla de esta manera (excepto en Kademlia)
Es posible utilizar anillos sobrepuestos para reducir el ruteo y el costo de las consultas	Las consultas de recursos no siempre pueden resolverse
Si se utiliza una decisión de diseño como MADPastry, las arquitecturas estructuradas se pueden utilizar en redes móviles *ad-hoc* de gran escala	En MADPastry las consultas populares tienen alta latencia, debido al incremento de las dimensiones del hipercubo
Algunos protocolos estructurados permiten añadir nuevos atributos sin requerir cambios significativos en la topología de la red (e.g., Mercury, MAAN y LORM)	No se recomienda utilizar Mercury en sistemas caracterizados por múltiples atributos, debido a que se generarían muchos anillos
Los protocolos estructurados que utilizan SADQ presentan bajo costo de resolución de consultas	A excepción de MADPastry y del *backbone* de hipercubos, las demás soluciones estructuradas de agregación de recursos no proporcionan la fase de empatamiento
	La mayoría de los protocolos estructurados de agregación de recursos solo proporcionan el descubrimiento y selección de recursos individuales
	Son aplicables en ambientes semi-dinámicos

Tabla 4.1: Ventajas y desventajas de las arquitecturas P2P estructuradas

Como se puede observar en la tabla 4.1, las arquitecturas estructuradas presentan más desventajas en comparación con las ventajas que ofrecen. La mayoría de los protocolos P2P estructurados, que permiten la agregación de recursos, están basados en DHTs, sin

embargo el uso de estos también conlleva una serie de desventajas:

- Originalmente fueron diseñados para indexar recursos caracterizados por un único atributo, por lo que se tuvieron que diseñar nuevos protocolos para soportar múltiples atributos (e.g., Mercury y LORM) [1].
- Un *peer* no puede saber qué *peer* es responsable de un elemento de datos en particular [12].
- Requieren que todas las copias de una misma llave sean almacenadas en un mismo *peer* o vecindario, lo que conduce a únicos puntos de fallo [12].
- Son ineficientes en ambientes altamente dinámicos [1].

En general, no es deseable emplear una arquitectura estructurada para el desarrollo de sistemas P2P colaborativos debido a lo siguiente:

- Las decisiones de diseño basadas en DHTs son obsoletas para el desarrollo de sistemas P2P colaborativos, debido a que son ineficientes en ambientes dinámicos (e.g., nubes P2P) y difícilmente soportan múltiples atributos [18].
- Solo proporcionan el descubrimiento y la selección de recursos individuales. Sin embargo, los sistemas P2P colaborativos deben descubrir y seleccionar recursos caracterizados por múltiples atributos [1].
- Son aplicables en ambientes semi-dinámicos, en donde los recursos no cambian constantemente. Sin embargo, las consultas populares de sistemas P2P colaborativos típicamente solicitan recursos dinámicos [5].
- Poseen una baja resiliencia [1].
- Poseen una baja flexibilidad [1].

RASupport debe permitir la agregación de recursos dinámicos y multi-atributo en sistemas P2P colaborativos, por lo que no se utilizará una arquitectura estructurada.

4.1.2. Ventajas y desventajas de las arquitecturas no estructuradas

Las arquitecturas no estructuradas son apropiadas en sistemas de pequeña a mediana escala y en ambientes altamente dinámicos (e.g., redes *ad-hoc* y redes sociales móviles). En la tabla 4.2 se presentan las ventajas y desventajas que poseen dichas arquitecturas.

Ventajas	**Desventajas**
Buena resiliencia	Algunos *peers* se pueden convertir en cuellos de botella
Los *peers* simplemente se unen a la red de acuerdo a un conjunto de reglas predeterminadas	Es difícil garantizar la conectividad entre *peers*, i.e., dificultan la fase de enlace
Garantizan el descubrimiento de contenido cuando se cuenta con una una base de datos centralizada (e.g., Napster)	Una base de datos centralizada constituye un único punto de fallo y limita la escalabilidad
Son altamente flexibles	El desempeño (e.g., *delay* o QoS) no se puede predecir
Se pueden utilizar en ambientes altamente dinámicos	El descubrimiento de las relaciones entre recursos al vuelo es costoso y toma tiempo (en la fase de empatamiento)
Pueden utilizar *super-peers* para rastrear las relaciones entre recursos	Es difícil garantizar la fase de enlace debido a que la topología de la red crece de una manera arbitraria y no estructurada
Poseen un buen balance de cargas debido a su topología aleatoria	
Facilitan el anunciamiento, el descubrimiento, la selección y el empatamiento de recursos multi-atributo	
Muchos protocolos no estructurados, que permiten la agregación de recursos, facilitan las fases de selección y enlace	

Tabla 4.2: Ventajas y desventajas de las arquitecturas P2P no estructuradas

En la actualidad, los protocolos no estructurados que facilitan la agregación de recursos, permiten llevar a cabo un mayor número de fases que los protocolos estructurados (ver tablas 2.1 y 2.2) [1]. Es deseable que *RASupport* mantenga cohesión entre las fases clave de la agregación de recursos, que funcione en ambientes altamente dinámicos y que permita la agregación de recursos multi-atributo. Por lo tanto, se optó por utilizar una arquitectura no estructurada, así como protocolos de este tipo. Sin embargo, se planea que en el futuro *RASupport* también permita el desarrollo de sistemas P2P colaborativos con arquitecturas estructuradas, debido a que estas garantizan un mejor desempeño en aplicaciones sensibles a la latencia (e.g., CASA) y son adecuadas para sistemas de gran escala, a pesar de que poseen puntos únicos de fallo y una baja resiliencia.

En particular, *RASupport* utilizará una topología basada en *super-peers*, debido a que estos [18]:

- Monitorean las especificaciones de recursos de los *peers* que se encuentran bajo su responsabilidad.
- Anuncian, descubren, seleccionan y enlazan recursos en nombre de sus *peers*.
- Incrementan la velocidad de las consultas y reducen la latencia inherente en las búsquedas de contenido.
- Disminuyen el costo de las fases de anunciamiento, descubrimiento y selección de recursos, ya que son los únicos implicados en la comunicación requerida para llevar a cabo dichas fases.
- Pueden rastrear las relaciones entre recursos de los *peers* que se encuentran bajo su responsabilidad, por lo que facilitan la fase de empatamiento.
- Son capaces de resolver consultas del mundo real con un costo moderado.
- Reducen el costo de la inundación de peticiones y el tráfico de la red, debido a que son los únicos implicados.

A pesar de las ventajas evidentes que nos ofrecen las arquitecturas no estructuradas basadas en *super-peers*, también presentan algunas desventajas:

- Pocos *peers* (solo *super-peers*) se encargan de resolver consultas e indexar recursos, lo cual acarrea problemas de balance de cargas.
- Existen puntos de fallo en cada nivel, e.g., si un *super-peer* falla, desconectará a los *peers* que se encuentran bajo su responsabilidad.

Características	**Arquitecturas no estructuradas**	**Arquitecturas estructuradas**
Flexibilidad	Alta	Baja
Recursos	Indexados localmente (típicamente)	Indexados remotamente en una DHT (típicamente)
Mensajes de consulta	*Broadcast* o paseos aleatorios	*Unicast*
Descubrimiento de contenido	Mejor esfuerzo	Garantizado
Desempeño	No predecible	Predecible
Balanceo de cargas	Buena distribución de cargas	La carga no está balanceada cuando existen consultas o recursos populares
Resiliencia	Alta	Puntos únicos de fallo, sin embargo soportan moderadas tasas de fallos
Consistencia de los índices	Alta (los *peers* conservan sus recursos)	Baja (los recursos son indexados remotamente)
Ambientes aplicables	Ambientes altamente dinámicos o de pequeña escala (e.g., redes sociales móviles)	Ambientes relativamente estables o de gran escala (e.g., compartición de archivos en computadoras de escritorio)
Ejemplos	Gnutella, LimeWire, Kazaa, BitTorrent	Chord, CAN, Pastry, Kademlia, BitTorrent

Tabla 4.3: Arquitecturas estructuradas vs arquitecturas no estructuradas

4.2. Arquitectura bio-inspirada basada en *super-peers*

En Ciencias de la Computación, las metáforas y modelos biológicamente inspirados han sido objeto de investigación en los últimos años, debido a que posibilitan ciertas propiedades (e.g., resiliencia, adaptación emergente y auto-organización) que son deseables en sistemas distribuidos y de gran escala [49]. *RASupport* utiliza el protocolo Myconet [40] para gestionar una arquitectura P2P no estructurada basada en *super-peers* e inspirada en el crecimiento de los micelios. Dicho protocolo permite crear sistemas P2P auto-organizables y tolerantes a fallos; por ello, ha cobrado gran interés dentro de la comunidad científica y actualmente continúa en proceso de investigación[1][2][3]. Además agrupa *peers*, dando prioridad a aquellos con mayores capacidades de cómputo, lo cual puede resultar en un mejor

[1]MycoCloud [49] es una extensión al protocolo Myconet propuesta en el año 2013, la cual reduce el tiempo de reconfiguración de un sistema en caso de un fallo o de altos picos de demanda. Así mismo, los *peers* son capaces de cambiar su propio servicio y de migrar fácilmente a otro clúster, haciendo más flexible al sistema y mejorando la calidad de servicio ante eventos adversos en la red.

[2]HITAP (*Hormone-Inspired Topology Adaptation Protection*) [45] es un mecanismo presentado en el año 2011, el cual funciona sobre el protocolo Myconet difundiendo hormonas para detectar y mitigar ataques que se podrían suscitar en contra de la red P2P.

[3]SODAP (*Self-Organized Degree Adaptation Protection*) [50] es un mecanismo propuesto en el año 2013 (cuyo artículo está por aparecer), el cual funciona sobre el protocolo Myconet y constituye una mejora a HITAP.

desempeño e incremento de la tasa de convergencia (i.e., el tiempo requerido para obtener una configuración óptima). Myconet es eficiente cuando la topología de la red necesita ser re-organizada —incluso bajo eventos adversos— por lo que permite construir redes P2P con una buena resiliencia. Este protocolo muestra buenas capacidades para mantener fuertemente conectada la red P2P, la cual puede escalar a un gran número de *peers* [40]. A diferencia de otros enfoques, Myconet ajusta dinámicamente las conexiones entre *super-peers*, incrementando la eficiencia, eficacia y resiliencia ante la pérdida de *peers* [49]. Myconet supera a SG-1, ERASP y muchos otros protocolos de gestión de *super-peers* [40].

La topología propuesta posee dos tipos de *peers*: *super-peer* y *peer normal*. Un *super-peer* es aquel que efectúa peticiones en nombre de los *peers* que se encuentran bajo su responsabilidad, de manera que se encarga de seleccionar, empatar y enlazar recursos. Un *peer normal* es aquel que solicita recursos a su *super-peer* y recibe respuesta de este. Los *super-peers* están conectados arbitrariamente mediante grafos dirigidos, por lo que no necesariamente todos están conectados entre sí. Los *super-peers* mantienen una tabla de ruteo a sus *peers normales* y a otros *super-peers*, mientras que los *peers normales* únicamente están conectados con el *super-peer* que los gestiona.

Definición 1. *Sea S el conjunto universo de super-peers existentes, E el conjunto universo de grafos dirigidos y C el conjunto universo de soportes existentes de agregación de recursos. La arquitectura del soporte P2P de agregación de recursos $c \in C$ está definida como un grafo $G = (S_c, E_c)$, en donde S_c y E_c son el conjunto de super-peers y grafos dirigidos del soporte c, respectivamente, tal que $S_c \subset S$ y $E_c \subset E$.*

Definición 2. *Sean SP_i y SP_j dos super-peers, tal que $SP_i, SP_j \in S_c$ y e denota un grafo dirigido, tal que $e \in E_c$. SP_i y SP_j son vecinos en la arquitectura P2P de c si y solo si $e = (SP_i, SP_j)$.*

La figura 4.1 ilustra el diseño de Myconet [40], el cual está inspirado en los patrones de crecimiento de las estructuras tipo raíz de las hifas micóticas [40]. Dichos patrones tienen la principal característica de recolectar nutrientes. La red entera de raíces (red P2P) se denomina micelio y a una hebra individual (*super-peer*) se le denomina hifa. Las hifas constantemente están en busca de biomasa (*peers normales*). Por otro lado, los micelios constantemente se adaptan a cambios en las condiciones ambientales, dirigiendo nutrientes y biomasa a áreas de interés, por lo que una red P2P inspirada en micelios constantemente se auto-organiza aplicando un conjunto de reglas.

En Myconet, los *super-peers* y *peers normales* están caracterizados por un valor entero que denota su grado de capacidad para satisfacer las necesidades de otros *peers*, sin embargo tal abstracción puede variar dependiendo del propósito de la aplicación. *RASupport* hace uso de la jerarquía de dominancia de las avispas de la especie *Polistes dominula*

[4] "Hyphae" por *TheAlphaWolf* - Trabajo originial. Licencia bajo *Creative Commons Attribution-Share Alike 3.0* vía Wikimedia Commons - http://commons.wikimedia.org/wiki File:Hyphae.JPG♯mediaviewer/File:Hyphae.JPG.

Figura 4.1: Crecimiento de un micelio[4].

(popularmente conocida como "avispa cartonera") [46], la cual se describe a continuación.

Las avispas (i.e., los *peers*) de la especie mencionada anteriormente poseen una jerarquía de dominancia lineal con respecto a las hembras, en donde la hembra α domina a todas las otras en el nido (i.e., el *overlay* que forma un *super-peer*). La hembra β domina a las demás avispas excepto a α. De esta manera, la jerarquía continúa hasta que la última hembra es dominada por todas las demás. Esta jerarquía se origina a partir de peleas que surgen entre las avispas. Las hembras dominantes son responsables de tareas importantes (e.g., la construcción del nido) por lo que tienden a pelear con otras hembras; mientras que las hembras de menor jerarquía llevan a cabo tareas externas (e.g., recolección de comida) por lo que no se involucran tan fácilmente en peleas. Una hembra dominante es aquella que toma la iniciativa de comenzar una pelea y, cuando esta gana una pelea, activa su sistema endocrino. Por lo tanto, las avispas que ganan una pelea tienden a ganar más peleas, mientras que aquellas que pierden peleas se debilitan y tienden a seguir perdiendo peleas [46].

Cada avispa i que pertenece a un determinado nido se encuentra caracterizada por un valor de fuerza (denotado como F_i) que influye en su habilidad para ganar peleas. En una avispa i, el valor de F_i refleja su actividad endocrina y su desarrolo ovárico. Además, F_i es proporcional a la cantidad de peleas que gana la avispa i (i.e., cuando i gana una pelea, el valor de F_i se incrementa y, por el contrario, cuando pierde una pelea, el valor de F_i se decrementa).

Empleando el modelo biológico descrito anteriormente, los *peers* que pertenecen a la

red P2P del soporte c mantienen un valor de fuerza, el cual denota su capacidad de satisfacer las necesidades de otros *peers*. Debido a que la fase de selección puede requerir un buen ancho de banda entre *super-peers* y a que la fase de empatamiento puede requerir una alta capacidad de procesamiento, en *RASupport* la fuerza de los *super-peers* y *peers normales* está definida en términos de almacenamiento[5], procesamiento (ecuación 5.1) y ancho de banda (ecuación 4.2). Los *super-peers* necesitan tener una alta capacidad de almacenamiento para gestionar apropiadamente las operaciones de sus *peers normales*. La capacidad de procesamiento de un *peer* (denotada como PR), ya sea *super-peer* o *peer normal*, queda definida de la siguiente manera:

$$PR = (CF + PF) \times CO \tag{4.1}$$

en donde CF es la velocidad en la que un procesador ejecuta instrucciones (i.e., la velocidad de reloj medida en Megahertz), PF es la frecuencia del procesador y CO es el número de núcleos (físicos y virtuales).

Por otro lado, la capacidad de ancho de banda de un *peer* (denotada como BW), ya sea *super-peer* o *peer normal*, queda definida de la siguiente manera:

$$BW = \frac{D + U}{2} \tag{4.2}$$

en donde D es el ancho de banda de bajada (medido en Mbps) y U es el ancho de banda de subida (medido en Mbps).

La capacidad de procesamiento varía con la velocidad de reloj y la arquitectura del CPU. Usualmente, los CPUs con mayores velocidades de reloj poseen mayor capacidad de procesamiento que aquellos con menores velocidades de reloj [6]. Por lo tanto, la fuerza de un *peer* i (denotada como F_i) se encuentra definida por la ecuación 4.3:

$$F(i) = \frac{C_{max} \log FD + PR + BW}{\log d_{max}} \tag{4.3}$$

en donde C_{max} es un parámetro configurable dentro de *RASupport* que determina la capacidad máxima de *peers* que pueden existir en la red P2P, FD es el espacio libre en disco duro (medido en MB), PR es la capacidad de procesamiento (determinada a partir de la ecuación 5.1), BW es el ancho de banda de la red (determinado a partir de la ecuación 4.2) y d_{max} es el máximo flotante de doble precisión, el cual varía de acuerdo a la arquitectura del procesador.

[5]El espacio total en el disco duro es el atributo que define la capacidad de almacenamiento de un *peer*.

[6]Información obtenida de *Red Hat Enterprise Linux 3: Introduction to System Administration. Fuente:* `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Introduction_to_System_Administration/s1-bandwidth-processing.html`

Myconet utiliza un conjunto de reglas locales que determinan el comportamiento de los estados de un *super-peer*, los cuales están definidos de forma jerárquica (ver figura 4.2): extendiendo, ramificando e inmóvil. De esta manera, el estado más alto (*inmóvil*) corresponde a un *super-peer* que ha alcanzado el número ideal de conexiones y que posee buenas capacidades, disponibilidad y estabilidad, en comparación con el resto de los *peers* [40]. Existen dos parámetros necesarios para definir la topología de la red: el número de conexiones máximas que puede tener un *super-peer* i a un conjunto de *peers normales* (max_PN), el cual es directamente proporcional a la fuerza de tal *super-peer* (denotada como $F(i)$) y el número máximo de enlaces que un *super-peer* puede tener a otros *super-peers* (max_SP), el cual es un parámetro configurable dentro de *RASupport*. Para diseminar información sobre *peers* vivos y sus respectivos estados, se utiliza el protocolo Newscast, el cual está basado en la técnica *gossiping* [44].

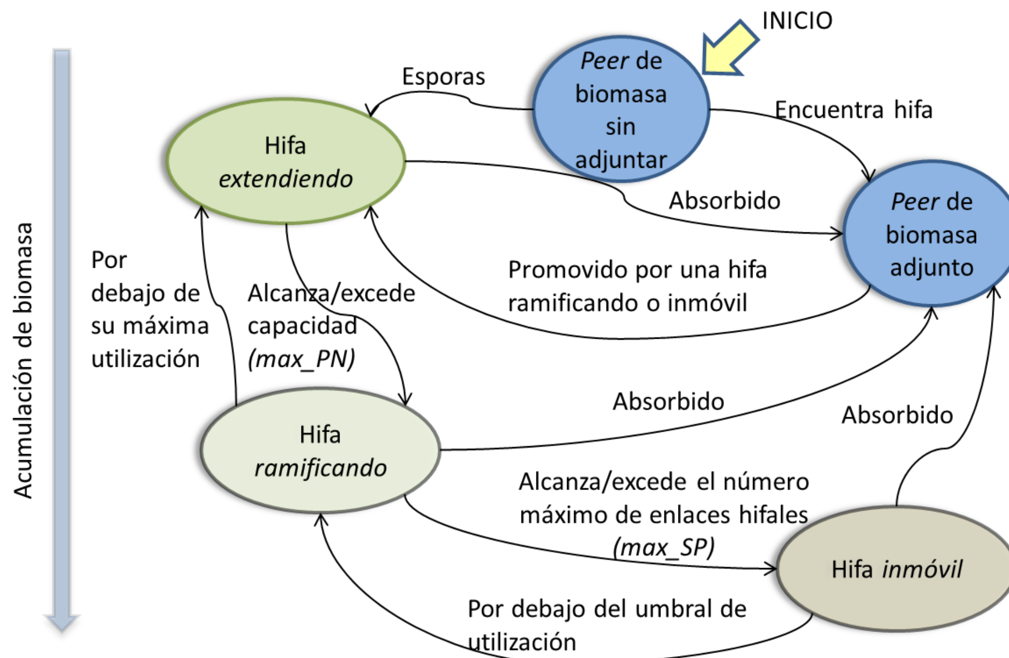


Figura 4.2: Transiciones de estado del protocolo Myconet [40]. Existen cuatro posibles estados (biomasa, extendiendo, ramificando e inmóvil) por los que puede transitar un *peer*, los cuales dependen de las condiciones de la red y de los valores de fuerza de dichos *peers*.

Cuando los *peers* se conectan por primera vez, automáticamente son considerados como *peers de biomasa* (i.e., *peers normales*) los cuales comienzan a buscar algún *super-peer* al cual se puedan conectar y, si no encuentran alguno (en cualquier estado), aplican las reglas pertinentes para transformarse en un *super-peer extendiendo*; posteriormente, dicho *super-peer extendiendo* se conecta a otros *super-peers* en caso de que estos últimos existan. Los *super-peers* que se encuentran en un estado *extendiendo* exploran constantemente

la red en busca de biomasa, i.e., en busca de *peers normales*[7]. Si no existen *super-peers inmóviles* o que estén *ramificando*, los *super-peers extendiendo* intentarán conectarse a otros *super-peers* de su misma jerarquía. Los *super-peers ramificando* son aquellos que alcanzan o exceden el número máximo de *peers normales* soportados, pero que aún no han alcanzado o excedido el número máximo de conexiones con otros *super-peers*. Además, dichos *super-peers ramificando* se encuentran en un estado intermedio (i.e., entre los estados *extendiendo* e *inmóvil*) y actúan como conducto para desplazar *peers normales* entre *super-peers inmóviles* y *super-peers extendiendo*, con la finalidad de mantener la robustez de la red P2P. Los *super-peers ramificando* ayudan a regular el número de *super-peers extendiendo*, aumentando o disminuyendo su número, con la finalidad de manipular adecuadamente los *peers normales* cuando se requiere. Los *super-peers inmóviles* son aquellos que han alcanzado el número ideal de conexiones con otros *super-peers* (max_SP) y, además, poseen suficientes *peers normales*, de manera que están utilizando toda su capacidad (max_PN). Las reglas de Myconet se aplican contínuamente, con la finalidad de mantener una red con *super-peers inmóviles* totalmente utilizados y un número óptimo de conexiones entre *peers inmóviles* y *peers ramificando*.

Myconet demuestra que la metáfora de los micelios es poderosa cuando se aplica en redes P2P y que puede ser usada para crear redes P2P auto-organizables, lo cual brinda mayores ventajas que otros enfoques en términos de robustez a fallas. Además, construye y mantiene efectivamente una red de *super-peers* fuertemente inter-conectados. Dicha red converge a un número óptimo de *super-peers* y altos niveles de capacidad de utilización. Una red gestionada por el protocolo Myconet se recupera fácilmente cuando se eliminan del 30 % al 50 % de los *peers* totales de la red e incluso cuando se elimina el 80 % de los *super-peers inmóviles*. A diferencia de otros enfoques, Myconet ajusta dinámicamente las conexiones entre *super-peers*, incrementando la eficiencia y resiliencia ante la pérdida de *peers*.

RASupport posee un componente de software encargado de gestionar apropiadamente la arquitectura P2P, utilizando el protocolo Myconet (ver figura 4.3). Los componentes de *RAToolkit* (*Advertisement*, *Selection*, *Matching* y *Binding*) se comunican con el componente encargado de gestionar la comunicación en la red (*Communication*) el cual, a su vez, se comunica con el componente de Myconet (*Myconet Support*).

[7]En los micelios, las hifas (i.e., *super-peers*) siempre están en busca de biomasa (i.e., *peers normales*) recolectando nutrientes y agua. Las hifas concentran biomasa y la utilizan para el crecimiento hifal (i.e., el crecimiento de la red P2P).

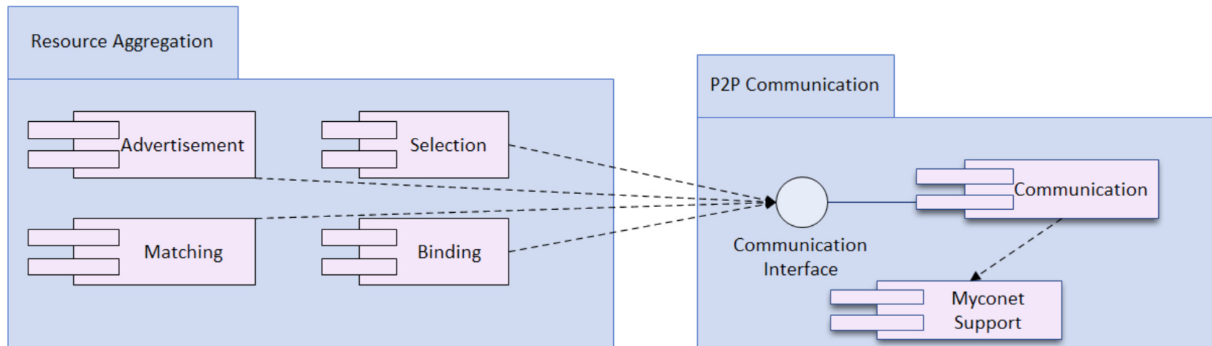


Figura 4.3: Diagrama de componentes de *RASupport*.

Como se puede observar en la figura 4.3, *RASupport* permite llevar a cabo las fases de anunciamiento, selección, empatamiento y enlace de recursos. La fase de descubrimiento se omite, debido a que los recursos son anunciados explícitamente. Con la finalidad de facilitar la comprensión y representación de las fases de la agregación de recursos, se diseñó una notación específica (ver tabla 4.4).

Notación	**Descripción**	**Ejemplo**	**Se lee**
$\Rightarrow$	Anunciamiento	$PN_1 \Rightarrow SP_4$	El *peer-normal* 1 anuncia sus recursos al *super-peer* 4
$\sigma = \psi$	Selección de todos los *peers* que satisfagan ψ	$\sigma = (memoria \geq 1GHz \wedge memoria \leq 2GHz) \wedge (so = \text{``}linux\text{''}) \wedge cores = 4$	Seleccionar todos los *peers* que satisfagan $\psi = (memoria \geq 1GHz \wedge memoria \leq 2GHz) \wedge (so = \text{``}linux\text{''}) \wedge cores = 4$
$=\sim$	Empatamiento	$PN_1 =\sim PN_5$	¿El *peer normal* 1 puede trabajar con el *peer normal* 5?
$>>=$	Enlace	$PN_0 >>= PN_2$	Enlazar al *peer normal* 0 con el *peer normal* 2
$\cup$	Unión	$SP_4 \cup PN_1$	El *peer normal* 1 se une al *super-peer* 4
$-$	Partida	$SP_4 - PN_1$	El *peer normal* 1 abandona al *super-peer* 4

Tabla 4.4: Notación para la agregación de recursos en *RASupport*.

4.3. Descripción de recursos

La especificación de recursos (RS, por sus siglas en inglés) es el documento mediante el cual un *peer* especifica los atributos que caracterizan a los recursos con los que cuenta. *RASupport* permite especificar atributos heterogéneos, dinámicos (e.g., ancho de banda) y estáticos (e.g., número de núcleos). Dichos atributos pueden ser de tipo flotante, entero o *string*. La figura 4.4 ilustra la estructura de los documentos XML de especificación de recursos, mientras que en el listado 4.1 se muestra un ejemplo detallado del mismo.

Listing 4.1: Ejemplo de un documento XML de especificación de recursos dentro de *RA-Support*.

```
<rs_initial>
	<use>
		<availability_start>00:00</availability_start>
		<availability_end>16:32</availability_end>
		<max_cpu_utilization>15</max_cpu_utilization>
		<only_for_friends>peer3,peer1</only_for_friends>
	</use>
	<processing>
		<free_mem>10319.03</free_mem>
		<busy_cpu>10.46</busy_cpu>
		<os_name>Linux</os_name>
		<cpu_speed>4261.15</cpu_speed>
		<cores>2</cores>
	</processing>
	<storage>
		<free_hdisk>52437.95</free_hdisk>
		<total_hdisk>98463.84</total_hdisk>
	</storage>
	<display>
		<bit_depth>6</bit_depth>
		<screen_width>707</screen_width>
		<screen_height>441</screen_height>
		<refresh_rate>24</refresh_rate>
	</display>
	<network>
		<latency>133.59</latency>
		<bandwidth>583.19</bandwidth>
	</network>
</rs_initial>
```

RASupport gestiona recursos de procesamiento (<processing>), de almacenamiento (<storage>), de despliegue (<display>) y de red (<network>). En particular, los atributos soportados que describen un recurso de procesamiento son: la velocidad de CPU (<cpu_speed>), la memoria RAM disponible (<free_mem>), el porcentaje de CPU ocupado (<busy_cpu>), el número de núcleos del procesador (<cores>) y el nombre del sistema operativo (<os_name>). Los atributos soportados que describen un recurso de almacenamiento son: el espacio disponible en disco duro (<free_hdisk>) y la capacidad total en disco duro (<total_hdisk>). Los atributos soportados que describen un recurso de despliegue son: la resolución de la pantalla (<screen_width> y <screen_height>), la profundidad de color (<bit_depth>) y la tasa de refresco (<refresh_rate>). Finalmente, los atributos soportados que describen un recurso de red son: el ancho de banda (<bandwidth>) y la latencia (<latency>). En la tabla 4.5 se muestra un resumen de los atributos anteriormente mencionados.

En particular, el ancho de banda de un *peer* se determina calculando el promedio entre el ancho de banda de subida y el ancho de banda de bajada, tal como se muestra en la

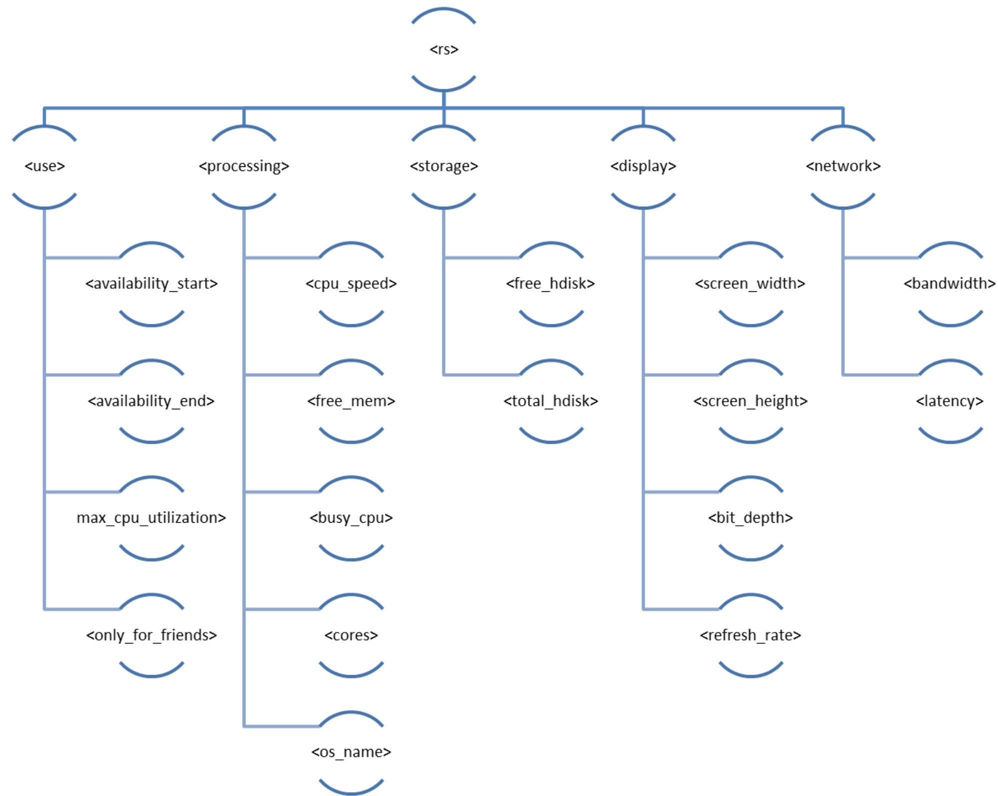


Figura 4.4: Estructura de los documentos XML de especificación de recursos dentro de *RASupport*.

ecuación 4.2. Por otro lado, la latencia y el ancho de banda entre dos *peers* se determina calculando el promedio entre la latencia del *peer* A al *peer* B y la latencia del *peer* B al *peer* A:

$$Latency = \frac{Latency_{AB} + Latency_{BA}}{2} \tag{4.4}$$

en donde *Latency* es la latencia final entre dos *peers*, $Latency_{AB}$ es la latencia del *peer* A al *peer* B y $Latency_{BA}$ es la latencia del *peer* B al *peer* A.

Atributo	**Descripción**	**Unidades**	**Tipo**	**Categoría**
		Atributos de procesamiento		
<cpu_speed>	Frecuencia de reloj del procesador	Megahertz (MHz)	Flotante	Estático
<free_mem>	Porcentaje de memoria RAM disponible	Megabytes (MB)	Flotante	Dinámico
<busy_cpu>	Porcentaje total de CPU ocupado	Porcentaje	Flotante	Dinámico
<cores>	Número de núcleos	N/A	Entero	Estático
<os_name>	Nombre del sistema operativo	N/A	String	Estático
		Atributos de almacenamiento		
<free_hdisk>	Espacio libre en el disco duro	Megabytes (MB)	Flotante	Dinámico
<total_hdisk>	Espacio total del disco duro	Megabytes (MB)	Flotante	Estático
		Atributos de despliegue		
<screen_width>	Ancho de la pantalla	Pixeles (px)	Entero	Estático
<screen_height>	Alto de la pantalla	Pixeles (px)	Entero	Estático
<bit_depth>	Número de bits utilizados para indicar el color de un pixel	Bits	Entero	Estático
<refresh_rate>	Tasa de refresco de la pantalla	Hertz (Hz)	Entero	Estático
		Atributos de red		
<bandwidth>	Ancho de banda de la red (subida y bajada)	Megabits por segundo (Mbps)	Flotante	Dinámico
<latency>	Latencia de la red (bidireccional)	Milisegundos (ms)	Flotante	Dinámico

N/A = No aplicable

Tabla 4.5: Atributos mantenidos por *RASupport*.

Además, *RASupport* permite especificar las restricciones de uso (`<use>`) de los recursos de *peers*, así como sus respectivos rangos de valores, mediante el documento de especificación de recursos. Las restricciones de uso soportadas son: el horario en el que se encuentran disponibles los recursos del *peer* (`<availability_start>` y `<availability_end>`), la utilización de CPU permitida (`<max_cpu_utilization>`) y la autorización de uso de recursos únicamente para ciertos *peers* amigos (`<only_for_friends>`). En la tabla 4.6 se muestra un resumen de las restricciones anteriormente mencionadas.

Restricción	Descripción	Unidades	Tipo
`<availability_start>`	Hora en la que un *peer* comienza a estar disponible	N/A	N/A
`<availability_end>`	Hora en la que un *peer* deja de estar disponible	N/A	N/A
`<max_cpu_utilization>`	Máximo porcentaje de CPU permitido	Porcentaje	Entero
`<only_for_friends>`	Indica si los recursos del *peer* pueden ser utilizados únicamente por amigos (se establece una lista con los alias de los *peers* amigos)	N/A	*String*

N/A = No aplicable

Tabla 4.6: Restricciones de uso de recursos mantenidas por *RASupport*.

4.4. Fase de anunciamiento

Definición 3. *Sean SP_f y SP_g dos super-peers, tal que $SP_f \in S_c$, $SP_g \in S$ y $SP_g \notin S_c$. Si SP_f abandona la red P2P del soporte $c \in C$ o reduce su jerarquía, se aplican las reglas pertinentes de acuerdo a su estado actual (extendiendo, ramificando o inmóvil) con la finalidad de que un nuevo super-peer, SP_g, lo reemplace.*

Un *peer normal* anuncia sus recursos a su *super-peer* a través de un documento RS como el que se muestra en el listado 4.1. Para llevar a cabo esta fase, un *peer normal* envía un *agente de anunciamiento* a su *super-peer*, el cual transporta el documento RS con un costo de $O(1)$ (ver figura 4.5). Debido a que los atributos dinámicos cambian frecuentemente en cargas de trabajo reales [5], los *peers normales* regularmente informan a sus *super-peers* acerca de las actualizaciones que suscitan. Por lo tanto, existen dos tipos de agentes de anunciamiento: el inicial y el de actualización. Los agentes de anunciamiento inicial transportan un documento RS que informa los recursos de un *peer normal* que se acaba de unir a un *super-peer*. Por otro lado, los agentes de anunciamiento de actualización transportan únicamente un mensaje que contiene el nombre del atributo dinámico que será actualizado, así como su nuevo valor; por lo tanto, no es necesario que transporten un documento XML como el que transportan los agentes de anunciamiento inicial.

En caso de que el *super-peer* SP_f abandone (voluntariamente o debido a una falla) la red P2P del soporte c, la arquitectura de dicho soporte se auto-organiza y los *peers normales* de SP_f anuncian sus especificaciones de recursos al nuevo *super-peer* SP_g. En este caso, SP_g se vuelve miembro de S_c, mientras que SP_f deja de pertenecer a S_c. En la tabla 4.7, se muestran los casos particulares en los que se envían agentes de anunciamiento inicial.

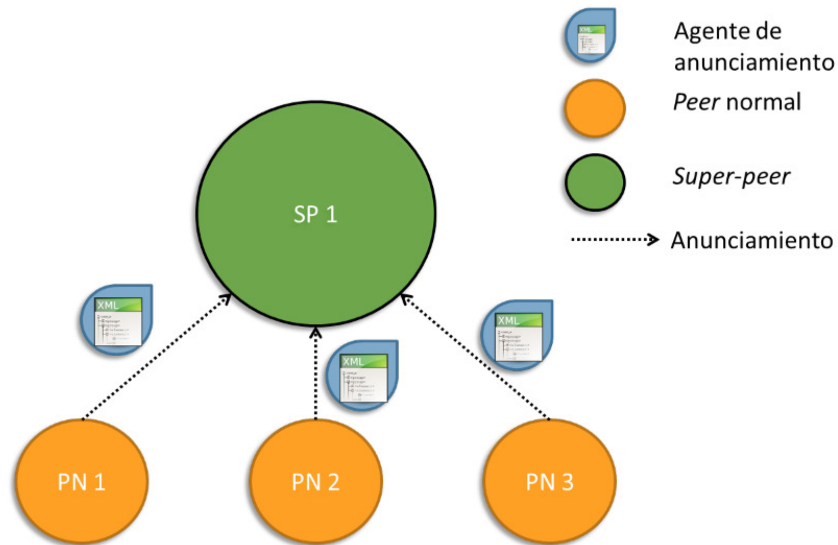


Figura 4.5: Anunciamiento de recursos en *RASupport*.

Caso	**Operación**
Cuando un *peer normal* PN_i se une a la red P2P del soporte c	$PN_i \Rightarrow SP_j$, tal que $SP_j \in S_c$
Cuando un *super-peer* SP_i adquiere un *peer normal* PN_j de otro *super-peer*	$PN_j \Rightarrow SP_i$
Cuando un *super-peer* SP_i se vuelve un *peer normal* debido a que es absorbido por un *peer normal* PN_j, tal que PN_j es un *peer normal* de SP_i	$SP_i \Rightarrow PN_j$
Cuando un *super-peer* SP_i es absorbido por un *super-peer* SP_j	$PN_k \Rightarrow SP_j \; \forall \; 0 < k <= n$, en donde n es el número de *peers normales* de SP_i

Tabla 4.7: Casos particulares en los que se envían agentes de anunciamiento inicial.

4.5. Fase de selección

Definición 4. *Sea Q el conjunto universo de consultas de recursos y Q_c el conjunto de consultas en el soporte c, tal que $Q_c \subset Q$.*

Definición 5. *Sea* G *el conjunto universo de grupos de atributos especificados en consultas de recursos y* G_q *el conjunto de grupos de atributos especificados en una consulta* q*, tal que* $q \in Q_c$ *y* $G_q \subset G$*.*

Definición 6. *Sea* A *el conjunto universo de atributos existentes,* A_q *el conjunto de atributos especificados en una consulta* q*,* A_g *el conjunto de atributos especificados en un grupo de atributos* g*,* a *un atributo especificado en el grupo* g *y* k *el número de atributos especificados en el grupo* g*, tal que* $g \in G_q$*,* $A_q \subset A$*,* $A_g \subseteq A_q$*,* $a \in A_g$ *y* $\mathbf{card}(A_g) = k$*.*

Definición 7. *Sea* $SP_{initiator}$ *el iniciador de una consulta, tal que* $SP_{initiator} \in S_c$*.*

Las consultas de recursos son personalizables dentro de *RASupport* y permiten especificar –mediante un documento XML– grupos de atributos que encapsulan los requerimientos y restricciones de una aplicación. Las consultas pueden ser de tipo *zero-attribute*, *single-attribute* o *multi-attribute*. Para cada grupo de la consulta se especifica el nombre (`<name>`), el número de *peers* requeridos (`<num_nodes>`), los atributos requeridos por *peer* y las restricciones entre *peers* (`<rest_betw_nodes>`). En la consulta, también se especifican las restricciones entre grupos de atributos (`<rest_betw_groups>`). Las restricciones entre *peers* y entre grupos permitidas por *RASupport* son el ancho de banda (`<bandwidth>`) y la latencia (`<latency>`), respectivamente. En el listado 4.2 se muestra un ejemplo de una consulta de recursos dentro de *RASupport*.

Listing 4.2: Ejemplo de una consulta de recursos dentro de *RASupport*.

```
<query>
    <option>requires:find_resources</option>
    <ttl>4</ttl>
    <group>
        <name>Processing</name>
        <num_nodes>8</num_nodes>
        <cpu_speed>1024.0,2500.0,4096.0,4096.0,0.2</cpu_speed>
        <free_mem>512.0,1024.0,4096.0,8192.0,0.05</free_mem>
        <busy_cpu>0,0,30,75,0.05</busy_cpu>
        <cores>1,2,4,4,0.03</cores>
        <free_hdisk>50.0,60.5,92.5,100.0,0.05</free_hdisk>
        <os_name>"Linux",0.0</os_name>
    </group>
    <group>
        <name>Display</name>
        <num_nodes>1</num_nodes>
        <screen_width>800,1024,1366,2560,0.02</screen_width>
        <screen_height>600,1024,2048,1700,0.02</screen_height>
        <bit_depth>8,16,48,48,0.0</bit_depth>
        <refresh_rate>40,60,120,120,0.05</refresh_rate>
    </group>
    <rest_betw_groups>
        <group_names>Processing, Display</group_names>
        <latency>0.0,0.0,50.0,100.0,0.2</latency>
    </rest_betw_groups>
</query>
```

A diferencia de otras soluciones de consulta de recursos, los rangos de valores de los atributos especificados en la consulta pueden ser enteros (e.g., el rango de núcleos necesitados) o flotantes (e.g., el rango de latencia requerida); de igual forma, es posible especificar atributos de tipo *string* (e.g., el nombre del sistema operativo). Para atributos númericos (enteros o flotantes) la consulta permite definir un rango de valores permitidos (*min_val* y *max_val*), un rango de valores ideales (*min_ideal* y *max_ideal*) y un valor de penalización[8] (P), los cuales se especifican en el siguiente orden: valor mínimo permitido, valor mínimo ideal, valor máximo ideal, valor máximo permitido y penalización (ver listado 4.2). Por el contrario, para cada atributo de tipo *string* únicamente se necesita especificar los siguientes valores: valor requerido y penalización P (ver listado 4.2).

Un *peer normal* o un *super-peer* puede solicitar recursos, sin embargo, en cualquier caso, los *super-peers* son los iniciadores de la consulta y, por lo tanto, son aquellos que envían agentes de consulta[9]. Estos agentes visitan *super-peers* empleando una técnica que depende del propósito específico de la aplicación; de hecho, una aplicación puede requerir buen rendimiento o sacrificar rendimiento con la finalidad de obtener siempre resultados en la fase de selección. De esta manera, *RASupport* permite especificar alguna de las siguientes opciones mediante la etiqueta `<option>`: encontrar recursos imperativamente (*requires:find_resources*) o mejorar el rendimiento de la red aunque no siempre se encuentren recursos (*requires:performance*). En ambas opciones, se utilizan protocolos propuestos en la presente tesis. En particular, en la primera opción se utiliza una modificación del protocolo de inundación, el cual se describe en la subsección 4.5.1; mientras que en la segunda opción, se utiliza un protocolo denominado "paseos aleatorios inteligentes" (*iRandomWalks*), el cual se describe en la subsección 4.5.2. En ambos protocolos, $SP_{initiator}$ inicia la selección de recursos enviando agentes de consulta con un TTL (*Time To Live*) limitado, el cual se especifica en la consulta de recursos mediante la etiqueta `<ttl>` (ver listado 4.2). Dichos agentes de consulta recolectan resultados en los *super-peers* que visitan y regresan a $SP_{initiator}$ cuando su TTL expira o cuando no encuentran algún *super-peer* al cual visitar.

4.5.1. Inundación con agentes de consulta

En la presente tesis, se propone el protocolo de inundación con agentes de consulta[10], en donde cada uno de los *super-peers* envía un agente de consulta a cada uno de sus vecinos *super-peers*, por lo que todos los *super-peers* del soporte c son inundados con dichos agentes. En la figura 4.6, se observa que $SP_{initiator}$ inicia la selección de recursos enviando copias de un mismo agente a todos sus vecinos ($SP7$, $SP6$, $SP4$ y $SP1$) y, cada uno de los

[8] P es la penalización que se aplica a un *peer* cuando no satisface el rango de valores ideales de un atributo en particular.

[9] Los agentes de consulta son aquellos que transportan un documento XML que contiene una consulta de recursos.

[10] El protocolo de inundación con agentes de consulta está basado en el protocolo de inundación de peticiones [51].

super-peers que reciben un agente de consulta, se comporta de manera similar a $SP_{initiator}$.

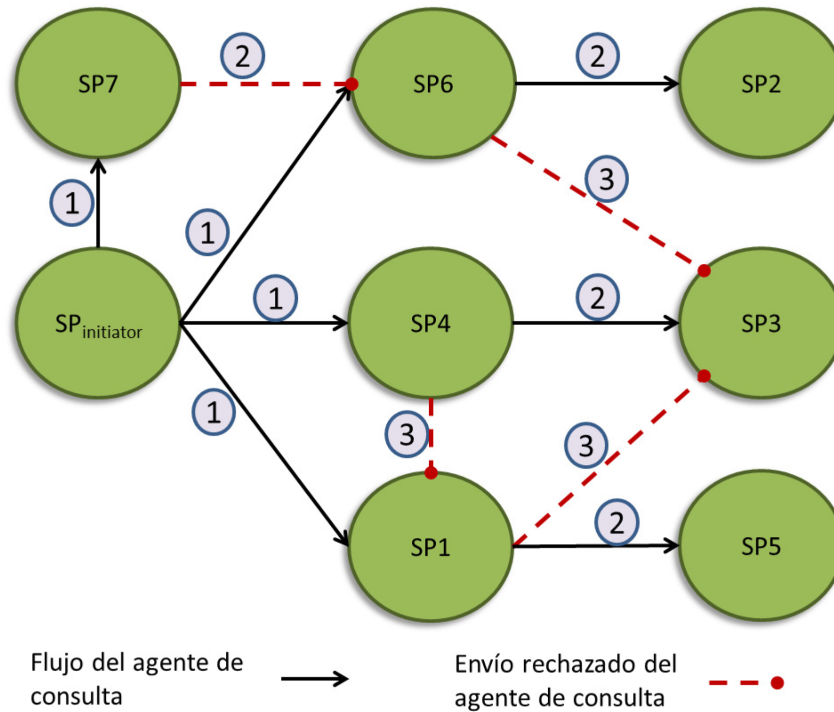


Figura 4.6: Protocolo de inundación con agentes de consulta.

Un documento XML de consulta de recursos puede llegar a tener un tamaño muy grande, por lo que el consumo de ancho de banda de la red puede llegar a ser significativo. El protocolo de inundación con agentes de consulta envía un mensaje de prueba (el cual evidentemente es de menor tamaño que un documento XML) antes de enviar el agente de consulta a un determinado *super-peer*, con la finalidad de reducir el ancho de banda de la red y de garantizar que un *super-peer* no reciba múltiples agentes que transportan una misma consulta. En la figura 4.6, se observa que los *super-peers* $SP6$ y $SP1$ rechazan agentes de consulta de $SP7$ y $SP4$, respectivamente, debido a que ya han recibido otros agentes provenientes de $SP_{initiator}$; así mismo, se observa que $SP3$ recibe un agente de consulta de $SP4$ por lo que rechaza agentes de consulta provenientes de $SP6$ y $SP1$, respectivamente.

A pesar de que este protocolo tiene la desventaja de causar tráfico en la red y de poseer una complejidad exponencial, garantiza que una consulta llegue a cada uno de los *super-peers* que existen en el soporte c.

4.5.2. Paseos aleatorios inteligentes (iRandomWalks)

Definición 8. *Sea* $Q_{initiator}$ *el conjunto de agentes de consulta enviados por* $SP_{initiator}$ *y* q_{agent} *un agente de consulta, tal que* q_{agent} *transporta la consulta* $q \in Q_c$ *y* $q_{agent} \in Q_{initiator}$.

Definición 9. *Sea VN_c el conjunto de super-peers visitados por q_{agent} en la red P2P del soporte $c \in C$, tal que $VN_c \subseteq S_c$. SP_v denota un super-peer visitado por q_{agent}, tal que $SP_v \in VN_c$ y $SP_v \neq SP_{initiator}$.*

Definición 10. *Sea NB_v el conjunto de super-peers vecinos de SP_v, tal que $NB_v \subseteq S_c$. SP_w denota un vecino de SP_v, tal que $SP_w \in NB_v$, $SP_w \neq SP_{initiator}$ y $SP_w \notin VN_c$.*

En este protocolo, cada *super-peer* mantiene una tabla de estadísticas en memoria RAM, cuyas entradas incluyen el par (grupo de atributos, lista de *super-peers*). La llave se determina con base en un grupo de atributos [11], mientras que el valor es una lista de referencias a *super-peers* que, en el pasado, regresaron resultados para dicho grupo.

Antes de enviar un agente de consulta, $SP_{initiator}$ ejecuta un algoritmo de similitud para cada grupo de atributos especificado en la consulta de recursos, con la finalidad de determinar si en su tabla de estadísticas existen grupos similares a los solicitados. $SP_{initiator}$ envía un agente de consulta a cada uno de los *super-peers* de su tabla de estadísticas, cuya similitud entre grupos sea alta (i.e., mayor al 75 %). Además, $SP_{initiator}$ envía un agente de consulta a n *super-peers* vecinos seleccionados aleatoriamente (en donde el valor de n es un parámetro configurable dentro de *RASupport*). Por lo tanto, el protocolo de paseos aleatorios inteligentes garantiza que una consulta de recursos llegue a aquellos *super-peers* que anteriormente obtuvieron resultados para grupos similares que existen en dicha consulta. Con la finalidad de optimizar el uso de la memoria RAM, las tablas de estadísticas se vacían cada t segundos (en donde el valor de t es un parámetro configurable dentro de *RASupport*). Por otro lado, un agente de consulta termina su trabajo cuando su TTL expira (i.e., cuando es igual a cero) o cuando SP_v no posee vecinos *super-peers*. Además, dichos agentes de consulta recolectan resultados de cada $SP_v \in VN_c$ y, una vez que han completado su trabajo, regresan sus resultados a $SP_{initiator}$ con la finalidad de que este último actualice su tabla de estadísticas.

Cada *super-peer* visitado ($SP_v \in VN_c$) por q_{agent}, reenvía a este último a uno y solo un *super-peer* vecino seleccionado aleatoriamente. El agente q_{agent} decrementa el valor de su TTL antes de visitar a $SP_v \in VN_c$ para prevenir que q_{agent} se envíe si su TTL ha expirado, reduciendo así el ancho de banda de la red. Antes de que q_{agent} visite a SP_w, verifica que SP_w siga vivo y, en caso de que no sea así, q_{agent} selecciona otro SP_w. De esta manera, *RASupport* garantiza que los agentes de consulta no se perderán en el camino.

En la figura 4.7 se ilustra un escenario que describe el funcionamiento del protocolo de paseos aleatorios inteligentes, en donde se observa que $SP_{initiator}$ envía cuatro agentes de consulta (i.e., $\mathbf{card}(Q_{initiator}) = 4$), los cuales poseen un $ttl = 3$, una lista de exclusión y una consulta compuesta por tres grupos de atributos requeridos: G_C, G_D y G_E. Además, se observa que $SP_{initiator}$ mantiene una tabla de estadísticas cuyas entradas iniciales son: $\{G_A, \{SP3, SP4\}\}$, $\{G_B, SP6\}$ y $\{G_C, SP1\}$. El grupo G_D de la consulta originada por

[11] Dos grupos de atributos son iguales si poseen el mismo valor *hash*, el cual se encuentra determinado por la cadena de caracteres que representa a dicho grupo.

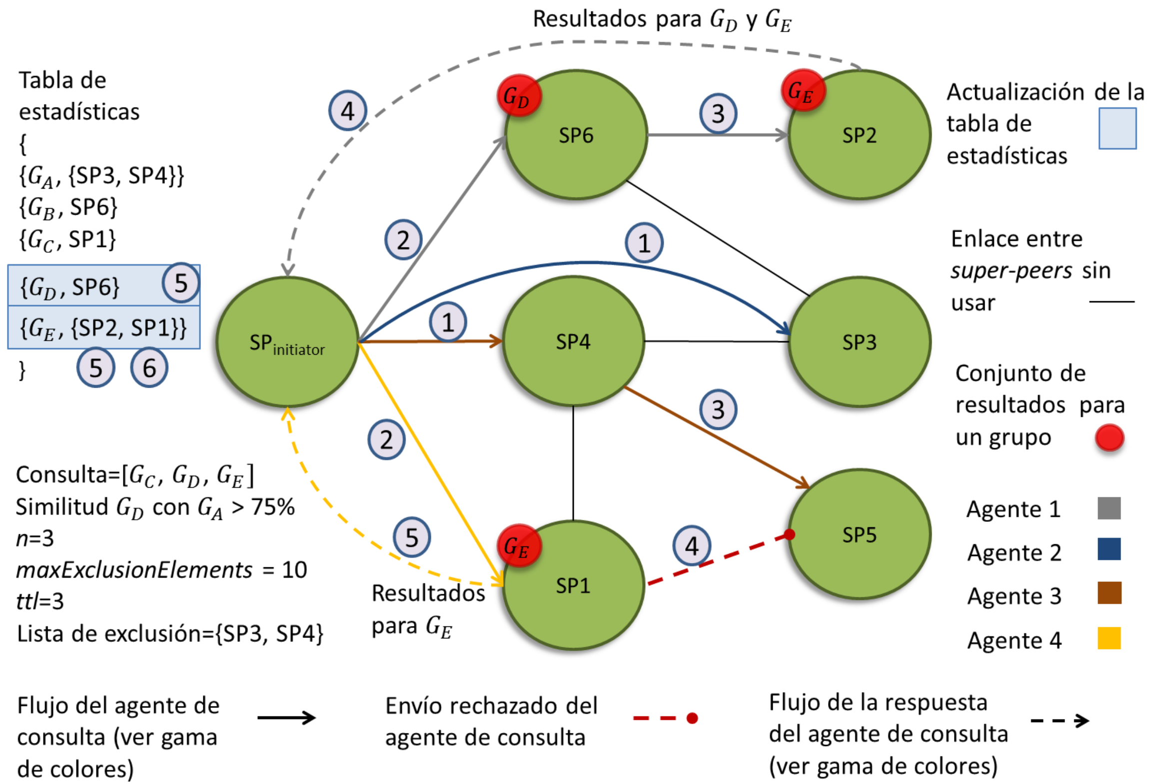


Figura 4.7: Protocolo de paseos aleatorios inteligentes. Los parámetros n (número máximo de *super-peers* que un agente de consulta puede visitar a partir de $SP_{initiator}$), $maxExclusionElements$ (número máximo de elementos que puede tener la lista de exclusión) y *ttl* (valor del TTL del agente de anunciamiento) son configurables dentro de *RASupport*.

$SP_{initiator}$ posee una alta similitud (mayor al 75 %) con el grupo G_A de la tabla de estadísticas; por lo tanto, los *super-peers* $SP3$ y $SP4$ son agregados a la lista de exclusión y $SP_{initiator}$ les envía directamente un agente de consulta. Posteriormente, $SP_{initiator}$ ejecuta el protocolo de paseos aleatorios con una ligera modificación. En la figura 4.8 se ilustra el protocolo tradicional de paseos aleatorios (sin modificación), en donde se observa que un *super-peer* puede ser visitado por varios agentes que transportan una misma consulta; para solucionar lo anterior, el protocolo de paseos aleatorios inteligentes propone un mecanismo que permite probar *super-peers* antes de enviar un agente de consulta. En la figura 4.7, se observa que a pesar de que n posee un valor de 3, $SP_{initiator}$ envía un agente de consulta únicamente a $SP1$ y $SP6$, debido a que $SP4$ se encuentra en la lista de exclusión. Finalmente, cada $SP_v \in VN_c$ reenvía un agente de consulta a uno y solo un vecino *super-peer* seleccionado aleatoriamente, siempre que dicho *super-peer* no haya recibido otro agente con la misma consulta. En la figura 4.7, también se observa que el agente 4 no se envía al *super-peer* $SP5$, debido a que este último ya ha recibido al agente 3. Todos los agentes de consulta recolectan resultados en cada $SP_v \in VN_c$ y regresan dichos

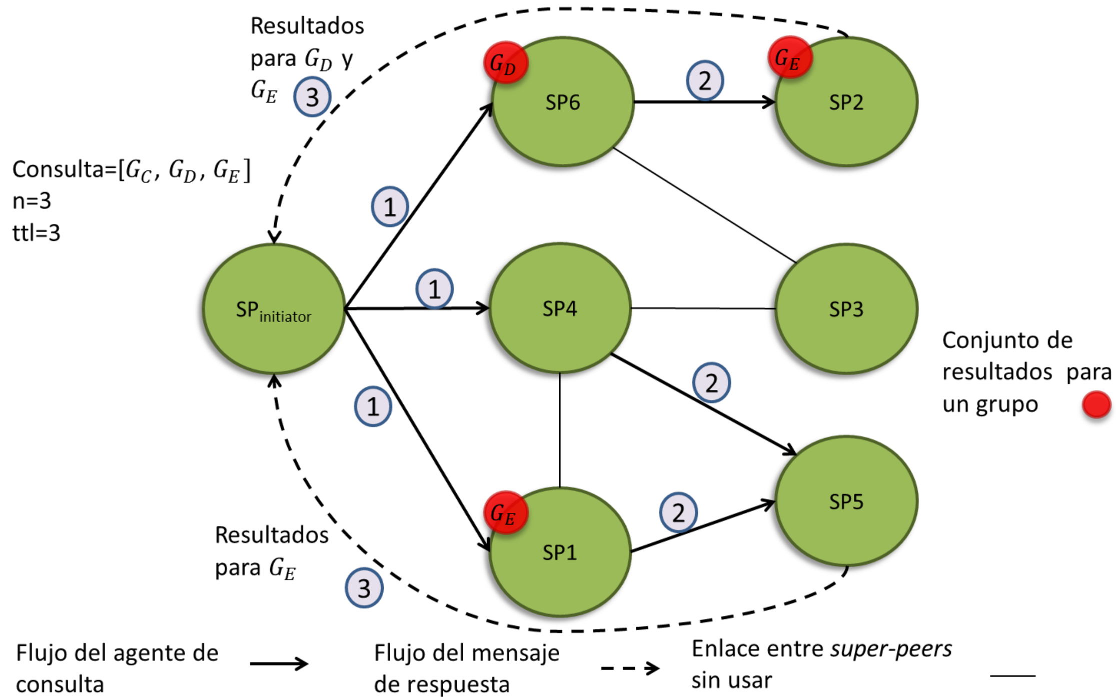


Figura 4.8: Protocolo tradicional de paseos aleatorios.

resultados a $SP_{initiator}$ (e.g., el agente 1 regresa resultados de los *super-peers* $SP6$ y $SP2$ para los grupos G_D y G_E, respectivamente). A continuación, se describen los principales algoritmos que componen al protocolo de paseos aleatorios inteligentes.

El algoritmo 1 es el núcleo del protocolo de paseos aleatorios inteligentes, cuyo parámetro de entrada es una consulta especificada en un documento XML (*xmlQuery*). Este algoritmo obtiene los grupos de atributos requeridos que existen en *xmlQuery* y utiliza una lista de *super-peers* (*intelligentList*) a los que se les enviará directamente un agente de consulta, en caso de que en el pasado hayan obtenido resultados para un grupo similar a alguno de los existentes en *xmlQuery*. Así mismo, se crea una lista de exclusión (*exclusionList*) que almacena el identificador de los *super-peers* que ya han resuelto una consulta similar en el pasado. Los agentes de consulta se crean a través del método *createQueryAgent* de un objeto que utiliza el patrón de diseño fábrica (*factory*) [14], cuyos parámetros de entrada son: una consulta de recursos y una lista de exclusión. Finalmente, el protocolo de paseos aleatorios inteligentes ejecuta el protocolo tradicional de paseos aleatorios con la modificación previamente mencionada, el cual se describe en el algoritmo 3.

Algoritmo 1 Protocolo de paseos aleatorios inteligentes

```
1: procedure PERFORMINTELLIGENTRANDOMWALK(xmlQuery)
2:     intelligentList ← newList()
3:     exclusionList ← newList()
4:     groupsArray ← getGroupsFromQuery(xmlQuery)
5:     for g ← 0, g < groupsArray.size(), g ← g + 1 do
6:         for i ← 0, i < statsTable.size(), i ← i + 1 do
7:             if getSimilarity(groupsArray[g], statsTable.getKeyAt(i)) > 75 then
8:                 intelligentList.union(statsTable.getValueAt(i))
9:                 exclusionList.union(statsTable.getValueAt(i))
10:            end if
11:        end for
12:    end for
13:    for i ← 0, i < intelligentList.size(), i ← i + 1 do
14:        queryAgent = factory.createQueryAgent(xmlQuery, exclusionList)
15:        queryAgent.sendTo(intelligentList[i])
16:    end for
17:    performModifiedRandomWalk(superPeerInitiator, xmlQuery, exclusionList)
18: end procedure
```

El algoritmo 2 sirve para determinar el porcentaje de similitud que existe entre un grupo de atributos requeridos mantenido en la tabla de estadísticas (*group1*) y un grupo especificado en una consulta de recursos (*group2*). En particular, se comparan los atributos especificados en *group1* con los atributos pertenecientes a *group2*. Para cada atributo, se compara un número determinado de *tokens* (i.e., factores) dependiendo del tipo de dicho atributo. Para atributos numéricos (i.e., enteros o flotantes) se comparan cuatro *tokens*: el valor mínimo permitido (*min_val*), el valor mínimo ideal (*min_ideal*), el valor máximo ideal (*max_ideal*) y el valor máximo permitido (*max_val*). Para atributos de tipo *string*, únicamente se compara un *token*: el valor especificado (*val*). En ambos casos, se omite el valor de penalización, debido a que esta es irrelevante para encontrar atributos en un nodo y únicamente sirve para evaluar dichos atributos. Por lo tanto, el algoritmo 2 determina la similitud que existe entre *group1* y *group2*, a partir de la siguiente ecuación:

$$Similarity = \frac{\sum_{i=1}^{z} \frac{100 count_i}{tokens_i}}{max(y, z)} \tag{4.5}$$

en donde y es el número de atributos que existen en *group1*, z es el número de atributos que existen en *group2*, $tokens_i$ es el número de factores a considerar en un atributo i (para atributos numéricos su valor es de 4 y para atributos de tipo *string* su valor es de 1), $count_i$ es el número de coincidencias de *tokens* que existe entre el atributo i de *group1* y el atributo i de *group2* y $max(y, z)$ es la función que determina el valor máximo entre y y z.

Algoritmo 2 Algoritmo de similitud entre grupos de atributos requeridos

```
1:  procedure getSimilarity(group1, group2)
2:      similarity ← 0.0
3:      for Attribute a2: group2.getAttributes() do
4:          if group1.containsAttribute(a2) then
5:              Attribute a1 ← group1.getAttribute(a2)
6:              if a2.isNumerical() then
7:                  count ← 0
8:                  if a1.min_val = a2.min_val then count ← count + 1
9:                  end if
10:                 if a1.min_ideal = a2.min_ideal then count ← count + 1
11:                 end if
12:                 if a1.max_ideal = a2.max_ideal then count ← count + 1
13:                 end if
14:                 if a1.max_val = a2.max_val then count ← count + 1
15:                 end if
16:                 similarity ← similarity + ((100 * count)/4)
17:             else
18:                 if a1.val = a2.val then similarity ← similarity + 100
19:                 end if
20:             end if
21:         end if
22:     end for
        return similarity/max(group1.countAttributes(), group2.countAttributes())
23: end procedure
```

El algoritmo 3 es un modificación del protocolo tradicional de paseos aleatorios [51], en el cual $SP_{initiator}$ envía n agentes de consulta a n *super-peers* vecinos, en caso de que el valor de n sea menor que el número total de dichos vecinos; en caso contrario, se envía un agente de consulta a todos y cada uno de los vecinos de $SP_{initiator}$. Los *super-peers* visitados envían el agente de consulta a uno y sólo a un vecino seleccionado aleatoriamente. Los métodos *sendAgentsToAllNeighbors* y *sendAgentsToRandomNeighbors* emplean la lista de exclusión y envían mensajes de prueba para determinar cuáles *super-peers* pueden ser visitados por un agente de consulta.

De manera similar al protocolo de inundación descrito en la sección anterior, un agente de consulta emplea el algoritmo 4 con la finalidad de evitar visitar a aquellos *super-peers* que ya hayan sido o que serán visitados por otros agentes de consulta, reduciendo así el consumo de ancho de banda de la red. Dicho algoritmo únicamente recibe como parámetro de entrada el objeto *super-peer* que se desea probar (sp). Con la finalidad de optimizar la búsqueda en una lista de exclusión, *RASupport* permite definir el número máximo de ele-

mentos que pueden existir en dicha lista por medio del parámetro *maxExclusionElements*. En caso de que el tamaño de la lista no haya excedido el máximo permitido y que *sp* no exista en la lista de exclusión, el agente de consulta envía un mensaje a *sp* (mediante el algoritmo 5) para determinar si sigue vivo y si no ha recibido previamente otro agente de consulta. En caso de que el tamaño de la lista haya excedido el tamaño máximo permitido, el agente de consulta no busca a *sp* en la lista de exclusión e inmediatamente le envía un mensaje de prueba, haciendo uso del algoritmo 5.

Algoritmo 3 Protocolo tradicional de paseos aleatorios

```
1: procedure performModifiedRandomWalk(sp, xmlQuery, exclusionList)
2:     if sp.isInitiator() then
3:         if supportConfig.n >= count(sp.getNeighbors()) then
4:             sp.sendAgentsToAllNeighbors(xmlQuery, exclusionList)
5:         else
6:             sp.sendAgentsToRandomNeighbors(xmlQuery, exclusionList,
7:             supportConfig.n)
8:         end if
9:     else
10:        sp.sendAgentsToRandomNeighbors(xmlQuery, exclusionList, 1)
11:    end if
12: end procedure
```

Algoritmo 4 Algoritmo para probar el estado de un *super-peer*

```
1: procedure queryAgent.testSuperpeer(sp)
2:     if exclusionList.size() ≤ supportConfig.maxExclusionElements then
3:         if !isInExclusionList(sp) then
4:             sendTestMessageTo(sp)
5:         end if
6:     else
7:         sendTestMessageTo(sp)
8:     end if
9: end procedure
```

Algoritmo 5 Algoritmo para enviar un mensaje de prueba a un *super-peer*

```
1: procedure queryAgent.sendTestMessageTo(sp)
2:     sendQueryIdMessageTo(sp);
3:     ack ← waitAckQueryIdFrom(sp)
4:     if ack then
5:         sendTo(sp)
6:     end if
7: end procedure
```

A diferencia del protocolo tradicional de paseos aleatorios (ver figura 4.8), el protocolo de paseos aleatorios inteligentes (ver figura 4.7) mantiene una lista de exclusión en los agentes de consulta para enviar mensajes de prueba antes de visitar a un *super-peer*, evitando así llegar a *super-peers* que ya hayan sido o que serán visitados por otros agentes que transportan una misma consulta, por lo que de esta manera se reduce el ancho de banda en la fase de selección. De igual forma, los *super-peers* mantienen tablas de estadísticas, lo cual permite alcanzar otros *super-peers* que probablemente regresarán resultados para un grupo especificado en una consulta. Por lo tanto, el protocolo de paseos aleatorios inteligentes se beneficia del aprendizaje de éxitos previos, aumentando así la probabilidad de éxito en una consulta en comparación con el protocolo tradicional de paseos aleatorios.

4.6. Fase de empatamiento

Definición 11. *Sea NP el conjunto universo de peers normales existentes y NP_c el conjunto de peers normales que existen dentro del soporte $c \in C$, tal que $NP_c \subset NP$. NP_v denota el conjunto de peers normales del super-peer SP_v, tal que $NP_v \subset NP_c$ y $SP_v \notin NP_v$.*

Definición 12. *Sea x un peer normal o un super-peer dentro del soporte c, tal que $x \in \{SP_v\} \cup NP_v$.*

Definición 13. *Sea CG_v la familia de conjuntos de peers candidatos (grupos de peers) determinada por q_{agent} en $SP_v \in VN_c$ para la consulta $q \in Q_c$ y h sea un conjunto de peers candidatos (grupo de peers), tal que $\mathbf{card}(CG_v) \leq \mathbf{card}(G_q)$ y $h \in CG_v$.*

Definición 14. *Sea R_{agent} el conjunto de resultados obtenidos por q_{agent}, tal que $R_{agent} = \bigcup_{i=1}^{\mathbf{card}(VN_c)} CG_v^i$.*

Definición 15. *Sea I_g una lista que denota una combinación inicial de peers candidatos para el grupo $g \in G_q$, formada por $SP_{initiator}$ a partir de los resultados obtenidos por cada uno de los agentes de consulta enviados.*

Definición 16. *Sea PC el conjunto universo de combinaciones de peers candidatos que existen, PC_g la familia de conjuntos de peers candidatos (combinaciones de peers) para el grupo $g \in G_q$, tal que $PC_g \subset PC$ y pc denota un conjunto de peers candidatos (combinación de peers) para el grupo g, tal que $pc \in PC_g$ y $pc \subset I_g$.*

Teorema 1. *El número de combinaciones de pares de peers que se pueden formar a partir de I_g es $\frac{n^2-n}{2}$, tal que $n = \mathbf{card}(I_g)$.*

Definición 17. *Sea K_g una lista de referencia para el grupo $g \in G_q$ y kg una combinación de peers que pueden trabajar entre sí, tal que $kg \in K_g$, $\mathbf{card}(kg) = 2$, $kg \subset I_g$ y $\mathbf{card}(kg) \leq \mathbf{card}(I_g)$. L_g sea una lista de combinaciones de peers que pueden trabajar entre sí en el grupo $g \in G_q$ y lg sea una combinación de peers, tal que $lg \in L_g$, $lg \subset I_g$ y $\mathbf{card}(lg) \leq \mathbf{card}(I_g)$. M_g denota una lista de combinaciones de peers que no pueden trabajar entre sí en el grupo $g \in G_q$ y mg denota una combinación de peers, tal que $mg \in M_g$, $\mathbf{card}(M_g) = 2$, $mg \subset I_g$ y $\mathbf{card}(mg) \leq \mathbf{card}(I_g)$.*

Definición 18. *Sea* R *el conjunto universo de restricciones entre peers,* R_g *el conjunto de restricciones entre peers especificadas en el grupo* $g \in G_q$*,* r *una restricción entre peers especificada en el grupo* $g \in G_q$ *y* m *el número de restricciones entre peers especificadas en el grupo* $g \in G_q$*, tal que* $R_g \subset R$*,* $\mathbf{card}(R_g) = m$ *y* $r \in R_g$*.*

Definición 19. *Sea* T *el conjunto universo de restricciones entre grupos de atributos requeridos,* T_q *el conjunto de restricciones entre grupos de atributos requeridos especificadas en la consulta* $q \in Q_c$*,* t *una restricción entre grupos de atributos requeridos especificada en la consulta* $q \in Q_c$ *y* p *el número de restricciones entre grupos especificadas en la consulta* $q \in Q_c$*, tal que* $T_q \subset T$*,* $\mathbf{card}(T_q) = p$ *y* $t \in T_q$*.*

Teorema 2. *El número de subagentes de empatamiento que se envían en la etapa de empatamiento de peers es* $\frac{n^2-n}{2} \cdot m$*, tal que* $n = \mathbf{card}(I_g)$ *y* $m = \mathbf{card}(R_g)$*.*

La carga computacional de esta fase está distribuida entre un agente de empatamiento[12] que se encuentra en $SP_{initiator}$ y cada agente de consulta enviado por dicho *super-peer*. El objetivo de esta fase es determinar el mejor conjunto de *peers* —a partir de los resultados de la fase de selección— que satisfagan los requerimientos y restricciones de una consulta de recursos. Se dice que q_{agent} obtiene resultados en $SP_v \in VN_c$, si y solo si $CG_v \neq \emptyset$. Para determinar si x es un *peer* candidato para un grupo $g \in G_q$, q_{agent} calcula la penalización de x para cada atributo $a \in A_g$. La ecuación 4.6 define la penalización P_x^a del *peer* x para un atributo a de cualquier tipo (entero, flotante o *string*), la cual está basada en la función de penalización utilizada en SWORD [36]:

$$P_x^a = P_g^a \delta_g^a \tag{4.6}$$

en donde P_g^a es la penalización especificada para el atributo a en el grupo g y δ_g^a es la desviación del atributo a en el grupo g.

La ecuación 4.7 define los casos particulares para el valor de δ_g^a (desviación medida en unidades nativas de un atributo, e.g., GB para el espacio libre en disco o milisegundos para la latencia) en un atributo a de tipo numérico (entero o flotante) especificado en el grupo g:

$$\delta_g^a = \begin{cases} min_ideal - min_val & \text{si } min_val \leq val_x^a < min_ideal \\ max_val - max_ideal & \text{si } max_ideal < val_x^a \leq max_val \\ 0 & \text{si } min_ideal \leq val_x^a \leq max_ideal \\ \infty & \text{en cualquier otro caso} \end{cases} \tag{4.7}$$

en donde min_val y max_val son los valores mínimo y máximo permitidos para un atributo a en un grupo g, respectivamente; min_ideal y max_ideal son los valores mínimo y máximo ideales para un atributo a en un grupo g y val_x^a es el valor real del atributo a en

[12] Un agente de empatamiento es aquel que se encarga de llevar a cabo el empatamiento de *peers* y de grupos, cuya finalidad es determinar cuáles *peers* satisfacen los requerimientos y restricciones de una consulta de recursos.

el *peer* x.

Por otro lado, la ecuación 4.8 define los casos particulares para el valor de δ_g^a en un atributo a de tipo *string* especificado en el grupo g:

$$\delta_g^a = \begin{cases} 0 & \text{si } val_x^a = val_g^a \\ \infty & \text{en cualquier otro caso} \end{cases} \tag{4.8}$$

en donde val_x^a es el valor real del atributo a en el *peer* x y val_g^a es el valor del atributo a especificado en el grupo g.

La penalización total del *peer* x para el grupo g se denota como P_x^g y está definida por la ecuación 4.9:

$$P_x^g = \sum_{i=1}^{k} P_x^i \tag{4.9}$$

en donde k es el número de atributos especificados en el grupo g y P_x^i es la penalización del *peer* x para un atributo i de cualquier tipo.

El *peer* x no satisface los requerimientos del grupo g, si y solo si P_x^g posee un valor infinito; en este caso, el *peer* es omitido y, en caso contrario, se agrega al grupo pertinente $h \in CG_v$. En la figura 4.9 se ilustran las regiones de los valores que puede tomar P_x^a a partir de la ecuación 4.6 [36].

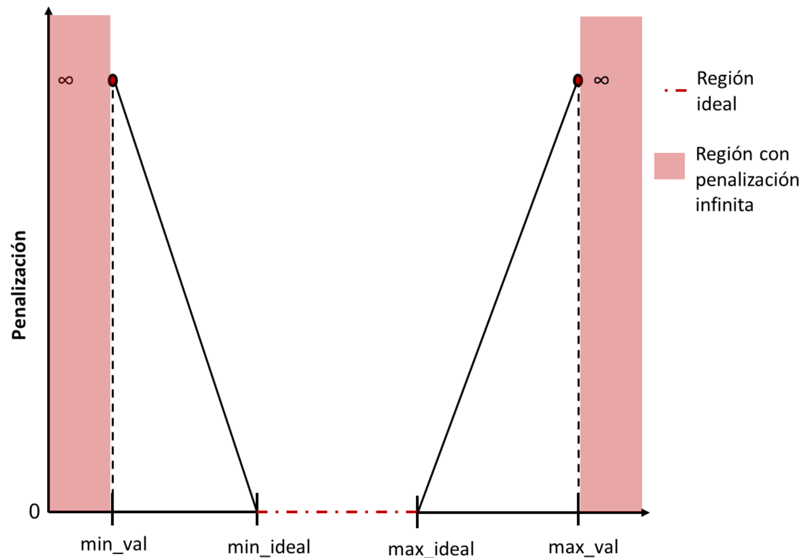


Figura 4.9: Regiones de los valores que puede tomar la penalización de un *peer* para un atributo a de cualquier tipo en un grupo g.

Una vez que $SP_{initiator}$ ha recibido los resultados (denotados como R_{agent}) del agente de consulta q_{agent}, el agente de empatamiento de $SP_{initiator}$ ejecuta un algoritmo basado en programación dinámica para generar todas las posibles combinaciones de *peers* candidatos para cada grupo $g \in G_q$, así como todas las posibles combinaciones de grupos candidatos para la consulta $q \in Q_c$. En la figura 4.10, se observa que el agente de empatamiento divide sus tareas en dos etapas: empatamiento de *peers* candidatos (subsección 4.6.1) y empatamiento de grupos candidatos (subsección 4.6.2). A continuación, se describen dichas etapas.

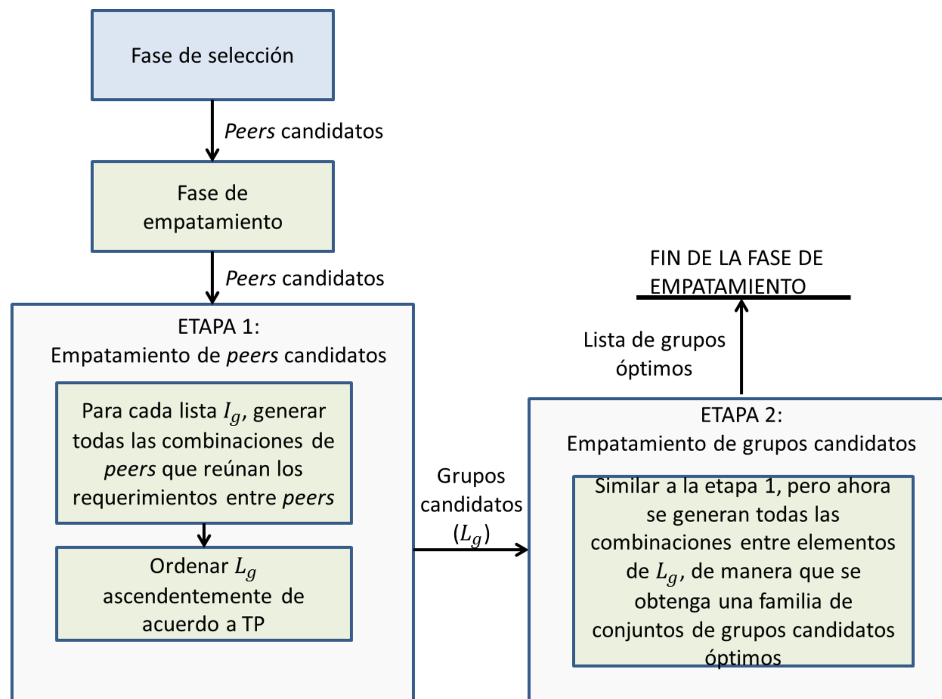


Figura 4.10: Etapas del agente de empatamiento.

4.6.1. Etapa 1: empatamiento de *peers* candidatos

El agente de empatamiento crea una lista I_g para cada grupo g especificado en la consulta q. Posteriormente, obtiene los *peers* candidatos de cada grupo $h \in CG_v$ y añade dichos *peers* a la respectiva lista I_g, de manera que esta lista hace referencia al grupo h. Finalmente, el agente de empatamiento ejecuta las siguientes iteraciones[13] para cada lista I_g cuya cardinalidad sea mayor a dos[14]:

[13] Las iteraciones se representan como $it = z$, en donde z es el número que corresponde a la iteración actual.

[14] En caso de que la cardinalidad de I_g sea menor o igual a dos, no es necesario llevar a cabo las iteraciones del empatamiento de *peers* e I_g se considera como el grupo óptimo para g.

Iteración 1 ($it = 1$):

1. Se crean las listas K_g, L_g y M_g.
2. Se calcula la penalización parcial PP por no satisfacer el número de *peers* especificados para el grupo g, empleando la siguiente ecuación:

$$PP = (n_g - (it + 1)) \times 100 \qquad (4.10)$$

   en donde n_g es el número de *peers* requeridos para el grupo g e it es la iteración actual.

3. Se llena la lista K_g con sub-combinaciones de dos *peers* (derivadas de I_g) que pueden trabajar entre sí, de manera que K_g sirve como una lista de referencia para futuras iteraciones. Para cada combinación de *peers* $pc \in PC_g$ que se puede formar a partir de la lista I_g, tal que $\mathbf{card}(pc) = 2$, se llevan a cabo los siguientes pasos:

   - Se envía un subagente de empatamiento[15] para evaluar las restricciones entre *peers* —especificadas en el grupo g— que existen entre los *peers* de la combinación pc. La penalización de una restricción r entre *peers* se denota como P_r y se determina mediante la siguiente ecuación:

$$P_r = P_g^r \delta_g^r \qquad (4.11)$$

   en donde P_g^r es la penalización especificada para la restricción r entre *peers* en el grupo g y δ_g^r es la desviación de la restricción r en el grupo g[16].

   - Se determina la sumatoria de las m penalizaciones entre *peers*, con la finalidad de establecer un valor de referencia (denotado como RK) que será almacenado en la lista K_g para ser utilizado en futuras iteraciones. Dicho valor de referencia se calcula empleando la siguiente ecuación:

$$RK = \sum_{i=1}^{m} P_r^i \qquad (4.12)$$

   en donde m es el número de restricciones entre *peers* especificadas en el grupo g y P_r^i es la penalización de la restricción i entre *peers*.

   - Se determina la penalización total de la combinación pc, empleando la siguiente ecuación:

$$TP = RK + PP \qquad (4.13)$$

[15] Los subagentes de empatamiento son aquellos que se encargan de determinar la relación que existe entre dos *peers*, e.g., la latencia y el ancho banda.

[16] Una restricción entre *peers* y entre grupos es considerada como un atributo. Las restricciones que se pueden especificar en las consultas de recursos dentro de *RASupport* se muestran en la tabla 4.8 y el valor de la desviación de la restricción r (denotada como δ_g^r) se calcula de manera similiar a las ecuaciones 4.7 (para restricciones de tipo entero y flotante) y 4.8 (para restricciones de tipo *string*).

en donde RK es la sumatoria de las penalizaciones de las m restricciones especificadas en el grupo g y PP es la penalización parcial con respecto al número de *peers*.

- Si el valor de TP es inifinito, la combinación *pc* se agrega a la lista M_g; en caso contrario, dicha combinación se agrega a las listas K_g y L_g. Cada elemento $mg \in M_g$ almacena una combinación de dos *peers* candidatos que no pueden trabajar entre sí, mientras que cada elemento $kg \in K_g$ almacena una combinación de dos *peers* candidatos que pueden trabajar entre sí, así como el valor de referencia (denotado como RK) asociado a kg. Finalmente, cada elemento $lg \in L_g$ almacena una combinación de *peers* candidatos que pueden trabajar entre sí (cuya cardinalidad varía dependiendo de la iteración que genere dicha combinación) y el valor de la penalización total (denotado como TP) asociado a lg.

Restricción	**Descripción**	**Unidades**	**Tipo**	**Ámbito**
`<latency>`	Latencia entre dos *peers* determinada mediante la ecuación 4.4	Milisegundos (ms)	Flotante	Entre *peers* y entre grupos
`<bandwidth>`	Ancho de banda entre dos *peers* determinado de manera similar a la ecuación 4.4	Megabits por segundo (Mbps)	Flotante	Entre *peers* y entre grupos

Tabla 4.8: Restricciones entre *peers* y entre grupos que se pueden especificar en consultas de recursos dentro de *RASupport*.

Iteración z ($it = z \; \forall 1 \leq z < \mathbf{card}(I_g)$):

1. Se crea una lista (denotada como GP) de combinaciones generadas durante la iteración actual, la cual sirve para no generar una misma combinación varias veces, e.g., la combinación $[NP_3, NP_6]$ es equivalente a la combinación $[NP_6, NP_3]$.

2. Se calcula la penalización parcial empleando la ecuación 4.10 con $it = z$.

3. Para cada elemento $kg \in K_g$, se crean todas las combinaciones posibles de *peers* y, para cada nueva sub-combinación *pc* de *peers* derivada de kg (compuesta por $it + 1$ elementos) que no exista en GP, se llevan a cabo los siguientes pasos:
   - Se agrega la sub-combinación *pc* a la lista GP.
   - El valor de la penalización total (denotado como TP) de la sub-combinación *pc* se inicializa en cero.
   - Se crean sub-combinaciones de dos elementos a partir de *pc*. Para dichas sub-combinaciones que no existan en la lista M_g, se toma el valor de RK de la lista K_g y se suma al valor de TP de la sub-combinación *pc*.

- Si el valor de TP de la sub-combinación pc no es infinito, la sub-combinación pc se agrega a la lista L_g junto con su respectivo valor de TP, de manera que pc se vuelve un elemento de L_g.

Iteración $\mathbf{card}(I_g)$ ($it = \mathbf{card}(I_g)$):

1. Se calcula la penalización parcial empleando la ecuación 4.10 con $it = \mathbf{card}(I_g)$.
2. La combinación final de *peers* candidatos (denotada como fc) posee los mismos elementos que I_g, tal que $fc = I_g$. Por lo tanto, si $\mathbf{card}(M_g) = 0$, I_g se vuelve un elemento de L_g; en caso contrario, I_g no puede existir en L_g.

4.6.2. Etapa 2: empatamiento de grupos candidatos

Una vez que el agente de empatamiento ha completado las $\mathbf{card}(I_g)$ iteraciones, la lista L_g se ordena ascendentemente de acuerdo a la penalización total (denotada como TP) de cada $lg \in L_g$. Posteriormente, cada combinación $lg \in L_g$ es considerada como un grupo candidato para $g \in G_q$. Finalmente, cada elemento lg (i.e., cada grupo candidato) se empata con los demás elementos de L_g, de manera que se evalúan las restricciones entre grupos especificadas en la consulta q. De manera similar al empatamiento de *peers*, se conserva una lista de grupos empatados, con la finalidad de evitar llevar a cabo un mismo empatamiento de grupos varias veces (e.g., el empatamiento del grupo 1 con el grupo 2 es equivalente al empatamiento del grupo 2 con el grupo 1).

La penalización de una restricción entre grupos se calcula mediante la ecuación 4.11 y la penalización total de dichas restricciones se calcula mediante la ecuación 4.13. La penalización del empatamiento entre dos grupos (denotada como P_{groups}) se calcula mediante la ecuación 4.14:

$$P_{groups} = \sum_{i=1}^{p} TP_t^i \tag{4.14}$$

en donde TP_r^i es la penalización total de la restricción i entre grupos y p es el número de restricciones entre grupos especificadas en la consulta q.

En la figura 4.11, se ilustra el empatamiento entre dos grupos: G1 y G2. El grupo G1 está compuesto por los *peers normales* NP_3, NP_6 y NP_7; mientras que el grupo $G2$ está compuesto por los *peers normales* NP_{19}, NP_{10} NP_{20}, NP_3, NP_{36} y NP_{37}. El agente de empatamiento emplea un subagente para evaluar una única restricción entre grupos que se especificó en la consulta $q \in Q_c$: latencia con una penalización $P = 0.05$. Por lo tanto, dicho subagente evalúa la latencia que existe entre los *peers* del grupo $G1$ y los *peers* del grupo $G2$, e.g., el subagente de empatamiento determina una latencia de $90ms$ entre los *peers normales* NP_3 y NP_{19}, por lo que el valor de la penalización total entre dichos *peers* es 2.5 (de acuerdo con las ecuaciones 4.11 y 4.13). El agente de empatamiento suma las penalizaciones totales obtenidas en la evaluación de las restricciones en cada una de

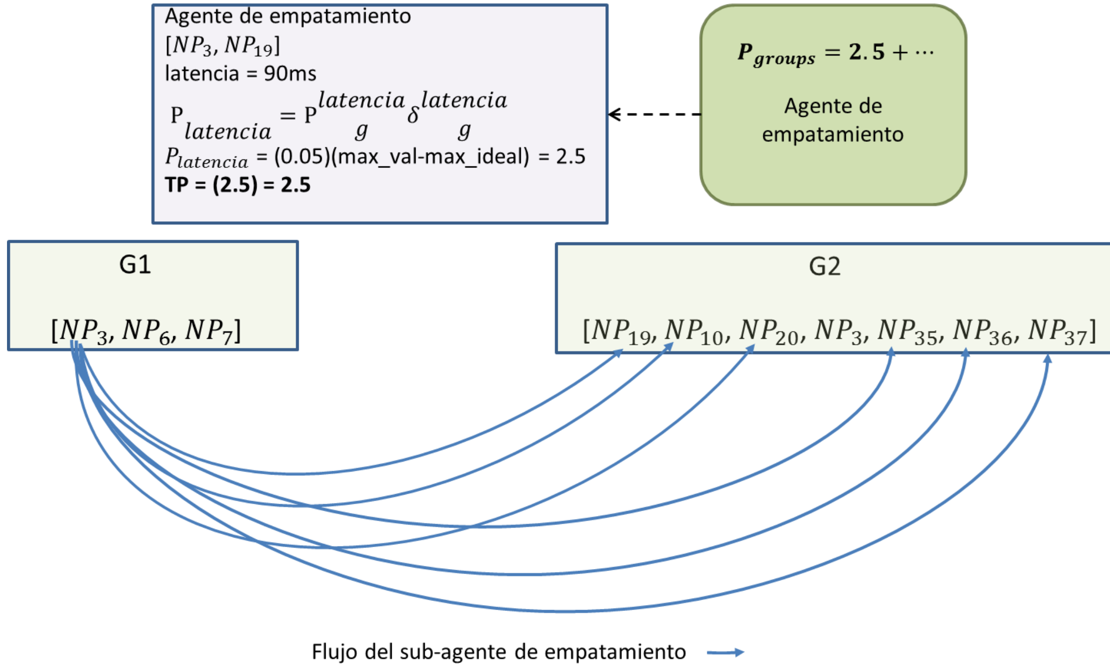


Figura 4.11: Empatamiento entre dos grupos: G1 y G2.

las posibles combinaciones de *peers* entre los grupos $G1$ y $G2$. En la figura 4.11, se observa que el subagente de empatamiento no evalúa la restricción entre los *peers normales* NP_3 y NP_3, debido a que dichos *peers* son el mismo.

La fase de empatamiento está compuesta por dos etapas: empatamiento de *peers* candidatos y empatamiento de grupos candidatos. Al finalizar la fase de empatamiento, se obtiene una lista (i.e., la combinación de grupos que posee la menor penalización entre grupos) de grupos óptimos de *peers* que satisface los requerimientos y restricciones especificadas en una consulta de recursos. En caso de que el agente de empatamiento no haya obtenido resultados, la lista de grupos óptimos estará compuesta por cada lista I_g que corresponde a cada grupo g especificado en la consulta q.

4.7. Fase de enlace

Definición 20. *Sea B_q una familia de conjuntos de peers (grupos óptimos) obtenida en la fase de empatamiento y b_g denota un conjunto de peers (grupo óptimo) que satisface los requerimientos y las restricciones especificadas en una consulta q para un grupo g, tal que $q \in Q_c$, $g \in G_q$ y $b_g \in B_q$.*

Definición 21. *Sea pn un peer proveedor de recursos para un grupo óptimo b_g, tal que $pn \in b_g$ y $pn \in x$.*

Definición 22. *Sea pr un peer consumidor de recursos que emite una consulta q, tal que* $q \in Q_c$ *y* $pr \in x$.

Una vez que el agente de empatamiento ha obtenido el conjunto (denotado como B_q) de los mejores grupos candidatos, es necesario asegurarse de que los recursos encontrados están disponibles para su uso. Debido a conexiones, desconexiones y fallas de *peers*, los recursos solicitados pueden no estar disponibles cuando requieren ser utilizados [1]. Por lo tanto, el objetivo de la fase de enlace es determinar si los recursos seleccionados y empatados están disponibles para su uso.

$SP_{initiator}$ determina cuáles *super-peers* están a cargo de los *peers* proveedores [17]. Para cada *super-peer* a cargo, se llevan a cabo los siguientes pasos:

1. $SP_{initiator}$ envía un agente de enlace.
2. El *super-peer* visitado por el agente de enlace, envía mensajes *ping* a los *peers* proveedores que gestiona, con la finalidad de determinar si estos continúan vivos.
3. El agente de enlace envía un mensaje de solicitud de recursos[18] a los *peers* proveedores que continúan vivos. Para cada grupo óptimo $b_g \in B_q$ en el que un *peer* proveedor *pn* debe trabajar, se siguen las siguientes reglas:
   a) Si *pn* está disponible, se encuentra en la disposición de compartir sus recursos y estos últimos se encuentran disponibles para trabajar en b_g, *pn* responde al agente de enlace con un mensaje de aceptación que permite usar sus recursos.
   b) En caso contrario, *pn* responde al agente de enlace con un mensaje de rechazo, de manera que *pn* deja de ser un elemento de b_g.
4. El agente de enlace regresa sus resultados a $SP_{initiator}$.

$SP_{initiator}$ emplea una operación *join* para esperar los resultados obtenidos por los agentes de enlace que envió, de manera que el resultado de la fase de enlace es la intersección entre el conjunto B_q y los resultados obtenidos por cada uno de los agentes de enlace. $SP_{initiator}$ regresa el resultado de la fase de enlace al *peer normal* que solicitó los recursos, en caso de ser necesario.

En la figura 4.12 se ilustra un escenario que describe la manera en que se lleva a cabo la fase de enlace dentro de *RASupport*. En dicha figura, se observa que el *peer* normal *pr* es el consumidor de recursos, cuya la lista de grupos candidatos para la consulta $q \in Q_c$ (denotada como B_q) inicialmente está compuesta por dos grupos de *peers* proveedores:

[17] Un agente de enlace es aquel que determina si los recursos seleccionados y empatados están disponibles para su uso. Estos agentes están conscientes de la relación que existe entre los *peers* proveedores y los grupos óptimos en los cuáles deben trabajar, ya que un *peer* proveedor puede pertenecer a dos o más grupos de B_q.

[18] Un mensaje de solicitud de recursos es aquel que especifica los recursos que se pretenden utilizar de un *peer* proveedor, de acuerdo a los grupos de B_q en los que dicho *peer* debe trabajar.

$B_q = \{b_c, b_d\}$. En particular, el grupo b_c está compuesto por los *peers* proveedores pn_{10} y pn_{11}, mientras que el grupo b_d está compuesto por el *peer* proveedor pn_9. En el paso 1, $SP_{initiator}$ envía dos agentes de anunciamiento (agente 1 y agente 2) a los *super-peers* SP_6 y SP_7, respectivamente, debido a que estos últimos están a cargo de los *peers* proveedores previamente mencionados. Posteriormente, en el paso 2 los *super-peers* visitados por los agentes de enlace, comprueban que los *peers* proveedores que están a su cargo continúen vivos y, los *peers* proveedores que ya no estén vivos y/o no se encuentren en la disposición de compartir sus recursos, se eliminan del respectivo grupo $b_g \in B_q$ (e.g., el *super-peer* SP_6 determina que pn_{11} ya no se encuentra vivo, por lo que es eliminado de b_c). En el paso 3, los agentes de enlace envían un mensaje de solicitud de recursos a los *peers* proveedores que se encuentran disponibles. Finalmente, en el paso 4 los agentes 1 y 2 regresan sus resultados a $SP_{initiator}$ para que este último envíe B_q —el cual ya no tiene al *peer* proveedor pn_{11} en el grupo b_1— al *peer* consumidor pr.

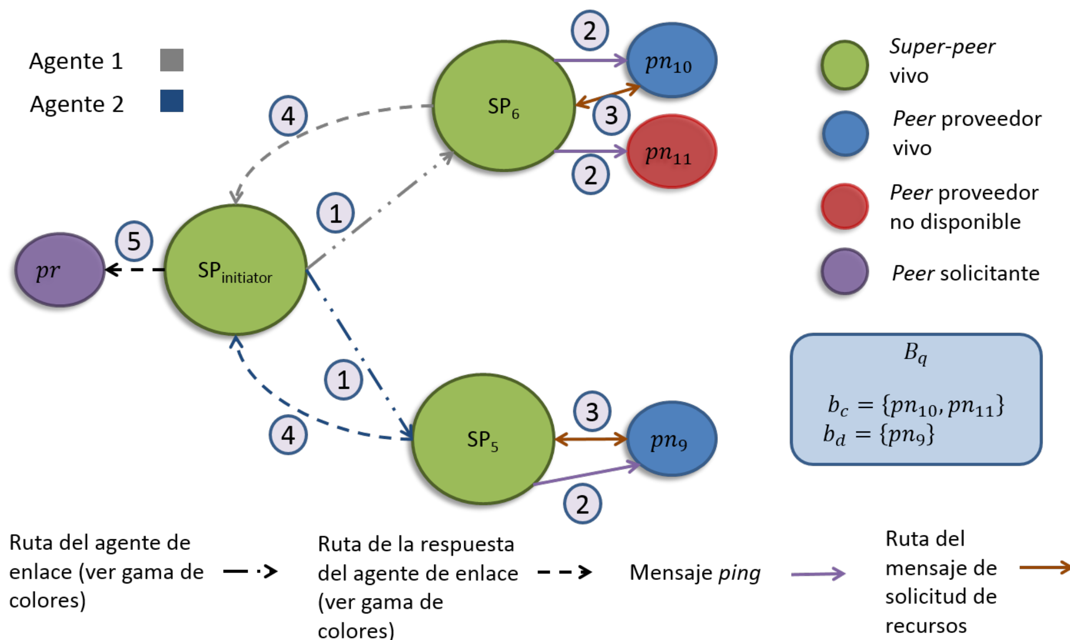


Figura 4.12: Fase de enlace de recursos dentro de *RASupport* para un conjunto compuesto por dos grupos de *peers* proveedores: $B_q = \{b_c, b_d\}$.

4.8. *Toolkit* de agregación de recursos en sistemas P2P colaborativos: *RAToolkit*

RAToolkit permite llevar a cabo la agregación de recursos en sistemas P2P colaborativos y está integrado por cuatro APIs, cada una de las cuales corresponde a las siguientes fases de la agregación de recursos: anunciamiento, selección, empatamiento y enlace. *RAToolkit*

se comunica con un componente denominado Soporte Myconet, el cual se encarga de gestionar la arquitectura de la red P2P y de permitir la comunicación entre *peers*, utilizando el protocolo Myconet. Cada *peer* mantiene una base de datos SQLite que administra las especificaciones de recursos diseminadas por los *peers normales*[19] (ver figura 4.13).

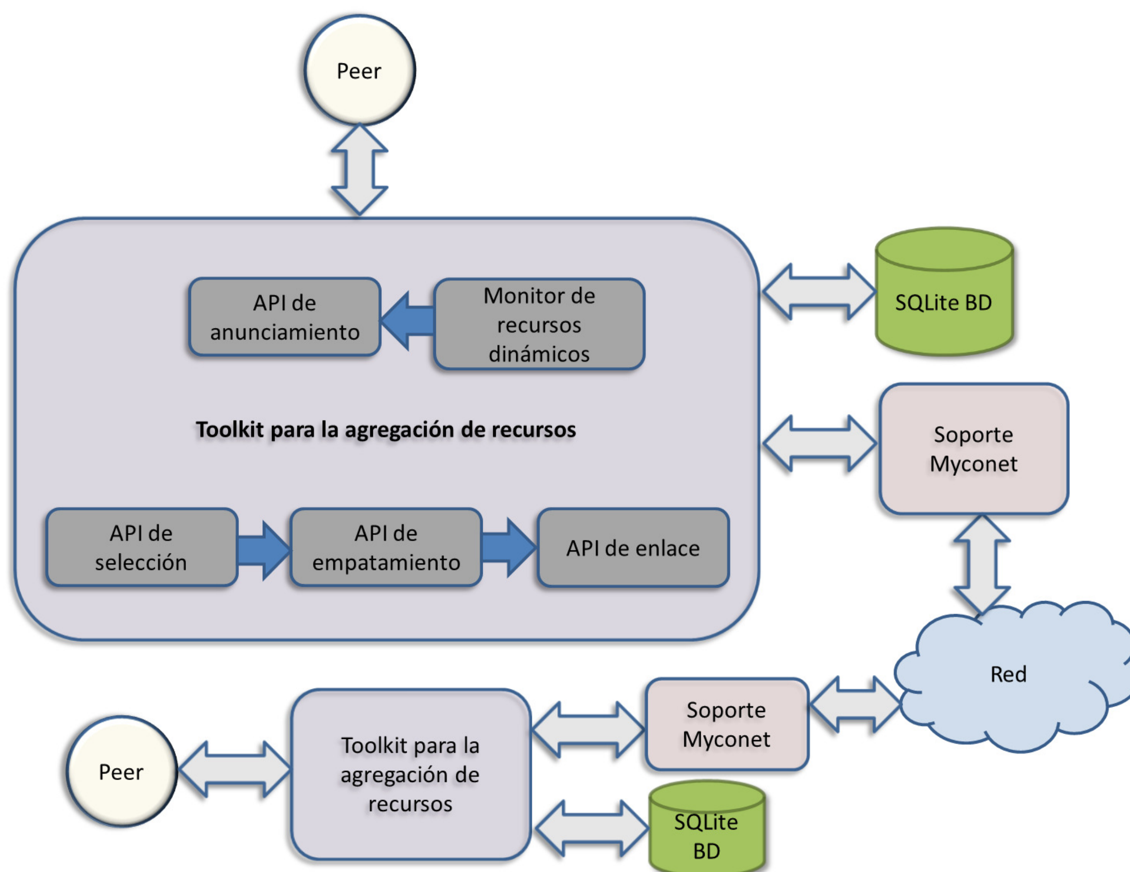


Figura 4.13: Arquitectura de *RAToolkit*.

La fase de descubrimiento se excluye, debido a que los *peers normales* anuncian explícitamente sus recursos a sus respectivos *super-peers*, así como las actualizaciones de los recursos dinámicos con los que cuentan dichos *peers normales*. De esta manera, los *super-peers* no necesitan llevar a cabo la fase de descubrimiento y, por ende, no existe un API que se encargue de llevar a cabo dicha fase. La fase de anunciamiento se lleva a cabo de manera independiente a las demás fases soportadas por *RAToolkit* y el API que se encarga de llevar a cabo dicha fase se comunica con un componente llamado monitor de recursos dinámicos, el cual se encarga de monitorear los recursos dinámicos de un *peer*. Sin embargo, las fases de selección, empatamiento y enlace necesitan comunicarse entre

[19]La base de datos empleada por *RAToolkit* hace uso del gestor SQLite y únicamente es utilizada por los *super-peers* para gestionar las especificaciones de recursos de sus *peers normales*.

sí, debido a que son las fases clave de la agregación de recursos. Los resultados de la fase de selección son utilizados por la fase de empatamiento, mientras que los resultados de la fase de empatamiento son utilizados por la fase de enlace; de esta manera, *RAToolkit* proporciona cohesión entre las fases de selección, empatamiento y enlace.

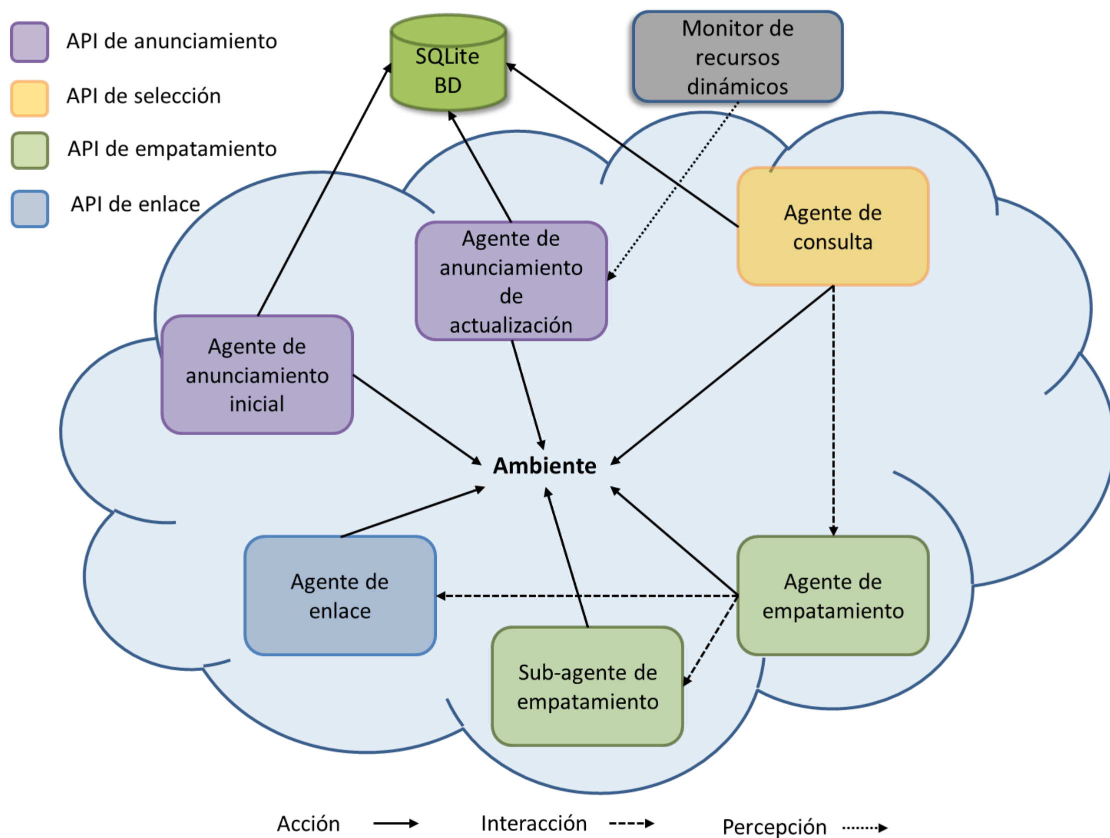


Figura 4.14: Relación entre los agentes y el ambiente.

RAToolkit está basado en un sistema multi-agente, en donde un conjunto de agentes de software activos[20] colaboran entre sí, con la finalidad de llevar a cabo las fases de la agregación de recursos de una manera satisfactoria, garantizando así la Calidad de Servicio deseada. Los agentes involucrados en *RAToolkit* se muestran en la tabla 4.9 y su relación se ilustra en la figura 4.14. Todos los agentes involucrados son reactivos, debido a que sus acciones están determinadas por ciertos eventos (ver figura 4.15), e.g., un agente de anunciamiento de actualización se acciona cuando el monitor de recursos dinámicos le notifica que un recurso ha cambiado. En el sistema multi-agente de *RAToolkit*, ningún agente posee una vista global del sistema.

[20]Un agente activo en un sistema multi-agente es aquel que tiene una meta simple, e.g., un pájaro en una parvada.

Agente	**Función**	**Comportamiento**	**Tipo**
Agente de anunciamiento inicial	Anuncia por primera vez los atributos dinámicos y estáticos de un *peer normal* a su respectivo *super-peer*	Transporta un documento XML de especificación de recursos con un costo de $O(1)$	Reactivo
Agente de anunciamiento de actualización	Anuncia una actualización de atributo(s) dinámico(s) de un *peer normal* a su respectivo *super-peer*	Transporta un documento XML de actualización de recursos con un costo de $O(1)$	Reactivo
Agente de consulta	Consulta los recursos solicitados por un *peer normal* o un *super-peer*, de manera que los resultados satisfagan los requerimientos y restricciones entre *peers* especificadas en la consulta de recursos	Transporta un documento XML que contiene una consulta de recursos. Su comportamiento varía dependiendo del propósito específico de la aplicación, la cual puede requerir encontrar recursos imperativamente o ahorrar ancho de banda	Reactivo
Agente de empatamiento	Lleva a cabo el empatamiento de *peers* y de grupos de atributos requeridos, con la finalidad de obtener un conjunto de *peers* que satisfagan los requerimientos y restricciones requeridas por un *peer* solicitante	Se encuentra en $SP_{initiator}$ y utiliza los resultados de los agentes de consulta para ejecutar un algoritmo basado en programación dinámica. Su comportamiento está dividido en dos fases: empatamiento de *peers* y empatamiento de grupos de atributos requeridos	Reactivo
Subagente de empatamiento	Establece la relación que existe entre dos *peers*, e.g., el ancho de banda	Auxilia a un agente de empatamiento, visitando dos *peers* para determinar la relación que existe entre ellos, e.g., la latencia (empleando la ecuación 4.4)	Reactivo
Agente de enlace	Determina si los recursos seleccionados y empatados están disponibles para su uso	Se envía a los *super-peers* que gestionan a los *peers* proveedores de recursos obtenidos por un agente de empatamiento	Reactivo

Tabla 4.9: Agentes involucrados en *RAToolkit*.

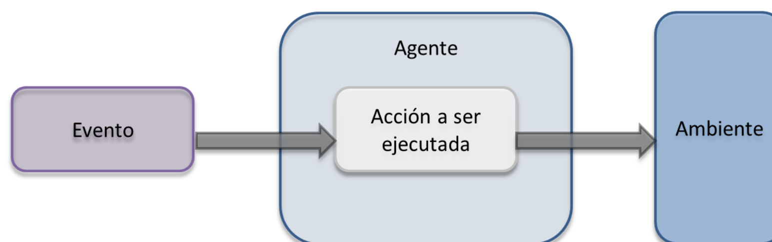


Figura 4.15: Comportamiento de los agentes dentro de *RAToolkit*.

En el sistema multi-agente propuesto, $SP_{initiator}$ actúa como coordinador para los agentes de consulta, los agentes de empatamiento, los subagentes de empatamiento y los agentes enlace. Dicho coordinador funciona de la siguiente manera:

1. Crea un conjunto de agentes de consulta.
2. Crea un agente de empatamiento que se encarga que esperar la respuesta de los agentes de consulta, los cuales se envían empleando el protocolo de inundación con agentes de consulta (subsección 4.5.1) o el protocolo de paseos aleatorios inteligentes (subsección 4.5.2).
3. Espera los resultados del agente de empatamiento.
4. Crea un conjunto de agentes de enlace con los resultados del agente de empatamiento.
5. Envía los agentes de enlace a los *super-peers* de cada nodo proveedor de recursos y emplea una operación *join* para esperar los resultados de dichos agentes de enlace.
6. En caso de ser necesario, regresa los resultados al *peer* normal que emitió la consulta de recursos.

En una orquesta sinfónica, una pieza musical surge a partir de la colaboración entre un conjunto músicos, cada uno de los cuales posee un objetivo bien definido: ejecutar su instrumento para interpretar una melodía y/o ritmo. Sin embargo, es posible que un subconjunto de dichos músicos (e.g., los violinistas) se encargue de un ensamble particular perteneciente a la pieza musical en cuestión. El sistema multi-agente de *RAToolkit* se basa en la analogía anterior, debido a que los agentes interactúan entre sí (la interacción entre agentes da origen a una misma pieza musical: la agregación de recursos), tienen objetivos particulares diferentes (cada agente ejecuta su propio instrumento) y un mismo objetivo global (interpretar una misma pieza musical: la agregación de recursos). De manera similar al ensamble de violines, los agentes de empatamiento y los subagentes de empatamiento tienen un mismo fin (ensamblar la fase de empatamiento), sin embargo, cada uno de ellos se comporta de manera diferente (retomando la analogía, en un ensamble de violines un violinista A podría ejecutar la tónica, mientras un violinista B ejecuta terceras). A lo anterior se le denomina “Orquestación de agentes”.

En la figura 4.16 se muestra el flujo de trabajo dentro de *RAToolkit*, en donde se observa la interacción entre los principales componentes, de manera que se proporciona cohesión entre las fases clave de la agregación de recursos: selección, empatamiento y enlace. En dicha figura se observa que los agentes de anunciamiento inicial y de actualización se encuentran en un ciclo infinito (mientras dichos agentes continúen vivos) enviando especificaciones de recursos (documentos XML) a la base de datos de los *super-peers* que están a cargo. Cuando el *peer* normal NP solicita recursos, envía la consulta a $SP_{initiator}$ para que este último lleve a cabo la fase de selección, mediante la generación de N agentes de

consulta. Posteriormente, los resultados de la fase de selección (i.e., los *peers* candidatos para la consulta q) son empleados por un agente de empatamiento que, en conjunto con un subagente de empatamiento, obtiene los grupos óptimos (de atributos requeridos) para la consulta q. Los agentes de enlace utilizan los resultados de la fase de empatamiento para determinar si los *peers* proveedores (i.e., los *peers* que se encuentran en los grupos óptimos), así como sus respectivos recursos, están disponibles para su uso. Finalmente, $SP_{initiator}$ recibe los resultados de cada uno de los agentes de enlace (i.e., los grupos óptimos enlazados para la consulta q) y regresa dichos resultados al *peer* normal NP, el cual emitió la consulta.

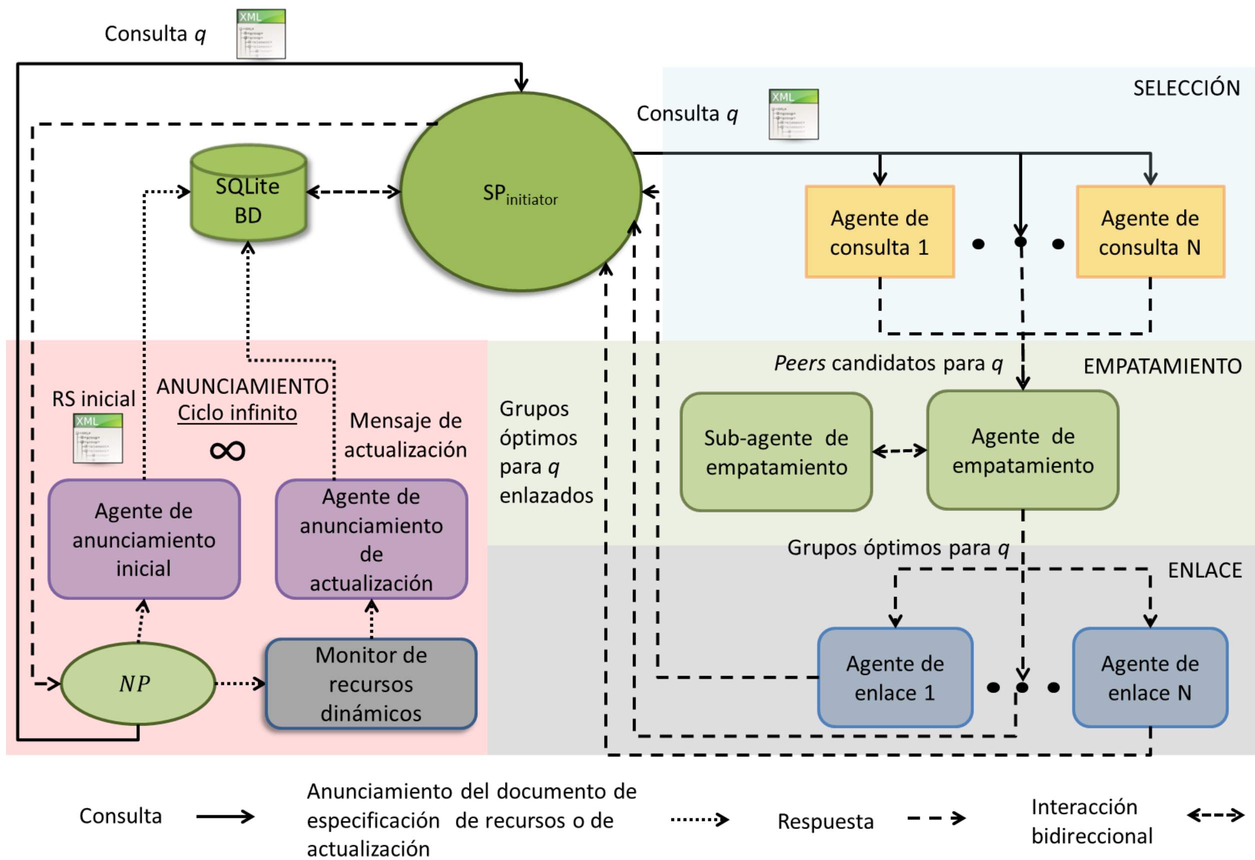


Figura 4.16: Flujo de trabajo de *RAToolkit*.

Capítulo 5

Implementación de *RASupport*

En este capítulo, se describen los principales módulos y componentes que conforman a *RASupport*, el cual se encuentra implementado en lenguaje Java, debido a que es el lenguaje orientado a objetos más popular que existe actualmente; dicho lenguaje ofrece portabilidad y una gran cantidad de bibliotecas que pueden ser utilizadas por desarrolladores de software. En particular, *RASupport* emplea la versión más reciente del JDK (*Java Development Kit 8*, por sus siglas en inglés) de la versión estándar de Java (Java SE). Además, todos los módulos y componentes de *RASupport* se encuentran bien documentados, poseen código fuente simple y reusable, se encuentran desacoplados y mantienen una alta cohesión entre sí. Por lo tanto, *RASupport* es flexible, robusto, comprensible y su comportamiento varía en función de los requerimientos y de las restricciones de una aplicación cliente. Los principales módulos de *RASupport* son los siguientes:

- **RASupportMain.** Es el módulo principal de *RASupport*, el cual es utilizado por los clientes que desean acceder a la funcionalidad que proporciona dicho soporte (sección 5.1).

- **RASupportConfig.** Es el módulo común que proporciona acceso a la configuración de *RASupport* (sección 5.2).

- **ResourceSim.** Es el módulo que permite llevar a cabo simulaciones de recursos heterogéneos caracterizados por atributos estáticos y dinámicos; además proporciona un monitor de recursos que detecta cambios en los recursos dinámicos de un nodo (sección 5.3).

- **RealResourceManager.** Es el módulo que permite la gestión de los recursos reales de un nodo (este módulo no se describe en la presente tesis, debido a que los experimentos se basan en simulaciones de una gran cantidad de nodos que poseen recursos heterogéneos, por lo cual se utilizó el módulo *ResourceSim*).

- **RAToolkit.** Es el módulo que permite llevar a cabo las fases de anunciamiento, selección, empatamiento y enlace de recursos (sección 5.4).

El diseño de *RASupport* se encuentra modelado con UML 2 (*Unified Modeling Language*, por sus siglas en inglés) y los diagramas se crearon con la herramienta de software *Visual Paradigm*. A excepción de *RASupportConfig*, todos los módulos principales de *RASupport* utilizan el patrón de diseño *Fachada* [14], de manera que se oculta la complejidad de dichos módulos. En los diagramas de paquetes (también conocidos como diagramas de módulos) las fachadas se representan mediante el símbolo de interfaz.

La implementación de *RASupport* se llevó a cabo utilizando la IDE (*Integrated Development Environment*, por sus siglas en inglés) Netbeans 8, mientras que para controlar las versiones del proyecto se utilizó la herramienta de software *Git*. En particular, el código fuente de *RASupport* se encuentra alojado en un repositorio de *GitHub*, el cual puede ser consultado en https://github.com/damianarellanes/RASupport.git.

5.1. Módulo principal de *RASupport*: *RASupportMain*

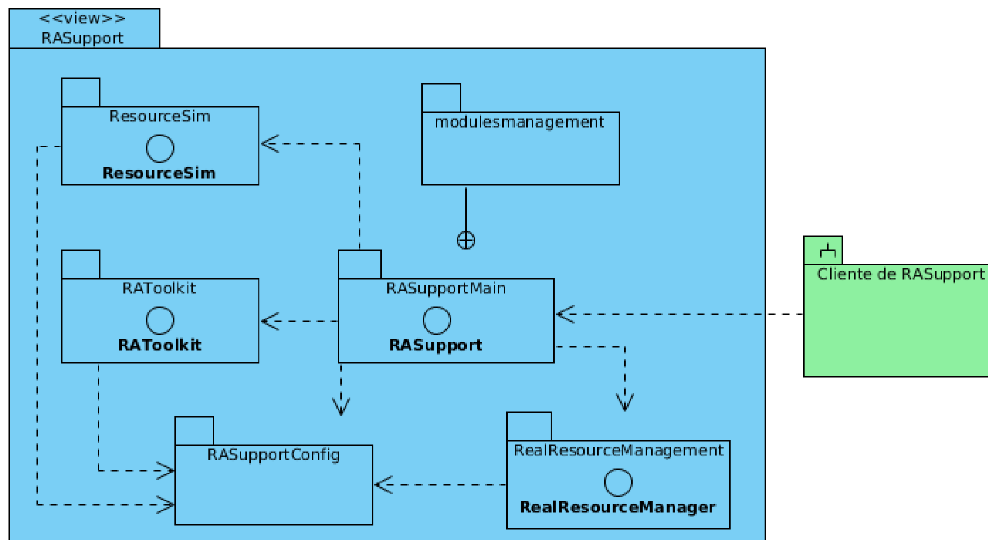


Figura 5.1: Diagrama de paquetes de *RASupport*.

RASupportMain es utilizado por los clientes de *RASupport* para acceder a la funcionalidad que este último proporciona. Por lo tanto, *RASupportMain* depende de todos los módulos del soporte propuesto (ver figura 5.1) y emplea el patrón de diseño *Fábrica* [14] para crear nuevos módulos (ver figura 5.2), de manera que estos se pueden agregar, eliminar y cambiar fácilmente (e.g., es posible substituir el módulo *ResourceSim* por el módulo *RealResourceManager*, con la finalidad de cambiar la manera en que se gestionan los recursos de un nodo). Además, *RASupportMain* facilita el uso del soporte propuesto, debido a que ofrece una fachada que oculta la complejidad de los demás módulos,

reduciendo así la cantidad de líneas de código del cliente.

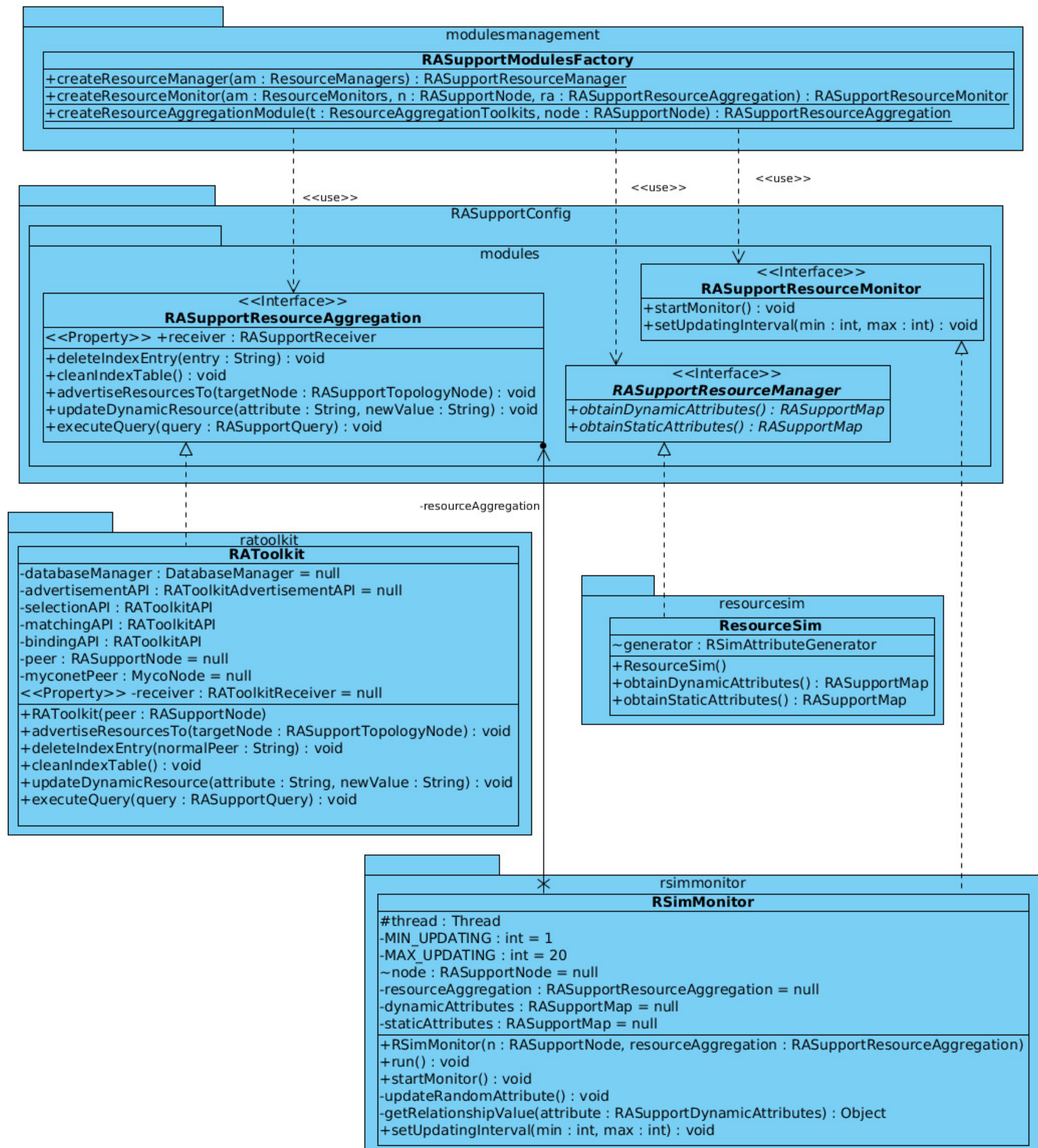


Figura 5.2: Diagrama de clases del submódulo *modulesmanagement*.

RASupportMain cuenta con un submódulo denominado *modulesmanagement*. En la figura 5.2 se muestra el diagrama de clases de tal submódulo, así como su relación con otros módulos de *RASupport*. Además, en esta figura también se observa que *modulesma-*

nagement cuenta con la clase *RASupportModulesFactory* que utiliza el patrón de diseño *Fábrica* [14] para crear módulos de gestión de recursos (*RASupportResourceManager*), módulos de monitoreo de recursos dinámicos (*RASupportResourceMonitor*) y módulos de agregación de recursos (*RASupportResourceAggregation*). *RASupportResourceManager* es una interfaz que implementan las fachadas de los módulos de gestión de recursos, los cuales se encargan de obtener recursos caracterizados por atributos estáticos y dinámicos en un nodo; en particular, *RASupport* ofrece dos módulos de gestión de recursos: un módulo para generar recursos aleatoriamente (*ResourceSim*) y otro módulo para obtener los recursos reales de un nodo (*RealResourceManager*). Por otro lado, *RASupportResourceMonitor* es una interfaz que implementan las fachadas de los monitores de recursos, los cuales se encargan de monitorear y detectar cambios en los recursos dinámicos de un nodo; en particular, *RASupport* únicamente ofrece el monitor de recursos (*RSimMonitor*) proporcionado por el módulo *ResourceSim*. Finalmente, *RASupportResourceAggregation* es una interfaz que implementan los módulos de agregación de recursos, los cuales llevan a cabo las fases de la agregación de recursos; dichas fases pueden variar dependiendo del módulo específico que se esté usando, e.g., para llevar a cabo las simulaciones en la presente tesis, se utilizó el módulo *RAToolkit*, el cual permite efectuar las fases de anunciamiento, selección, empatamiento y enlace, empleando los mecanismos descritos en el capítulo 4.

5.2. Módulo de configuración y modelado de recursos: *RASupportConfig*

El módulo *RASupportConfig* es utilizado por los demás módulos, debido a que contiene la configuración especificada por el cliente y la configuración interna de *RASupport*. En la tabla 5.1 se describen los parámetros que un cliente puede configurar, cuyos valores se definen en un archivo de configuración con extensión *.properties*.

Parámetro	**Descripción**	**Valores permitidos**
peer_alias	Define el alias del *peer* dentro de *RASupport*	Cadena de caracteres que identifica al *peer*
mode	Define el modo de operación de *RASupport*	▪ *random*. Permite la generación de recursos heterogéneos caracterizados por atributos estáticos y dinámicos con valores aleatorios; además, crea un monitor que simula actualizaciones en recursos dinámicos ▪ *real*. Permite la obtención de los recursos reales de un *peer* y crea un monitor que detecta cambios en los recursos dinámicos de este *peer*
toolkit	Define el *toolkit* que se utilizará para llevar a cabo la agregación de recursos	▪ *ratoolkit*. Utiliza *RAToolkit*, el cual se encuentra descrito en el capítulo 4 y lleva a cabo las fases de anunciamiento, selección, empatamiento y enlace de recursos en un ambiente P2P altamente dinámico
use_restriction: establece una restricción de uso para los recursos de un *peer*		
availability	Establece el horario en que se encuentran disponibles los recursos de un *peer*, especificando la hora de inicio (mediante la propiedad *start*) y la hora de finalización (mediante la propiedad *end*)	Para las propiedades *start* y *end*, se especifica un rango de hora con el siguiente formato: hh:mm (hora:minutos)
max_cpu_utilization	Establece el porcentaje máximo de utilización del CPU de un *peer*	Se especifica un valor entero en el intervalo [0,99]
only_for_friends	Determina si los recursos de un *peer* solo serán compartidos por *peers* amigos	Se especifican los alias de los *peers* amigos con los que se desea compartir recursos, separándolos con coma. Dichos alias no se pueden repetir

Tabla 5.1: Parámetros configurables de *RASupport*.

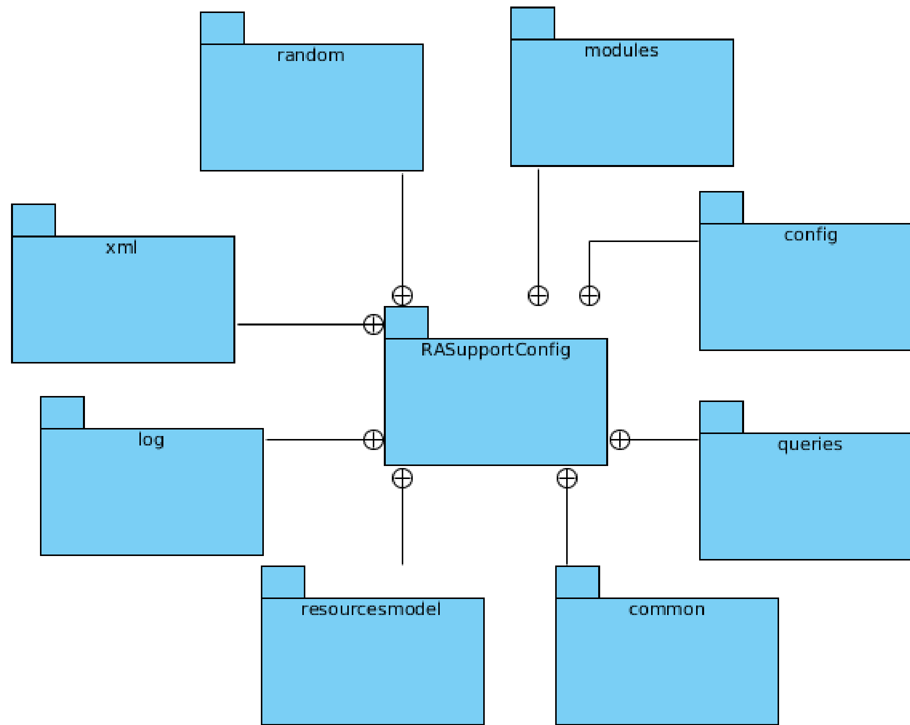


Figura 5.3: Diagrama de paquetes del módulo *RASupportConfig*.

En la figura 5.3 se muestra el diagrama de paquetes del módulo *RASupportConfig*, el cual está conformado por ocho submódulos: *common*, *config*, *log*, *modules*, *queries*, *random*, *resourcesmodel* y *xml*. El submódulo *common* proporciona las clases que definen constantes de configuración generales para *RASupport*, e.g., los tipos de recursos mantenidos y los modos en que dicho soporte puede operar. El submódulo *config* define las clases que administran la configuración especificada por un cliente de *RASupport* en el archivo *.properties*. El submódulo *log* se encarga de gestionar los mensajes (informativos, de error y de advertencia) que son mostrados al cliente, en caso de que se desee proporcionar información sobre algún evento (e.g., se muestra un mensaje de error cuando se especifica un parámetro inexistente en el archivo de configuración); así mismo, es posible rastrear dichos mensajes, debido a que se conocen las clases que los originan. El submódulo *modules* se encarga de gestionar la configuración sobre los módulos de *RASupport*; en particular, contiene las interfaces que deben implementar las fachadas de los módulos que existen y llegasen a existir, garantizando así la escalabilidad y la fácil adición de nuevos módulos. El submódulo *queries* proporciona las clases que representan consultas de recursos dentro de *RASupport*, de manera que estas puedan ser mapeadas a objetos; en particular, contiene la enumeración *RASupportGroupRestrictions* en donde se especifican las restricciones entre grupos de atributos requeridos, la enumeración *RASupportNodeRestrictions* en donde se especifican las restricciones entre nodos, la clase *RASupportQuery* que representa una consulta de recursos, la clase *RASupportQueryGroup* que representa un grupo de atributos requeridos dentro de una consulta, la enumeración *RASupportQueryOptions* en donde

se especifican las opciones que definen el comportamiento de la fase de selección, la clase *RASupportQueryRequirement* que representa un requerimiento (atributo o restricción) de una consulta de recursos, la clase *RASupportQueryRestrictionSet* que representa un conjunto de restricciones (entre nodos o entre grupos de atributos requeridos) y la interfaz *RASupportQueryRestrictions* que define los métodos que deben implementar las enumeraciones que especifican las restricciones entre nodos y entre grupos. El submódulo *random* proporciona las clases que permiten gestionar la generación de números pseudoaleatorios dentro de *RASupport*.

En particular, las clases *RASupportQueryGroup* y *RASupportQueryRestrictionSet* proporcionadas por el submódulo *queries* poseen fachadas que permiten agregar atributos y restricciones entre nodos y entre grupos. La clase *RASupportQuery* valida que no existan grupos de atributos requeridos con el mismo nombre y que únicamente exista una opción (*requires:performance* o *requires:find_resources*) empleada en la fase de selección, un TTL y un conjunto de restricciones entre grupos. Además, *RASupportQuery* valida que en un grupo de atributos requeridos se haya especificado un nombre y el número de nodos requeridos. *RASupportQuery* puede poseer más de un conjunto de restricciones entre grupos, sin embargo estas deben poseer al menos dos grupos bajo restricción[1] y al menos una restricción entre grupos. Con la finalidad de mantener un orden en las consultas de recursos, *RASupport* crea un directorio denominado *RASupportQueries*, el cual es utilizado para almacenar todos los documentos XML de consulta de recursos que se crean.

En la figura 5.4 se muestra el diagrama de clases del submódulo *resourcesmodel*, el cual se encarga de gestionar la descripción de recursos dentro de *RASupport*, de manera que se pueden añadir fácilmente modelos de recursos, atributos estáticos, atributos dinámicos y restricciones de uso para dichos recursos, e.g., para dar soporte a un nuevo atributo dinámico, solo basta con agregar un nuevo elemento en la enumeración *RASupportDynamicAttributes* (ver figura 5.5). Además, *resourcesmodel* proporciona la clase *RASupportMap* que permite la gestión concurrente de un conjunto de objetos (e.g., un conjunto de atributos dinámicos) debido a que hereda de un *mapa hash concurrente* de Java (ver figura 5.4).

[1]Un grupo bajo restricción es aquel que participa en la evaluación de un conjunto de restricciones entre grupos

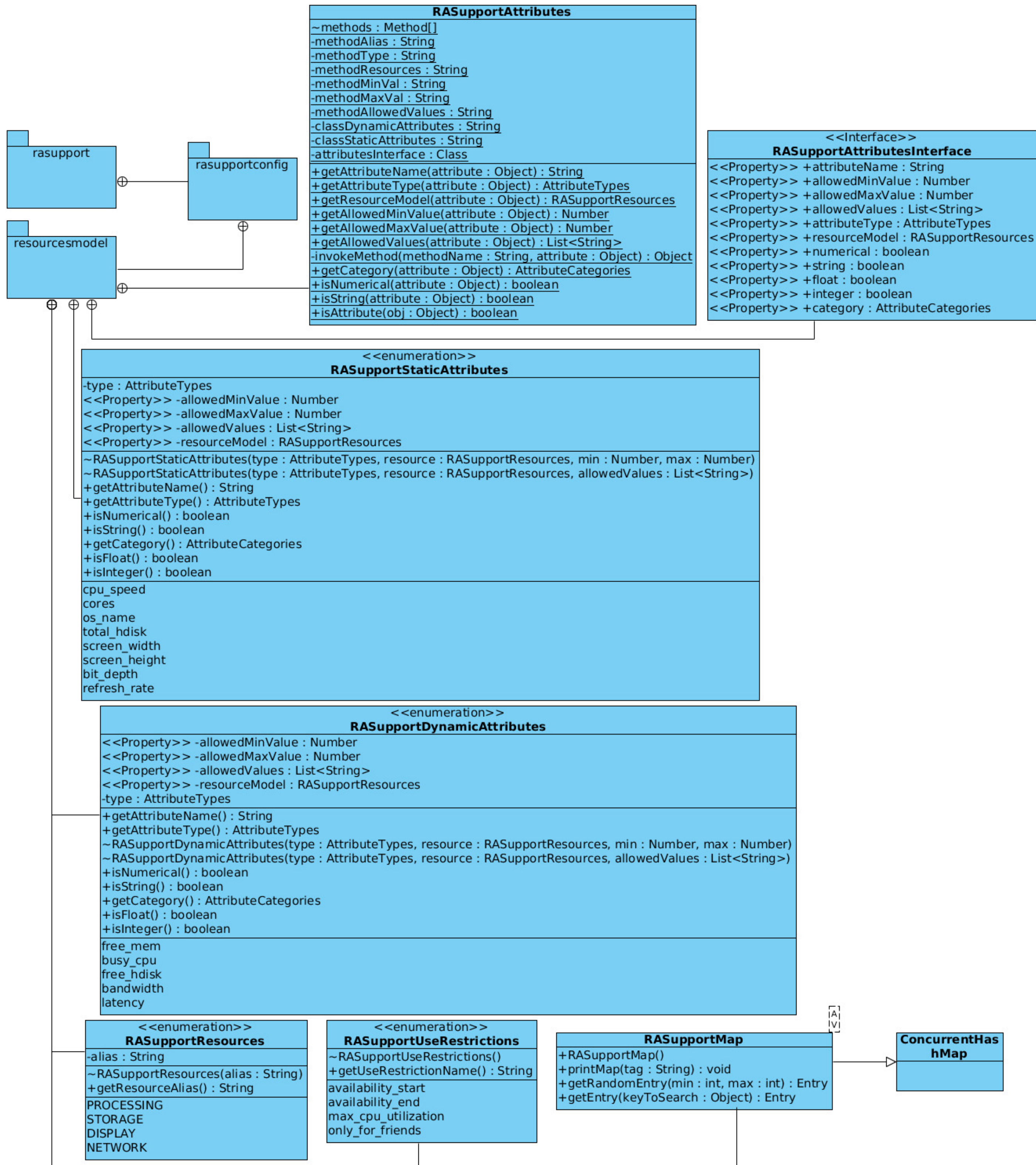


Figura 5.4: Diagrama de clases del submódulo *resourcesmodel*.

```
1  package RASupport.rasupport.rasupportconfig.resourcesmodel;
2
3  import RASupport.rasupport.rasupportconfig.common.RASupportCommon;
4  import RASupport.rasupport.rasupportconfig.common.RASupportCommon.AttributeCategories;
5  import static RASupport.rasupport.rasupportconfig.common.RASupportCommon.AttributeCategories.*;
6  import RASupport.rasupport.rasupportconfig.common.RASupportCommon.AttributeTypes;
7  import static RASupport.rasupport.rasupportconfig.common.RASupportCommon.AttributeTypes.*;
8  import static RASupport.rasupport.rasupportconfig.resourcesmodel.RASupportResources.*;
9  import java.util.List;
10
11 /**
12  * RASupportConfig: allowed dynamic attributes in RASupport
13  * Add new dynamic attributes into RASupport
14  * @author Damian Arellanes
15  */
16 public enum RASupportDynamicAttributes implements RASupportAttributesInterface {
17
18     // PROCESSING ATTRIBUTES
19     free_mem (FLOAT_ATTRIBUTE, PROCESSING, 10f, 16000f),
20     busy_cpu (FLOAT_ATTRIBUTE, PROCESSING, 0f, 100f),
21
22     // STORAGE ATTRIBUTES
23     free_hdisk (FLOAT_ATTRIBUTE, STORAGE, 0f, 1000000f),
24
25     // DISPLAY ATTRIBUTES
26
27     // NETWORK ATTRIBUTES
28     bandwidth (FLOAT_ATTRIBUTE, NETWORK, 100f, 1000f),
29     latency (FLOAT_ATTRIBUTE, NETWORK, 100f, 1000f);
```

Nuevo atributo

Figura 5.5: Ejemplo de adición de un nuevo atributo dinámico (espacio libre en disco: *free_hdisk*) en *RASupport*, en el cual se especifica el tipo de atributo (flotante), el recurso al que caracteriza (recurso de almacenamiento: *STORAGE*) y el intervalo ([0,1000000]) utilizado por *ResourceSim* para generar valores aleatorios para dicho atributo.

Finalmente, el submódulo *xml* contiene las clases necesarias para gestionar documentos XML dentro de *RASupport*. En particular, dicho submódulo emplea el API *StAX* (*Streaming API for XML*) para llevar a cabo el análisis y la generación de documentos XML. A diferencia de DOM (*Document Object Model*, por sus siglas en inglés), StAX permite manejar documentos de gran tamaño sin necesidad de cargarlos en memoria. Así mismo, StAX ofrece la funcionalidad para escribir documentos XML, lo cual no sucede con el API SAX (*Simple API for XML*). En la tabla 5.2 se muestra una comparación entre las APIs previamente mencionadas. El submódulo *xml* posee dos clases que permiten mapear documentos XML de consulta de recursos a objetos y viceversa. *XMLQueryWriter* permite escribir documentos XML de consulta de recursos a partir de un objeto de la clase *RASupportQuery*, mientras que *XMLReader* permite leer documentos XML de consulta de recursos para posteriormente mapearlos (de forma automática) a objetos de la clase *RASupportQuery*.

Característica	**StAX**	**SAX**	**DOM**
Tipo de API	*Pull*, *streaming*	*Push*, *streaming*	Árbol en memoria
Facilidad de uso	Alta	Media	Alta
Eficiencia en CPU y memoria	Buena	Buena	Varía
Lectura de documentos XML	Sí	Sí	Sí
Escritura de documentos XML	Sí	No	Sí

Tabla 5.2: Tabla comparativa entre StAX, SAX y DOM.

Con la finalidad de estandarizar los documentos XML de consultas de recursos dentro de *RASupport*, se diseñaron una serie de reglas que validan que dichos documentos estén bien formados y que cumplan con los requisitos establecidos por el soporte propuesto (ver tabla 5.3). Para ello, se utilizan documentos *XSD* (*XML Schema Definition*, por sus siglas en inglés) debido a que estos ayudan a los documentos XML de consultas de recursos a:

- Validar que la estructura del documento sea correcta, e.g., únicamente debe existir una etiqueta `<ttl>`.
- Definir los elementos permitidos, e.g., solo se permite especificar atributos para los que *RASupport* da soporte.
- Definir la secuencia de aparición de los elementos, e.g., los grupos de atributos requeridos (i.e., las etiquetas `<group>`) deben aparecer antes de las restricciones entre grupos (i.e., la etiqueta `<rest_betw_groups>`).
- Establecer las reglas de cada elemento, e.g., los atributos de tipo *string* deben empatar con el patrón $..*, \backslash d + (\backslash . \backslash d+)$.
- Definir los tipos de dato de cada elemento, e.g., las restricciones entre nodos (i.e., las etiquetas `<rest_betw_nodes>`) pueden ser de tipo entero, flotante o *string*.

La clase *XMLQueryReader* emplea las mismas reglas descritas en la tabla 5.3 y añade otras más para la lectura de documentos XML de consultas de recursos, las cuales se describen en la tabla 5.4.

Regla	Descripción	Motivo	Clases en las que se aplica
1	Las restricciones entre nodos no deben poseer nodos bajo restricción	Los nombres de los nodos que se obtienen en un grupo de atributos requeridos se desconocen cuando se origina una consulta	*RASupportQueryGroup*
2	Los métodos que solicitan atributos numéricos (enteros o flotantes) no permiten atributos de tipo *string*	En las consultas de recursos, los atributos de tipo numérico poseen un valor mínimo requerido, un valor mínimo ideal, un valor máximo ideal, un valor máximo requerido y la penalización; por lo tanto, dichos valores son necesarios en atributos numéricos	*RASupportQueryGroup* y *RASupportQueryRestrictionSet*
3	Los métodos que solicitan atributos de tipo *string* no permiten atributos numéricos (enteros o flotantes)	En las consultas de recursos, los atributos de tipo *string* poseen el valor requerido y la penalización; por lo tanto, no tiene caso especificar un valor mínimo requerido, un valor mínimo ideal, un valor máximo ideal y un valor máximo requerido	*RASupportQueryGroup* y *RASupportQueryRestrictionSet*
4	La duplicidad de atributos no está permitida	Los grupos de atributos requeridos en una consulta, las restricciones entre nodos y las restricciones entre grupos no permiten la duplicidad de atributos, con la finalidad de evitar conflictos en la evaluación de los últimos	*RASupportQueryGroup* y *RASupportQueryRestrictionSet*
5	Las restricciones entre nodos y entre grupos de atributos deben contar al menos con una restricción	No tiene sentido contar con un conjunto de restricciones si no se especificó al menos una restricción	*RASupportQueryGroup* y *RASupportQuery*
6	Las consultas de recursos solo deben poseer una opción y un TTL	Las consultas de recursos no deben poseer ambiguedades	*RASupportQuery*
7	Los grupos de atributos requeridos en una consulta deben poseer un nombre y el número de nodos requeridos	Las consultas de recursos identifican a un grupo de atributos requeridos a través de su nombre y obtienen un conjunto de resultados para estos basándose en el número de nodos que se requieren	*RASupportQuery*
8	La duplicidad de los nombres de los grupos de atributos requeridos en una consulta no está permitida	Las consultas de recursos identifican a los grupos de atributos requeridos a través del nombre de estos últimos	*RASupportQuery*
9	Las restricciones entre grupos de atributos requeridos deben poseer al menos dos grupos bajo restricción	No tiene sentido evaluar las restricciones entre grupos de atributos requeridos si no se cuenta con al menos dos grupos	*RASupportQuery*
10	Las consultas de recursos deben poseer al menos un grupo de atributos requeridos, una única opción y un solo TTL	La fase de selección necesita saber cuáles son los requerimientos de una consulta	*RASupportQuery*
11	Los documentos XML de consulta de recursos no se pueden generar si no existe una consulta de recursos consistente	Los documentos XML de consulta de recursos deben ser consistentes	*RASupportQuery*

Tabla 5.3: Reglas para la creación de consultas de recursos en *RASupport*.

Regla	Descripción	Motivo	Clases en las que se aplica
12	Los atributos y restricciones que no tienen soporte en *RASupport* se omiten	*RASupport* mejora el rendimiento de la fase de selección omitiendo atributos y restricciones que nunca son utilizados	*XMLQueryReader*
13	Los documentos XML de consulta de recursos solo pueden especificar una consulta por documento	Los objetos de la clase *RASupportQuery* están asociados directamente con un documento XML de consulta de recursos	*XMLQueryReader*
14	Los atributos y restricciones que se especifiquen deben seguir la nomenclatura de *RASupport*	Los atributos y restricciones son evaluadas de acuerdo con la nomenclatura de *RASupport*. Para tipos numéricos se utiliza la siguiente nomenclatura: valor mínimo permitido, valor mínimo ideal, valor máximo ideal, valor máximo permitido y penalización. Para tipos *string* se utiliza únicamente el valor permitido y la penalización	*XMLQueryReader*
15	El valor mínimo permitido, el valor mínimo ideal, el valor máximo ideal, el valor máximo permitido y la penalización deben ser válidos	Los atributos y restricciones deben tener requerimientos válidos para poder ser evaluados correctamente	*XMLQueryReader*

Tabla 5.4: Reglas para la lectura de documentos XML de consultas de recursos en *RASupport*.

5.3. Módulo de simulación de recursos heterogéneos caracterizados por atributos estáticos y dinámicos: ResourceSim

El módulo *ResourceSim* es un gestor de recursos dentro de *RASupport*, por lo que su fachada implementa la interfaz *RASupportResourceManager*. *ResourceSim* permite la generación aleatoria de múltiples recursos caracterizados por atributos heterogéneos, estáticos y dinámicos, haciendo uso de la descripción de recursos (*resourcesmodel*) proporcionada por el módulo de configuración de *RASupport* (*RASupportConfig*). La generación de dichos atributos se lleva a cabo sin necesidad de contar con nodos físicos, lo cual aumenta la robustez de *RASupport* para efectuar simulaciones de recursos heterogéneos en grandes cantidades de nodos. El módulo *ResourceSim* está compuesto por tres submódulos: *rsimcommon*, *rsimmonitor* y *resourcesim*. El submódulo *rsimcommon* únicamente posee la clase *RSimCommon*, la cual se encarga de proporcionar la configuración común para

los demás submódulos de *ResourceSim*. Además, en la configuración proporcionada por *RSimCommon*, se especifican las reglas que determinan la relación que existe entre un conjunto de atributos numéricos (e.g., una regla para que el valor del espacio libre en disco siempre sea menor al valor del espacio total en disco). En la figura 5.6 se muestra el diagrama de clases del submódulo *resourcesim*, el cual contiene la fachada de *ResourceSim* y la clase que se encarga de generar atributos estáticos y dinámicos (*RSimAttributeGenerator*), empleando las reglas previamente mencionadas. En esa misma figura, se observa que la fachada *ResourceSim* regresa un objeto de tipo *RASupportMap* cuando se desea obtener atributos estáticos (a través del método *obtainStaticAttributes()*) o atributos dinámicos (empleando el método *obtainDynamicAttributes()*).

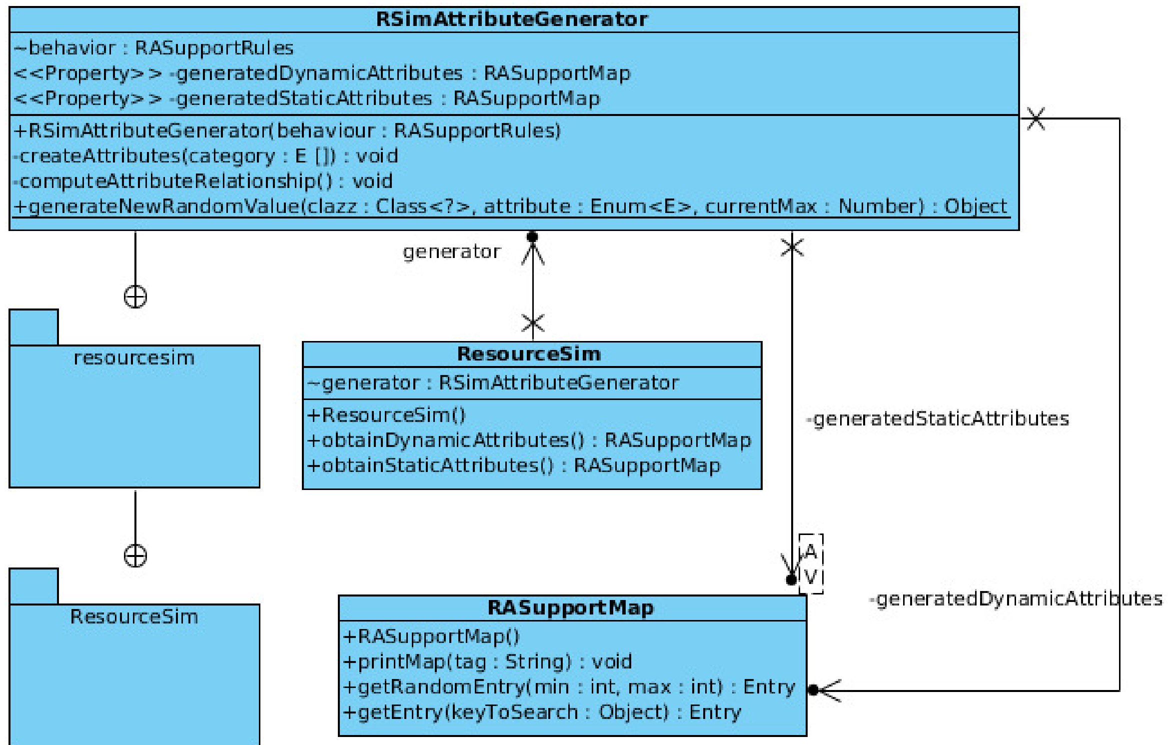


Figura 5.6: Diagrama de clases del submódulo *resourcesim*.

En la figura 5.7, se muestra el diagrama de clases del submódulo *rsimmonitor*, en donde se observa que este último es un monitor de recursos dentro de *RASupport*, por lo que ofrece una fachada que implementa la interfaz *RASupportResourceMonitor*. Además, *rsimmonitor* permite la generación aleatoria de actualizaciones de recursos dinámicos, las cuales se generan cada *ms* milisegundos, en donde el valor de *ms* se genera aleatoriamente en un intervalo definido por dos variables: retardo mínimo de actualización (*MIN_UPDATING*) y retardo máximo de actualización (*MAX_UPDATING*).

Con la finalidad de validar y verificar el correcto funcionamiento del monitor de recursos, el desarrollo del módulo *rsimmonitor* fue dirigido por pruebas automáticas basadas en un oráculo, el cual determina que las actualizaciones en atributos dinámicos cumplan con las reglas especificadas en la clase *RSimCommon*. En el algoritmo 6, se observa que el oráculo recibe como entrada la lista de atributos estáticos (*staticAttributes*) y la lista de atributos dinámicos (*dynamicAttributes*) que posee un nodo en particular, así como el atributo (*attribute*) seleccionado aleatoriamente y el nuevo valor (*value*) que se pretende establecer en dicho atributo. Cada elemento de la tabla de relaciones entre atributos numéricos (*attributesRelation*) constituye una lista ordenada (de menor a mayor) con respecto a los valores de los atributos a relacionar. Para cada lista, el oráculo obtiene el atributo (*next*) que se encuentra inmediatamente después de *attribute*. Posteriormente, se determina si *next* es un atributo estático o dinámico. Finalmente, el oráculo comprueba que *value* no exceda el valor de *next*. En caso de que suscite algún error, el oráculo lo despliega en pantalla y termina la ejecución del programa.

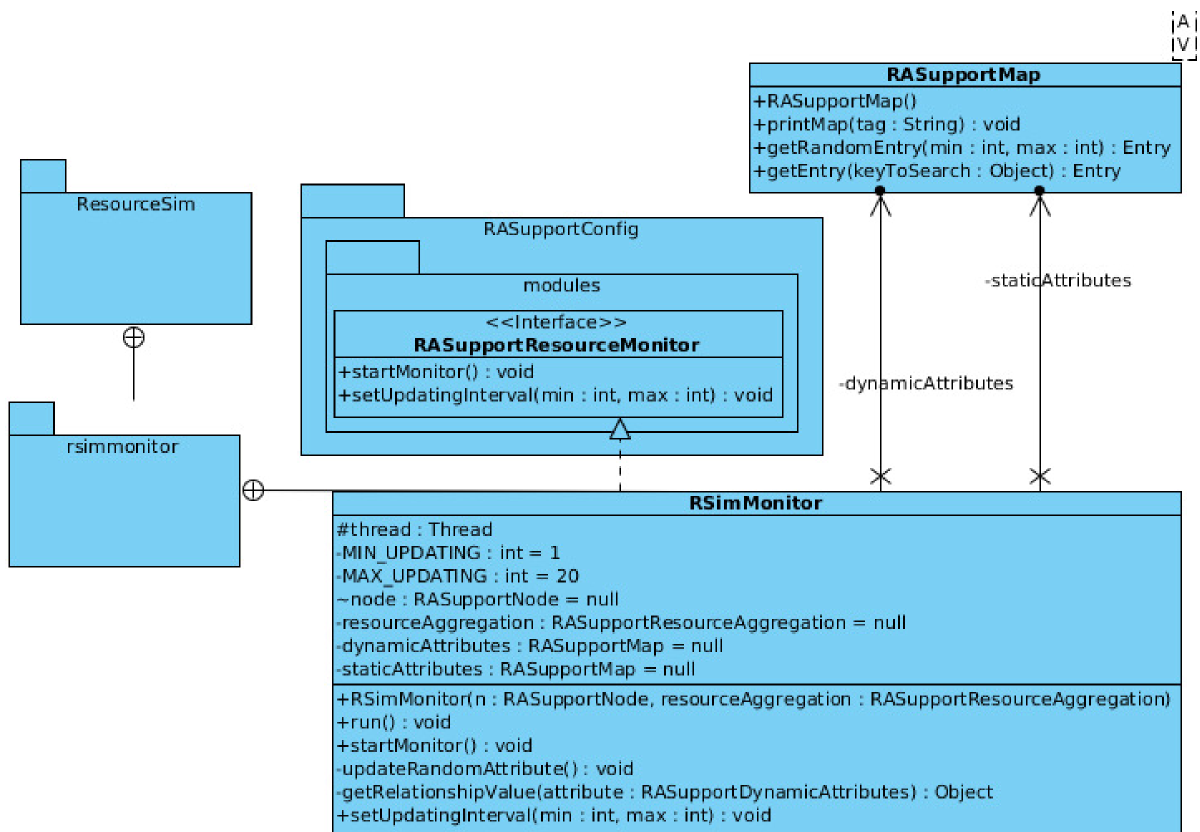


Figura 5.7: Diagrama de clases del submódulo *rsimmonitor*.

Algoritmo 6 Algoritmo del oráculo que valida el funcionamiento de *rsimmonitor*

```
1: procedure getSimilarity(attribute, value, staticAttributes, dynamicAttributes)
2:     for relation r: attributesRelation do
3:         index ← r.indexOf(attribute)
4:         if index exists and !r.isLast(index) then
5:             next ← r.get(index + 1)
6:             nextValue ← 0
7:             if staticAttributes.contains(next) then
8:                 nextValue ← staticAttributes.get(next)
9:             else if dynamicAttributes.contains(next) then
10:                nextValue ← dynamicAttributes.get(next)
11:            else
12:                showError(next is not a static or a dynamic attribute)
13:                exit(0)
14:            end if
15:        end if
16:        if value > nextValue then
17:            showError(value is greater than nextValue)
18:            exit(0)
19:        end if
20:    end for
21: end procedure
```

5.4. *Toolkit* para la agregación de recursos en sistemas P2P colaborativos: *RAToolkit*

En esta sección, se describen los principales componentes que integran a *RAToolkit*, el cual es un módulo de agregación de recursos dentro de *RASupport*, por lo que ofrece una fachada que implementa la interfaz *RASupportResourceAggregation*. *RAToolkit* está basado en los mecanismos descritos en el capítulo 4; por lo tanto, proporciona cuatro APIs que permiten llevar a cabo las fases de anunciamiento, selección, empatamiento y enlace, respectivamente. Además, dicho *toolkit* está compuesto por ocho submódulos: *AdvertisementAPI*, *ApisManagement*, *BindingAPI*, *Common*, *DatabasesManagement*, *MatchingAPI*, *SelectionAPI* y *TransportLayer*. Los módulos *AdvertisementAPI*, *SelectionAPI*, *MatchingAPI* y *BindingAPI* son las APIs que permiten llevar a cabo la agregación de recursos.

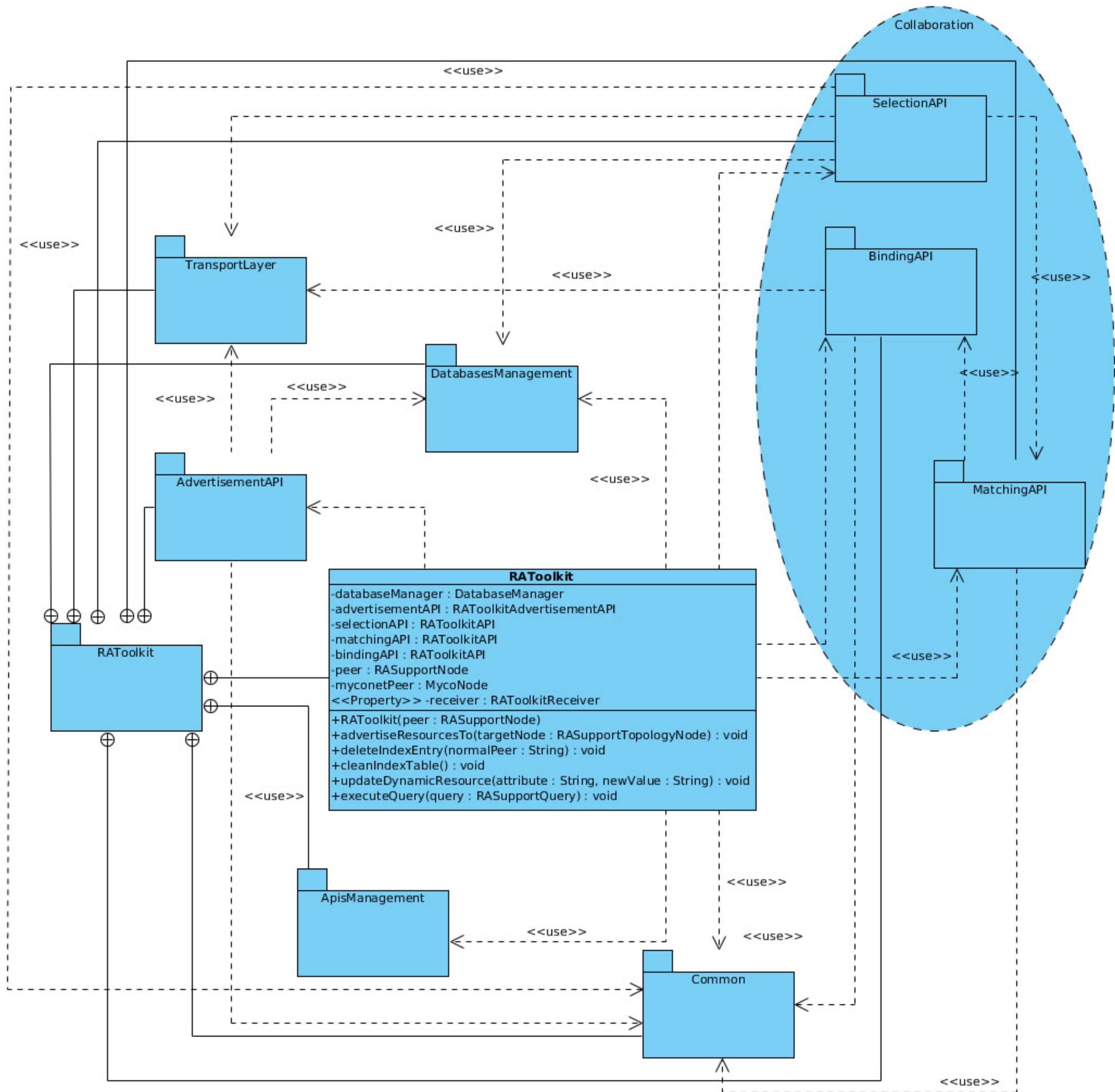


Figura 5.8: Diagrama de paquetes de *RAToolkit*.

En la figura 5.8, se muestra el diagrama de paquetes de *RAToolkit*, en donde se observa que los módulos *SelectionAPI*, *MatchingAPI* y *BindingAPI* colaboran entre sí; en esa misma figura, se observa que la fachada de *RAToolkit* ofrece cinco métodos públicos, los cuales se describen a continuación:

- *advertiseResourcesTo()*. Permite anunciar recursos explícitamente desde un *peer normal* a algún *super-peer* (*targetNode*).
- *deleteIndexEntry()*. Permite eliminar un registro (*normalPeer*) de la tabla de índices.
- *cleanIndexTable()*. Permite suprimir todos los registros de la tabla de índices.
- *updateDynamicResources()*. Permite actualizar un recurso dinámico de algún *peer* que se encuentre en la tabla de índices, proporcionando el atributo (*attribute*) que se desea actualizar, así como su nuevo valor (*newValue*).
- *executeQuery()*. Permite llevar a cabo las fases clave de la agregación de recursos a partir de una instancia (*query*) que representa una consulta de recursos.

Debido a que los sistemas P2P colaborativos únicamente necesitan obtener resultados a partir de una consulta de recursos, dichos sistemas solo deben emplear el método *executeQuery()* proporcionado por *RASupportMain*, el cual se encarga de ocultar toda la complejidad que se encuentra detrás de las fases clave de la agregación de recursos (ver figura 5.9).

En la figura 5.10, se muestra el diagrama de secuencia de *RASupport*, en donde un sistema P2P colaborativo solicita un conjunto de recursos mediante el método *executeQuery()* del módulo *RASupportMain*, especificando el objeto *query* que representa una consulta de recursos. *RASupportMain* inicia el proceso llamando al método *executeQuery()* de *RAToolkit*, el cual lleva a cabo las fases clave de la agregación de recursos, manteniendo cohesión entre estas. En primer lugar, *RAToolkit* llama al método *selectResources()* proporcionado por *SelectionAPI* para seleccionar un conjunto de recursos, de acuerdo a los requerimientos y a las restricciones especificados en el objeto *query*. Posteriormente, *SelectionAPI* llama al método *matchResources()* proporcionado por *MatchingAPI*, con la finalidad de obtener un conjunto de grupos óptimos de atributos requeridos. Finalmente, *MatchingAPI* llama al método *bindResources()* de *BindingAPI* para enlazar dichos grupos. Esta última API regresa (al *peer* que solicitó la consulta) un conjunto de resultados (*ResultSet*) que satisfacen los requerimientos y las restricciones especificadas en el objeto *query*.

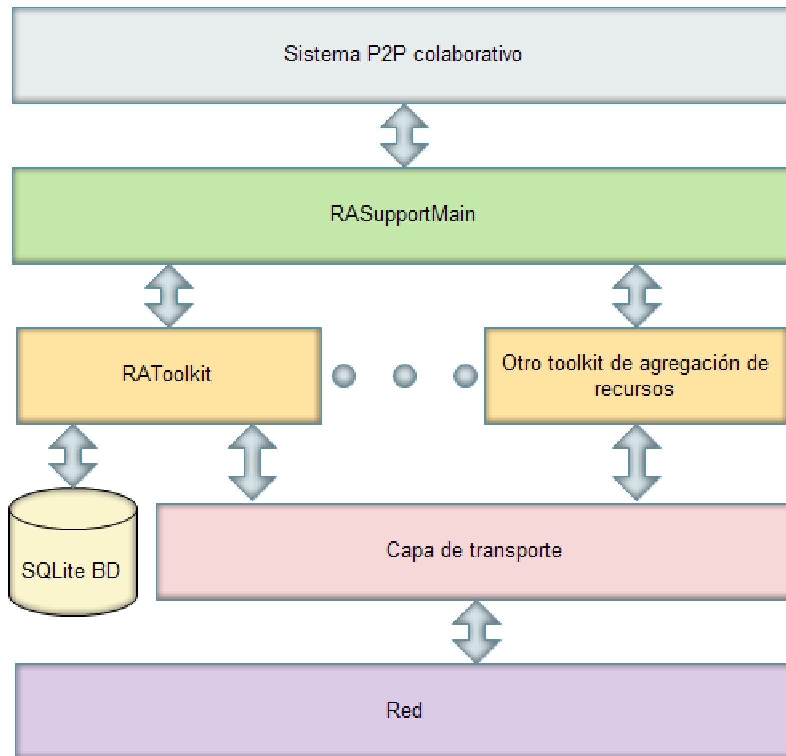


Figura 5.9: Capas de *RASupport*.

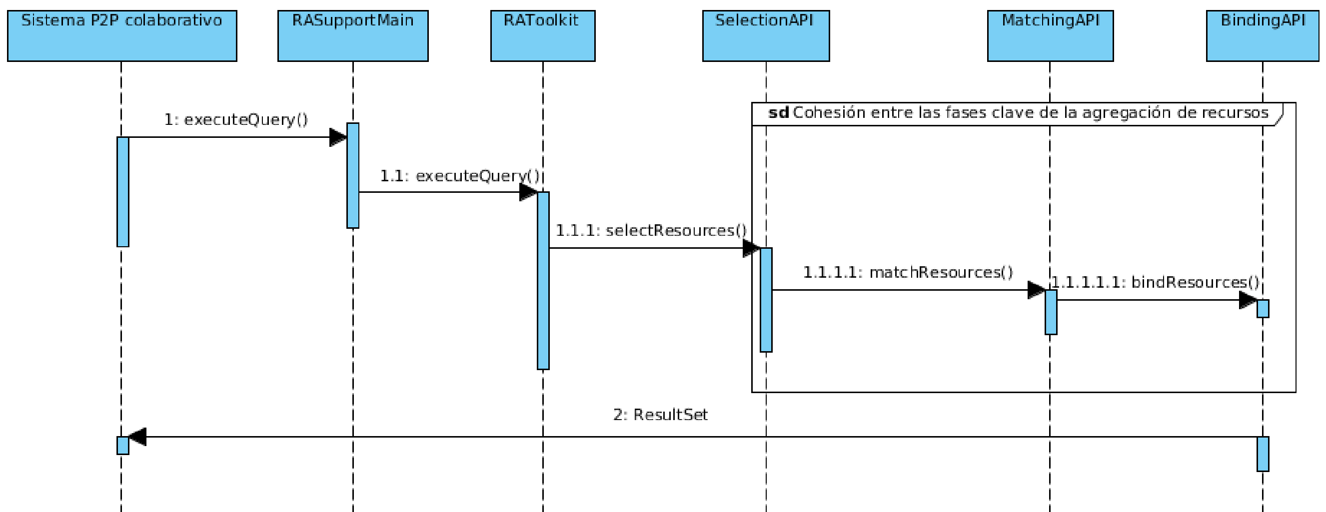


Figura 5.10: Diagrama de secuencia de *RASupport*.

El submódulo *ApisManagement* permite crear nuevas APIs de agregación de recursos, empleando el patrón de diseño *Fábrica* [14] (ver figura 5.11). Actualmente, *RAToolkit* posee cuatro APIs (*AdvertisementAPI*, *SelectionAPI*, *MatchingAPI* y *BindingAPI*), las cuales se pueden substituir fácilmente (e.g., es posible cambiar el API de la fase de empata-

miento por otra que exhiba una menor complejidad). Las APIs previamente mencionadas ofrecen fachadas para ocultar la complejidad del proceso interno que llevan a cabo; además, implementan alguna de las siguientes interfaces dependiendo de la fase de la agregación de recursos que deseen proporcionar: *RAToolkitAdvertisementAPI*, *RAToolkitSelectionAPI*, *RAToolkitMatchingAPI* o *RAToolkitBindingAPI*.

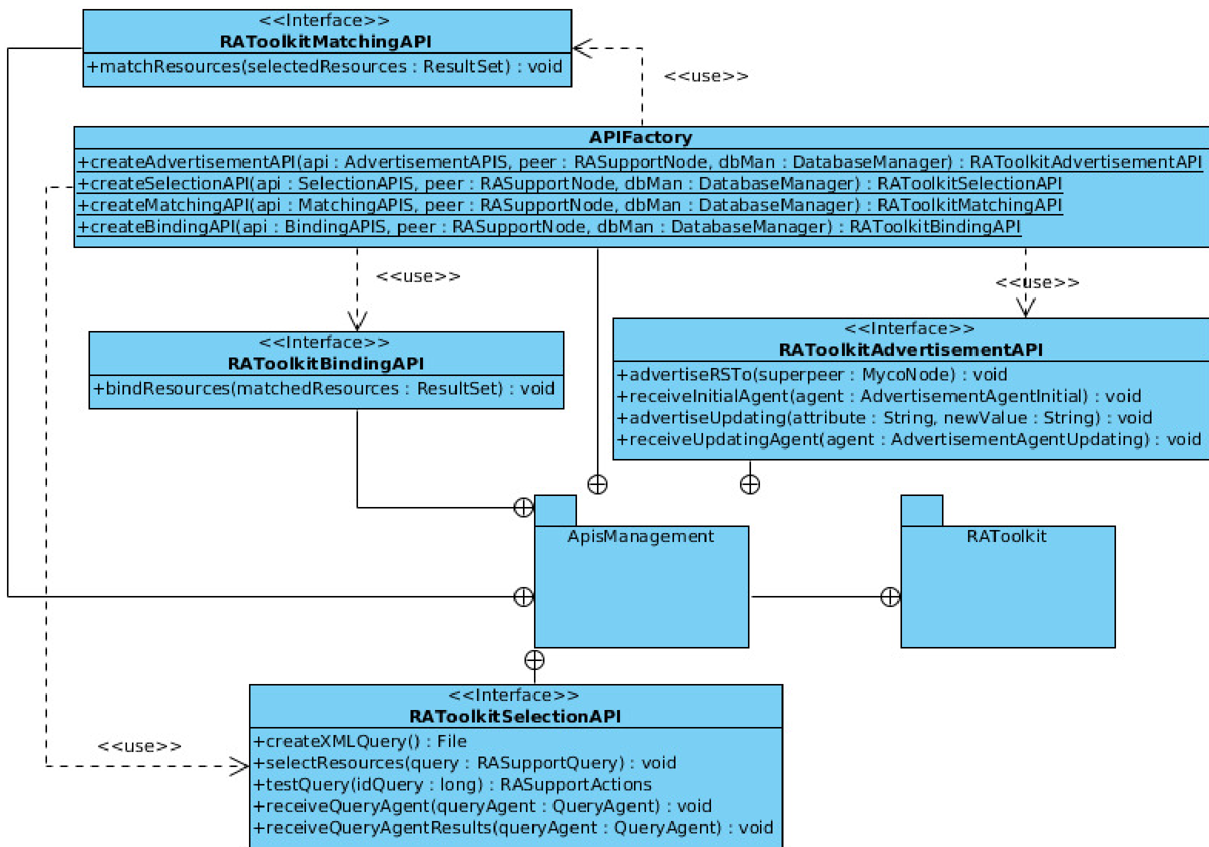


Figura 5.11: Diagrama de clases del submódulo *ApisManagement*.

El submódulo *Common* posee seis componentes: *Agent*, *AgentsFactory*, *Common*, *ErrorsManager*, *RAToolkitConfigParser* y *ResultSet*, los cuales son comúnes a todos los submódulos de *RAToolkit*. En particular, *Agent* es una interfaz que deben implementar todos los agentes que existen y llegasen a existir dentro de *RAToolkit*; además, dicha clase hereda de la interfaz *RASupportNetworkTraveler* proporcionada por *RASupportConfig*. Los agentes que interactúan en *RAToolkit* son creados a traves de la clase *AgentsFactory* que emplea el patrón de diseño *Fábrica* [14]. La interfaz *Common* establece la configuración tanto para los agentes como para las APIs que existen y llegasen a existir en *RASupport*. La clase *ErrorsManagement* define las variables que representan errores dentro de *RAToolkit* (e.g., *UNKNOWN_AGENT* hace referencia a que se especificó un agente desconocido en la clase *AgentsFactory*). *RAToolkitConfigParser* extrae la configuración especificada para *RAToolkit* en el archivo de configuración de *RASupport*, cuyos parámetros poseen

el prefijo “ratoolkit”. Finalmente, la clase *ResultSet* es la representación de un conjunto de resultados obtenidos para una consulta en particular. En la figura 5.12 se muestra el diagrama de clases del submódulo *Common*.

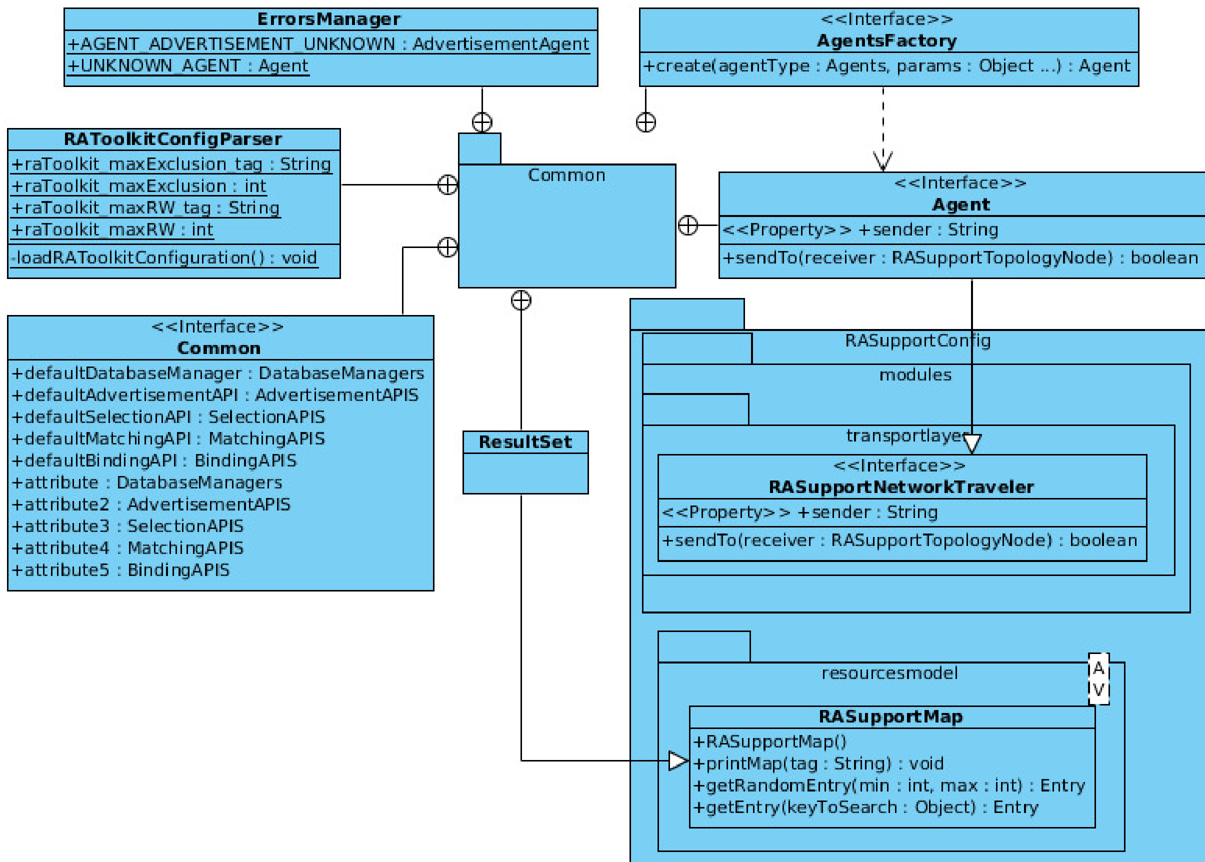


Figura 5.12: Diagrama de clases del submódulo *Common*.

El submódulo *DatabasesManagement* es aquel que se encarga de gestionar la base de datos que existe en *RAToolkit*. Para ello, dicho submódulo proporciona tres componentes: *DatabaseErrors*, *DatabaseManager* y *DatabaseManagerFactory*. El primero de ellos es una interfaz que proporciona las variables que definen errores que se pueden presentar al momento de llevar a cabo una operación sobre la base de datos. Actualmente, *RAToolkit* únicamente posee soporte para el gestor de bases de datos SQLite, sin embargo es posible cambiar fácilmente dicho gestor debido a que *DatabaseManagement* proporciona la clase *DatabaseManagerFactory* que utiliza el patrón de diseño *Fábrica* [14] para crear instancias de gestores de bases de datos, los cuales deben implementar los métodos proporcionados por la interfaz *DatabaseManager*. En la figura 5.13, se muestra el diagrama de clases del submódulo *DatabasesManagement*, en donde se observa que un gestor de bases de datos emplea el patrón de diseño *Singleton* [14], con la finalidad de mantener una única instancia de la clase *DatabaseManager*.

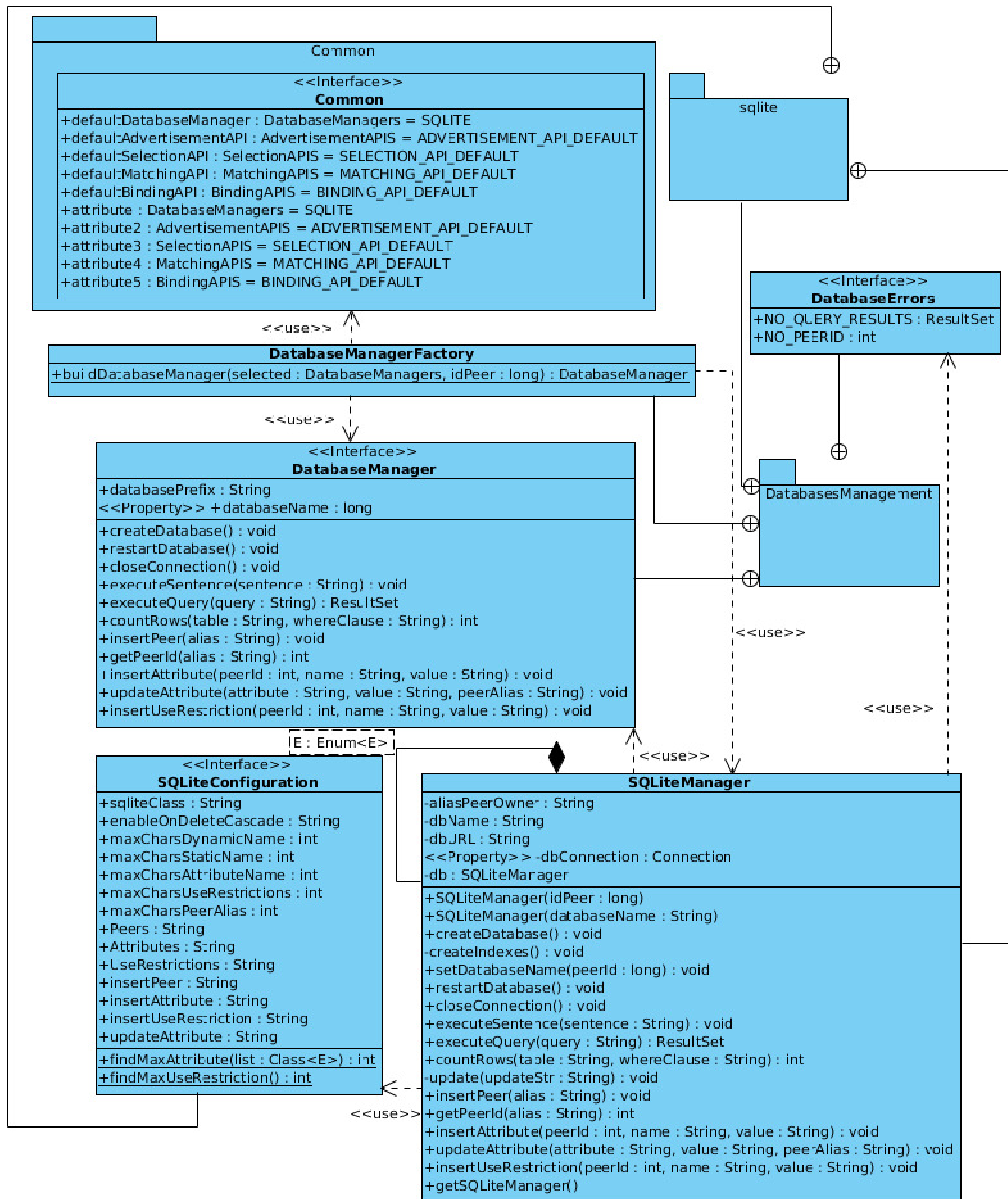


Figura 5.13: Diagrama de clases del submódulo *DatabasesManagement*.

Los protocolos P2P de agregación de recursos que posee *RAToolkit* necesitan una capa de transporte para comunicarse a través de la red. *RAToolkit* ofrece un submódulo denominado *TransportLayer*, el cual contiene la enumeración *RAToolkitMessages* (que hereda de la interfaz *RASupportActions* proporcionada por *RASupportConfig*) en donde se especifican los mensajes que se pueden recibir a través de la red, la clase *RAToolki-*

tReceiver encargada de atender peticiones de otros *peers* y la clase *RAToolkitSender* que ofrece los métodos para enviar mensajes y agentes a través de la red. En la figura 5.14, se muestra el diagrama de clases de *TransportLayer*, en donde se observa que las clases *RAToolkitReceiver* y *RAToolkitSender* implementan las interfaces *RASupportReceiver* y *RASupportSender*, respectivamente, las cuales son proporcionadas por *RASupportConfig*.

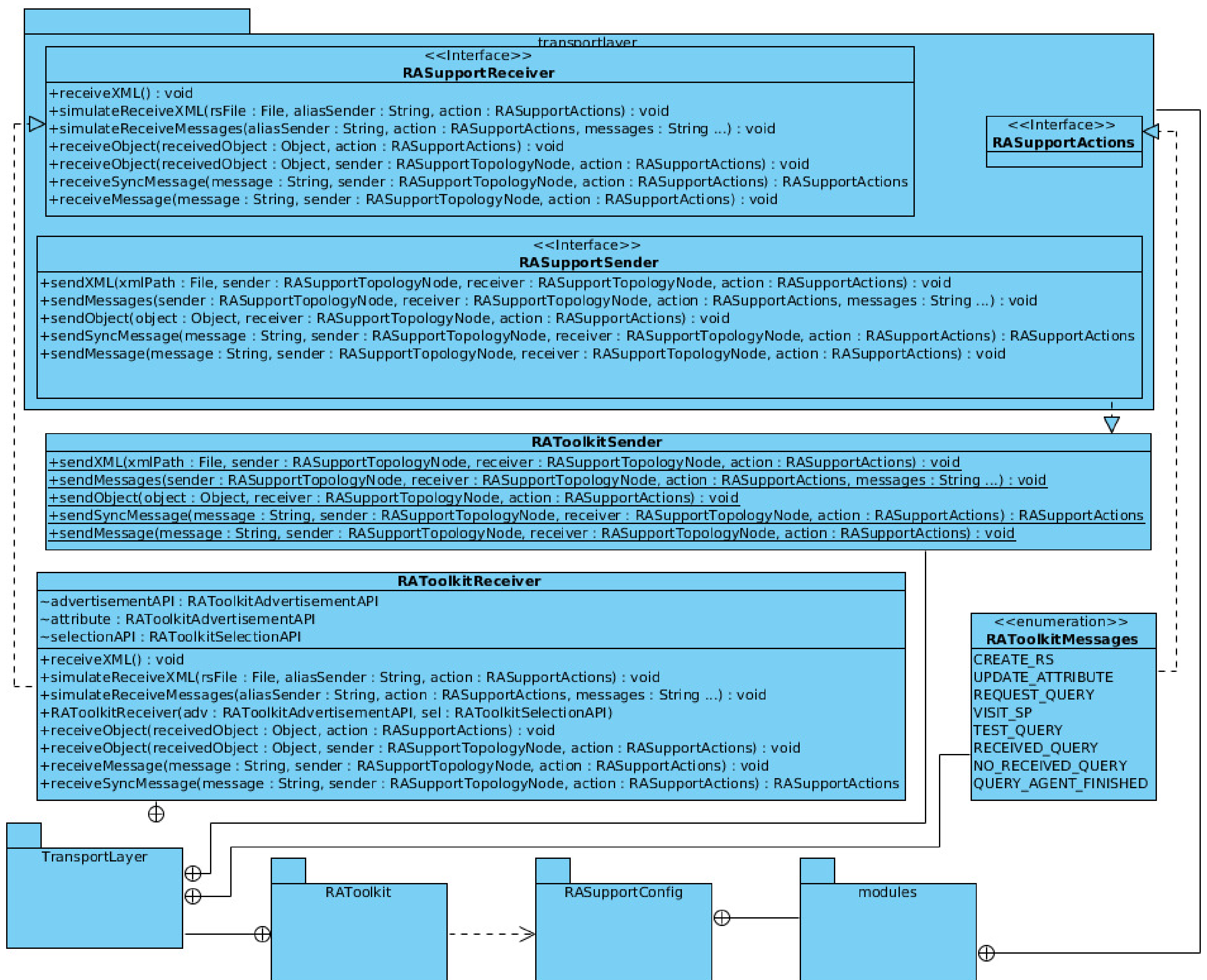


Figura 5.14: Diagrama de clases del submódulo *TransportLayer*.

RAToolkit se desarrolló de forma modular, por lo que fácilmente se pueden agregar, modificar y eliminar módulos sin necesidad de afectar otras partes del diseño. Debido a lo anterior, *RAToolkit* ofrece alta cohesión y bajo acoplamiento. A continuación, se describe el diseño e implementación de la base de datos SQLite, el soporte para el protocolo Myconet y la implementación de las APIs *AdvertisementAPI*, *SelectionAPI*, *MatchingAPI* y

BindingAPI.

5.4.1. Implementación y diseño de la base de datos SQLite

Como se mencionó en el capítulo 4, es necesario que los *super-peers* indexen las especificaciones de recursos (RS, por sus siglas en inglés) de los *peers normales* que tienen a su cargo. Una estructura de datos (e.g., una tabla *hash*) no proporciona la robustez necesaria para mantener grandes volúmenes de datos ni para llevar a cabo consultas complejas de recursos (e.g., seleccionar *peers* con memoria RAM entre 4GB y 8GB). Para solucionar lo anterior, *RAToolkit* aprovecha las ventajas que ofrece el lenguaje SQL (*Structured Query Language*), empleando una base de datos gestionada por SQLite 3, debido a que esta:

- Proporciona acceso concurrente a los datos.
- Funciona en múltiples plataformas, por lo que es fácil de portar a otros sistemas.
- Proporciona un rápido acceso a los datos (SQLite es el gestor de bases de datos más rápido que existe actualmente)[2].
- Permite la generación de consultas en lenguaje SQL.
- Genera bases de datos en archivos de tamaño pequeño.
- Lleva a cabo transacciones atómicas, consistentes, aisladas y durables, incluso después de que el sistema falló.
- No requiere configuración previa.
- Proporciona una API para Java sencilla de usar.
- Es auto-contenido, i.e., no requiere dependencias externas.
- Permite el almacenamiento en memoria o en disco.
- Posee código fuente de dominio público.
- Puede ser usado para cualquier propósito (incluso comercial).

En la sección 5.4, se mencionó que *RAToolkit* posee un módulo (*DatabasesManagement*) de administración de bases de datos, de manera que los gestores (representados por la clase *DatabaseManager*) se pueden substituir fácilmente, sin tener que modificar el código en donde se efectúan operaciones sobre la base de datos.

La base de datos de *RAToolkit* es utilizada por los *super-peers* para almacenar referencias a sus *peers normales*, así como los atributos y las restricciones de uso (i.e., los índices

[2]https://www.sqlite.org/speed.html.

de recursos) que poseen dichos *peers normales*. En la figura 5.15, se muestra el diagrama relacional de la base de datos, en donde se observa que esta posee tres tablas: *Peers*, *Attributes* y *UseRestrictions*. La primera tabla mantiene una referencia a los *peers normales* que el *super-peer* tiene a su cargo; mientras que las otras dos almacenan los atributos y las restricciones de uso que poseen tales *peers normales*, respectivamente. Los valores de los atributos y de las restricciones de uso se almacenan en campos de tipo *char*, debido a que es impráctico tener diferentes campos para cada tipo de atributo. En particular, cuando se desea obtener y comparar el valor numérico de un atributo o una restricción de uso, una consulta debe emplear la función *CAST* para efectuar la conversión de acuerdo al tipo (flotante o entero), e.g., *select * from Attributes where nameAttribute='busy_cpu' and CAST(valueAttribute AS INTEGER) between 0 and 100*.

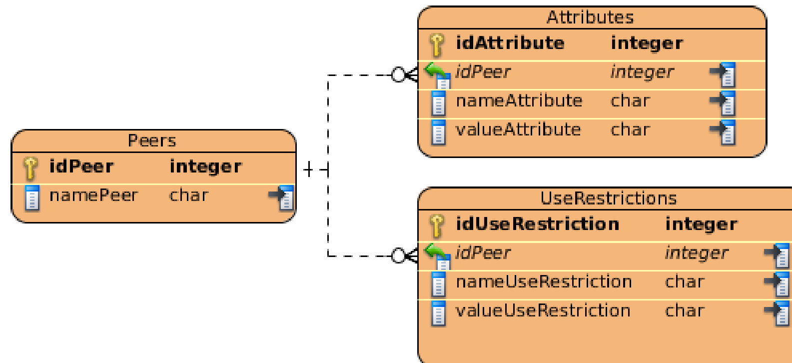


Figura 5.15: Diagrama relacional de la base de datos de *RAToolkit*.

Con la finalidad de mejorar la eficiencia de las consultas de recursos, se utilizan índices en todas las tablas[3]. En la figura 5.15, se observa que todas las tablas mantienen índices en todas sus columnas. Por lo tanto, la base de datos de *RAToolkit* utiliza índices de cobertura[4] para consultas con cláusulas *WHERE*. La complejidad de las consultas para I registros en la tabla índice es de $O(log_2(I))$. Para consultas que contienen cláusulas *OR*, el motor de consultas de la base de datos emplea la técnica *OR mediante unión*.

Por otro lado, la base de datos de *RAToolkit* mantiene una integridad referencial en cascada (*on delete on update cascade*), lo cual garantiza que la información almacenada siempre sea confiable. La restricción en cascada asegura que cuando se elimina o actualiza un registro de una tabla padre (e.g., un registro de la tabla *Peers*), todos los registros de las tablas hijo relacionadas (e.g., registros de la tabla *Attributes*) también se eliminarán o actualizarán.

[3]SQLite emplea búsquedas binarias cuando se utilizan índices en las tablas. Si el número de registros de salida (i.e., aquellos registros que se encuentran en la tabla índice) es K y el número de registros en la tabla de búsqueda es N, entonces el costo general de un consulta es $O(logN(K+1))$.

[4]Un índice de cobertura es aquel que cubre todas las columnas de una tabla, incluyendo la llave primaria.

La base de datos empleada por *RAToolkit* es útil en las fases de anunciamiento y selección. En particular, en la fase de anunciamiento los *super-peers* efectúan operaciones de inserción, mientras que en la fase de selección ejecutan consultas SQL.

5.4.2. Soporte Myconet

A diferencia de otros protocolos de gestión de *super-peers*, Myconet mantiene el código fuente de sus simulaciones[5] en un repositorio de GitHub, el cual se encuentra disponible en la dirección https://github.com/pataprogramming/myconet-peersim. Para llevar a cabo las simulaciones en la presente tesis, se reutilizó dicho código y se modificó con la finalidad de integrarlo en *RASupport*.

El soporte Myconet está compuesto por 124 clases, la cuales interactúan entre sí para organizar y gestionar una red P2P no estructurada basada en *super-peers*. En particular, los *peers* se representan mediante la clase *MycoNode*. En la figura 5.16, se observa que dicha clase implementa la interfaz *RASupportTopologyNode* proporcionada por *RASupportConfig*, debido a que esta contiene los métodos que deben implementar las clases que representan nodos dentro de una topología[6]. En esa misma figura, se observa que *RASupportMain* no depende de la clase *MycoNode* sino de la interfaz *RASupportTopologyNode*, por lo que *RASupportMain* se encuentra desacoplado del soporte Myconet.

El soporte Myconet permite la visualización de los grafos (i.e., las redes P2P) que se generan en una ejecución, lo cual constituye una ventaja fundamental al momento de depurar y analizar el comportamiento y el estado de una red P2P (ver figura 5.17). En dicho soporte, los *peers* mantienen un identificador numérico y se encuentran representados de la siguiente manera:

- *Peers normales*: círculo azul.
- *Super-peers* extendiendo: triángulo rojo.
- *Super-peers* ramificando: rombo amarillo.
- *Super-peers* inmóviles: pentágono verde.
- *Super-peers* muertos: círculo verde.

[5] Los creadores de Myconet desarrollaron el código del simulador (i.e., el soporte Myconet) en lenguaje Java.

[6] En la presente tesis, la topología es una red P2P no estructurada basada en *super-peers*, la cual puede variar dependiendo del módulo de agregación de recursos que se esté usando (e.g., es posible añadir un módulo que utilice una topología centralizada).

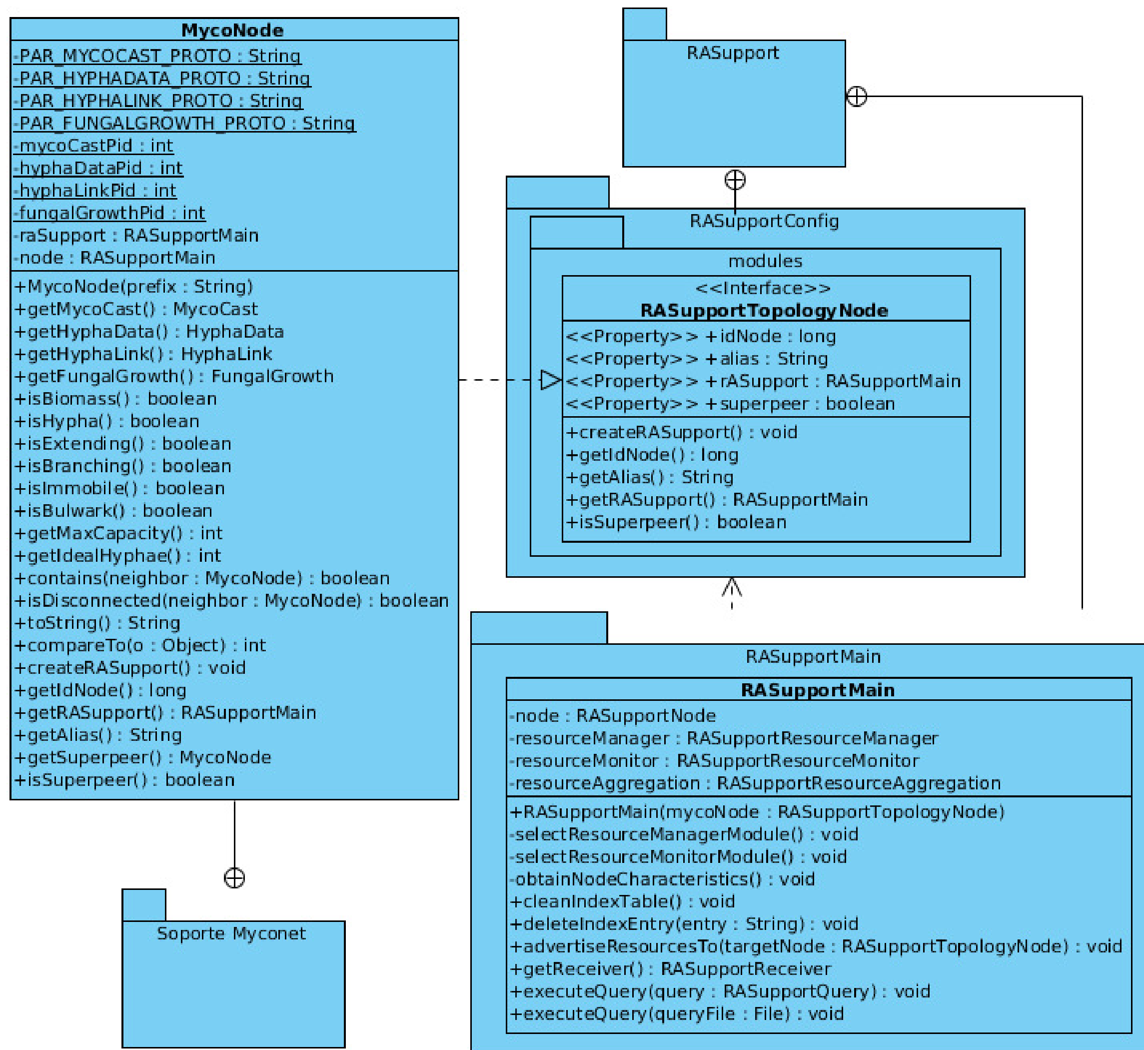


Figura 5.16: Diagrama de clases en donde se muestra que *RASupport* se encuentra desacoplado del soporte Myconet.

Los sistemas P2P colaborativos únicamente necesitan obtener resultados a partir de una consulta de recursos, sin necesidad de enterarse cómo está estructurada la topología de la red. Estos sistemas utilizan la fachada *RASupportMain* para acceder a la funcionalidad que ofrece *RASupport*, el cual oculta la manera en que se gestiona dicha topología. *RAToolkit* emplea el soporte Myconet para obtener información (e.g., para conocer cuáles son los *peers normales* de un determinado *super-peer*) y para gestionar la red P2P. En la figura 5.18, se muestra la relación que existe entre un sistema P2P colaborativo, *RASupport*, *RAToolkit* y el soporte Myconet.

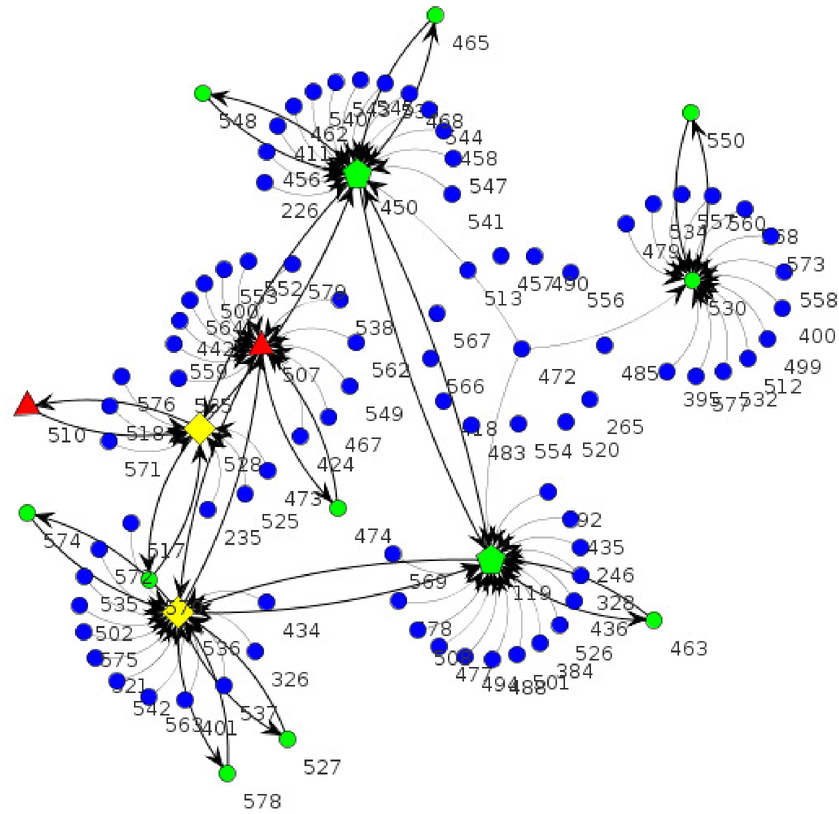


Figura 5.17: Ejemplo de una red P2P generada con el soporte Myconet.

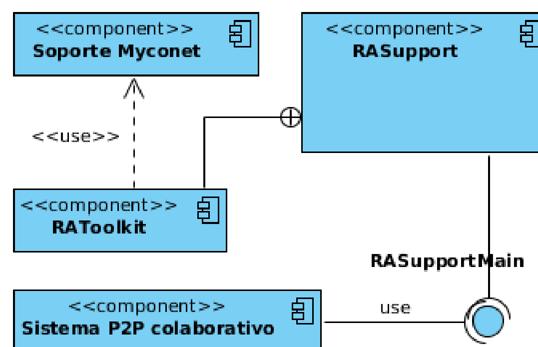


Figura 5.18: Diagrama de componentes que muestra la relación entre un sistema P2P colaborativo, *RASupport*, *RAToolkit* y el soporte Myconet.

5.4.3. API de anunciamiento de recursos: AdvertisementAPI

AdvertisementAPI es una API de *RAToolkit* que permite llevar a cabo la fase de anunciamiento de recursos, por lo que ofrece una fachada que implementa la interfaz *RAToolkitAdvertisementAPI*. Dicha API tiene el objetivo de anunciar y de actualizar especificaciones de recursos empleando agentes de anunciamiento inicial y agentes de anunciamiento de actualización, respectivamente. En la figura 5.19, se observa que *AdvertisementAPI* está compuesta por los módulos *agents* y *rs*. El primero contiene las clases que representan a los agentes de anunciamiento (*AdvertisementAgentInitial* y *AdvertisementAgentUpdating*), las clases que se encargan de gestionar a estos agentes (*InitialManager* y *UpdatingManager*) y la clase que utiliza el patrón de diseño *Fábrica* [14] (*AdvertisementAgentsFactory*) para crear dichos agentes (ver figura 5.20). El segundo módulo únicamente contiene la clase *RSpec*, la cual representa una especificación de recursos compartida entre las clases *InitialManager* y *UpdatingManager* (ver figura 5.21).

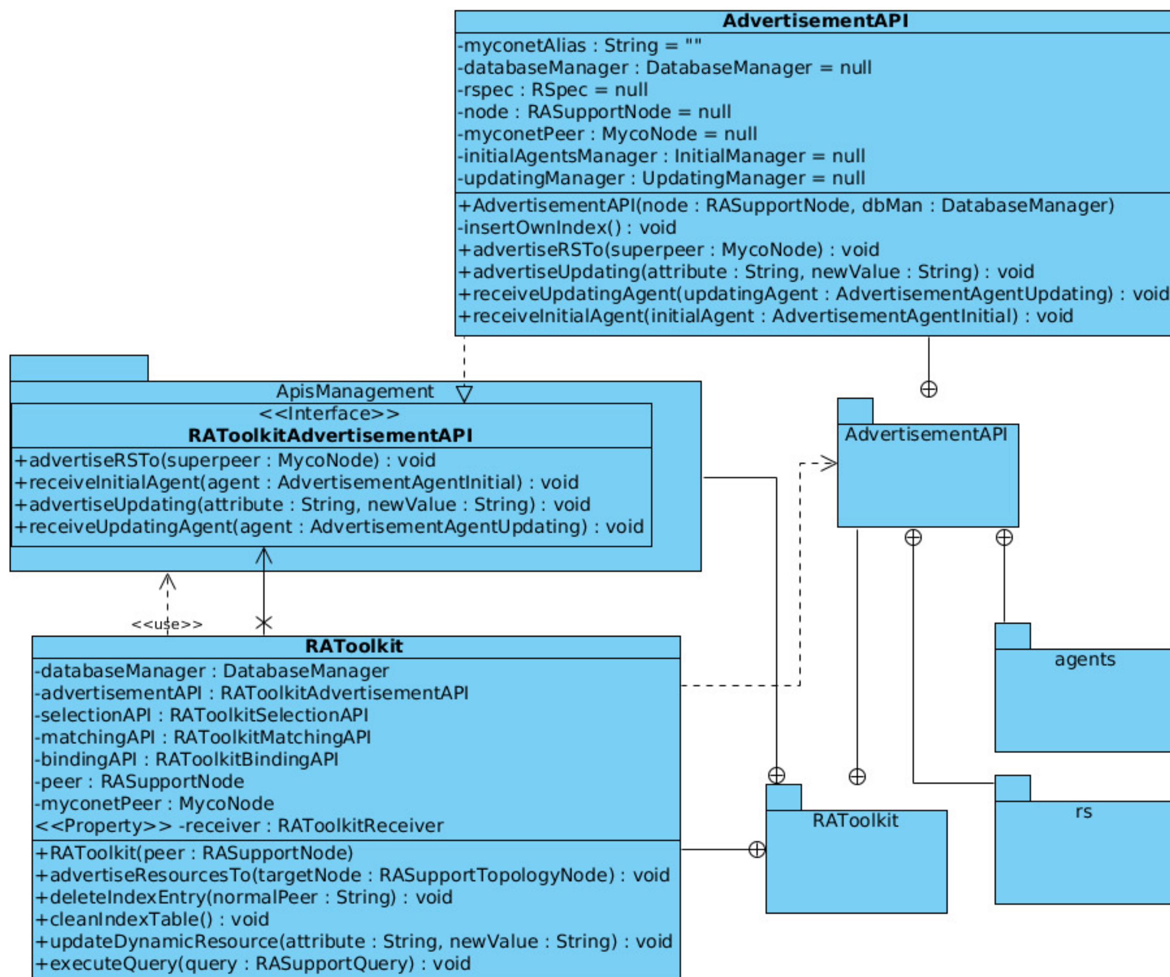


Figura 5.19: Diagrama de paquetes de *AdvertisementAPI*.

Figura 5.20: Diagrama de clases del módulo *agents*.

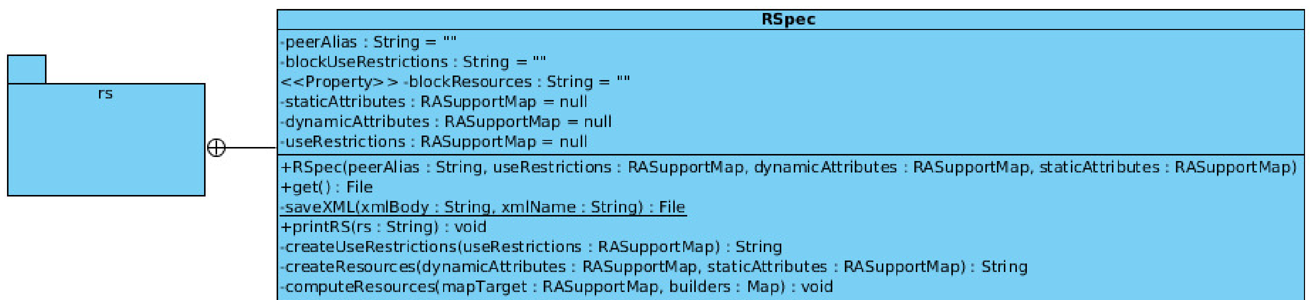


Figura 5.21: Diagrama de clases del módulo *rs*.

AdvertisementAPI se inicializa con dos instancias: una de la clase *RASupportNode* y otra de la clase *DatabaseManager*. La primera sirve para obtener información sobre el *peer* que se está gestionando, mientras que la segunda permite insertar *peers normales* (junto sus respectivos atributos y sus restricciones de uso) en la base de datos. El constructor de *AdvertisementAPI* inicializa una especificación de recursos (i.e., una instancia de la clase *RSpec*) para crear un gestor de agentes de anunciamiento inicial (i.e., una instancia de la clase *InitialManager*) y un gestor de agentes de anunciamiento de actualización (i.e., una instancia de la clase *UpdatingManager*).

En la figura 5.19, se observa que la fachada de *AdvertisementAPI* ofrece los métodos *advertiseRSTo()*, *receiveInitialAgent()*, *advertiseUpdating()* y *receiveUpdatingAgent()*. El método *advertiseRSTo()* permite anunciar recursos de un *peer normal* a su respectivo *supeer-peer* empleando un agente de anunciamiento inicial, mientras que el método *receiveInitialAgent()* sirve para recibir a dicho agente, el cual inserta una especificación de recursos en la base de datos del *super-peer* que visita. Por otro lado, el método *advertiseUpdating()* sirve para enviar un agente de anunciamiento de actualización, mientras que el método *receiveUpdatingAgent()* sirve para recibirlo en un *super-peer*.

Los agentes de anunciamiento son creados por un *peer normal* y su tiempo de vida es directamente proporcional al tiempo que tardan en llevar a cabo su cometido. El recolector de basura de la máquina virtual de Java es quien se encarga de liberar la memoria ocupada por estos agentes. En particular, los agentes de anunciamiento inicial transportan documentos XML de especificación de recursos y son envíados cuando un *peer normal* se une a un *super-peer*, mientras que los agentes de anunciamiento de actualización tienen el objetivo de anunciar a un *super-peer* cambios que suscitan en los recursos dinámicos de un *peer normal*. En la figura 5.22, se muestran los estados por los cuales transitan los agentes de anunciamiento durante su tiempo de vida.

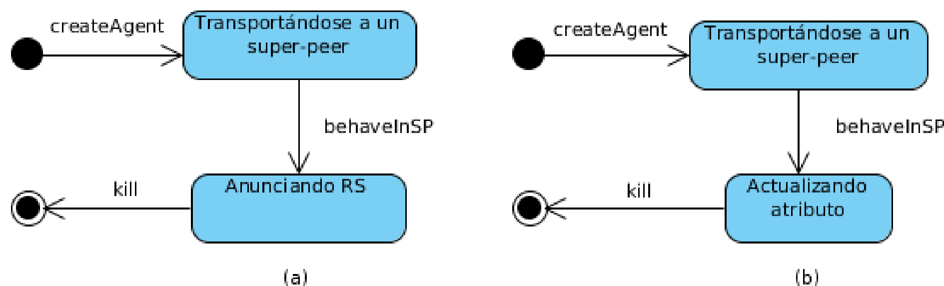


Figura 5.22: Diagrama de estados de los agentes de anunciamiento: (a) agente de anunciamiento inicial y (b) agente de anunciamiento de actualización.

Clase	**Método**	**Motivo**
BasicBiomassStrategy	*doDynamics()*	1. Cuando un *peer normal* $NP1$ se une por primera vez a algún *super-peer* $SP1$, $NP1$ anuncia sus recursos a $SP1$ 2. Cuando un *peer normal* $NP1$ posee conexiones a dos *super-peers*, alguno de los últimos borra a $NP1$ de su base de datos
HyphaLink	*transferNeighbor()*	Cuando un *super-peer* $SP1$ roba un *peer normal* $NP1$ de un *super-peer* $SP2$: 1. $NP1$ anuncia sus recursos a $SP1$ 2. $SP1$ borra a $NP1$ de su base de datos
HyphaLink	*transferNeighbor()*	Cuando un *super-peer* $SP1$ absorbe a un *super-peer* $SP2$, los *peers normales* de $SP2$ anuncian sus recursos a $SP1$
HyphaLink	*swapHyphae()*	Cuando un *super-peer* $SP1$ se transforma en un *peer normal*, $SP1$ anuncia sus recursos al *super-peer* al cual se une
BasicExtendingStrategy	*doDynamics()*	Cuando un *peer normal* $NP1$ aún no está conectado a un *super-peer* estable, busca alguno que se encuentre en ese estado y le anuncia sus recursos
HyphaData	*becomeBiomass()*	Cuando un *peer normal* $NP1$ absorbe a su *super-peer* $SP1$, este último anuncia sus recursos a $NP1$
HyphaData	*become()*	1. Cuando un *super-peer* se convierte en un *peer normal* borra el contenido de las tablas de su base de datos 2. Cuando un *super-peer* $SP1$ detecta que uno de sus *peers normales* ya no está disponible, $SP1$ lo borra de su base de datos
HyphaLink	*transferNeighbor()*	Cuando un *peer normal* absorbe a su *super-peer*, este último anuncia sus recursos al primero
HyphaLink	*growHypha()*	Cuando un *peer normal* $NP1$ es promovido a *super-peer* extendiendo, el *super-peer* a cargo lo borra de su base de datos

Tabla 5.5: Modificaciones realizadas en el código del soporte Myconet para implementar la fase de anunciamiento de recursos.

En la figura 5.23, se muestra la comunicación que existe en *RAToolkit* para llevar a cabo el anunciamiento inicial de recursos, en donde se observa que el soporte Myconet (*Myconet*) es quien inicia el proceso cuando se presenta alguno de los casos descritos en el capítulo 4. Así mismo, en la figura 5.24 se ilustra la comunicación entre objetos que se lleva a cabo para actualizar recursos dinámicos, en donde se observa que una vez que el monitor *RASupportResourceMonitor* ha detectado un cambio, solicita al módulo de agregación de recursos de *RAToolkit* que inicie el proceso de actualización del recurso dinámico en cuestión (tanto en la base de datos local como en la base de datos del *super-peer* que visita el agente de anunciamiento de actualización). La tabla 5.5 describe las modificaciones que se realizaron sobre el soporte Myconet para llevar a cabo la fase de anunciamiento de recursos.

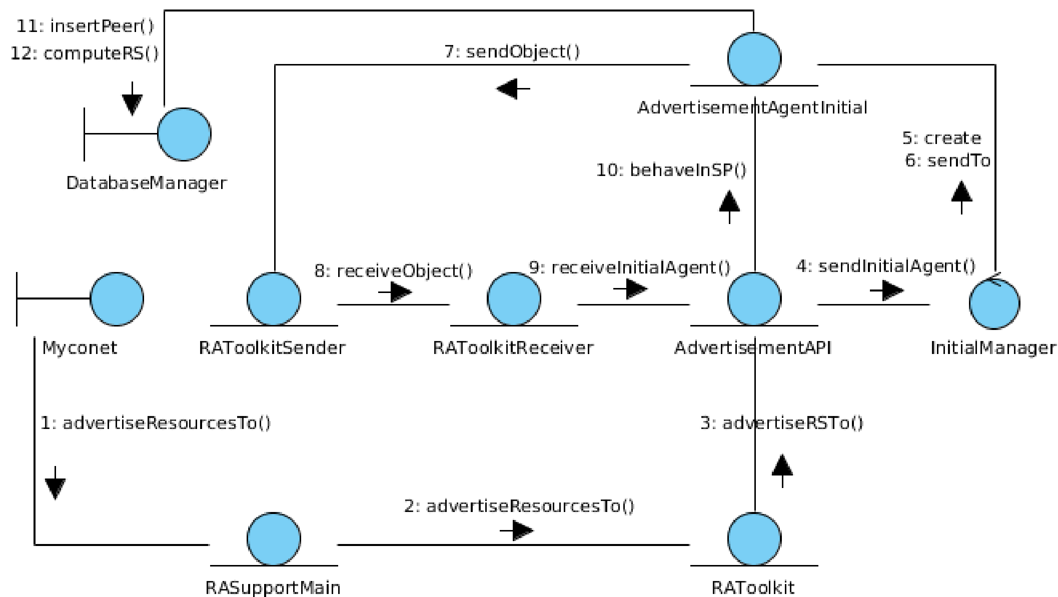


Figura 5.23: Diagrama de comunicación para el anunciamiento inicial de recursos.

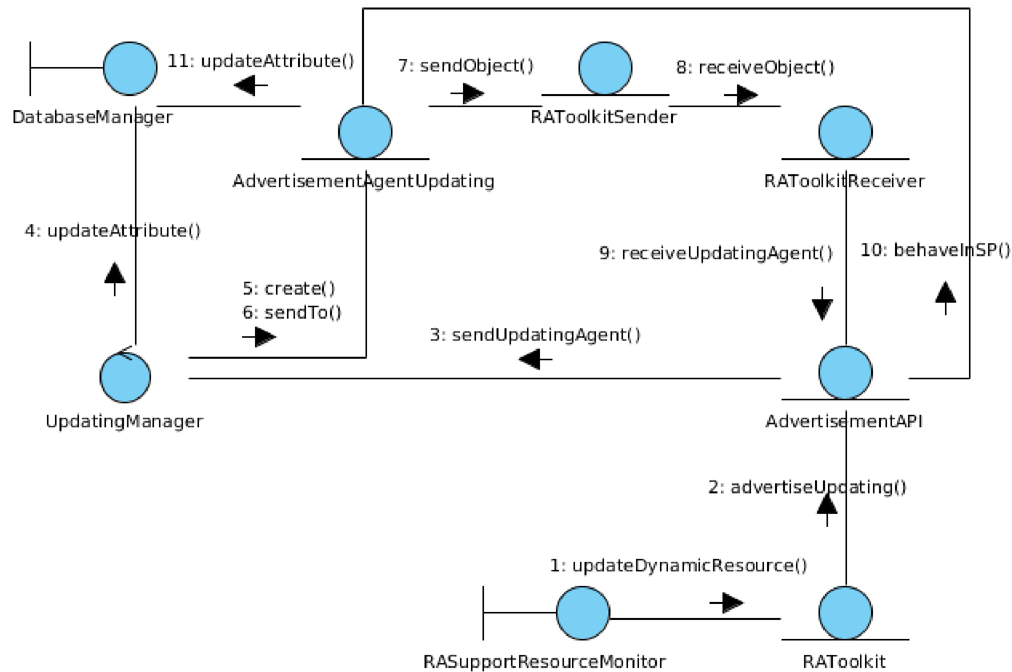


Figura 5.24: Diagrama de comunicación para la actualización de un recurso dinámico.

5.4.4. API de selección de recursos: SelectionAPI

SelectionAPI es una API de *RAToolkit* que permite llevar a cabo la fase de selección de recursos, por lo ofrece una fachada que implementa la interfaz *RAToolkitSelectionAPI*. Dicha API tiene el objetivo de obtener un conjunto de resultados que satisfagan los requerimientos y las restricciones de una consulta de recursos, empleando un agente de consulta. En la figura 5.25, se observa que *SelectionAPI* está compuesta por los módulos *agents*, *common* y *protocols*. El primero contiene la clase *QueryAgent* que representa un agente de consulta, la clase *QueryAgentResult* que representa los resultados obtenidos por dicho agente, la clase *QueryAgentsWaiter* que atiende una consulta de recursos, la clase *QueryEvaluator* que evalúa una consulta de recursos en un *super-peer* visitado y la clase *SelectionManager* que gestiona a los agentes de consulta (ver figura 5.26). Por otro lado, el módulo *common* únicamente contiene la interfaz *SelectionAPIConfig*, la cual especifica la configuración global de *SelectionAPI* (ver figura 5.25). Finalmente, el módulo *protocols* contiene la clase *Flooding* que posee la lógica del protocolo de inundación con agentes de consulta, la clase *IRandomWalk* que posee la lógica del protocolo de paseos aleatorios inteligentes (*iRandomWalks*), la clase *ProtocolContext* que emplea el patrón de diseño *Estrategia* [14] para ejecutar un protocolo de selección (dependiendo de la opción especificada en la consulta de recursos), la enumeración *QueryAgentsBehaviour* en donde se especifican los protocolos soportados por *SelectionAPI* y la interfaz *SelectionProtocol* que define los métodos que deben implementar los protocolos que existen o que llegasen a existir en dicha API (ver figura 5.27). Los protocolos de selección previamente mencionados emplean el patrón de diseño *Prototipo* [14] para llevar a cabo la creación de agentes

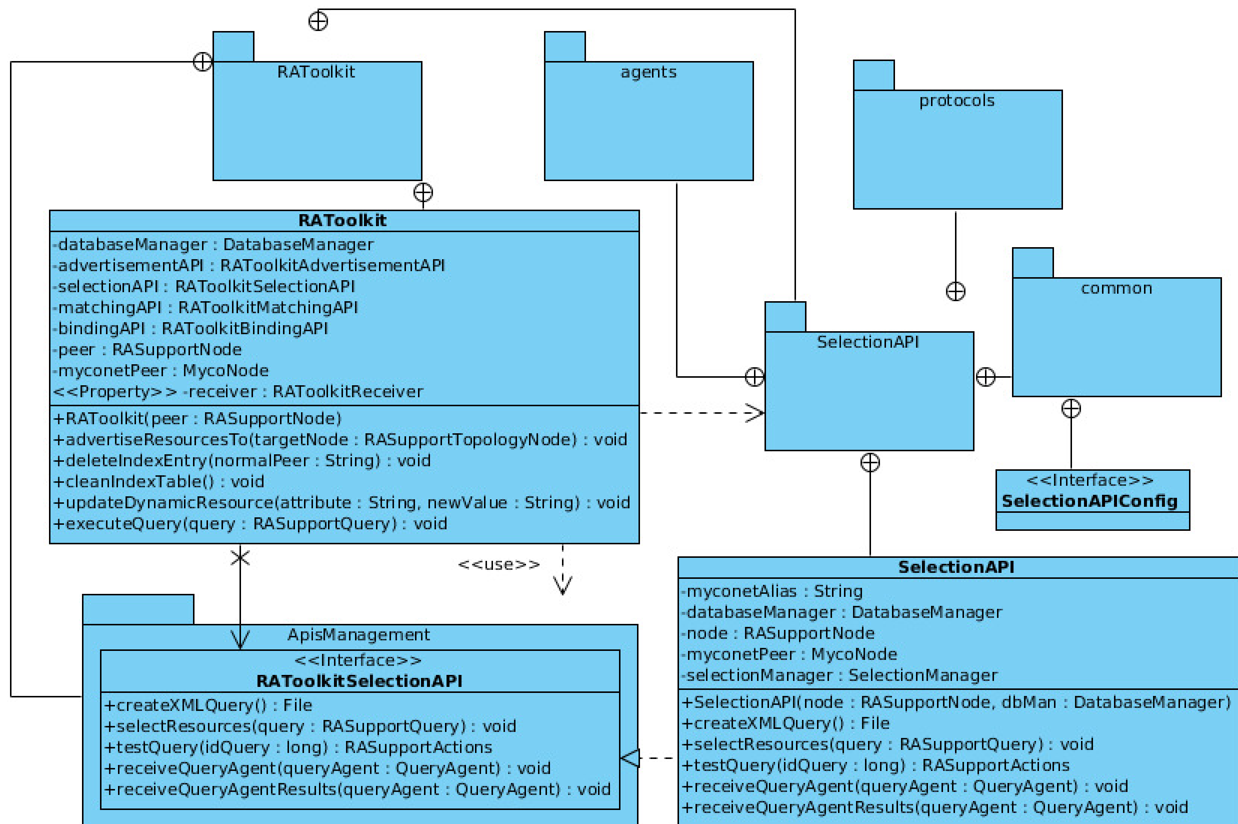


Figura 5.25: Diagrama de paquetes de *SelectionAPI*.

de consulta, con la finalidad de enviar una copia del agente que reciben como parámetro de entrada.

De manera similar a *AdvertisementAPI*, *SelectionAPI* se inicializa con dos instancias: una de la clase *RASupportNode* y otra de la clase *DatabaseManager*. La primera sirve para obtener información sobre el *peer* que se está gestionando, mientras que la segunda permite llevar a cabo consultas SQL sobre la base de datos. El constructor *SelectionAPI* crea una instancia de un gestor de agentes de consulta (i.e., una instancia de la clase *SelectionManager*).

En la figura 5.25, se observa que la fachada de *SelectionAPI* ofrece los métodos *createXMLQuery()*, *selectResources()*, *testQuery()*, *receiveQueryAgent()* y *receiveQueryAgentResults()*. El método *createXMLQuery()* es útil en sistemas P2P colaborativos en donde siempre se trabaja con una misma consulta de recursos, debido a que permite crear una única instancia de la clase *RASupportQuery*. El método *selectResources()* sirve para obtener resultados para una consulta de recursos, en función de los requerimientos y de las restricciones especificadas en el documento XML de dicha consulta. El método *testQuery()* permite determinar si un *super-peer* ya ha recibido una consulta de recursos en

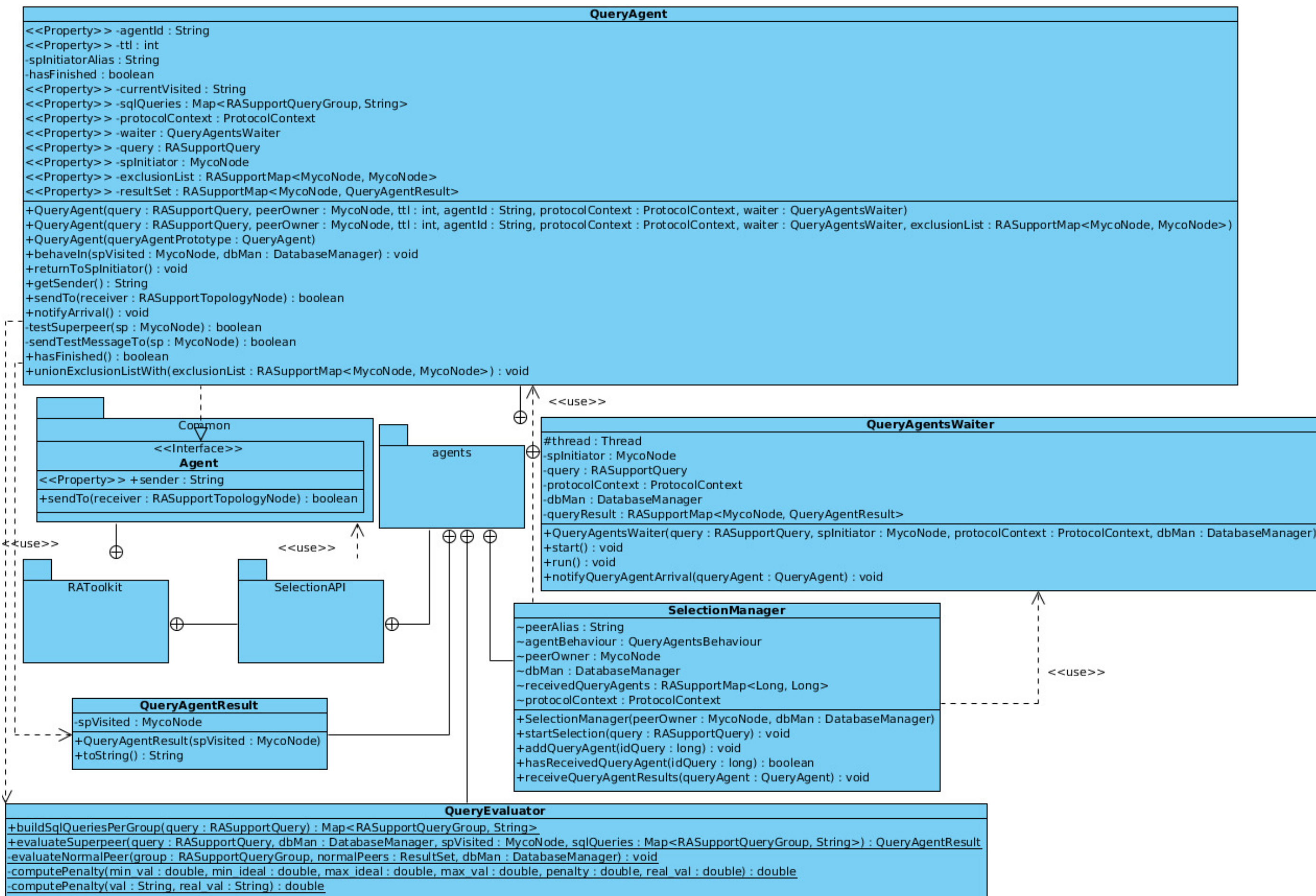


Figura 5.26: Diagrama de clases del módulo *agents*.

particular. El método *receiveQueryAgent()* sirve para recibir un agente de consulta en un *super-peer*, mientras que el método *receiveQueryAgentResults()* es empleado por un *super-peer* iniciador para recibir los resultados de un agente de consulta.

Los agentes de consulta transportan documentos XML de consultas de recursos y son creados por una instancia de la clase *QueryAgentsWaiter*, la cual se encarga de atender una consulta de recursos y de esperar los resultados (mediante una operación *join*) de los agentes de consulta que envía. El tiempo de vida de dichos agentes es directamente proporcional al tiempo que tardan en llevar a cabo su cometido. Al igual que los agentes de anunciamiento, el recolector de basura de la máquina virtual de Java es quien se encarga de liberar la memoria ocupada por estos agentes. En la figura 5.28, se muestran los estados por los cuales transita un agente de consulta durante su tiempo de vida.

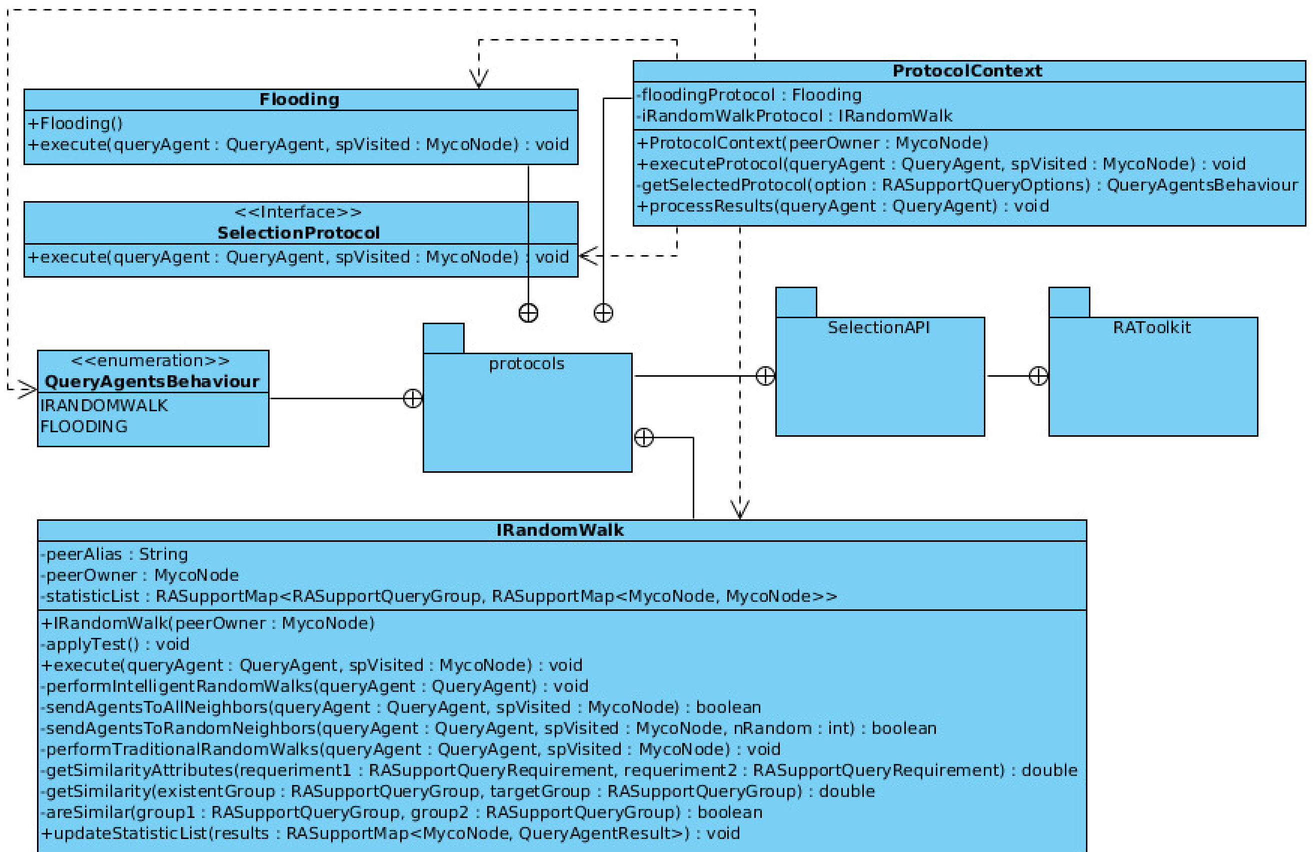


Figura 5.27: Diagrama de clases del módulo *protocols*.

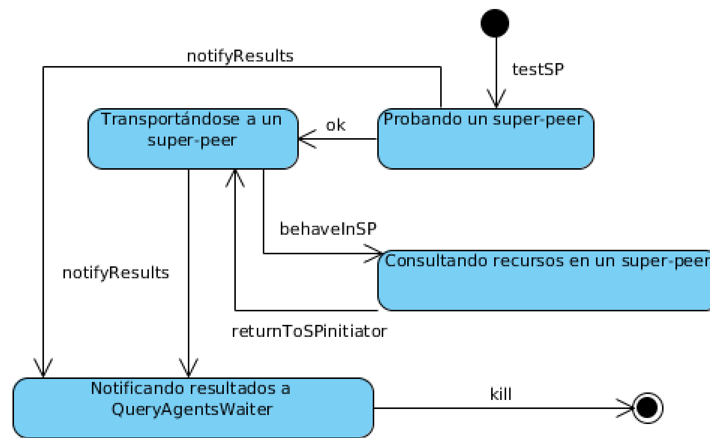


Figura 5.28: Diagrama de estados de un agente de consulta.

En la figura 5.29 se muestra la comunicación que existe en *RASupport* para llevar a cabo la fase de selección de recursos, en donde se observa que un sistema P2P colaborativo es quien inicia el proceso una vez que desea efectuar una consulta de recursos. Además, en esa misma figura se observa que *RASupport* oculta la complejidad de *RAToolkit*, mediante

la fachada *RASupportMain*.

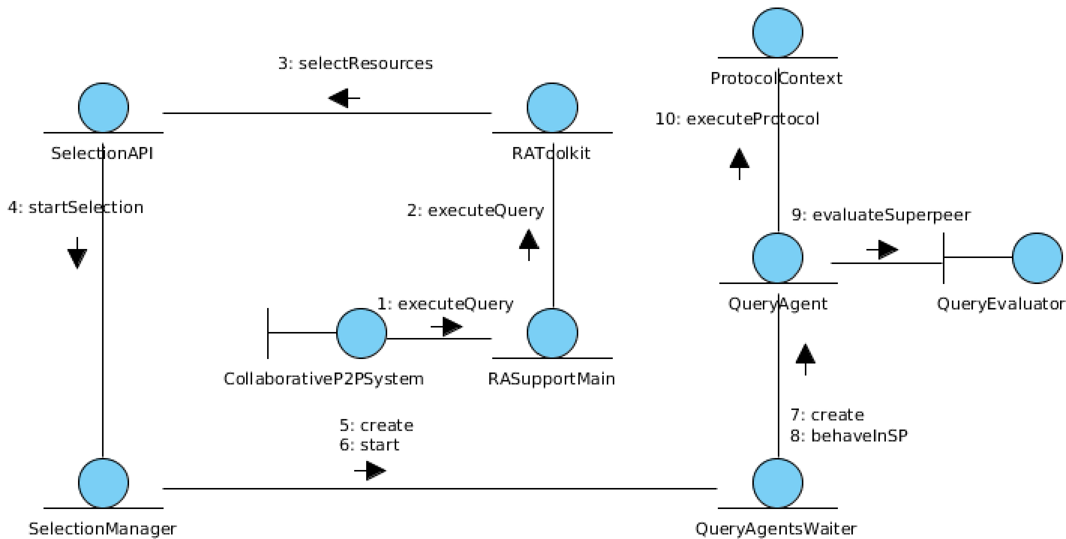


Figura 5.29: Diagrama de comunicación para la selección de recursos.

Además, en el archivo de configuración *.properties* de *RASupport* se permite especificar la configuración de los parámetros que determinan el comportamiento de *SelectionAPI*, con la finalidad de proporcionar flexibilidad al cliente de dicha API. En la tabla 5.6, se describen estos parámetros.

Parámetro	**Descripción**	**Valores permitidos**
ratoolkit.max_exclusion	Define el número máximo de elementos que pueden existir en la lista de exclusión de los agentes de consulta	Cualquier número natural que se encuentre en el intervalo [0, $MAX_INTEGER$), en donde $MAX_INTEGER$ es el máximo entero permitido de acuerdo con la arquitectura de la computadora en donde se encuentre *RASupport*
ratoolkit.max_rw	Define el número máximo de *superpeers* vecinos a los que $SP_{initiator}$ puede enviar agentes de consulta	Cualquier número natural que se encuentre en el intervalo [0, $MAX_INTEGER$), en donde $MAX_INTEGER$ es el máximo entero permitido de acuerdo con la arquitectura de la computadora en donde se encuentre *RASupport*

Tabla 5.6: Parámetros configurables de *RAToolkit* que definen el comportamiento de *SelectionAPI*.

Las consultas de recursos que transportan los agentes de consulta poseen un identifi-

cador único, el cual se encuentra compuesto por el identificador de un objeto de la clase *RASupportQuery*, el identificador de un objeto de la clase *MycoNode* y la marca de tiempo *timestamp* que corresponde al momento en que fue creada la consulta. Para generar dicho identificador se utiliza la siguiente función *hash*[7]:

$$h(q) = \sum_{i=0}^{n-1} 31^{n-1-i} s1_i + \sum_{i=0}^{m-1} 31^{m-1-i} s2_i + \sum_{i=0}^{o-1} 31^{o-1-i} s3_i \tag{5.1}$$

en donde $h(q)$ denota la función *hash* para una consulta de recursos q, $s1_i$ denota el i-ésimo carácter de la cadena de caracteres que representa un objeto de la clase *RASupportQuery*, $s2_i$ denota el i-ésimo carácter de la cadena de caracteres que representa un objeto de la clase *MycoNode*, $s3_i$ denota el i-ésimo carácter de la cadena de caracteres que representa un objeto de la clase *Timestamp*, n es la longitud de la cadena $s1$, m es la longitud de la cadena $s2$ y o es la longitud de la cadena $s3$.

En la topología construída por el soporte Myconet, los *super-peers* se encuentran conectados mediante enlaces simétricos (i.e., un *super-peer* A se encuentra conectado a un *super-peer* B y viceversa). Por lo tanto, los agentes de consulta mantienen una lista de exclusión que contiene los *super-peers* (i.e., instancias de la clase *MycoNode*) que ya han visitado. En caso de que un agente de consulta pretenda visitar a un *super-peer* y este último no se encuentre en la lista de exclusión, se le envía un mensaje de prueba para saber si no ha recibido a otro agente transportando una consulta con el mismo identificador. De esta manera, se garantiza que los *super-peers* que conforman a la red P2P no reciban múltiples agentes que transportan una consulta con el mismo identificador.

5.4.5. API de empatamiento de recursos: MatchingAPI

MatchingAPI es una API de *RAToolkit* que permite llevar a cabo la fase de empatamiento de recursos, por lo que ofrece una fachada que implementa la interfaz *RAToolkit-MatchingAPI*. Dicha API está basada en el mecanismo propuesto en el capítulo 4 y tiene el objetivo de determinar cuáles son los *peers* que mejor satisfacen los requerimientos y las restricciones especificadas en cada grupo de atributos requeridos de una consulta de recursos. En la figura 5.30, se observa que *MatchingAPI* está compuesta por los módulos *agents* y *common*. El primero contiene las clases que representan a los agentes de subempatamiento, los cuales se encargan de evaluar las restricciones entre *peers* y entre grupos de atributos requeridos (ver figura 5.31). El segundo módulo contiene la interfaz *Matcher* que implementan las clases *GroupMatcher* y *PeerMatcher*, las cuales se encargan de empatar las restricciones entre grupos de atributos requeridos y las restricciones entre *peers*, respectivamente (ver figura 5.32). A diferencia de *AdvertisementAPI* y *SelectionAPI*, el constructor de *MatchingAPI* no tiene argumentos ya que las entradas son los resultados obtenidos por cada uno de los agentes de consulta que una instancia de la clase *QueryAgentsWaiter* ha enviado.

[7]https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html.

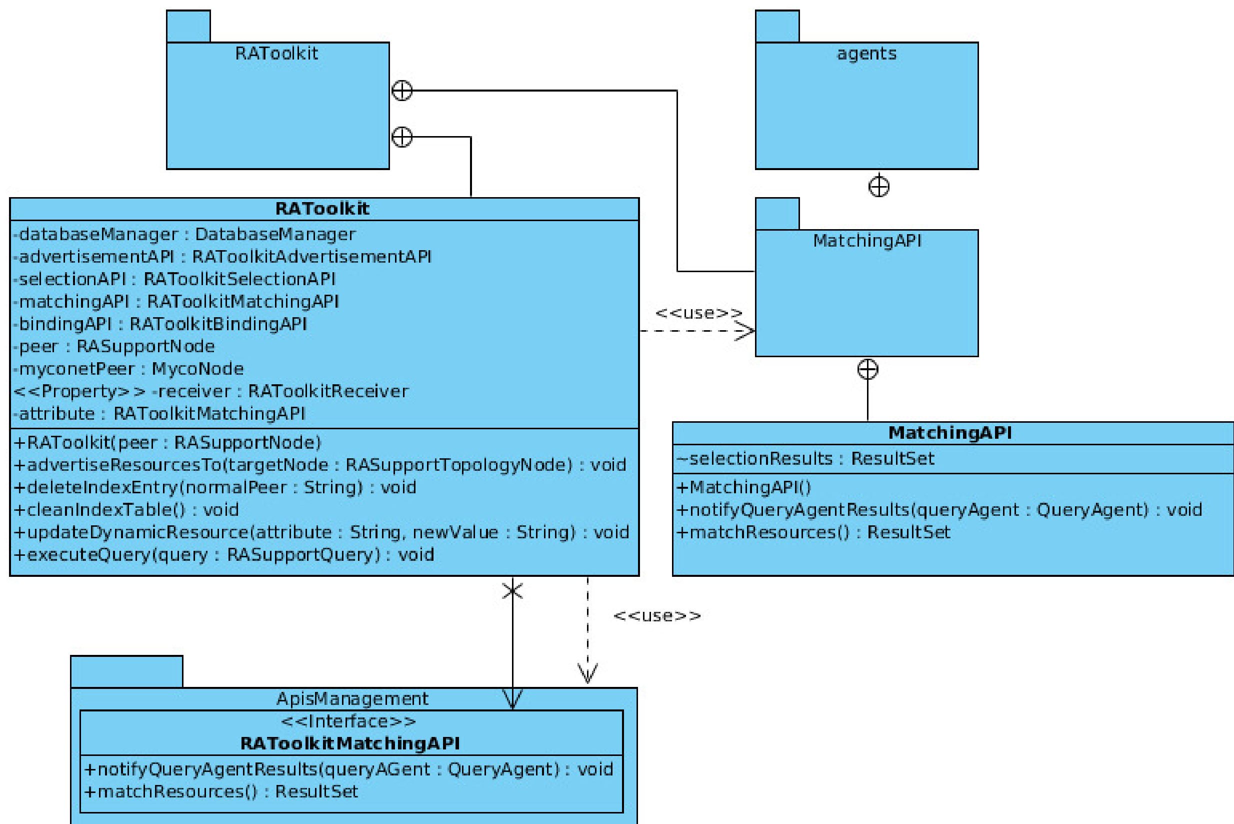


Figura 5.30: Diagrama de paquetes de *MatchingAPI*.

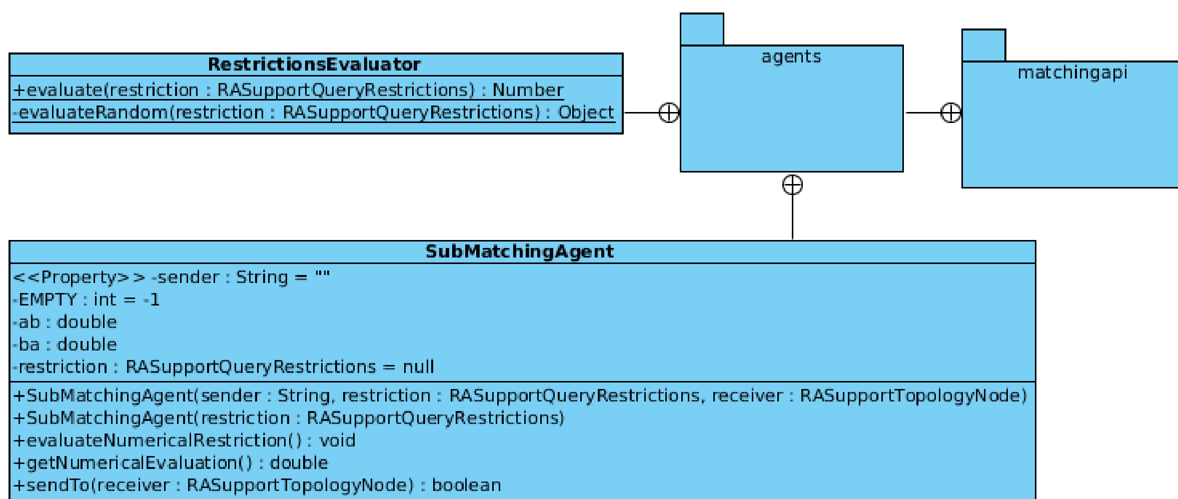


Figura 5.31: Diagrama de clases del módulo *agents*.

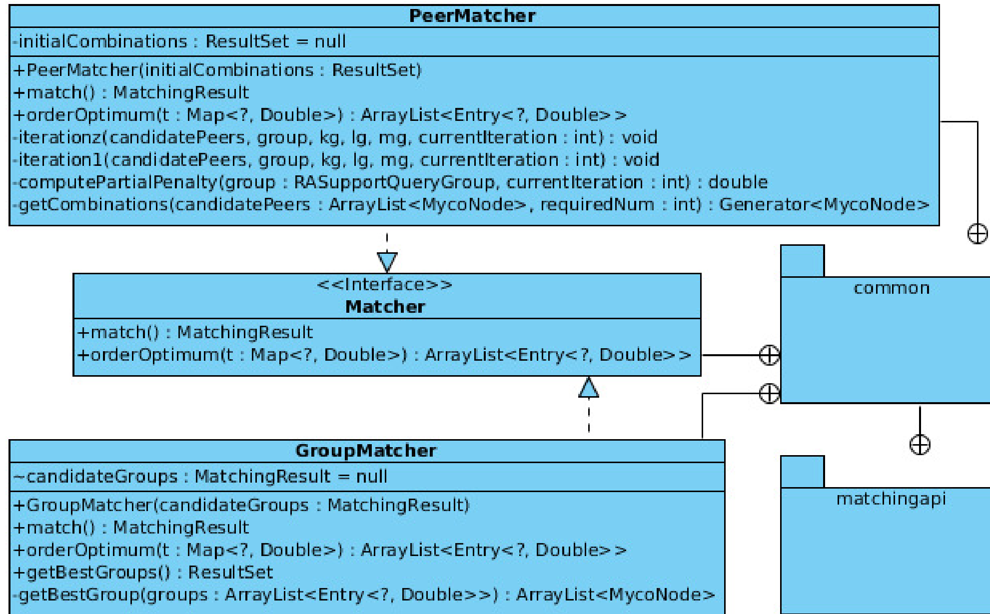


Figura 5.32: Diagrama de clases del módulo *common*.

En la figura 5.30 se observa que la fachada de *MatchingAPI* ofrece los métodos *notifyQueryAgentResults()* y *matchResources()*. El primero de ellos sirve para notificar los resultados de un agente de consulta a un agente de empatamiento, mientras que el segundo se utiliza para iniciar la fase de empatamiento de recursos. *MatchingAPI* representa un agente de empatamiento *per-se*, por lo que las instancias de esta API son creadas por instancias de la clase *QueryAgentsWaiter*. Un agente de empatamiento tiene la finalidad de esperar (mediante una operación *join*) los resultados obtenidos por los agentes de consulta que han sido enviados por $SP_{initiator}$. Cuando un agente de consulta regresa a $SP_{initiator}$, dicho agente emplea el método *notifyQueryAgent()* para notificar sus resultados al agente de empatamiento. Una vez que se han recibido los resultados de todos los agentes de consulta enviados, el objeto de la clase *QueryAgentsWaiter* ejecuta el método *matchResources()* del agente de empatamiento para llevar a cabo las dos etapas descritas en el capítulo 4 (i.e., empatamiento de *peers* candidatos y de grupos candidatos), mediante los métodos *match()* de las clases *PeerMatcher* y *GroupMatcher*, respectivamente. El agente de empatamiento obtiene como salida un objeto *ResultSet*, el cual contiene un mapa *hash* concurrente con los *peers* que mejor satisfacen los requerimientos y las restricciones para cada grupo de atributos requeridos especificado en la consulta de recursos.

Los subagentes de empatamiento se encuentran representados por la clase *SubMatchingAgent* y emplean la clase *RestrictionsEvaluator* para evaluar las restricciones entre nodos y entre grupos, especificadas en una consulta de recursos. Con la finalidad de proporcionar robustez y heterogeneidad en los experimentos, *RestrictionsEvaluator* permite generar evaluaciones aleatorias que dependen del tipo atributo y de las ecuaciones presentadas en el capítulo 4. De manera similar a los agentes de consulta, el tiempo de vida

de los subagentes de empatamiento es directamente proporcional al tiempo que tardan en llevar a cabo su cometido y es el recolector de basura de la máquina virtual de Java quien se encarga de liberar la memoria ocupada por estos agentes. En la figura 5.33, se muestran los estados por los cuales transitan los agentes y los subagentes de empatamiento durante su tiempo de vida.

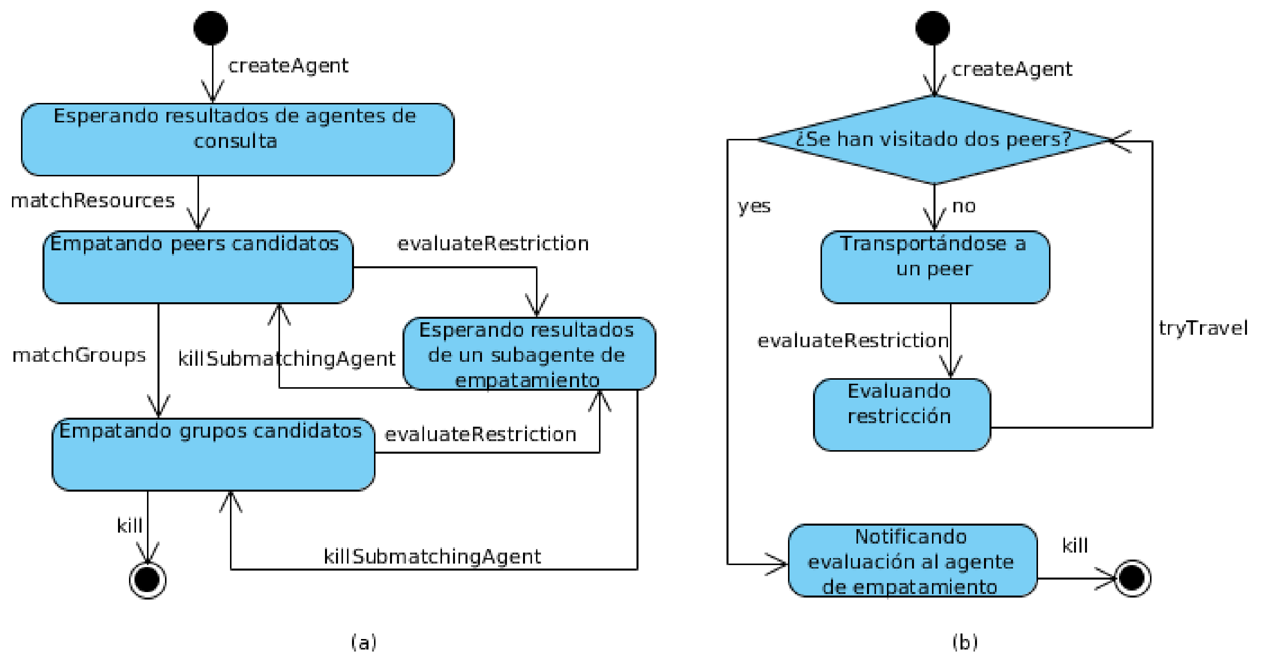


Figura 5.33: Diagrama de estados de (a) un agente de empatamiento y (b) un agente de subempatamiento.

En la figura 5.34 se muestra la comunicación que existe en *RASupport* para llevar a cabo la fase de empatamiento de recursos, en donde se observa que un objeto *QueryAgentsWaiter* es quien inicia el proceso una vez que todos los agentes de consulta han regresado resultados para una consulta de recursos en particular. Además, en esa misma figura se observa que *RASupport* oculta la complejidad de *MatchingAPI*, mediante el método *matchResources()* proporcionado por la fachada que implementa la interfaz *RAToolkitMatchingAPI*.

MatchingAPI genera combinaciones de *peers* y de grupos candidatos, mediante algoritmos de clase NP-difícil; por lo tanto, es necesario mejorar dicha API con heurísticas que permitan detener la búsqueda de combinaciones, evitando así cuellos de botella al momento de llevar a cabo la fase de empatamiento de recursos.

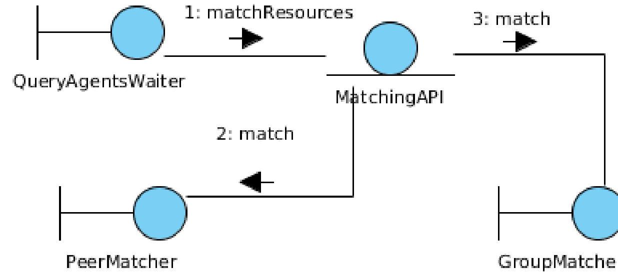


Figura 5.34: Diagrama de comunicación para el empatamiento de recursos.

5.4.6. API de enlace de recursos: Binding API

BindingAPI es una API de *RAToolkit* que permite llevar a cabo la fase de enlace de recursos, por lo que ofrece una fachada que implementa la interfaz *RAToolkitBindingAPI*. Dicha API está basada en el mecanismo propuesto en el capítulo 4 y tiene el objetivo de determinar si los recursos de los *peers* proveedores están disponibles para su uso, a partir de los resultados obtenidos en la fase de empatamiento de recursos. En la figura 5.35, se observa que *BindingAPI* está compuesta por el módulo *agents*, el cual contiene las clase *BindingAgent* que representa un agente de enlace y la clase *BindingAgentAsset* que representa el activo que transportan los agentes de enlace (ver figura 5.36). La clase *BindingAgent* implementa la interfaz *Agent* debido a que representa un agente dentro de *RASupport*. Los agentes de enlace emplean las reglas descritas en el capítulo 4 y utilizan los métodos (1) *behaveInSP()* para comportarse en un *super-peer* visitado, (2) *sendResourceRequestMessage()* para enviar un mensaje de solicitud de recursos, (3) *returnToSpInitiator()* para regresar al *super-peer* iniciador, (4) *getVisited()* para obtener una referencia al *super-peer* visitado y (5) *getAsset()* para obtener una referencia al activo transportado. Dicho activo está conformado por una consulta de recursos (*query*), una lista de *peers proveedores* (*superpeerProviders*) que están a cargo del *super-peer* visitado por el agente de enlace y un mapa (*providerGroups*) que asocia cada *peer* proveedor con los grupos solicitados en la consulta de recursos. El constructor de *BindingAPI* toma una instancia de la clase *RASupportNode* como argumento de entrada, ya que es necesario mantener una referencia a los *super-peers*.

Adicionalmente, en la figura 5.35 se observa que la fachada de *BindingAPI* ofrece los métodos *receiveBindingAgent()*, *RequestResources()*, *receiveBindingResults()* y *bindResources()*. El método *receiveBindingAgent()* sirve para recibir un agente de enlace en un determinado *super-peer*, el método *RequestResources* se utiliza para atender un mensaje de solicitud de recursos enviado por un agente de enlace, el método *receiveBindingResults* es utilizado por $SP_{initiator}$ para recibir los resultados de un agente de enlace que ha enviado y, finalmente, el método *bindResources()* es aquel que es empleado por una instancia de *RAToolkit* para solicitar la fase de enlace de recursos. La salida de *BindingAPI* es una instancia de la clase *RASupportResults*, la cual contiene los resultados de las fases de selección, empatamiento y enlace, i.e., los *peers* proveedores que mejor satisfacen los

requerimientos y las restricciones especificadas en la consulta de recursos, cuyos recursos están disponibles para ser utilizados.

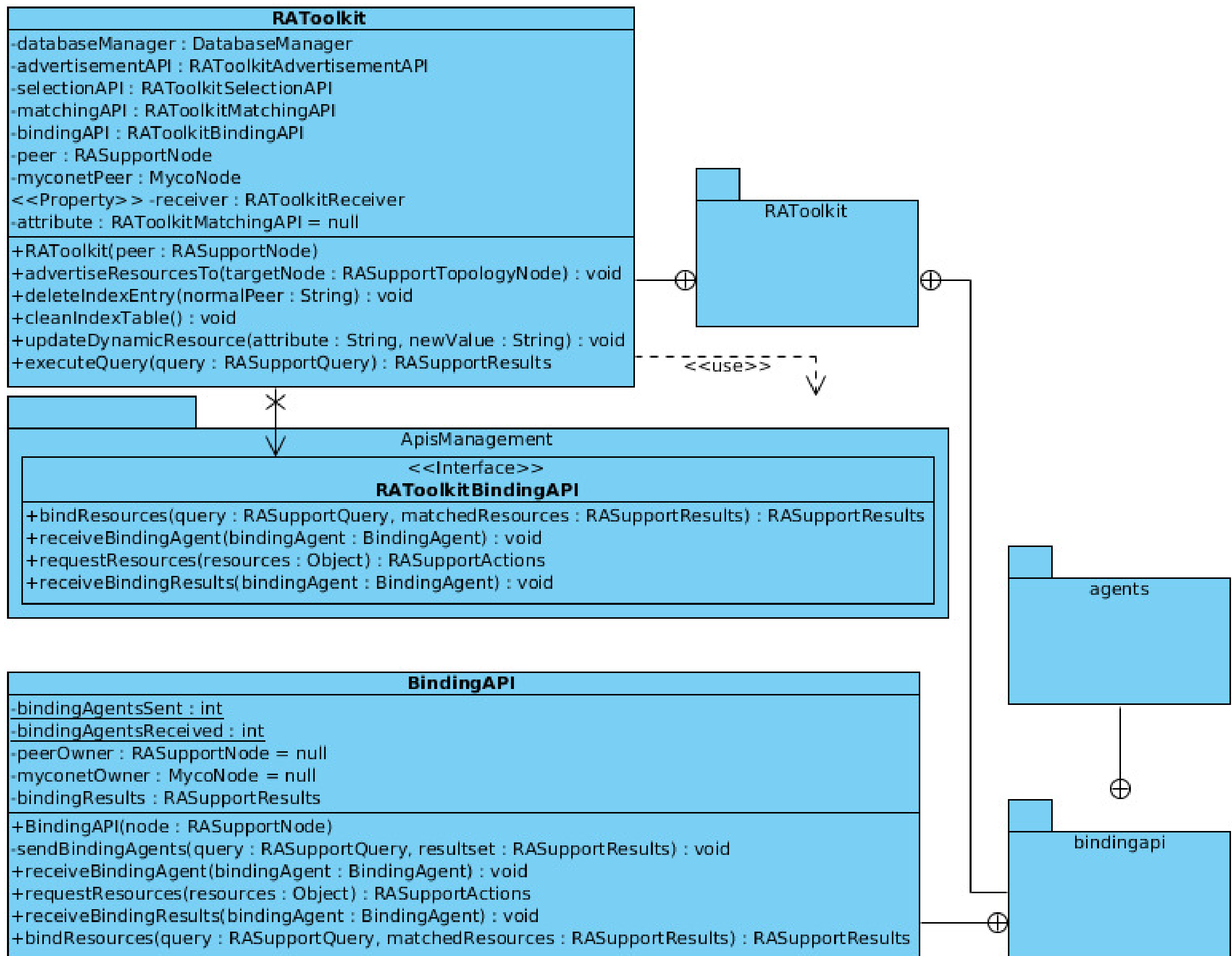


Figura 5.35: Diagrama de paquetes de *BindingAPI*.

El tiempo de vida de los agentes de enlace es directamente proporcional al tiempo que tardan en llevar a cabo su cometido y, al igual que los demás agentes de *RAToolkit*, es la máquina virtual de Java quien se encarga de liberar la memoria ocupada por dichos agentes. En la figura 5.37 se muestran los estados por los cuales transitan los agentes de enlace durante su tiempo de vida.

En la figura 5.38 se ilustra la comunicación que existe en *RASupport* para llevar a cabo la fase de enlace de recursos, en donde se observa que un objeto *RAToolkit* es quien inicia el proceso una vez que ha completado las fases de selección y de empatamiento, respectivamente. Además, en esa misma figura se observa que *RASupport* oculta la complejidad de *BindingAPI*, mediante el método *bindResources()* proporcionado por la fachada de *RAToolkitBindingAPI*.

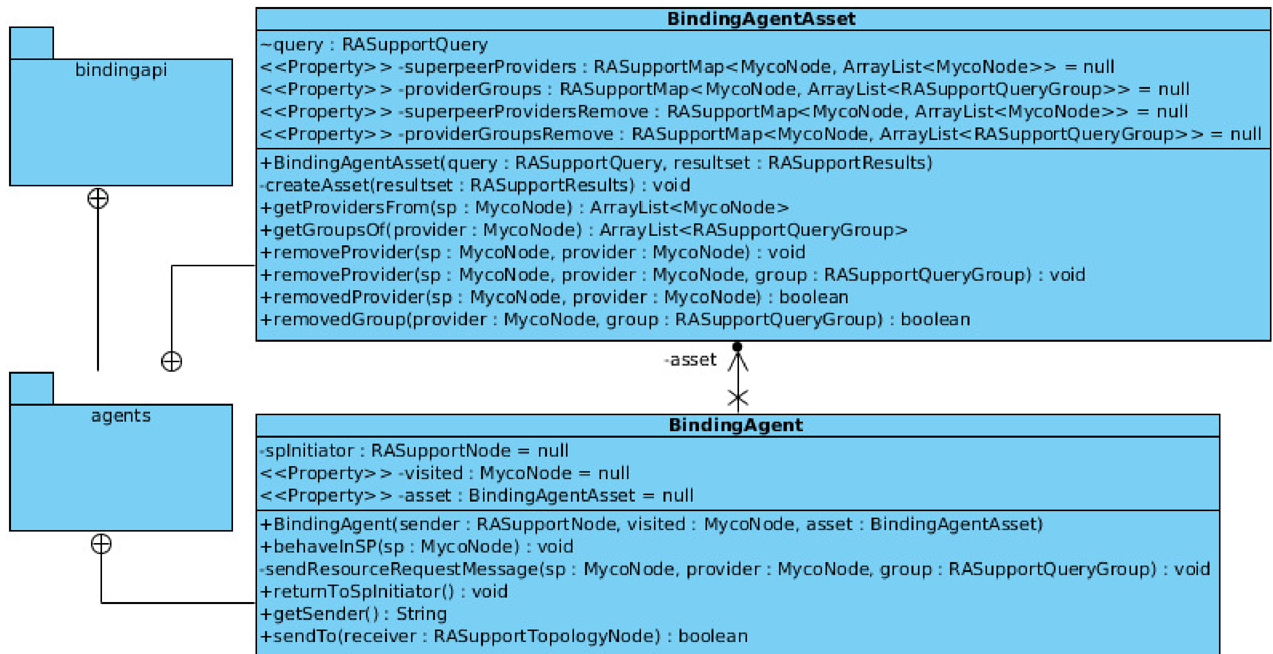


Figura 5.36: Diagrama de clases del módulo *Agents*.

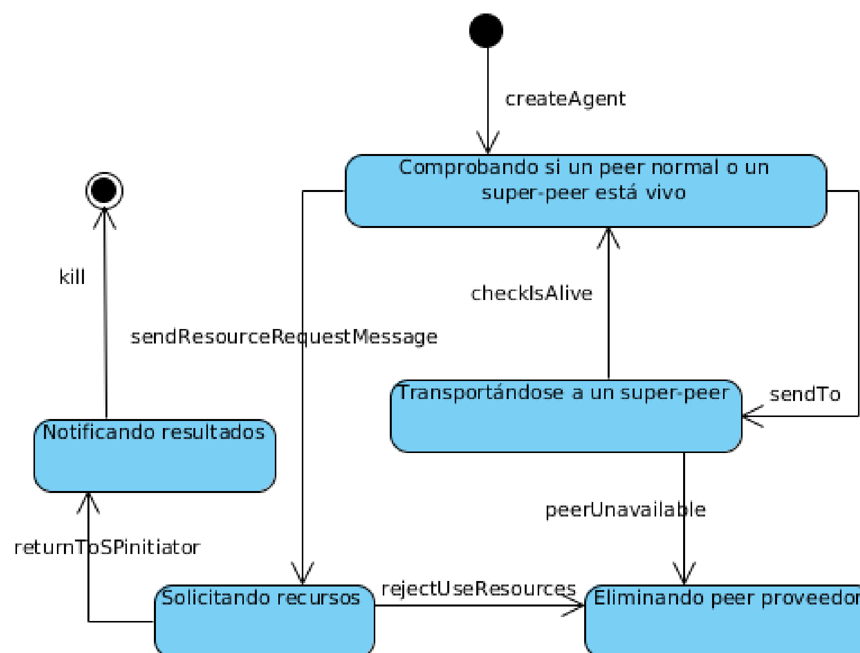


Figura 5.37: Diagrama de estados de un agente de enlace.

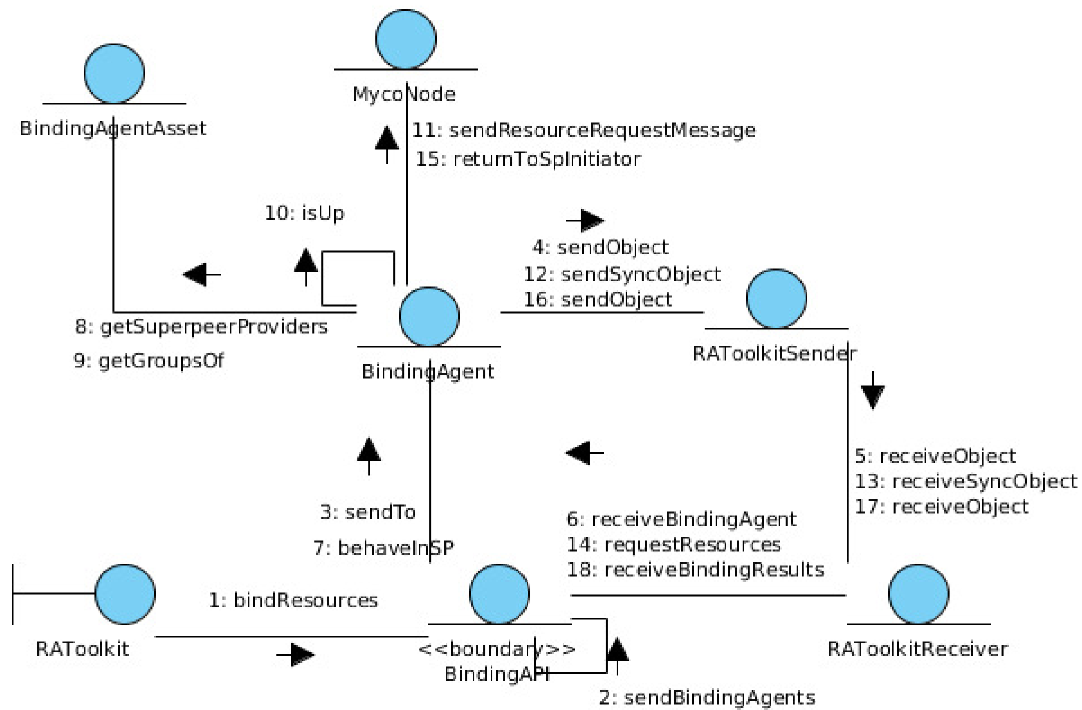


Figura 5.38: Diagrama de comunicación para el enlace de recursos.

BindingAPI envía mensajes *ping* para verificar que un *peer proveedor* o un *super-peer* continúen vivos; para ello, emplea el método *isUp()* de un objeto *MycoNode*. Cada vez que se determina que un *peer proveedor* de un grupo –de los resultados obtenidos en la fase de empatamiento– no está disponible, no se encuentra en la disposición de compartir sus recursos o estos últimos no están disponibles, los agentes de enlace eliminan al *peer proveedor* del grupo correspondiente, garantizando así Calidad de Servicio.

Capítulo 6

Resultados y caso de estudio

En este capítulo, se presentan los resultados obtenidos a partir de la presente tesis. En particular, se muestran los resultados de las fases de anunciamiento (sección 6.1), selección (sección 6.2), empatamiento (sección 6.3) y enlace (sección 6.4), utilizando el *toolkit* proporcionado por *RASupport*, i.e., *RAToolkit*.

Con la finalidad de llevar a cabo los experimentos que validan y verifican el funcionamiento de *RASupport*, se empleó la herramienta de software *PeerSim*, la cual está escrita en lenguaje Java y permite realizar simulaciones de redes P2P utilizando grandes cantidades de *peers*. En particular, se empleó el motor basado en ciclos y, para cada uno de los *peers* creados por *PeerSim*, se generaron aleatoriamente recursos heterogéneos (caracterizados por atributos dinámicos y estáticos) mediante el módulo *ResourceSim*. Además, se utilizó el monitor *rsimmonitor* para generar actualizaciones aleatorias en los atributos dinámicos de los *peers* previamente mencionados.

Es necesario validar el funcionamiento de *RASupport* en ambientes semi-dinámicos (e.g., cómputo en *grid*) y en ambientes altamente dinámicos (e.g., sistemas sensibles a la latencia); por ello, *ResourceSim* permite modelar ambientes dinámicos empleando intervalos de actualización de uno a cinco segundos, mientras que los ambientes semi-dinámicos se modelan con intervalos de actualización de seis a 20 segundos. La computadora en la que se ejecutaron los experimentos, posee las siguientes características:

- Procesador Intel© Core™ i7-3630QM CPU @ 2.40 GHz con 6 núcleos.
- Memoria caché de 6144 KB.
- Memoria RAM con una capacidad de 6 GB y una velocidad de 1600 MHz.
- Sistema operativo Fedora 19 (Gato de Schrodinger), el cual está basado en Linux.
- Arquitectura AMD de 64 bits.
- Máquina virtual de Java (JVM, por sus siglas en inglés) de 64 bits en modo mezclado.

- Disco duro de 1 TB.
- Conexión Wi-Fi a un módem que utiliza el estándar IEEE 802.11b y posee una potencia de transmisión de 16 dBm, una tasa de bits de 36 Mb/s y una frecuencia de 2.427 GHz.

Con la finalidad de mantener homogeneidad en la presente tesis, en los experimentos descritos en este capítulo se emplean las definiciones formales que fueron presentadas en el capítulo 4, las cuales se resumen a continuación:

- Definiciones formales para la arquitectura P2P: (1) soporte P2P de agregación de recursos - c, (2) *peer normal* - NP y (3) *super-peer* - SP.
- Definiciones formales para la fase de selección de recursos: (1) *super-peer* iniciador de una consulta de recursos - $SP_{initiator}$, (2) *super-peer* visitado por un agente de consulta - SP_v, (3) conjunto de *super-peers* visitados por agentes de consulta dentro de *RASupport* - VN_c, (4) conjunto de consultas de recursos dentro de *RASupport* - Q_c, (5) consulta de recursos específica dentro de *RASupport* - q, (6) conjunto de grupos solicitados en una consulta $q \in Q_c$ - G_q y (7) grupo específico solicitado en una consulta $q \in Q_c$ - g.
- Definiciones formales para la fase de empatamiento de recursos: (1) lista inicial de *peers* candidatos para un grupo $g \in G_q$ - I_g, (2) lista de referencia de *peers* que pueden trabajar entre sí en un grupo $g \in G_q$ - K_g, (3) lista de referencia de *peers* que no pueden trabajar entre sí en un grupo $g \in G_q$ - M_g, (4) lista de grupos de *peers* candidatos para un grupo $g \in G_q$ - L_g, (5) número de grupos de *peers* candidatos para un grupo $g \in G_q$ - n, (6) número de restricciones entre nodos[1] para un grupo $g \in G_q$ - m y (7) penalización total de un grupo de *peers* candidatos - P_{groups}.
- Definiciones formales para la fase de enlace de recursos: (1) grupo óptimo de *peers* proveedores para un grupo $g \in G_q$ - b_g, (2) *peer* proveedor de recursos - pn y (3) *peer* consumidor de recursos - pr.

Las definiciones formales presentadas en el capítulo 4 son irrelevantes en los experimentos llevados a cabo en la fase de anunciamiento de recursos. Además, los índices de NP, SP, pn y pr corresponden a los identificadores de los *peers* dentro de la topología P2P formada por *RASupport*; mientras que los índices de I_g, K_g, M_g, L_g, n, m y b_g corresponden al grupo al cual hacen referencia.

En la sección 6.5 se presenta un caso de estudio que valida el funcionamiento y uso de *RASupport*. En particular, se describe el análisis, el diseño y la implementación de un sistema P2P colaborativo, cuyo objetivo es la generación del conjunto de Mandelbrot

[1]Es necesario recordar que, en la presente tesis, un nodo es equivalente a un *peer* en una topología P2P.

a partir de la distribución de tareas en nodos de procesamiento, los cuales envían sus resultados a un nodo de despliegue con la finalidad de que este último muestre el fractal en su pantalla. En esta sección, también se presentan los resultados obtenidos a partir de la ejecución del sistema P2P colaborativo de validación, el cual emplea *RASupport* para llevar a cabo las fases clave de la agregación de recursos.

6.1. Resultados de la fase de anunciamiento de recursos

Los experimentos que validan el funcionamiento de *AdvertisementAPI* poseen la siguiente configuración global:

- Duración: 50 ciclos.
- Duración de cada ciclo: 500 milisegundos.
- Probabilidad de que un *super-peer* en un estado inmóvil promueva a alguno de sus *peers normales* en un determinado ciclo: 0.1.

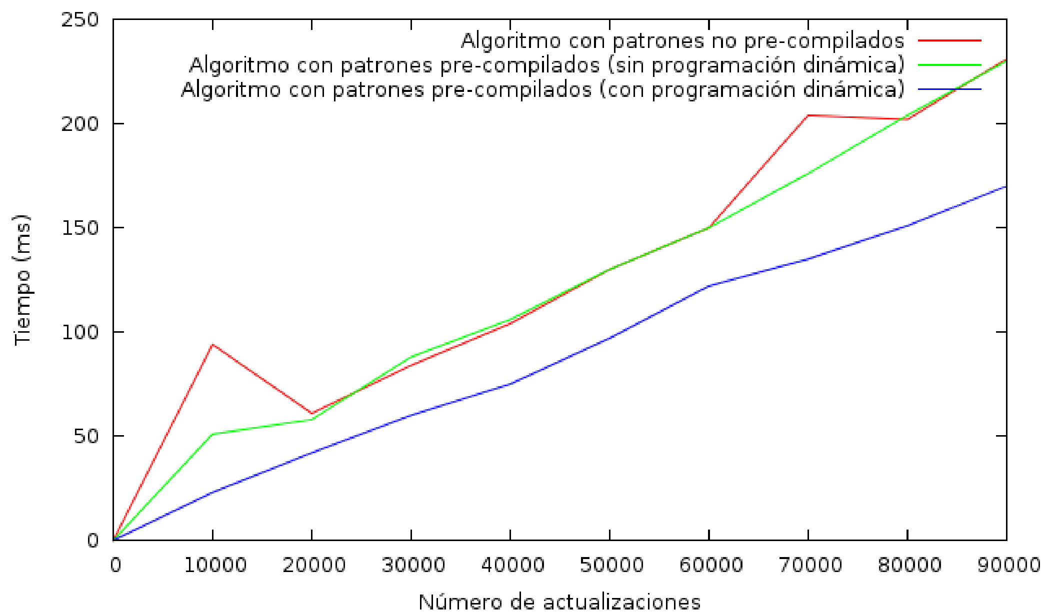


Figura 6.1: Análisis de los algoritmos propuestos para la actualización de recursos dinámicos.

Cada vez que el monitor de recursos detecta una actualización en un recurso dinámico, un objeto de la clase *UpdatingManager* actualiza el documento de especificación de recursos que mantiene y envía un agente de anunciamiento de actualización al respectivo *super-peer*. Con la finalidad de mejorar la eficiencia en dichas actualizaciones, se

propusieron y analizaron tres algoritmos: (1) algoritmo con patrones no pre-compilados, (2) algoritmo con patrones pre-compilados —sin programación dinámica— y (3) algoritmo con patrones pre-compilados —con programación dinámica—[2]. Estos algoritmos permiten actualizar recursos dinámicos en un documento XML de especificación de recursos (RS, por sus siglas en inglés). Para un experimento de 100,000 actualizaciones en un documento RS, como el que se muestra en el listado 6.1, se obtuvieron los resultados de la figura 6.1, en donde se observa que el algoritmo con patrones pre-compilados —con programación dinámica— se desempeña mejor que los otros dos algoritmos. Por lo tanto, la clase *UpdatingManager* emplea dicho algoritmo para generar, compilar y almacenar una expresión regular (e.g., la expresión regular para el espacio libre en disco es `<free_hdisk>(.*)</free_hdisk>`) en una tabla *hash* para cada atributo soportado por *RASupport*. Estas expresiones regulares se utilizan para reemplazar coincidencias en un documento XML de especificación de recursos.

Listing 6.1: Documento XML de especificación de recursos, empleado en el análisis de los algoritmos propuestos para la actualización de recursos dinámicos.

```
<rs_initial>
	<use>
		<availability_start>00:00</availability_start>
		<availability_end>16:32</availability_end>
		<max_cpu_utilization>15</max_cpu_utilization>
		<only_for_friends>peer3,peer1</only_for_friends>
	</use>
	<processing>
		<free_mem>15855.58</free_mem>
		<busy_cpu>31.96</busy_cpu>
		<os_name>Linux</os_name>
		<cpu_speed>3963.54</cpu_speed>
		<cores>5</cores>
	</processing>
	<storage>
		<free_hdisk>85024.1</free_hdisk>
		<total_hdisk>442877.28</total_hdisk>
	</storage>
	<display>
		<bit_depth>6</bit_depth>
		<refresh_rate>641</refresh_rate>
		<screen_height>721</screen_height>
		<screen_width>408</screen_width>
	</display>
	<network>
		<bandwidth>420.76</bandwidth>
		<latency>189.77</latency>
	</network>
</rs_initial>
```

[2]Los algoritmos con patrones pre-compilados compilan dichos patrones una sola vez, por lo que no se requiere la creación de múltiples objetos *Pattern*, ahorrando así memoria RAM.

El monitor de recursos *rsimmonitor* permite variar aleatoriamente el intervalo de actualización de recursos dinámicos con una distribución uniforme discreta; por lo tanto, se pueden simular tanto ambientes dinámicos (en donde los recursos cambian rápidamente) como ambientes semi-dinámicos. Los ambientes dinámicos se modelan con un intervalo de actualización de uno a cinco segundos, mientras que los ambientes semi-dinámicos se modelan con un intervalo de actualización de seis a 20 segundos.

Con la finalidad de validar y verificar que el anunciamiento de recursos efectuado por *AdvertisementAPI* se lleva a cabo aún cuando existe *churn*[3], se realizó un experimento en una red P2P altamente dinámica compuesta por 100 *peers*, en donde los recursos dinámicos se actualizan de uno a dos segundos. En particular, en cada ciclo existe *churn* de 0.02, i.e., en cada ciclo entran y abandonan el 2 % de los *peers*, los cuales pueden ser de cualquier tipo (i.e., *super-peers* o *peers normales*). En la figura 6.2, se muestran los resultados obtenidos a partir de dicho experimento, en donde se observa que los agentes de anunciamiento inicial se envían constantemente debido a que la red P2P se auto-organiza como consecuencia del *churn* que existe. El número de agentes de anunciamiento inicial que se envían alcanza su máximo pico en los primeros ciclos, debido a la gran cantidad de *peers* que se unen a la red P2P y a que estos necesitan anunciar sus recursos a los *super-peers* que se encuentran a cargo. En general, en la figura 6.2 se observa que se envían más agentes de anunciamiento de actualización que agentes de anunciamiento inicial, debido al ambiente altamente dinámico que fue simulado en este experimento.

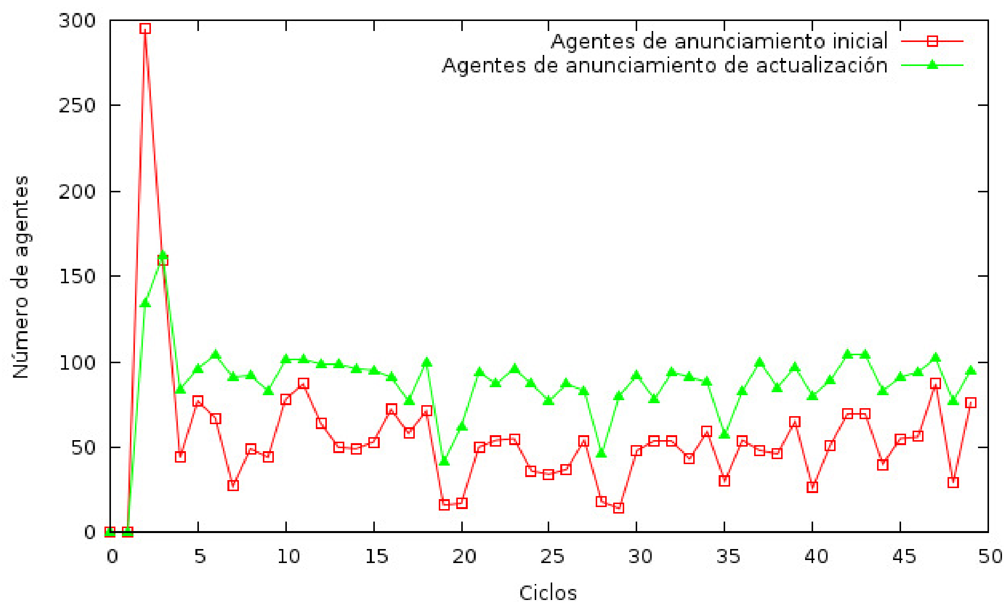


Figura 6.2: Número de agentes de anunciamiento en un experimento de 50 ciclos con *churn* del 2 %.

[3]En redes P2P, el término *churn* se refiere al efecto colectivo que se produce como consecuencia de la llegada y la partida de miles o millones de *peers*.

AdvertisementAPI permite el anunciamiento de recursos tanto en ambientes altamente dinámicos como en ambientes semi-dinámicos. Para validar esto, se llevó a cabo un experimento en una red de 100 *peers* con *churn* de 0.10 (i.e., en cada ciclo entran y salen el 10 % de los *peers*). Para tal experimento, un ambiente dinámico se modela con un intervalo de actualización de uno a cinco segundos, mientras que un ambiente semi-dinámico se modela con un intervalo de actualización de seis a 20 segundos. En la figura 6.3, se muestra el resultado de este experimento, en donde se oberva que en ambientes altamente dinámicos se envía una mayor cantidad de agentes de anunciamiento de actualización que en ambientes semi-dinámicos.

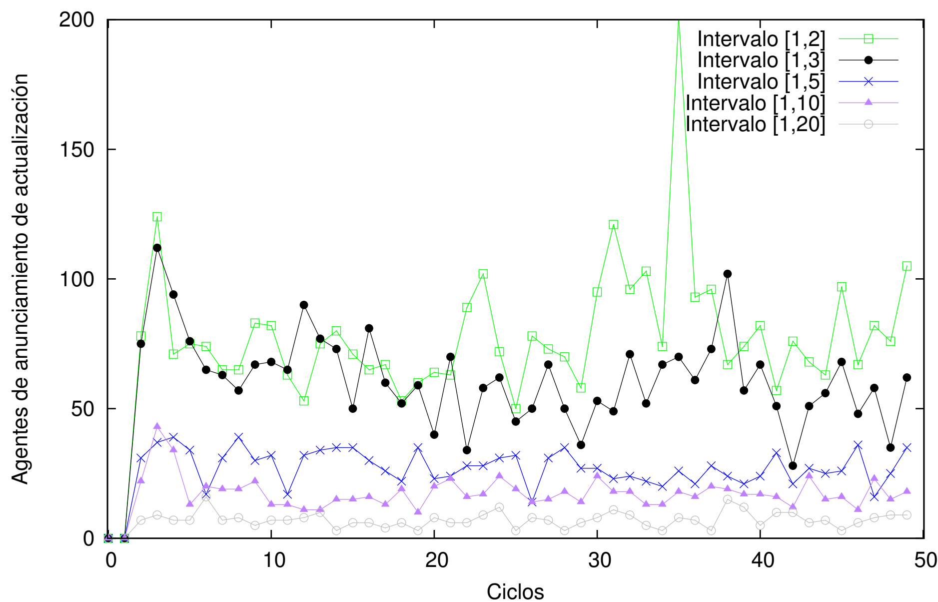


Figura 6.3: Agentes de anunciamiento de actualización en ambientes dinámicos y semi-dinámicos.

Con la finalidad de determinar el número de agentes de anunciamiento que se envían en una red P2P altamente dinámica, en donde los recursos cambian en un intervalo de uno a dos segundos, se llevó a cabo un experimento en el que se varía el tamaño de la red de cero a 500 *peers* con la suposición de que no existe *churn*. En la figura 6.4, se muestra el resultado de dicho experimento, en donde se observa que el número de agentes de anunciamiento de actualización crece considerablemente conforme crece el tamaño de la red, mientras que el número de agentes de anunciamiento inicial se mantiene casi estable, debido a que no existe *churn* en la red P2P.

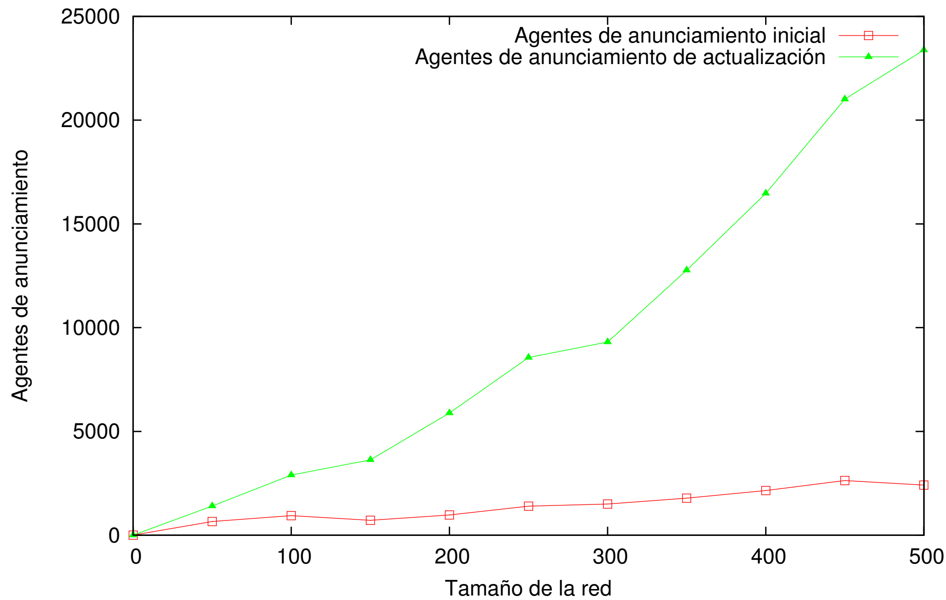


Figura 6.4: Agentes de anunciamiento vs el tamaño de la red.

6.2. Resultados de la fase de selección de recursos

Las consultas de recursos dentro de *RASupport* permiten especificar opciones, mediante la etiqueta `<option>`, que determinan el protocolo a utilizar en la fase de selección: encontrar recursos imperativamente (*requires:find_resources*) o sacrificar convergencia de resultados para obtener rendimiento (*requires:performance*). En el primer caso, *SelectionAPI* emplea el protocolo de inundación con agentes de consulta, mientras que en el segundo caso emplea el protocolo de paseos aleatorios inteligentes (*iRandomWalks*). Ambos protocolos fueron descritos en el capítulo 4.

iRandomWalks almacena estadísticas en una tabla *hash* concurrente (i.e., una instancia de la clase *RASupportMap*) en donde cada entrada contiene: un objeto de la clase *RASupportQueryGroup* (llave) y una lista de objetos de la clase *MycoNode* (valor). Un grupo de atributos requeridos (i.e., una instancia de la clase *RASupportQueryGroup*) puede ser representado por una cadena de caracteres; sin embargo, una métrica de similitud de cadenas de caracteres es insuficiente para determinar la similitud que existe entre dos grupos de atributos requeridos, debido a que estas métricas no toman en cuenta la semántica de los requerimientos ni las restricciones especificadas para dichos atributos (i.e., no toman

en cuenta los valores permitidos ni los valores de penalización especificados en una consulta de recursos). Con la finalidad de determinar si el algoritmo de similitud entre grupos de atributos requeridos, presentado en el capítulo 4, se desempeña mejor y encuentra un porcentaje de similitud más certero que los principales algoritmos de similitud de cadenas de caracteres (i.e., la distancia de Levenshtein [47], la distancia de Jaro-Winkler [48] y el algoritmo de Ian Oliver [52])[4] se llevaron a cabo tres experimentos.

El primer experimento consta de 500 iteraciones y emplea los grupos de atributos requeridos que se muestran en el listado 6.2. En la figura 6.5, se muestran los resultados obtenidos a partir de este experimento, en donde se observa que el algoritmo propuesto supera considerablemente a los demás algoritmos en términos de rendimiento.

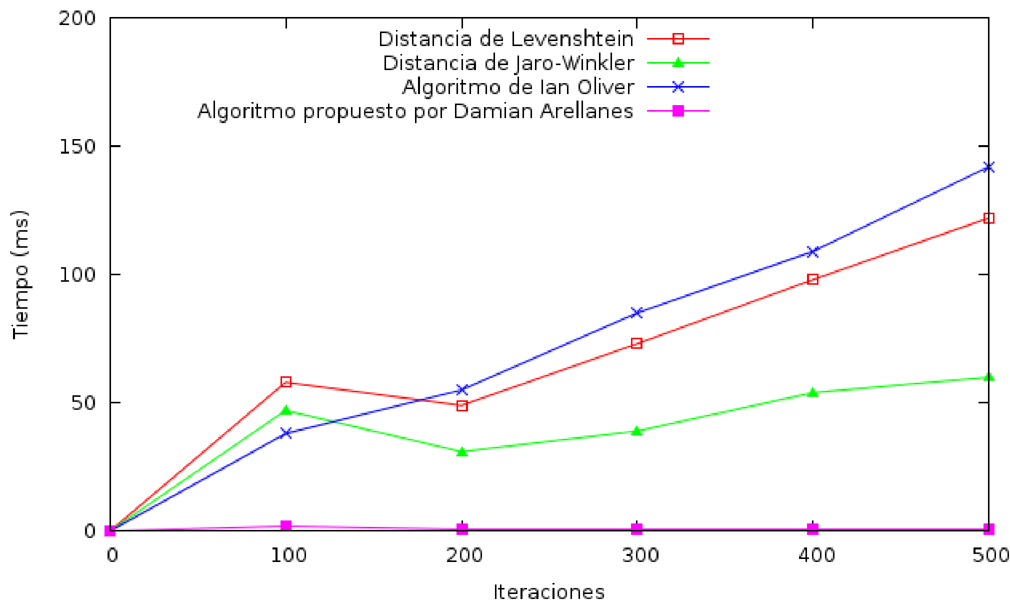


Figura 6.5: Análisis de rendimiento de los algoritmos que permiten determinar la similitud que existe entre dos grupos de atributos requeridos.

Listing 6.2: Grupos de atributos requeridos empleados en los primeros dos experimentos.

```
<group>
        <name>Processing</name>
        <num_nodes>8</num_nodes>
        <os_name>Linux,0.0</os_name>
        <busy_cpu>0.0,0.0,30.0,100.0,0.05</busy_cpu>
        <free_mem>10.0,1024.0,4096.0,16000.0,0.05</free_mem>
```

[4]La diferencia entre el algoritmo propuesto y los otros algoritmos (i.e., la distancia de Levenshtein, la distancia de Jaro-Winkler y el algoritmo de Ian Oliver) radica en que el primero toma en cuenta los tipos de atributos requeridos (i.e., si son numéricos o *string*) mientras que los otros algoritmos no lo hacen, ya que únicamente se enfocan en determinar qué tanto se parecen las cadenas de caracteres que representan a dos grupos de atributos requeridos.

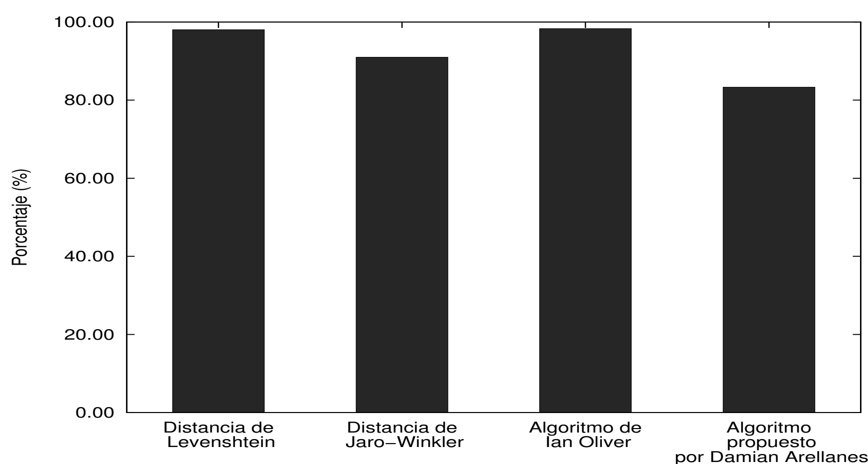
```
        <cpu_speed>500.0,2500.0,4096.0,5000.0,0.2</cpu_speed>
        <free_hdisk>0.0,60.5,92.5,1000000.0,0.05</free_hdisk>
        <cores>1,2,4,8,0.03</cores>
        <rest_betw_nodes>
                <bandwidth>10.199999809265137,22.100000381469727,30.0,50.0,0.02</bandwidth>
                <latency>10.199999809265137,22.100000381469727,30.0,50.0,0.02</latency>
        </rest_betw_nodes>
</group>
<group>
        <name>Processing</name>
        <num_nodes>8</num_nodes>
        <os_name>Windows,0.0</os_name>
        <busy_cpu>0.0,0.0,30.0,100.0,0.05</busy_cpu>
        <free_mem>10.0,1024.0,4096.0,16000.0,0.05</free_mem>
        <cpu_speed>500.0,2500.0,4096.0,5000.0,0.2</cpu_speed>
        <free_hdisk>0.0,60.5,92.5,1000000.0,0.05</free_hdisk>
        <cores>1,2,4,8,0.03</cores>
</group>
```

Listing 6.3: Grupos de atributos requeridos empleados en el tercer experimento.

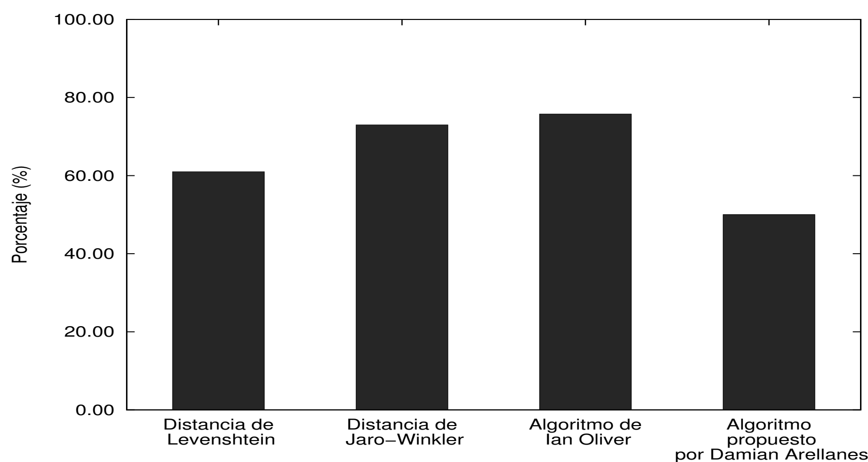
```
<group>
        <name>Processing</name>
        <num_nodes>8</num_nodes>
        <free_mem>10.0,1024.0,4096.0,16000.0,0.05</free_mem>
        <cpu_speed>500.0,2500.0,4096.0,5000.0,0.2</cpu_speed>
        <free_hdisk>0.0,60.5,92.5,1000000.0,0.05</free_hdisk>
        <rest_betw_nodes>
                <bandwidth>10.199999809265137,22.100000381469727,30.0,50.0,0.02</bandwidth>
                <latency>10.199999809265137,22.100000381469727,30.0,50.0,0.02</latency>
        </rest_betw_nodes>
</group>
<group>
        <name>Processing</name>
        <num_nodes>8</num_nodes>
        <busy_cpu>0.0,0.0,30.0,100.0,0.05</busy_cpu>
        <free_mem>10.0,1024.0,4096.0,16000.0,0.05</free_mem>
        <cpu_speed>500.0,2500.0,4096.0,5000.0,0.2</cpu_speed>
        <os_name>Windows,0.0</os_name>
        <cores>1,2,4,8,0.03</cores>
        <free_hdisk>0.0,60.5,92.5,1000000.0,0.05</free_hdisk>
</group>
```

Los siguientes dos experimentos se llevaron a cabo en una sola iteración, debido a que se enfocan en determinar el porcentaje de similitud que arrojan los algoritmos analizados. En la la figura 6.6 (a), se muestran los resultados del segundo experimento, en el cual se comparan los grupos de atributos requeridos del listado 6.2. En esa figura, se observa que el algoritmo propuesto logra un porcentaje de similitud equivalente al 83.33 %, mientras que el porcentaje de similitud obtenido por los otros tres algoritmos se aproxima al 100 %. Ambos grupos del listado 6.2 cuentan con 6 atributos, sin embargo, difieren en el sistema operativo (`<os_name>`); de esta manera, $\frac{5}{6}$ de los atributos coinciden (i.e., existe

un porcentaje de similitud del 83.33 % entre ambos grupos). En la figura 6.6 (b), se muestran los resultados del tercer experimento, en el cual se comparan los grupos de atributos requeridos del listado 6.3. En esa figura, se observa que el algoritmo propuesto arroja un porcentaje de similitud del 50 %, lo cual es razonable debido a que ambos grupos coinciden en exactamente la mitad de atributos (`<free_mem>`, `<cpu_speed>` y `<free_hdisk>`); mientras que los otros tres algoritmos obtienen porcentajes de similitud que oscilan entre el 60 % y el 75 %.



(a) Porcentaje de similitud entre los grupos de atributos requeridos del listado 6.2.



(b) Porcentaje de similitud entre los grupos de atributos requeridos del listado 6.3.

Figura 6.6: Correctitud del algoritmo propuesto que permite determinar la similitud entre grupos de atributos requeridos.

A partir de los resultados obtenidos en los primeros tres experimentos, se concluye que el algoritmo propuesto ofrece un mejor rendimiento y es más certero que los principales algoritmos que permiten determinar la similitud entre grupos de atributos requeridos.

Por lo tanto, el protocolo *iRandomWalks* emplea el algoritmo de similitud entre grupos propuesto en la presente tesis.

Para validar el funcionamiento de *SelectionAPI*, se llevó a cabo un experimento en una red P2P de 50 *peers*, como la que se ilustra en la figura 6.7. En este experimento, el *peer normal* NP_{19} emite una consulta de recursos como la que se muestra en el listado 6.4. Para ello, el simulador de *RASupport* proporciona una interfaz gráfica, la cual permite seleccionar el archivo XML que especifica la consulta de recursos (ver figura 6.8).

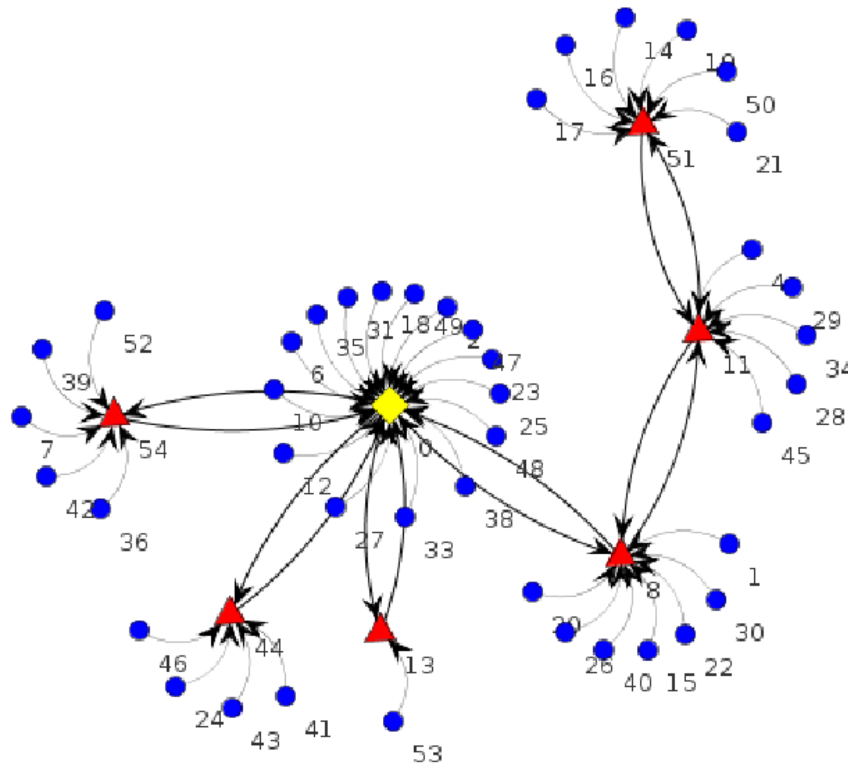


Figura 6.7: Red P2P inicial del experimento que valida el funcionamiento de *SelectionAPI*.

Listing 6.4: Consulta de recursos con un TTL de 10, la cual requiere desempeño (i.e., la ejecución del protocolo iRandomWalks) y un grupo de procesamiento compuesto por ocho nodos que satisfagan cuatro atributos y dos restricciones entre nodos.

```
<?xml version="1.0"?>
<query>
        <option>requires:performance</option>
        <ttl>10</ttl>
        <group>
                <name>Processing</name>
                <num_nodes>8</num_nodes>
                <cpu_speed>500.0,2500.0,4096.0,5000.0,0.2</cpu_speed>
                <free_mem>10.0,1024.0,4096.0,16000.0,0.05</free_mem>
```

```
            <free_hdisk>0.0,60.5,92.5,1000000.0,0.05</free_hdisk>
            <rest_betw_nodes>
                  <latency>10.2,22.1,30.0,50.0,0.02</latency>
                  <bandwidth>10.2,22.1,30.0,50.0,0.02</bandwidth>
            </rest_betw_nodes>
      </group>
</query>
```

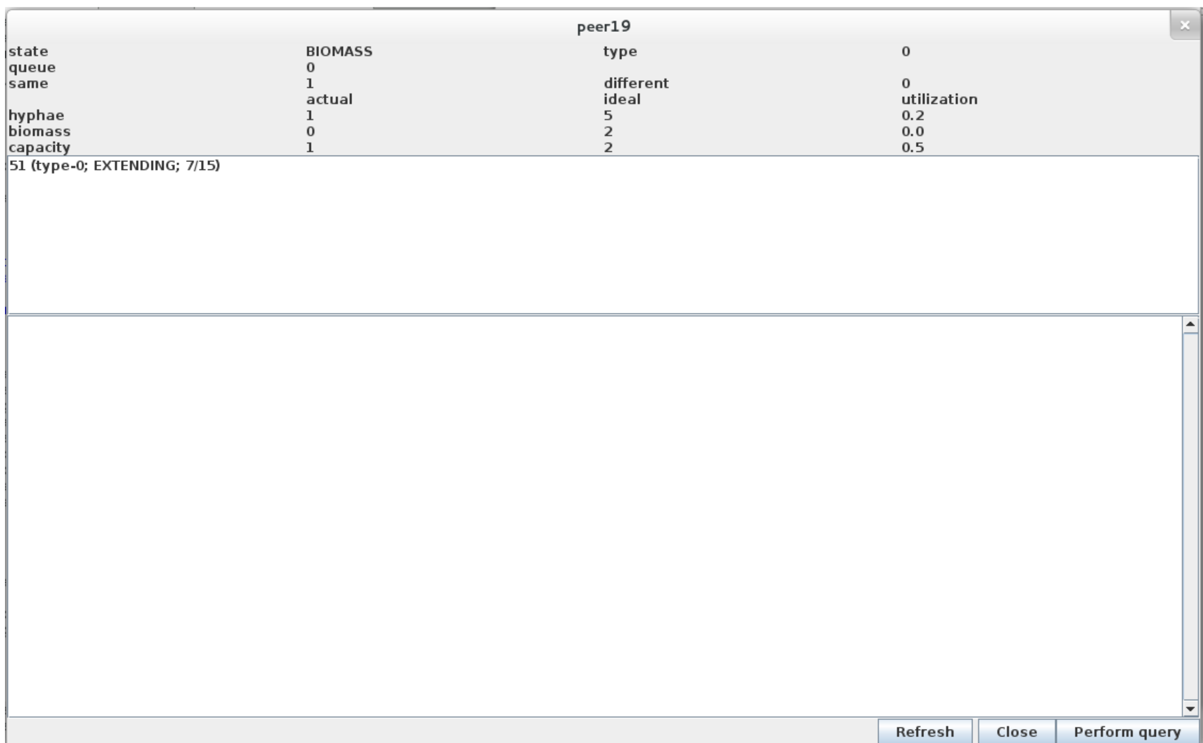


Figura 6.8: Interfaz gráfica para emitir una consulta de recursos en el simulador de *RA-Support*.

En el recuadro verde de la figura 6.9, se observa que NP_{19} envía una consulta de recursos (cuyo identificador es -1121016612) a SP_{51} (i.e., $SP_{initiator} = SP_{51}$), el cual ejecuta el protocolo *iRandomWalks* debido a que dicha consulta requiere desempeño. $SP_{initiator}$ no posee entradas en su tabla de estadísticas, por lo que intenta enviar agentes de consulta a todos sus vecinos (ver recuadro azul de la figura 6.9). Posteriormente, los *super-peers* que reciben un agente de consulta lo reenvían a uno y solo un vecino *super-peer* seleccionado aleatoriamente (ver recuadro café de la figura 6.9); por lo tanto, el agente de consulta sigue la ruta $SP_{51} \rightarrow SP_{11} \rightarrow SP_8 \rightarrow SP_0 \rightarrow SP_{44}$ (i.e., $VN_c = \{SP_{51}, SP_{11}, SP_8, SP_0, SP_{44}\}$). En los recuadros negros de la figura 6.9, se observa que algunos *super-peers* no envían agentes de consulta debido a que el *super-peer* al que intentan enviar se encuentra en la lista de exclusión, e.g., SP_8 no puede enviar un agente de consulta a SP_{11}. Al llegar a SP_{44}, el agente de consulta aún no ha expirado su TTL; sin embargo, ya no encuentra algún *super-peer* vecino al cual visitar, por lo que el agente de consulta regresa sus resultados

a $SP_{initiator}$ (ver recuadros morados de la figura 6.9). La consulta de recursos del listado 6.4 especifica los mismos límites que *ResourceSim* para cada atributo, de manera que los resultados obtenidos por *selectionAPI* están compuestos por todos los *peers normales* (incluyendo al *super-peer*) de cada $SP_v \in VN_c$ (ver figura 6.10). En la figura 6.11, se muestra el resultado de una segunda ejecución del protocolo *iRandomWalks*, empleando la consulta de recursos del listado 6.4, en donde se observa que esta vez la tabla de estadísticas de SP_{51} posee entradas (con los *super-peers* de la ejecución anterior, i.e., VN_c) cuyas llaves (i.e., grupos de atributos requeridos) poseen una similitud mayor al 75 % con respecto al grupo especificado en el listado 6.4. Por lo tanto, $SP_{initiator}$ envía directamente agentes de consulta a cada elemento de VN_c y, finalmente, ejecuta el protocolo de paseos aleatorios con la modificación propuesta en el capítulo 4.

```
Output - RASupport (run)
CDSimulator: cycle 1 done
myconet.StateObserver (execute): MYCOCAST STATS: 43 biomass, 6 extending, 1 branching, 0 immobile, 0 bulwark.
nov 24, 2014 12:38:59 PM myconet.StateObserver execute
INFORMACIÓN: MYCOCAST STATS: 43 biomass, 6 extending, 1 branching, 0 immobile, 0 bulwark.
RASupport: peer19->peer19 has requested resources
RASupport: queryID -> -1121016612
RASupport: peer51->peer19 has requested resources
RASupport: queryID -> -1121016612
GROUP Processing:
iRandomWalk executed!
SP peer51 sends to all neighbors
SP peer51 tries to peer11
RASupport: SUPER-PEER peer11 has received the query -1121016612
GROUP Processing:
SP peer11 sends to 1 neighbors
SP peer11 tries to peer8
RASupport: SUPER-PEER peer8 has received the query -1121016612
GROUP Processing:
SP peer8 sends to 1 neighbors
SP peer8 tries to peer11
SP peer8 tries to peer0
RASupport: SUPER-PEER peer0 has received the query -1121016612
GROUP Processing:
SP peer0 sends to 1 neighbors
SP peer0 tries to peer44
RASupport: SUPER-PEER peer44 has received the query -1121016612
GROUP Processing:
SP peer44 sends to 1 neighbors
SP peer44 tries to peer0
SP peer44 returns a query agent
RASupport: SUPER-PEER peer51 has received a result set for the query -1121016612
Adding result from query agent received...
IMPOSSIBLE TO SEND TO SUPER-PEER peer11 -> exclusion list
IMPOSSIBLE TO SEND TO SUPER-PEER peer0 -> exclusion list
SP_initiator has received all query agents
```

Figura 6.9: Mensajes obtenidos por *RASupport* durante la ejecución del protocolo de paseos aleatorios inteligentes (*iRandomWalks*).

```
------------------------------------------------------------------------
SP_visited = peer0
Candidate peers = Group: Processing -> [0 (type-0; BRANCHING; 19/15), 47 (type-0; BIOMASS; 1/4), 2 (type-0; BIOMASS; 1/3), 38 (typ
------------------------------------------------------------------------
------------------------------------------------------------------------
SP_visited = peer8
Candidate peers = Group: Processing -> [8 (type-0; EXTENDING; 9/9), 30 (type-0; BIOMASS; 1/3), 1 (type-0; BIOMASS; 1/2), 15 (type-
------------------------------------------------------------------------
------------------------------------------------------------------------
SP_visited = peer11
Candidate peers = Group: Processing -> [11 (type-0; EXTENDING; 7/9), 28 (type-0; BIOMASS; 1/5), 29 (type-0; BIOMASS; 1/1), 4 (type
------------------------------------------------------------------------
------------------------------------------------------------------------
SP_visited = peer44
Candidate peers = Group: Processing -> [44 (type-0; EXTENDING; 5/13), 24 (type-0; BIOMASS; 1/9), 41 (type-0; BIOMASS; 1/7), 46 (ty
------------------------------------------------------------------------
------------------------------------------------------------------------
SP_visited = peer51
Candidate peers = Group: Processing -> [51 (type-0; EXTENDING; 7/15), 17 (type-0; BIOMASS; 1/7), 21 (type-0; BIOMASS; 1/3), 16 (ty
------------------------------------------------------------------------
```

Figura 6.10: Resultados de la primera ejecución del protocolo de paseos aleatorios inteligentes (*iRandomWalks*).

Figura 6.11: Resultados de la segunda ejecución del protocolo de paseos aleatorios inteligentes (*iRandomWalks*).

6.3. Resultados de la fase de empatamiento de recursos

En esta sección, se presenta un experimento que valida el funcionamiento de *MatchingAPI*, la cual emplea el mecanismo propuesto en el capítulo 4. Inicialmente se tiene una red P2P con 10 *peers* (el estado de la red P2P es irrelevante para este experimento), una consulta de recursos como la que se muestra en el listado 6.5, dos restricciones entre nodos para el grupo *ProcessingGroup* (i.e., $m_{ProcessingGroup} = 2$) y una lista inicial para este grupo compuesta por cuatro *peers* candidatos (i.e., $n_{ProcessingGroup} = 4$): $I_{ProcessingGroup} = \{NP_{30}, NP_{52}, NP_{51}, NP_{53}\}$ (ver recuadro verde de la figura 6.12). La lista de referencia de *peers* que pueden trabajar entre sí en el grupo *ProcessingGroup* está conformada por cuatro combinaciones: $K_{ProcessingGroup} = \{[NP_{52}, NP_{53}], [NP_{30}, NP_{52}], [NP_{30}, NP_{53}]$ $[NP_{52}, NP_{51}]\}$ (ver recuadro azul de la figura 6.12). La lista de referencia de *peers* que no pueden trabajar entre sí en el grupo *ProcessingGroup* está conformada por dos combinaciones: $M_{ProcessingGroup} = \{[NP_{51}, NP_{53}], [NP_{30}, NP_{51}]\}$ (ver recuadro amarillo de la figura 6.12). Por lo tanto, $\mathbf{card}(K_{ProcessingGroup}) + \mathbf{card}(M_{ProcessingGroup}) = 4 + 2 = 6 = \frac{n^2_{ProcessingGroup} - n_{ProcessingGroup}}{2}$ demuestra que el teorema 1 se cumple. En la etapa de empatamiento de *peers* candidatos, *MatchingAPI* determina cinco grupos candidatos (ver recuadro rojo de la figura 6.12): $L_{ProcessingGroup} = \{\{NP_{30}, NP_{52}, NP_{53}\}, \{NP_{30}, NP_{52}\}, \{NP_{52}, NP_{53}\},$ $\{NP_{52}, NP_{51}\}, \{NP_{30}, NP_{53}\}\}$. En el recuadro violeta de la figura 6.12 se observa que, en la etapa de empatamiento de grupos candidato, *MatchingAPI* determina que el grupo $\{NP_{30}, NP_{52}, NP_{53}\}$ es aquel que posee la menor penalización entre grupos ($P_{groups} =$

525.5); por lo tanto, es el grupo óptimo para *ProcessingGroup*. Adicionalmente, en el recuadro café de la figura 6.12 se observa que se envían 12 agentes de subempatamiento, cumpliendo así el teorema 2: $\frac{n^2_{ProcessingGroup} - n_{ProcessingGroup}}{2} \cdot m_{ProcessingGroup} = 6 \cdot 2 = 12$.

Listing 6.5: Consulta de recursos del experimento que valida el funcionamiento de *MatchingAPI*.

```
<?xml version="1.0" encoding="utf-8"?>
<query>
	<option>requires:find_resources</option>
	<ttl>3</ttl>
	<group>
		<name>ProcessingGroup</name>
		<num_nodes>5</num_nodes>
		<busy_cpu>25.5,30.0,40.0,75.0,0.05</busy_cpu>
		<cores>1,2,4,4,0.03</cores>
		<rest_betw_nodes>
			<latency>0.0,0.0,25.0,50.0,0.5</latency>
			<bandwidth>0.0,0.0,50.0,100.0,0.3</bandwidth>
		</rest_betw_nodes>
	</group>
	<group>
		<name>StorageGroup</name>
		<num_nodes>4</num_nodes>
		<free_hdisk>50.0,60.5,92.5,100.0,0.05</free_hdisk>
		<rest_betw_nodes>
			<latency>0.0,0.0,25.0,50.0,0.5</latency>
			<bandwidth>0.0,0.0,50.0,100.0,0.3</bandwidth>
		</rest_betw_nodes>
	</group>
	<rest_betw_groups>
		<group_names>ProcessingGroup,StorageGroup</group_names>
		<latency>0.0,0.0,50.0,100.0,0.05</latency>
	</rest_betw_groups>
</query>
```

6.4. Resultados de la fase de enlace de recursos

El experimento que valida el funcionamiento de *BindingAPI* posee la siguiente configuración global:

```
Output ×
Debugger Console ×   RASupport (run) ×
GROUP StorageGroup:
GROUP ProcessingGroup:
RASupport: SUPER-PEER peer50 has received a result set for the query 1896399665
Adding result from query agent received...
RASupport: SUPER-PEER peer50 has received a result set for the query 1896399665
Adding result from query agent received...
SP_initiator has received all query agents
---------------------------------------------------------------------------
*******************SELECTION PHASE RESULTS*******************
---------------------------------------------------------------------------
Group(4 peers) = ProcessingGroup
Candidate peers = [30 (type-0; BIOMASS; 1/15), 52 (type-0; BIOMASS; 1/15), 51 (type-0; BIOMASS; 1/15), 53 (type-0; BIOMASS; 1/15
---------------------------------------------------------------------------
*******************MATCHING PHASE RESULTS*******************
K_g(4): {CombinatoricsVector=([52 (type-0; BIOMASS; 1/15), 53 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([30 (
M_g(2): {CombinatoricsVector=([51 (type-0; BIOMASS; 1/15), 53 (type-0; BIOMASS; 1/15)], size=2)=Infinity, CombinatoricsVector=([
Sent 12 sub-matching agents
---------------------------------------------------------------------------
FIRST PHASE -> CANDIDATE GROUPS
Group: ProcessingGroup
Candidate groups(5): [CombinatoricsVector=([30 (type-0; BIOMASS; 1/15), 52 (type-0; BIOMASS; 1/15), 53 (type-0; BIOMASS; 1/15)],
---------------------------------------------------------------------------
SECOND PHASE -> BEST GROUPS
Group: ProcessingGroup
peers(3): [30 (type-0; BIOMASS; 1/15), 52 (type-0; BIOMASS; 1/15), 53 (type-0; BIOMASS; 1/15)]
```

Figura 6.12: Resultados del experimento que valida el funcionamiento de *MatchingAPI*.

- Tamaño de la red: 25 *peers* (ver figura 6.13).
- Duración del experimento: 50 ciclos.
- Duración de cada ciclo: 500 milisegundos.
- Probabilidad de que un *super-peer* en un estado inmóvil promueva a alguno de sus *peers normales* en cada ciclo: 0.1.

En este experimento, NP_{168} emite una consulta de recursos (i.e., $SP_{initiator} = SP_{130}$) como la que se muestra en el listado 6.6; por lo tanto, NP_{168} es el *peer* consumidor pr_{168}. Inicialmente, en la fase de empatamiento de recursos se obtuvo únicamente un grupo compuesto por ocho *peers* proveedores para $America$: $b_{America} = \{pn_{164}, pn_{166}, pn_{169}, pn_{150}, pn_{147}, pn_{108}, pn_{143}, pn_{157}\}$ (ver recuadro violeta de la figura 6.14). En la figura 6.13, se puede observar que el *super-peer* SP_{130} tiene a su cargo a los *peers* proveedores pn_{164}, pn_{166} y pn_{169}, mientras que el *super-peer* SP_{121} tiene a su cargo a los *peers* proveedores pn_{150}, pn_{147}, pn_{108}, pn_{143} y pn_{157}. Por lo tanto $SP_{initiator}$ envía dos agentes de enlace: uno al *super-peer* SP_{130} y otro al *super-peer* SP_{121} (ver recuadro azul de la figura 6.14). En la figura 6.14 se muestra el flujo de mensajes de *BindingAPI* para llevar a cabo la fase de enlace, en donde se observa que los recursos de los *peers* proveedores pn_{166} y pn_{157} para el grupo $America$ no están disponibles, por lo que dichos *peers* son eliminados de $b_{America}$ (ver recuadros rojos de la figura 6.14). De esta manera, el resultado final de *BindingAPI* es un grupo de seis *peers* proveedores para el grupo $America$: $b_{America} = \{pn_{164}, pn_{169}, pn_{150}, pn_{147}, pn_{108}, pn_{143}\}$ (ver recuadro verde de la figura 6.14).

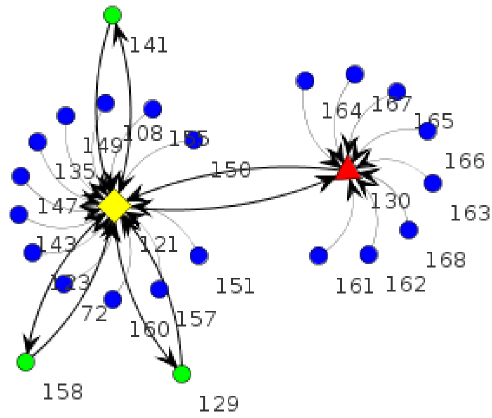


Figura 6.13: Estado inicial de la red P2P del experimento que valida el funcionamiento de *BindingAPI*.

Listing 6.6: Consulta de recursos del experimento que valida el funcionamiento de *BindingAPI*.

```
<?xml version="1.0" encoding="utf-8"?>
<query>
        <option>requires:performance</option>
        <ttl>2</ttl>
        <group>
                <name>America</name>
                <num_nodes>2</num_nodes>
                <busy_cpu>0.0,25.5,40.0,90.0,0.05</busy_cpu>
                <cores>1,3,4,4,0.03</cores>
                <rest_betw_nodes>
                        <latency>0.0,0.0,25.0,50.0,0.5</latency>
                </rest_betw_nodes>
        </group>
        <group>
                <name>Europe</name>
                <num_nodes>5</num_nodes>
                <busy_cpu>0.0,25.5,40.0,90.0,0.05</busy_cpu>
                <free_hdisk>0.0,60.5,100.0,200.0,0.2</free_hdisk>
                <cores>1,2,4,4,0.01</cores>
        </group>
        <rest_betw_groups>
                <group_names>America,Europe</group_names>
                <latency>0.0,0.0,50.0,100.0,0.05</latency>
        </rest_betw_groups>
</query>
```

```
Output - RASupport (run)
)=12.5, CombinatoricsVector=([171 (type-0; BIOMASS; 1/15), 147 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([171 (type-0; BIOMASS; 1/15), 169 (t
ype-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([143 (type-0; BIOMASS; 1/15), 157 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([147 (
type-0; BIOMASS; 1/15), 108 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([150 (type-0; BIOMASS; 1/15), 108 (type-0; BIOMASS; 1/15)], size=2)=12.
5, CombinatoricsVector=([150 (type-0; BIOMASS; 1/15), 143 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([150 (type-0; BIOMASS; 1/15), 147 (type-0
; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([150 (type-0; BIOMASS; 1/15), 157 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([121 (type-
0; BRANCHING; 16/15), 150 (type-0; BIOMASS; 1/15)], size=2)=12.5, CombinatoricsVector=([121 (type-0; BRANCHING; 16/15), 157 (type-0; BIOMASS; 1/15)], size=2)=12
.5, CombinatoricsVector=([157 (type-0; BIOMASS; 1/15), 162 (type-0; BIOMASS; 1/15)], size=2)=12.5]
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
SECOND PHASE -> BEST GROUPS
-----------------------------------------------------------------------------
Group: America
peers(8): [164 (type-0; BIOMASS; 1/15), 166 (type-0; BIOMASS; 1/15), 169 (type-0; BIOMASS; 1/15), 150 (type-0; BIOMASS; 1/15), 147 (type-0; BIOMASS; 1/15), 108
(type-0; BIOMASS; 1/15), 143 (type-0; BIOMASS; 1/15), 157 (type-0; BIOMASS; 1/15)]
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
*******************BINDING PHASE RESULTS*******************

Binding agent to SP 130 (type-0; EXTENDING; 9/15)
Providers: [164 (type-0; BIOMASS; 1/15), 166 (type-0; BIOMASS; 1/15), 169 (type-0; BIOMASS; 1/15)]
-----------------------------------------------------------------------------
Binding agent to SP 121 (type-0; BRANCHING; 16/15)
Providers: [150 (type-0; BIOMASS; 1/15), 147 (type-0; BIOMASS; 1/15), 108 (type-0; BIOMASS; 1/15), 143 (type-0; BIOMASS; 1/15), 157 (type-0; BIOMASS; 1/15)]

peer130 has received a binding agent!
Requesting resources to provider 164 (type-0; BIOMASS; 1/15)
Provider 164 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP America!
Requesting resources to provider 166 (type-0; BIOMASS; 1/15)
Provider 166 (type-0; BIOMASS; 1/15) has REJECTED the use of its resources in GROUP America (provider peer has been REMOVED from that group)
Requesting resources to provider 169 (type-0; BIOMASS; 1/15)
Provider 169 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP America!
peer168 has received results from a binding agent! (1/2)
peer121 has received a binding agent!
Requesting resources to provider 150 (type-0; BIOMASS; 1/15)
Provider 150 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP America!
Requesting resources to provider 147 (type-0; BIOMASS; 1/15)
Provider 147 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP America!
Requesting resources to provider 108 (type-0; BIOMASS; 1/15)
Provider 108 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP America!
Requesting resources to provider 143 (type-0; BIOMASS; 1/15)
Provider 143 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP America!
Requesting resources to provider 157 (type-0; BIOMASS; 1/15)
Provider 157 (type-0; BIOMASS; 1/15) has REJECTED the use of its resources in GROUP America (provider peer has been REMOVED from that group)
peer168 has received results from a binding agent! (2/2)

Final Results:
Group: America
peers(6): [164 (type-0; BIOMASS; 1/15), 169 (type-0; BIOMASS; 1/15), 150 (type-0; BIOMASS; 1/15), 147 (type-0; BIOMASS; 1/15), 108 (type-0; BIOMASS; 1/15), 143
(type-0; BIOMASS; 1/15)]
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
```

Figura 6.14: Resultados del experimento que valida el funcionamiento de *BindingAPI*.

6.5. Caso de estudio: sistema P2P colaborativo para la generación del Conjunto de Mandelbrot

Los fractales son objetos geométricos cuya estructura fundamental se puede encontrar a diferentes escalas del mismo. El término fue acuñado por Benoit B. Mandelbrot en 1975 y deriva del adjetivo en Latín *fractus*, que significa quebrado o irregular. Muchos patrones que se presentan en la Naturaleza son irregulares, infinitos y están fragmentados, por lo que estos pueden ser representados empleando geometría fractal [15]. La generación de fractales tiene requerimientos especiales de procesamiento, almacenamiento y despliegue. Comúnmente, los fractales son desplegados y procesados en una única computadora; sin embargo, la generación de los mismos se puede llevar a cabo mediante la colaboración de diversos recursos distribuidos alrededor de múltiples computadoras. De esta manera, un sistema P2P colaborativo representa una opción para generar fractales mediante la colaboración de recursos. Los fractales poseen tres características principales [16]:

1. No se pueden describir con geometría euclidiana debido a que son muy irregulares.

2. Poseen autosemejanza[5], por lo que los fractales son recursivos y paralelizables.

[5]La autosemejanza es la simetría que existe dentro de una escala.

3. Se originan a partir de reglas iniciales que dan lugar a figuras complejas.

El conjunto de Mandelbrot es el fractal más estudiado en la actualidad y tiene la principal característica de que cada punto calculado es completamente independiente de los demás [16]. Debido a lo anterior, se determinó que la generación del conjunto de Mandelbrot es el problema ideal para desarrollar un sistema P2P colaborativo, mediante la distribución del algoritmo que permite la generación de dicho fractal. El desarrollo de este sistema se auxilió de *RASupport* para agregar recursos en una red P2P basada en *super-peers*. La definición formal del conjunto de Mandelbrot es la siguiente:

Definición 23. *Sea D un número complejo cualquiera. A partir de D, se construye una sucesión por recursión.*

$$\begin{cases} Z_0 = 0 & \text{(término inicial)} \\ Z_{n+1} = Z_n^2 + D & \text{(relación de inducción)} \end{cases} \tag{6.1}$$

6.5.1. Análisis del sistema P2P colaborativo de validación

El sistema P2P colaborativo de validación considera tres tipos de nodos: un nodo maestro, nodos de procesamiento y un nodo despliegue. El nodo maestro es el consumidor de recursos que utiliza la fachada de *RASupport* para determinar cuál es el mejor nodo de despliegue y cuáles son los mejores nodos de procesamiento de acuerdo a una consulta de recursos; adicionalmente, el nodo maestro se encarga de distribuir tareas a los nodos de procesamiento. Los nodos de procesamiento tienen la finalidad de procesar un fragmento del conjunto de Mandelbrot y de comunicar sus resultados al nodo de despliegue. El nodo de despliegue se encarga de reunir los resultados de los nodos de procesamiento y de mostrar en pantalla el conjunto de Mandelbrot. El nodo maestro particiona el plano x, y en filas y distribuye tareas de la siguiente forma:

Definición 24. *Sea N el número de nodos de procesamiento, M un nodo de procesamiento cualquiera, W el ancho del* frame *del nodo de despliegue, H la altura del* frame *del nodo de despliegue, C_M el número de columnas asignadas a M y R_M el número de filas asignadas a M.*

$$C_M = W \forall M \in [1, N] \tag{6.2}$$

$$R_M = \begin{cases} \frac{H}{N} & \text{si } 1 \leq M < N \\ \frac{H}{N} + (H \bmod N) & \text{si } M = N \end{cases} \tag{6.3}$$

Los nodos de procesamiento emplean el algoritmo de secuencia de escape para procesar el rango que les fue asignado de acuerdo a las ecuaciones 6.2 y 6.3. Dicho algoritmo se

describe a continuación:

Algoritmo 7 Algoritmo de secuencia de escape

```
1: procedure COMPUTEMANDELBROT(xmin, xmax, ymin, ymax)
2:     for row ← ymin,row < ymax, row ← row + 1 do
3:         for column ← xmin,column < xmax,column ← column + 1 do
4:             z ← 0
5:             for i ← maxiterations,i > 0 AND (SQRT(z^2)≤ escape),i ← i − 1 do
6:                 z ← z^2 + c
7:             end for
8:             if i mód 2 = 0 then
9:                 pixels[row, column] ← mandelbrotColor
10:            else
11:                pixels[row, column] ← nonMandelbrotColor
12:            end if
13:        end for
14:    end for
       return pixels
15: end procedure
```

RASupport juega un papel importante en el sistema P2P colaborativo de validación, ya que permite llevar a cabo las fases de la agregación de recursos de una manera transparente al cliente. En general, las fases de la colaboración de recursos se llevan a cabo de la siguiente manera dentro del sistema P2P colaborativo de validación:

1. Anunciamiento: *RASupport* anuncia los recursos de cada *peer*, haciendo uso de una o más Especificación de recursos (RSs) y empleando el mecanismo propuesto en el capítulo 4.

2. Descubrimiento: esta fase no es necesaria, ya que *RASupport* anuncia recursos explícitamente a los *super-peers*.

3. Selección: *RASupport* determina cuál es el mejor *peer* de despliegue y cuáles son los mejores *peers* de procesamiento que se pueden tener a partir de los existentes en la red P2P, empleando cualquiera de los protocolos propuestos en el capítulo 4. La consulta de recursos es especificada por el *peer* maestro, mediante un documento XML.

4. Empatamiento: *RASupport* determina los grupos de *peers* que mejor satisfacen los requerimientos y las restricciones especificadas en cada grupo solicitado en la consulta de recursos.

5. Enlace: *RASupport* determina si los recursos de los *peers* proveedores, que cumplen con los requerimientos y las restricciones del *peer* maestro, están disponibles para

su uso. En caso de ser así, *RASupport* establece un enlace entre el *peer* maestro (consumidor) y los *peers* proveedores (i.e., los *peers* de procesamiento y los de despliegue). Los recursos de los *peers* de procesamiento se enlazan con los recursos del *peer* de despliegue. En general, *RASupport* determina si los recursos de los *peers* proveedores están disponibles para ser usados en los grupos de procesamiento y de despliegue, respectivamente.

6. Uso: el *peer* maestro utiliza *RASupport* para solicitar 10 *peers* de procesamiento y uno de despliegue. Posteriormente, solicita la creación del *frame* al *peer* de despliegue, mediante el envío de un mensaje *createDisplay*. Consecutivamente, el *peer* maestro emplea las ecuaciones 6.2 y 6.3 para distribuir tareas a los *peers* de procesamiento, mediante el envío de mensajes *startProcessing*. Una vez que un *peer* de procesamiento recibe el mensaje *startProcessing*, ejecuta el algoritmo 7 para procesar la región del plano que le fue asignada. Finalmente, los *peers* de procesamiento envían sus resultados al *peer* de despliegue —mediante mensajes *receiveProcessingResults*— para que este último tipo de *peer* recolecte resultados y, una vez que haya recibido resultados de todos los *peers* de procesamiento, muestre el conjunto de Mandelbrot en su *frame*. Este proceso se ilustra en la figura 6.15.

7. Liberación: los recursos de los *peers* de procesamiento dejan de ser utilizados una vez que estos han procesado la región del conjunto de Mandelbrot que les fue asignada y que han enviado sus resultados al *peer* de despliegue. Los recursos del *peer* de despliegue dejan de ser utilizados una vez que se destruye el hilo del *frame* que muestra el conjunto de Mandelbrot.

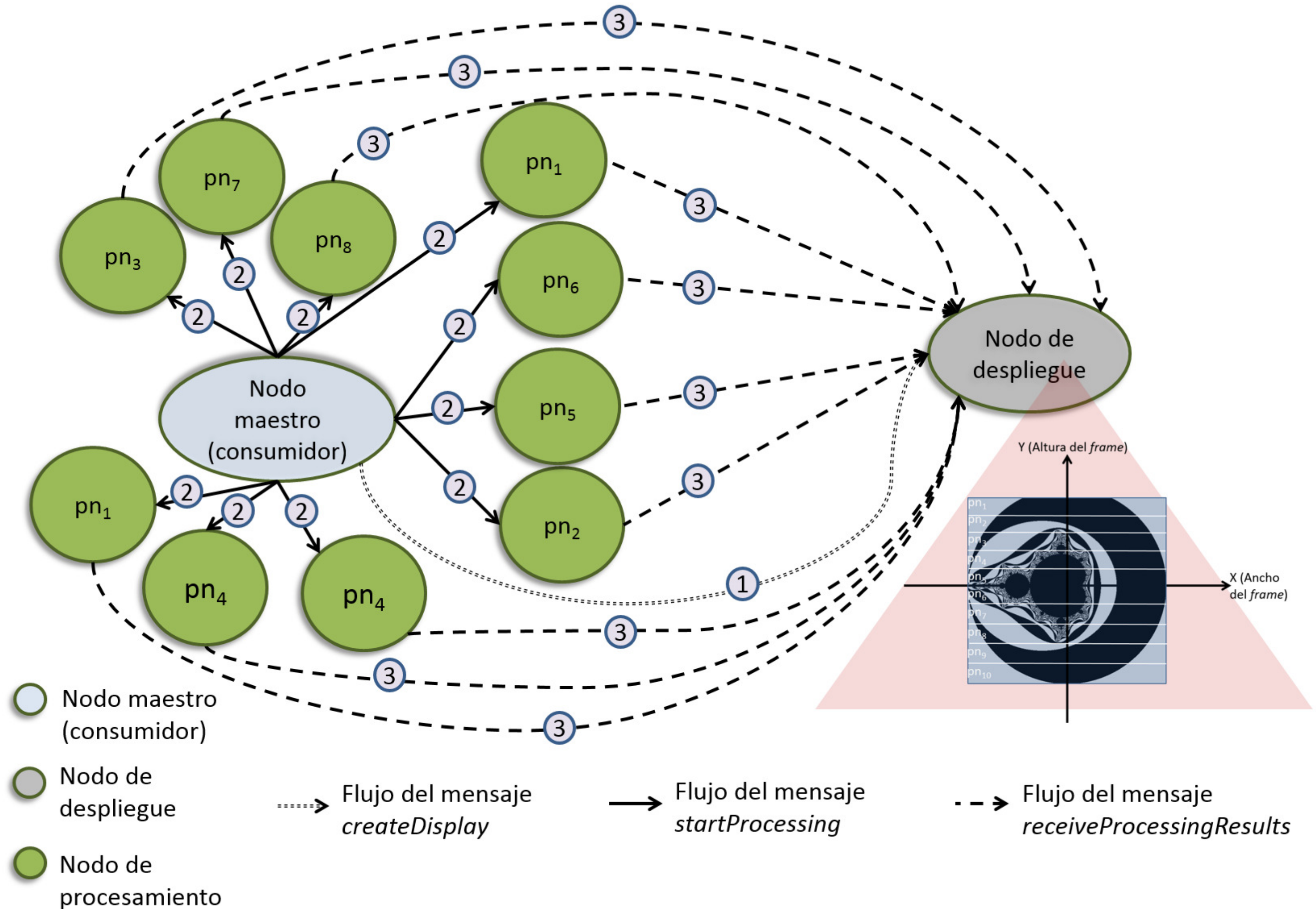


Figura 6.15: Colaboración de recursos en el sistema P2P colaborativo de validación que permite la generación del conjunto de Mandelbrot.

6.5.2. Diseño e implementación del sistema P2P colaborativo de validación

El sistema P2P colaborativo de validación se encuentra desarrollado en lenguaje Java, empleando la versión más reciente del JDK (*Java Development Kit 8*) de la versión estándar de Java (Java SE). En la figura 6.16 se observa que *MandelbrotSetP2PCollaborative* es la clase principal de dicho sistema, cuyo constructor recibe como argumentos de entrada instancias de las clases *RASupportTopologyNode* y *RASupportMain*. En esa misma figura, se observa que *MandelbrotSetP2PCollaborative* posee los métodos públicos *receiveXMLMessage()* y *generateMandelbrotSet()*, los cuales se utilizan para recibir documentos XML y para solicitar la generación del conjunto de Mandelbrot, respectivamente. Los documentos XML que se reciben especifican contratos de comportamiento de acuerdo al tipo de nodo; en el listado 6.7 se muestra un ejemplo de un contrato de comportamiento para un nodo de procesamiento, mientras que en los listados 6.8 y 6.9 se muestran dos ejemplos de contratos de comportamiento para un nodo de despliegue. El método *generateMan-*

delbrotSet() recibe como argumento de entrada un archivo XML, el cual representa una consulta de recursos necesaria para llevar a cabo las fases clave de la agregación de recursos, haciendo uso únicamente del método *executeQuery()* proporcionado por la fachada principal de *RASupport* (i.e., *RASupportMain*). En la figura 6.16, se observa que *RASupportMain* oculta la complejidad que existe detrás de las fases de la agregación de recursos, de manera que el sistema P2P colaborativo se enfoca en el propósito del sistema, i.e., en las fases de uso y liberación.

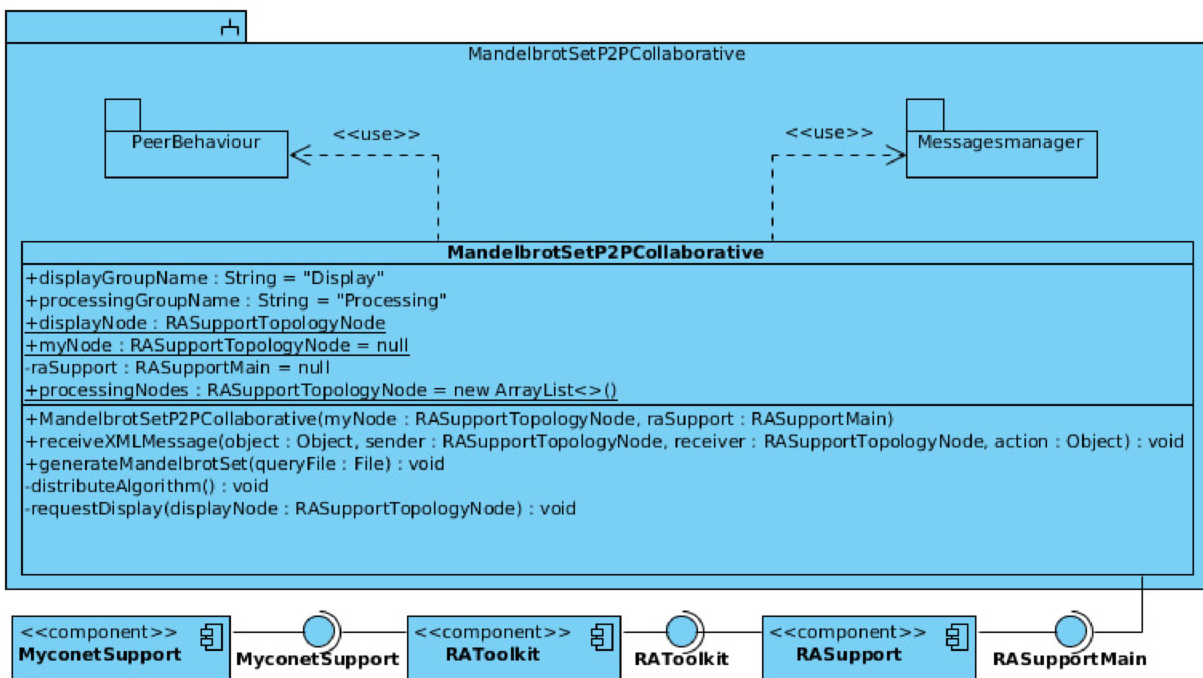


Figura 6.16: Diagrama de componentes del sistema P2P colaborativo de validación.

Listing 6.7: Ejemplo de un contrato de comportamiento que especifica el rango de puntos a procesar por un nodo de procesamiento.

```
<?xml version="1.0"?>
<message>
	<resource>
		<name>processing</name>
		<parameterSet>
			<parameter name="width">800</parameter>
			<parameter name="height">600</parameter>
			<parameter name="xmin">0</parameter>
			<parameter name="xmax">400</parameter>
			<parameter name="ymin">0</parameter>
			<parameter name="ymax">300</parameter>
		</parameterSet>
	</resource>
</message>
```

Listing 6.8: Ejemplo de un contrato de comportamiento que especifica los puntos a dibujar en el *frame* del nodo de despliegue.

```
<?xml version="1.0"?>
<message>
	<resource>
		<name>display</name>
		<parameterSet>
			<parameter name="xmin">0</parameter>
			<parameter name="xmax">1</parameter>
			<parameter name="ymin">0</parameter>
			<parameter name="ymax">1</parameter>
		</parameterSet>
		<parameterSet>
			<parameter name="x">0</parameter>
			<parameter name="y">0</parameter>
			<parameter name="isMandelbrot">true</parameter>
		</parameterSet>
		<parameterSet>
			<parameter name="x">1</parameter>
			<parameter name="y">0</parameter>
			<parameter name="isMandelbrot">true</parameter>
		</parameterSet>
		<parameterSet>
			<parameter name="x">0</parameter>
			<parameter name="y">1</parameter>
			<parameter name="isMandelbrot">false</parameter>
		</parameterSet>
		<parameterSet>
			<parameter name="x">1</parameter>
			<parameter name="y">1</parameter>
			<parameter name="isMandelbrot">true</parameter>
		</parameterSet>
</message>
```

Listing 6.9: Ejemplo de un contrato de comportamiento que especifica las dimensiones del *frame* que se desea crear en un nodo de despliegue.

```
<?xml version="1.0"?>
<message>
	<resource>
		<name>display</name>
		<parameterSet>
			<parameter name="width">800</parameter>
			<parameter name="height">600</parameter>
		</parameterSet>
	</resource>
</message>
```

En la figura 6.16 se observa que el sistema P2P colaborativo de validación posee dos módulos: *Messagesmanager* y *PeerBehaviour*. En la figura 6.17, se muestra el diagrama de clases de *Messagesmanager*, en donde se observa que este módulo contiene las clases *Para-*

meter, *ParameterSet*, *Resource*, *XMLConfiguration*, *XMLManager* y *XMLMessageWriter*, las cuales representan contratos de comportamiento y, además, sirven para crear y analizar dichos contratos. El módulo *PeerBehaviour* está compuesto por los submódulos *Common*, *Processing* y *Display*. En la figura 6.18 se muestra el diagrama de clases del submódulo *Common*, en donde se observa que el sistema P2P colaborativo utiliza el patrón *Estrategia* para definir el comportamiento de un nodo cuando este recibe un mensaje en particular; la interfaz *PeerBehaviour* es aquella que deben implementar las clases de comportamiento, mientras que las clases *BehaviourManager* y *Context* definen la estrategia a seguir. El submódulo *Processing* únicamente contiene la clase *ProcessingBehaviour* que representa el comportamiento de un nodo de procesamiento, en el que se calcula una región del conjunto de Mandelbrot empleando el algoritmo 7. El submódulo *Display* contiene las clases *Display* y *DisplayBehaviour*, las cuales representan una instancia única de un *frame* y el comportamiento de un nodo de despliegue al recibir un mensaje, respectivamente. Adicionalmente, el sistema P2P colaborativo de validación utiliza el patrón *Singleton* para definir una única instancia de un *frame* en un nodo de despliegue. Así, existen tres tipos de comportamiento:

1. Como nodo de cómputo cuando se recibe un mensaje *startProcesing*, el cual contiene un contrato de comportamiento como el de la figura 6.7, i.e., la instrucción para que el nodo de procesamiento procese una región del plano x, y del conjunto de Mandelbrot.

2. Como nodo de despliegue cuando se recibe un mensaje *receiveProcessingResults* que contiene un contrato de comportamiento como el de la figura 6.8, i.e., la instrucción para dibujar pixeles en el *frame* del nodo de despliegue.

3. Como nodo de despliegue cuando se recibe un mensaje *createDisplay* que contiene un contrato de comportamiento como el de la figura 6.9, i.e., la instrucción para que el nodo de despliegue cree el *frame*.

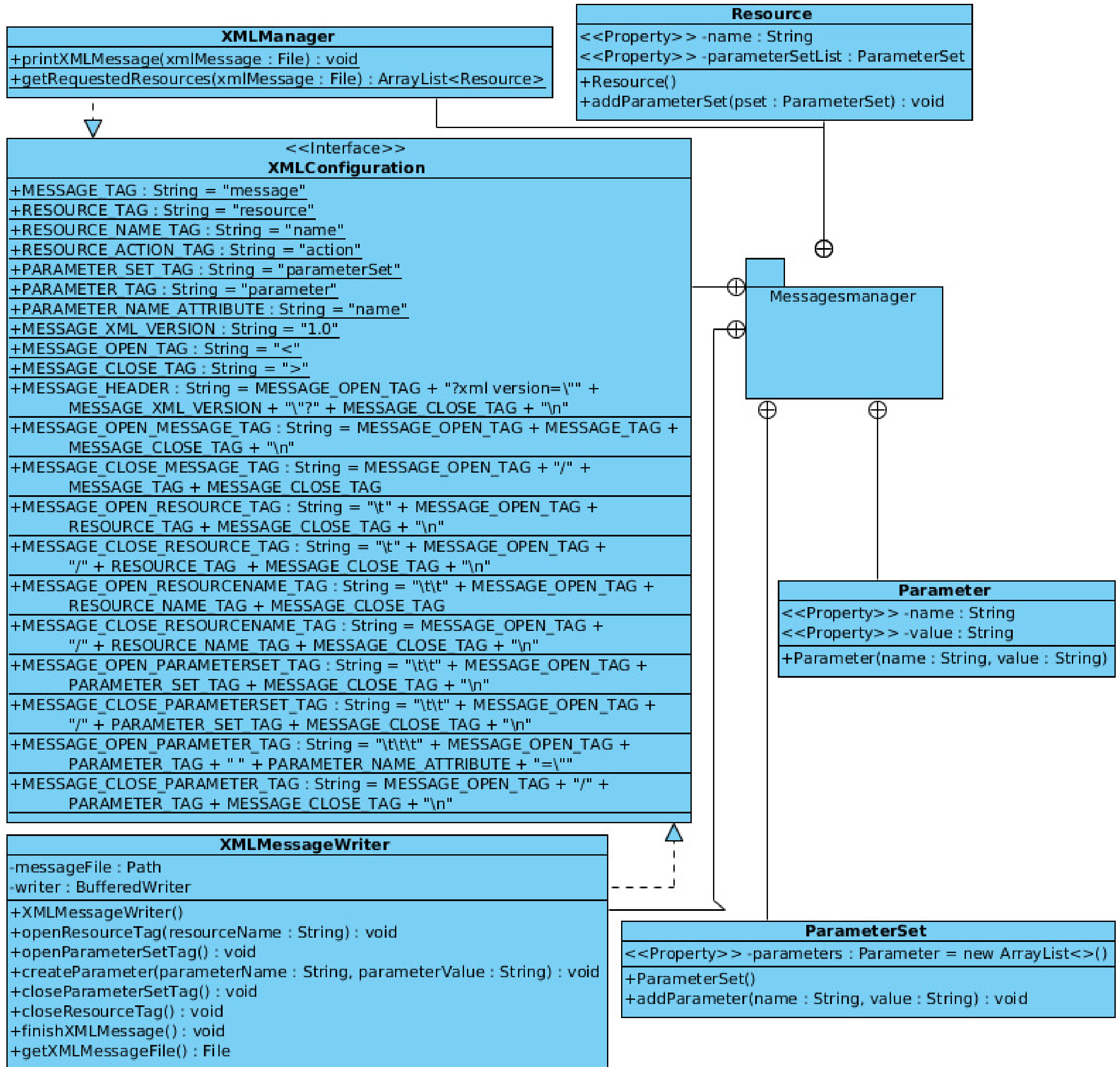


Figura 6.17: Diagrama de clases del módulo *Messagesmanager*.

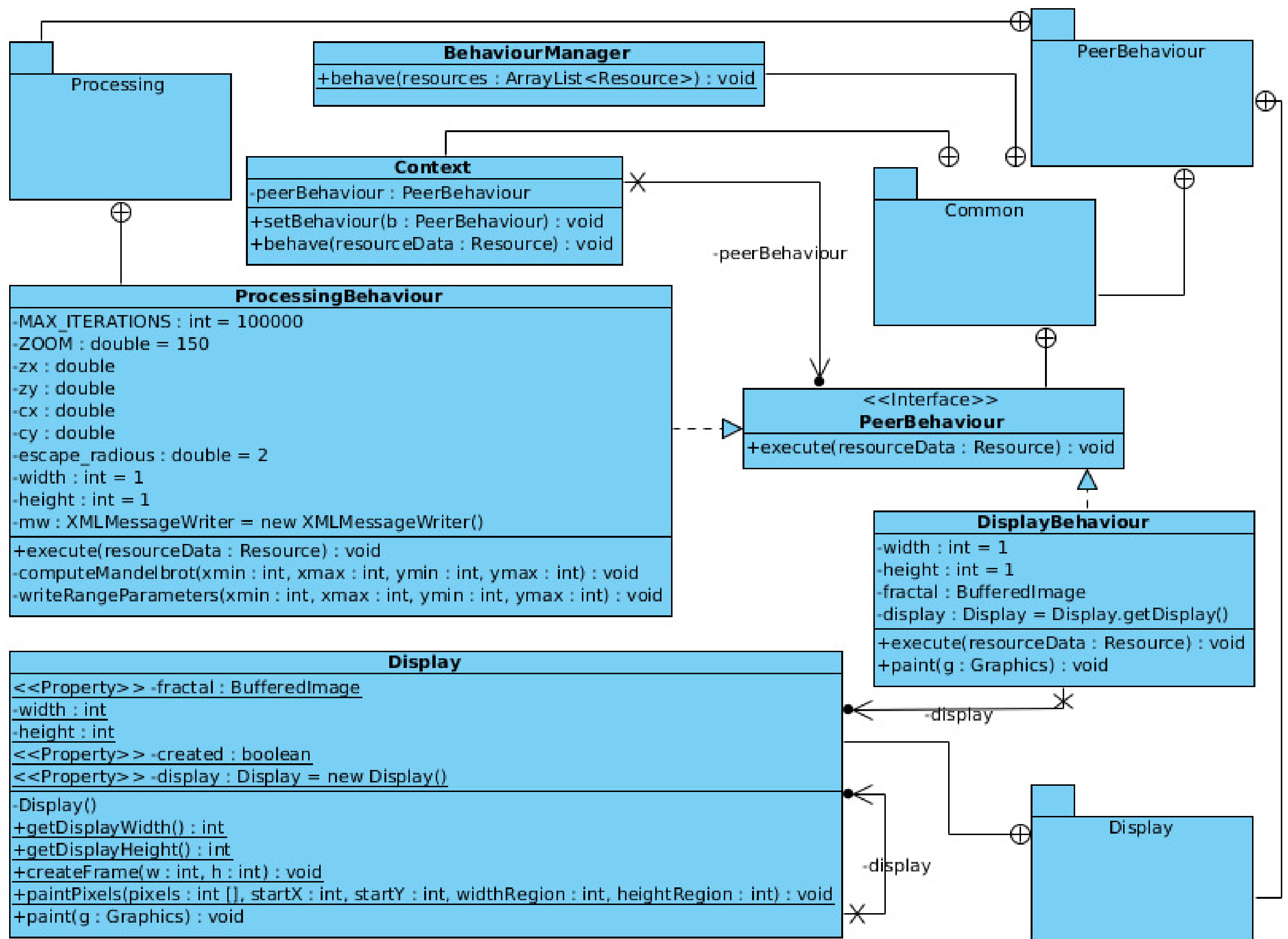


Figura 6.18: Diagrama de clases del módulo *PeerBehaviour*.

En la figura 6.19 se muestra el diagrama de comunicación del sistema P2P colaborativo de validación, en donde se observa que los patrones *Estrategia* y *Singleton* facilitan la gestión de comportamientos dentro del sistema P2P colaborativo. Además, se observa que el método *generateMandelbrotSet* oculta la complejidad que existe detrás del sistema P2P colaborativo. De manera similar, se observa que las fases clave de la agregación de recursos son llevadas a cabo de una forma transparente al cliente, gracias al método *executeQuery()* proporcionado por la fachada de *RASupport*, i.e., *RASupportMain*.

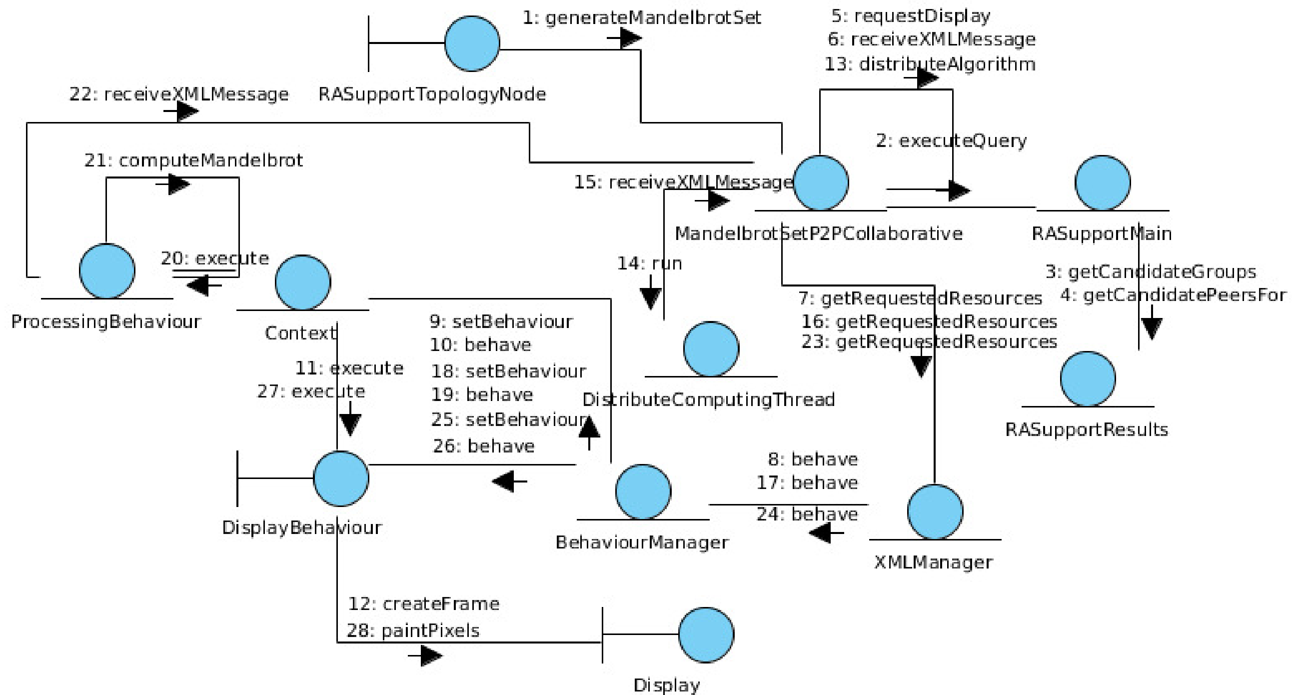


Figura 6.19: Diagrama de comunicación del sistema P2P colaborativo de validación.

6.5.3. Resultados del caso de estudio

Para validar el uso y el funcionamiento de *RASupport*, se llevaron a cabo una serie de experimentos. A continuación se describen detalladamente dos de estos experimentos, haciendo énfasis en las fases de la agregación de recursos. Las fases clave son efectuadas por *RASupport*, mientras que las fases que dependen del propósito específico de la aplicación (i.e., uso y liberación) son llevadas a cabo por el sistema P2P colaborativo de validación. La configuración global para ambos experimentos es la siguiente:

- Tamaño de la red: 25 *peers*.
- Intervalo de actualización de recursos dinámicos (red P2P altamente dinámica): 1 a 3 segundos.
- Duración de los experimentos: 50 ciclos.
- Duración de cada ciclo: 400 milisegundos.
- Probabilidad de que un *super-peer* en un estado inmóvil promueva a alguno de sus *peers normales* en un determinado ciclo: 0.2.

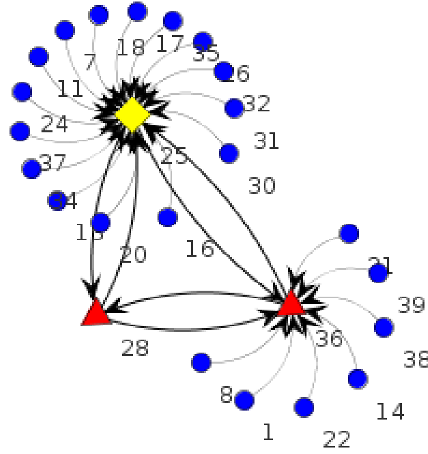


Figura 6.20: Estado inicial de la red P2P durante el primer experimento que valida el funcionamiento y uso de *RASupport*.

El primer experimento inicia una vez que el nodo maestro NP_{22} solicita la generación del conjunto de Mandelbrot, empleando el sistema P2P colaborativo que tiene instalado. La figura 6.20 ilustra el estado inicial de la red P2P, en donde se observa que algunos *peers* poseen identificadores superiores a 25, debido que algunos *peers* han abandonado la red, mientras que otros se han unido a ella. Inicialmente, se tienen tres *super-peers* ($SP_c = \{SP_{25}, SP_{28}, SP_{36}\}$) y 22 *peers normales* ($NP_c = \{NP_1, NP_8, NP_{14}, NP_{21}, NP_{22}, NP_{38}, NP_{39}, NP_7, NP_{11}, NP_{13}, NP_{16}, NP_{17}, NP_{18}, NP_{20}, NP_{24}, NP_{26}, NP_{30}, NP_{31}, NP_{32}, NP_{34}, NP_{35}, NP_{37}\}$), organizados de la siguiente manera: $SP_{36} = \{NP_1, NP_8, NP_{14}, NP_{21}, NP_{22}, NP_{38}, NP_{39}\}, SP_{28} = \emptyset, SP_{25} = \{NP_7, NP_{11}, NP_{13}, NP_{16}, NP_{17}, NP_{18}, NP_{20}, NP_{24}, NP_{26}, NP_{30}, NP_{31}, NP_{32}, NP_{34}, NP_{35}, NP_{37}\}$. El listado 6.10 muestra el archivo de configuración que el nodo maestro especifica para *RASupport*, en donde se observa que el soporte se ejecuta en modo aleatorio ($mode$ = $random$) debido a que el experimento se lleva a cabo a cabo en un entorno simulado. Adicionalmente, se observa que: (1) *RAToolkit* ha sido seleccionado como el *toolkit* de agregación de recursos ($toolkit$ = $ratoolkit$), (2) los recursos del nodo maestro están disponibles entre las 00:00 y las 16:32 ($use_restriction.availability.start$ =00:00 y $use_restriction.availability.end$ =16:32), (3) el porcentaje máximo de CPU que se puede utilizar del nodo maestro es de 15 % ($use_restriction.max_cpu_utilization$ = 15) y (4) los recursos del nodo maestro únicamente pueden ser utilizados por los *peers* amigos *peer3* y *peer1* ($use_restriction.only_for_friends = peer3, peer1$). La configuración de *RAToolkit* es la siguiente: (1) el número máximo de elementos permitidos en la lista de exclusión es de 2147483647 ($ratoolkit.max_exclusion$ = 2147483647) y (2) el número máximo de *super-peers* vecinos a los que $SP_{initiator}$ puede enviar agentes de consulta empleando *iRandomWalks* es de 2 ($ratoolkit.max_rw$ = 2).

Listing 6.10: Archivo de configuración para *RASupport* especificado por el nodo maestro.

```
# Configuration of the master node

# Execution mode of RASupport
mode=random

# Toolkit for resource aggregation to use
toolkit=ratoolkit

# Use restriction: availability of the peer
use_restriction.availability.start=00:00
use_restriction.availability.end=16:32

# Use restriction: allowed percentage of CPU
use_restriction.max_cpu_utilization=15

# Use restriction: determinates if the resources are going to be shared only with friends. You need t
use_restriction.only_for_friends = peer3,peer1

### RAToolkit configuration
# Max allowed elements in the exclusion list from the query agents
ratoolkit.max_exclusion=2147483647
# Neighbors to send query agents from Sp_initiator using iRandomWalks
ratoolkit.max_rw=2
```

La consulta de recursos del listado 6.11 describe la manera en que el nodo maestro modela grupos de procesamiento (i.e., *Processing*) y de despliegue (i.e., *Display*). En esta consulta se puede observar que dicho nodo desea encontrar recursos imperativamente (`<option>`requires:find_resources`</option>`), por lo que *RASupport* utiliza el protocolo de inundación con agentes de consulta durante la fase de selección, sacrificando así rendimiento por convergencia de resultados. En particular, el grupo *Processing* requiere 10 nodos de procesamiento caracterizados por la velocidad de procesador (`<cpu_speed>`), la memoria RAM disponible (`<free_mem>`) y el espacio libre en el disco duro (`<free_hdisk>`). El grupo *Display* requiere un nodo de despliegue caracterizado por la resolución de la pantalla (`<screen_height>` y `<screen_width>`), la tasa de refresco de la pantalla (`<refresh_rate>`) y la profundidad de color (`<bit_depth>`). Cada uno de los agentes de consulta que se envían posee un $TTL = 8$. En ambos grupos, no se especifican restricciones entre nodos, debido a que los nodos de un mismo grupo no necesitan comunicarse entre sí. Sin embargo, la consulta de recursos requiere se satisfaga la restricción de latencia entre grupos, debido a que los nodos de procesamiento del grupo *Processing* comunican sus resultados al nodo de despliegue del grupo *Display*.

Listing 6.11: Consulta de recursos emitida por el nodo maestro.

```
<?xml version="1.0"?>
<query>
	<option>requires:performance</option>
	<ttl>10</ttl>
	<group>
		<name>Processing</name>
		<num_nodes>10</num_nodes>
		<cpu_speed>500.0,2500.0,4096.0,5000.0,0.2</cpu_speed>
		<free_mem>10.0,1024.0,4096.0,16000.0,0.05</free_mem>
		<free_hdisk>0.0,60.5,92.5,1000000.0,0.05</free_hdisk>
	</group>
	<group>
		<name>Display</name>
		<num_nodes>1</num_nodes>
		<screen_height>0,1024,2048,3000,0.02</screen_height>
		<refresh_rate>0,60,120,1000,0.05</refresh_rate>
		<bit_depth>0,16,48,1000,0.0</bit_depth>
		<screen_width>0,1024,1366,3000,0.02</screen_width>
	</group>
	<rest_betw_groups>
		<group_names>Processing,Display</group_names>
		<latency>0.0,0.0,50.0,100.0,0.2</latency>
	</rest_betw_groups>
</query>
```

En la figura 6.21, se muestran los resultados obtenidos por *RASupport* en la fase de selección de recursos, en donde se observa que el nodo maestro NP_{22} solicita recursos, por lo que el *super-peer* a cargo es el iniciador de la consulta (i.e., $SP_{initiator} = SP_{36}$). De esta manera, $SP_{initiator}$ inicia el protocolo de inundación con agentes de consulta, cuya ejecución se describe a continuación. SP_{36} envía un agente de consulta a SP_{25}, ya que este último aún no ha recibido un agente transportando la consulta emitida por $SP_{initiator}$ (ver recuadro verde de la fig 6.21). Posteriormente, SP_{25} intenta inundar con agentes de consulta a sus vecinos *superpeers* (i.e., SP_{36} y SP_{28}); sin embargo, SP_{25} se percata de que SP_{36} se encuentra en la lista de exclusión (debido a que SP_{36} ya ha recibido al mismo agente de consulta), por lo que SP_{25} únicamente envía el agente de consulta a SP_{28} (ver recuadros azules de la figura 6.21). SP_{28} intenta inundar con agentes de consulta a sus vecinos *super-peers* (i.e., SP_{36} y SP_{28}); sin embargo, como ambos vecinos ya han recibido al mismo agente de consulta, este último no se reenvía y regresa a $SP_{initiator}$ para notificar sus resultados (ver recuadros amarillos de la figura 6.21). Como SP_{36} solo ha enviado un agente de consulta a uno de sus dos vecinos *superpeers* (i.e., SP_{25}), intenta enviar un agente de consulta al vecino faltante (i.e., SP_{28}); sin embargo, dicho vecino ya ha recibido a un agente transportando la misma consulta de recursos, por lo que no se le envía agente alguno (ver recuadros cafés de la figura 6.21). En el recuadro violeta de la figura 6.21, se muestran los resultados obtenidos en la fase de selección, en donde se observa que solo un *peer* satisface los requerimientos y las restricciones especificadas para el grupo *Processing*, mientras que todos los *peers* de la red P2P satisfacen los requerimientos y las restricciones

especificadas para el grupo *Display*. Lo anterior sucede debido a que los requerimientos del grupo *Processing* son muy específicos y a que los requerimientos del grupo *Display* poseen amplios rangos de valores permitidos.

```
Output - RASupport (run)
INFORMACIÓN: MYCOCAST STATS: 22 biomass, 2 extending, 1 branching, 0 immobile, 0 bulwark.
RASupport: peer22->peer22 has requested resources
RASupport: queryID -> 1197368818
RASupport: peer36->peer22 has requested resources
RASupport: queryID -> 1197368818
GROUP Processing:
GROUP Display:
SP peer36 tries to send AgentXXX to peer25
RASupport: SUPER-PEER peer25 has received the query 1197368818
GROUP Processing:
GROUP Display:
SP peer25 tries to send AgentXXX to peer36
IMPOSSIBLE TO SEND TO SUPER-PEER peer36 -> exclusion list
SP peer25 tries to send AgentXXX to peer28
IMPOSSIBLE TO SEND TO SUPER-PEER peer36 -> exclusion list
IMPOSSIBLE TO SEND TO SUPER-PEER peer25 -> exclusion list
RASupport: SUPER-PEER peer28 has received the query 1197368818
GROUP Processing:
GROUP Display:
SP peer28 tries to send AgentXXX to peer36
SP peer28 tries to send AgentXXX to peer25
RASupport: SUPER-PEER peer36 has received a result set for the query 1197368818
IMPOSSIBLE TO SEND TO SUPER-PEER peer28 -> exclusion list
Adding result from query agent received...
SP peer36 tries to send AgentXXX to peer28
SP_initiator has received all query agents

********************SELECTION PHASE RESULTS********************
----------------------------------------------------------------------------
----------------------------------------------------------------------------
Group(1 peers) = Processing
Candidate peers = [39 (type-0; BIOMASS; 1/15)]
----------------------------------------------------------------------------
----------------------------------------------------------------------------
Group(25 peers) = Display
Candidate peers = [36 (type-0; EXTENDING; 9/15), 22 (type-0; BIOMASS; 1/6), 8 (type-0; BIOMASS; 1/2), 1 (type-0; BIOMASS; 1/1), 14 (type-0; BIOMASS; 1/11), 21 (
type-0; BIOMASS; 1/9), 38 (type-0; BIOMASS; 1/15), 39 (type-0; BIOMASS; 1/15), 25 (type-0; BRANCHING; 17/15), 16 (type-0; BIOMASS; 1/8), 20 (type-0; BIOMASS; 1/
5), 30 (type-0; BIOMASS; 1/15), 32 (type-0; BIOMASS; 1/15), 31 (type-0; BIOMASS; 1/15), 7 (type-0; BIOMASS; 1/8), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS
; 1/15), 35 (type-0; BIOMASS; 1/15), 17 (type-0; BIOMASS; 1/6), 34 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1), 24 (type-0; BIOMASS; 1/2), 37 (type-0; BI
OMASS; 1/15), 11 (type-0; BIOMASS; 1/7), 28 (type-0; EXTENDING; 2/15)]
```

Figura 6.21: Resultados obtenidos por *RASupport* en la fase de selección de recursos.

Una vez que se han seleccionado los nodos que satisfacen los requerimientos y las restricciones especificadas para los grupos *Processing* y *Display*, respectivamente, se lleva a cabo la fase de empatamiento (i.e., el empatamiento de *peers* candidatos y de grupos candidatos). La lista inicial de *peers* candidatos para el grupo *Processing* está conformada por un *peer* (i.e., $n_{Processing} = 1$): $I_{Processing} = \{NP_{39}\}$ (ver recuadros morados de la figura 6.22). La lista inicial de *peers* candidatos para el grupo *Display* está conformada por 25 *peers* (i.e., $n_{Display} = 25$): $I_{Display} = \{SP_{36}, NP_{22}, NP_8, NP_1, NP_{14}, NP_{21}, NP_{38}, NP_{39}, SP_{25}, NP_{16}, NP_{20}, NP_{30}, NP_{32}, NP_{31}, NP_7, NP_{13}, NP_{26}, NP_{35}, NP_{17}, NP_{34}, NP_{18}, NP_{24}, NP_{37}, SP_{11}, NP_{28}\}$ (ver recuadros azules de la figura 6.22). Debido a que $n_{Processing} = 1$, en el grupo *Processing* únicamente existe un grupo óptimo de *peers* candidatos: $L_{Processing} = \{NP_{39}\}$ (ver recuadro violeta de la figura 6.23). Por otro lado, para el grupo *Display* se requiere la generación de combinaciones de *peers*, ya que $n_{Display} > 1$. La lista de referencia de *peers* que pueden trabajar entre sí en el grupo *Display* (i.e., $K_{Display}$) está conformada por 211 combinaciones, mientras que la lista de referencia de *peers* que no pueden trabajar entre sí en el grupo *Display* (i.e., $M_{Display}$) está conformada por 89 combinaciones (ver recuadro verde de la figura 6.24). Con la finalidad de mostrar claridad en el presente experimento, las listas $K_{Display}$ y $M_{Display}$ no se detallan; sin embargo, el teorema 1 se cumple, ya que $\mathbf{card}(K_{Display}) + \mathbf{card}(M_{Display}) = 211 + 89 = 300 = \frac{n^2_{Display} - n_{Display}}{2}$. En la etapa de empatamiento de *peers* candidatos, *RASupport* determina 5463 grupos candidatos para *Display* (i.e., $L_{Display}$), los cuales no se detallan por la razón antes mencionada

(ver recuadro verde de la figura 6.25). En la etapa de empatamiento de grupos candidatos, *RASupport* determina que el grupo $\{SP_{36}, NP_{13}, NP_{26}, NP_{18}\}$ es aquel que posee la menor penalización entre grupos ($P_{groups} = 260.0$); por lo tanto, es el grupo óptimo para *Display* (ver recuadro violeta de la figura 6.23).

Figura 6.22: Listas iniciales para los grupos *Processing* ($I_{Processing}$) y *Display* ($I_{Display}$) y lista de referencia de *peers* que pueden trabajar entre sí en el grupo *Display* ($K_{Display}$).

Figura 6.23: Resultados obtenidos por *RASupport* en la fase de empatamiento de recursos.

```
Output - RASupport (run)
e-0; BIOMASS; 1/8), 17 (type-0; BIOMASS; 1/6)], size=2)=0.0, CombinatoricsVector=([16 (type-0; BIOMASS; 1/8), 18 (type-0; BIOMASS; 1/1)], size=2)=40.0, Combinat
oricsVector=([32 (type-0; BIOMASS; 1/15), 35 (type-0; BIOMASS; 1/15)], size=2)=80.0, CombinatoricsVector=([16 (type-0; BIOMASS; 1/8), 20 (type-0; BIOMASS; 1/5)]
, size=2)=40.0, CombinatoricsVector=([32 (type-0; BIOMASS; 1/15), 37 (type-0; BIOMASS; 1/15)], size=2)=40.0, CombinatoricsVector=([16 (type-0; BIOMASS; 1/8), 26
 (type-0; BIOMASS; 1/15)], size=2)=0.0, CombinatoricsVector=([16 (type-0; BIOMASS; 1/8), 30 (type-0; BIOMASS; 1/15)], size=2)=40.0, CombinatoricsVector=([16 (ty
pe-0; BIOMASS; 1/8), 31 (type-0; BIOMASS; 1/15)], size=2)=80.0, CombinatoricsVector=([16 (type-0; BIOMASS; 1/8), 35 (type-0; BIOMASS; 1/15)], size=2)=80.0, Comb
inatoricsVector=([17 (type-0; BIOMASS; 1/6), 11 (type-0; BIOMASS; 1/7)], size=2)=40.0}
M_g(89): {CombinatoricsVector=([38 (type-0; BIOMASS; 1/15), 7 (type-0; BIOMASS; 1/8)], size=2)=Infinity, CombinatoricsVector=([21 (type-0; BIOMASS; 1/9), 25 (ty
pe-0; BRANCHING; 17/15)], size=2)=Infinity, CombinatoricsVector=([25 (type-0; BRANCHING; 17/15), 30 (type-0; BIOMASS; 1/15)], size=2)=Infinity, CombinatoricsVec
tor=([1 (type-0; BIOMASS; 1/1), 7 (type-0; BIOMASS; 1/8)], size=2)=Infinity, CombinatoricsVector=([21 (type-0; BIOMASS; 1/9), 28 (type-0; EXTENDING; 2/15)], siz
e=2)=Infinity, CombinatoricsVector=([17 (type-0; BIOMASS; 1/6), 24 (type-0; BIOMASS; 1/2)], size=2)=Infinity, CombinatoricsVector=([30 (type-0; BIOMASS; 1/15),
7 (type-0; BIOMASS; 1/8)], size=2)=Infinity, CombinatoricsVector=([1 (type-0; BIOMASS; 1/1), 11 (type-0; BIOMASS; 1/7)], size=2)=Infinity, CombinatoricsVector=(
[22 (type-0; BIOMASS; 1/6), 1 (type-0; BIOMASS; 1/1)], size=2)=Infinity, CombinatoricsVector=([13 (type-0; BIOMASS; 1/4), 24 (type-0; BIOMASS; 1/2)], size=2)=In
finity, CombinatoricsVector=([1 (type-0; BIOMASS; 1/1), 14 (type-0; BIOMASS; 1/11)], size=2)=Infinity, CombinatoricsVector=([30 (type-0; BIOMASS; 1/15), 13 (typ
```

Figura 6.24: Lista de referencia de *peers* que no pueden trabajar entre sí en el grupo *Display* ($M_{Display}$).

```
Output - RASupport (run)
FIRST PHASE -> CANDIDATE GROUPS
------------------------------------------------------------------------
Group: Processing
Candidate groups(1): [CombinatoricsVector=([39 (type-0; BIOMASS; 1/15)], size=1)=0.0]
------------------------------------------------------------------------
Group: Display
Candidate groups(5463): [CombinatoricsVector=([36 (type-0; EXTENDING; 9/15), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/1
5), 18 (type-0; BIOMASS; 1/1)], size=4)=-260.0, CombinatoricsVector=([1 (type-0; BIOMASS; 1/1), 16 (type-0; BIOMASS; 1/8), 26 (t
ype-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)], size=4)=-260.0, CombinatoricsVector=([21 (type-0; BIOMASS; 1/9), 13 (type-0;
BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)], size=4)=-260.0, CombinatoricsVector=([1 (type-0; BIOMASS;
 1/1), 16 (type-0; BIOMASS; 1/8), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)], size=5)=-24
0.0, CombinatoricsVector=([36 (type-0; EXTENDING; 9/15), 1 (type-0; BIOMASS; 1/1), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMAS
S; 1/15), 18 (type-0; BIOMASS; 1/1)], size=5)=-240.0, CombinatoricsVector=([1 (type-0; BIOMASS; 1/1), 21 (type-0; BIOMASS; 1/9),
 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)], size=5)=-240.0, CombinatoricsVector=([1 (typ
```

Figura 6.25: Grupos candidatos para *Display* ($L_{Display}$).

Ahora que se cuenta con los grupos que mejor satisfacen los requerimientos y las restricciones de la consulta de recursos, *RASupport* procede a llevar a cabo la fase de enlace. A partir de la figura 6.23, se deduce que *RASupport* ha obtenido un grupo compuesto por un *peer* proveedor para *Processing* ($b_{Processing} = \{pn_{39}\}$) y un grupo compuesto por cuatro *peers* proveedores para *Display* ($b_{Display} = \{pn_{36}, pn_{13}, pn_{26}, pn_{18}\}$). En la figura 6.20, se puede observar que el *super-peer* SP_{36} tiene a su cargo al *peer* proveedor pn_{39}, mientras que el *super-peer* SP_{25} tiene a su cargo a los *peers* proveedores pn_{13}, pn_{26} y pn_{18}. Por lo tanto, $SP_{initiator}$ envía dos agentes de enlace: uno al *super-peer* SP_{36} y otro al *super-peer* SP_{25} (ver recuadro azul de la figura 6.26). En la figura 6.26, se muestra el flujo de mensajes enviados y recibidos en *RASupport* para llevar a cabo la fase de enlace, en donde se observa que los recursos de los *peers* proveedores pn_{13} y pn_{26} para el grupo *Display* no están disponibles, por lo que dichos *peers* proveedores son eliminados de $b_{Display}$ (ver recuadros rojos de la figura 6.26). Además, el agente de enlace que visita a SP_{36} solo envía un mensaje de solicitud de recursos al *peer* proveedor pn_{39} (ver recuadro café de la figura 6.26), debido a que el *super-peer* visitado (i.e., SP_{36}) es también un nodo proveedor (i.e., pn_{36}). De esta manera, el resultado proporcionado por *RASupport* al sistema P2P colaborativo de validación son dos grupos (ver recuadro verde de la figura 6.26): (1) $b_{Processing} = \{pn_{39}\}$ con un único *peer* proveedor y (2) $b_{Display} = \{pn_{36}, pn_{18}\}$ con dos *peers* proveedores. De esta manera, se concluye que el nodo de procesamiento será pn_{39}, mientras que el nodo de despliegue será pn_{36}, ya que la consulta de recursos requiere un único nodo para el grupo *Display* y pn_{36} satisface mejor la consulta de recursos (de acuerdo a su valor de penalización calculado en la fase de empatamiento de recursos) que pn_{18}. Lo anterior resulta razonable ya que pn_{36} es un *super-peer*, mientras que pn_{18} es un *peer normal* (ver figura 6.20).

```
Output - RASupport (run)
Group: Processing
peers(1): [39 (type-0; BIOMASS; 1/15)]
--------------------------------------------------------------------------
Group: Display
peers(4): [36 (type-0; EXTENDING; 9/15), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)]
--------------------------------------------------------------------------
--------------------------------------------------------------------------
*******************BINDING PHASE RESULTS*******************

Binding agent to SP 36 (type-0; EXTENDING; 9/15)
Providers: [39 (type-0; BIOMASS; 1/15), 36 (type-0; EXTENDING; 9/15)]
--------------------------------------------------------------------------
Binding agent to SP 25 (type-0; BRANCHING; 17/15)
Providers: [13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)]

peer36 has received a binding agent!
Requesting resources to provider 39 (type-0; BIOMASS; 1/15)
Provider 39 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP Processing!
Requesting resources to provider 36 (type-0; EXTENDING; 9/15) (no travel needed in binding agent because receiver is the super-peer)
Provider 36 (type-0; EXTENDING; 9/15) has ACCEPTED the use of its resources in GROUP Display!
peer22 has received results from a binding agent! (1/2)
peer25 has received a binding agent!
Requesting resources to provider 13 (type-0; BIOMASS; 1/4)
Provider 13 (type-0; BIOMASS; 1/4) has REJECTED the use of its resources in GROUP Display (provider peer has been REMOVED from that group)
Requesting resources to provider 26 (type-0; BIOMASS; 1/15)
Provider 26 (type-0; BIOMASS; 1/15) has REJECTED the use of its resources in GROUP Display (provider peer has been REMOVED from that group)
Requesting resources to provider 18 (type-0; BIOMASS; 1/1)
Provider 18 (type-0; BIOMASS; 1/1) has ACCEPTED the use of its resources in GROUP Display!
peer22 has received results from a binding agent! (2/2)

Final Results:
Group: Processing
peers(1): [39 (type-0; BIOMASS; 1/15)]
--------------------------------------------------------------------------
Group: Display
peers(2): [36 (type-0; EXTENDING; 9/15), 18 (type-0; BIOMASS; 1/1)]

--------------------------------------------------------------------------
```

Figura 6.26: Resultados obtenidos por *RASupport* en la fase de enlace de recursos.

RASupport regresa al sistema P2P colaborativo de validación los resultados obtenidos (i.e., un grupo con los mejores nodos de procesamiento y otro con los mejores nodos de despliegue) en las fases clave de la agregación de recursos (i.e., selección, empatamiento y enlace). De esta manera, el nodo maestro o nodo consumidor (i.e., pr_{22}) de dicho sistema puede generar el conjunto de Mandelbrot, mediante la colaboración entre los recursos de los nodos de procesamiento y de despliegue. En el recuadro violeta de la figura 6.27, se muestran los resultados de la ejecución del sistema P2P colaborativo de validación, en donde se observa que, en primera instancia, el nodo maestro envía un mensaje *createDisplay* —que contiene un contrato de comportamiento (archivo */tmp/3686592185076322978.xml*) como el que se muestra en el listado 6.9— al *peer* de despliegue pn_{36} para que este cree un *frame* de 800×600. Posteriormente, el nodo maestro envía un mensaje *startProcessing* —que contiene un contrato de comportamiento (archivo */tmp/4542316051485094882.xml*) como el que se muestra en el listado 6.7— al *peer* de procesamiento pn_{39} para que este procese toda la región del plano del conjunto de Mandelbrot. Finalmente, el *peer* de procesamiento pn_{39} envía un mensaje *receiveProcessingResults* —que contiene un contrato de comportamiento (archivo */tmp/8920786685438656054.xml*) similar al que se muestra en el listado 6.8— al *peer* de despliegue pn_{36} para que este dibuje el conjunto de Mandelbrot en su *frame* (ver figura 6.28).

```
Output - RASupport (run)
Group: Processing
peers(1): [39 (type-0; BIOMASS; 1/15)]
---------------------------------------------------------------------------
Group: Display
peers(4): [36 (type-0; EXTENDING; 9/15), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)]
---------------------------------------------------------------------------
---------------------------------------------------------------------------
*******************BINDING PHASE RESULTS*******************
---------------------------------------------------------------------------
Binding agent to SP 36 (type-0; EXTENDING; 9/15)
Providers: [39 (type-0; BIOMASS; 1/15), 36 (type-0; EXTENDING; 9/15)]
---------------------------------------------------------------------------
Binding agent to SP 25 (type-0; BRANCHING; 17/15)
Providers: [13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)]
---------------------------------------------------------------------------
peer36 has received a binding agent!
Requesting resources to provider 39 (type-0; BIOMASS; 1/15)
Provider 39 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP Processing!
Requesting resources to provider 36 (type-0; EXTENDING; 9/15) (no travel needed in binding agent because receiver is the super-peer)
Provider 36 (type-0; EXTENDING; 9/15) has ACCEPTED the use of its resources in GROUP Display!
peer22 has received results from a binding agent! (1/2)
peer25 has received a binding agent!
Requesting resources to provider 13 (type-0; BIOMASS; 1/4)
Provider 13 (type-0; BIOMASS; 1/4) has REJECTED the use of its resources in GROUP Display (provider peer has been REMOVED from that group)
Requesting resources to provider 26 (type-0; BIOMASS; 1/15)
Provider 26 (type-0; BIOMASS; 1/15) has REJECTED the use of its resources in GROUP Display (provider peer has been REMOVED from that group)
Requesting resources to provider 18 (type-0; BIOMASS; 1/1)
Provider 18 (type-0; BIOMASS; 1/1) has ACCEPTED the use of its resources in GROUP Display!
peer22 has received results from a binding agent! (2/2)
---------------------------------------------------------------------------
Final Results:
Group: Processing
peers(1): [39 (type-0; BIOMASS; 1/15)]
---------------------------------------------------------------------------
Group: Display
peers(2): [36 (type-0; EXTENDING; 9/15), 18 (type-0; BIOMASS; 1/1)]
---------------------------------------------------------------------------
---------------------------------------------------------------------------
MESSAGE RECEIVED IN peer36
File: /tmp/3686592185076322978.xml
Action: createDisplay
Node 0:
X = 0-800
Y = 0-600
MESSAGE RECEIVED IN peer39
File: /tmp/4542316051485094882.xml
Action: startProcessing
Executing computing node...
MESSAGE RECEIVED IN peer36
File: /tmp/8920786685438656054.xml
Action: receiveProcessingResults
```

Figura 6.27: Mensajes enviados en el sistema P2P colaborativo de validación, durante la generación del conjunto de Mandelbrot.

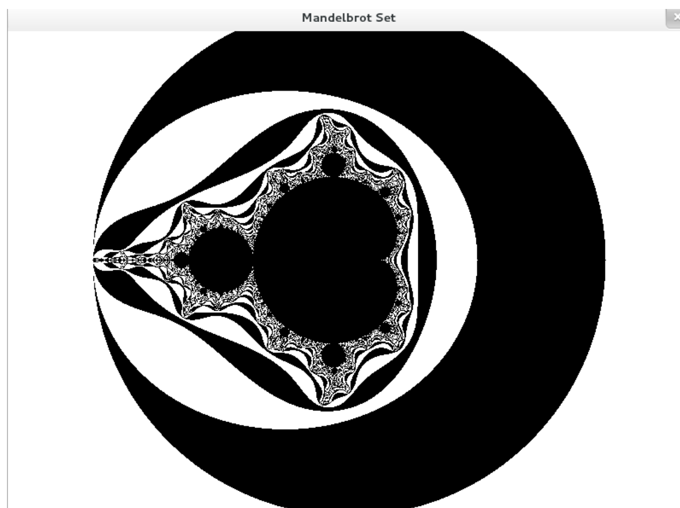


Figura 6.28: *Frame* con una resolución de 800×600, en donde se despliega el conjunto de Mandelbrot como resultado de la ejecución del sistema P2P colaborativo de validación.

En un segundo experimento, se utilizó la configuración global, la configuración para *RA-Support* del listado 6.10 y la consulta de recursos del listado 6.11. Esta vez, el sistema P2P colaborativo de validación se ejecutó en una red P2P como la que se muestra en la figura 6.29, teniendo como nodo maestro al *super-peer* SP_{24} (i.e., $SP_{initiator} = SP_{24}$) y obteniendo como resultado dos nodos de procesamiento ($b_{Processing} = \{pn_0, pn_{19}\}$) y 18 nodos de despliegue ($b_{Display} = \{pn_0, pn_{17}, pn_5, pn_{13}, pn_{21}, pn_{14}, pn_{22}, pn_{12}, pn_{24}, pn_1, pn_{23}, pn_{25}, pn_{18}, pn_4, pn_{26}, pn_{15}, pn_7, pn_{16}\}$) (ver recuadro verde la figura 6.30). Por lo tanto, el *peer* proveedor pn_0 actuará como nodo de procesamiento y despliegue, ya que este es un *super-peer* que posee altas capacidades de cómputo (de acuerdo con la ecuación 4.3 especificada en el capítulo 4), además de satisfacer mejor los requerimientos y las restricciones especificadas en los grupos *Processing* y *Display*, en comparación con los demás *peers* proveedores de $b_{Processing}$ y $b_{Display}$, respectivamente. En el recuadro violeta de la figura 6.30, se muestra el flujo de mensajes de este experimento, en donde se observa que, en primera instancia, el nodo maestro envía un mensaje *createDisplay* —que contiene un contrato de comportamiento (archivo */tmp/6845954228611468846.xml*) como el que se muestra en el listado 6.9— al *peer* de despliegue pn_0 para que este cree un *frame* de 800×600. Posteriormente, el nodo maestro envía dos mensajes *startProcessing* —que contienen contratos de comportamiento (archivos */tmp/4846087343993489398.xml* y */tmp/7152535199785552925.xml*) como el que se muestra en el listado 6.7— a los *peers* de procesamiento pn_0 y pn_{19} para que cada uno de ellos procese una mitad del plano x, y del conjunto de Mandelbrot. Finalmente, ambos *peers* de procesamiento envían mensajes *receiveProcessingResults* —que contienen contratos de comportamiento (archivos */tmp/8180254768908535677.xml* y */tmp/5035249368573627897.xml*) similares al que se muestra en el listado 6.8— al nodo de despliegue pn_0, para que este dibuje el conjunto de Mandelbrot en su *frame* (ver figura 6.28).

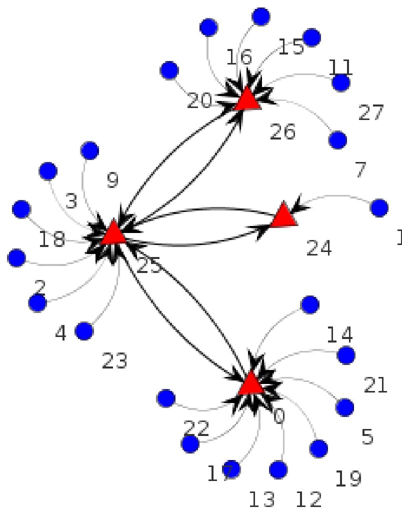


Figura 6.29: Estado de la red P2P durante el segundo experimento que valida el funcionamiento y uso de *RASupport*.

```
Output - RASupport (run) ×
Group: Processing
peers(1): [39 (type-0; BIOMASS; 1/15)]
--------------------------------------------------------------------------
Group: Display
peers(4): [36 (type-0; EXTENDING; 9/15), 13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)]
--------------------------------------------------------------------------
--------------------------------------------------------------------------
*******************BINDING PHASE RESULTS*******************
--------------------------------------------------------------------------
Binding agent to SP 36 (type-0; EXTENDING; 9/15)
Providers: [39 (type-0; BIOMASS; 1/15), 36 (type-0; EXTENDING; 9/15)]
--------------------------------------------------------------------------
Binding agent to SP 25 (type-0; BRANCHING; 17/15)
Providers: [13 (type-0; BIOMASS; 1/4), 26 (type-0; BIOMASS; 1/15), 18 (type-0; BIOMASS; 1/1)]
--------------------------------------------------------------------------
peer36 has received a binding agent!
Requesting resources to provider 39 (type-0; BIOMASS; 1/15)
Provider 39 (type-0; BIOMASS; 1/15) has ACCEPTED the use of its resources in GROUP Processing!
Requesting resources to provider 36 (type-0; EXTENDING; 9/15) (no travel needed in binding agent because receiver is the super-peer)
Provider 36 (type-0; EXTENDING; 9/15) has ACCEPTED the use of its resources in GROUP Display!
peer22 has received results from a binding agent! (1/2)
peer25 has received a binding agent!
Requesting resources to provider 13 (type-0; BIOMASS; 1/4)
Provider 13 (type-0; BIOMASS; 1/4) has REJECTED the use of its resources in GROUP Display (provider peer has been REMOVED from that group)
Requesting resources to provider 26 (type-0; BIOMASS; 1/15)
Provider 26 (type-0; BIOMASS; 1/15) has REJECTED the use of its resources in GROUP Display (provider peer has been REMOVED from that group)
Requesting resources to provider 18 (type-0; BIOMASS; 1/1)
Provider 18 (type-0; BIOMASS; 1/1) has ACCEPTED the use of its resources in GROUP Display!
peer22 has received results from a binding agent! (2/2)
--------------------------------------------------------------------------
Final Results:
Group: Processing
peers(1): [39 (type-0; BIOMASS; 1/15)]
--------------------------------------------------------------------------
Group: Display
peers(2): [36 (type-0; EXTENDING; 9/15), 18 (type-0; BIOMASS; 1/1)]
--------------------------------------------------------------------------
--------------------------------------------------------------------------
MESSAGE RECEIVED IN peer36
File: /tmp/3686592185076322978.xml
Action: createDisplay
Node 0:
X = 0-800
Y = 0-600
MESSAGE RECEIVED IN peer39
File: /tmp/4542316051485094882.xml
Action: startProcessing
Executing computing node...
MESSAGE RECEIVED IN peer36
File: /tmp/8920786685438656054.xml
Action: receiveProcessingResults
```

Figura 6.30: Mensajes enviados en el segundo experimento durante la generación del conjunto de Mandelbrot.

Capítulo 7

Conclusiones y trabajo a futuro

El objetivo de este trabajo de tesis fue el diseño y la implementación de un soporte que permite llevar a cabo la agregación de recursos en sistemas P2P colaborativos. En este capítulo se hace una recapitulación del planteamiento del problema (sección 7.1), el cual se refiere a la falta de herramientas de *software* que faciliten el desarrollo de sistemas P2P colaborativos, empleando algún lenguaje de programación específico. Posteriormente, se presentan las conclusiones de la presente tesis (sección 7.2) y, finalmente, se proponen algunas mejoras y futuras extensiones que se pueden aplicar en *RASupport* (sección 7.3).

7.1. Recapitulación del planteamiento del problema

Actualmente, la mayoría de los protocolos P2P que permiten llevar a cabo la agregación de recursos se encuentran especificados teóricamente y solo implementan un subconjunto de las fases que comprenden dicho proceso, sin proporcionar cohesión entre las fases clave (selección, empatamiento y enlace). Además, no es trivial descubrir, agregar y utilizar recursos dinámicos y heterogéneos que están distribuidos, por lo que es necesario crear soluciones eficientes para direccionar esta problemática [1]. Aunado a lo anterior, actualmente no existen herramientas de *software* que faciliten el desarrollo de sistemas P2P colaborativos. En particular, no existen *toolkits* que permitan llevar a cabo la agregación de recursos en este tipo de sistemas, usando algún lenguaje de programación específico.

Aunque se han propuesto protocolos P2P para anunciar, descubrir y seleccionar recursos individuales, éstos proporcionan una forma primitiva de agregación de recursos debido a que no capturan correctamente las relaciones entre recursos y, por ende, son incapaces de satisfacer los requerimientos de una aplicación. Además, ninguno de estos protocolos soporta de manera escalable y eficiente los requerimientos de sistemas P2P colaborativos del mundo real. Por lo tanto, se necesitan soluciones de agregación de recursos que sean más eficientes y que se adapten a las necesidades del mundo real [18].

Los protocolos P2P no estructurados de agregación de recursos presentan un alto costo en la fase de anunciamiento y en muchos de éstos ni siquiera hay garantía de encontrar re-

cursos. Todos los protocolos P2P de agregación de recursos existentes tienden a presentar fallas al gestionar recursos dinámicos y al momento de realizar consultas multi-atributo.

De acuerdo con Bandara y Jayasumana [1], actualmente se requiere una propuesta para agregar grupos de recursos heterogéneos, distribuidos, dinámicos y multi-atributo. Aunque se han propuesto muchas soluciones basadas en P2P para llevar a cabo el descubrimiento de recursos multi-atributo, aún existe la necesidad de contar con una solución que permita llevar a cabo las fases clave de la agregación de recursos (selección, empatamiento y enlace). Se debe tener cuidado al combinar estas fases ya que, de otra manera, las optimizaciones en una fase podrían provocar conflictos en las siguientes fases.

7.2. Conclusiones

La *colaboración de recursos* es el proceso en el que los recursos de diversos nodos de un sistema distribuido son compartidos e interactúan entre sí para lograr grandes y complejas tareas, difíciles de conseguir utilizando un solo nodo. Un recurso puede ser un ciclo de procesador, capacidad de almacenamiento, ancho de banda de la red, un sensor/actuador, hardware especial, un *middleware*, un algoritmo científico, software de aplicación o un servicio (e.g., Web y de una nube). La colaboración de recursos es fundamental en sistemas en los que se requiere realizar ciertas tareas que resultan difíciles de llevar a cabo en una sola computadora (e.g., tareas que involucran alto procesamiento de cómputo o que requieren gran espacio en disco) [1]. En particular, la agregación de recursos es un subconjunto de la colaboración de recursos, el cual comprende las fases de anunciamiento, selección, empatamiento y enlace.

En la presente tesis se presentó el diseño y la implementación de *RASupport*, el cual es un soporte flexible, auto-configurable, bio-inspirado y multi-agente que permite agregar recursos en un entorno P2P, facilitando así el desarrollo de sistemas P2P colaborativos. Dicho soporte, cuenta con un *toolkit* (denominado *RAToolkit*) para llevar a cabo las fases de anunciamiento, selección, empatamiento y enlace, manteniendo cohesión entre las fases clave de la agregación de recursos (selección, empatamiento y enlace). *RASupport* es flexible, debido a que permite agregar fácilmente nuevos módulos (e.g., se puede añadir un módulo que permita balancear cargas de una manera eficiente) y se pueden cambiar o eliminar los ya existentes (e.g., es posible substituir el módulo de simulación de recursos por el módulo de administración de recursos reales). *RASupport* es auto-configurable, debido a que emplea técnicas de programación generativa que hacen que su comportamiento varíe en función de los requerimientos y las restricciones de una aplicación cliente, los cuales son especificados en un archivo de configuración y en documentos XML. *RASupport* es bio-inspirado, debido a que su arquitectura no estructurada basada en *super-peers* se encuentra gestionada por el protocolo Myconet, el cual permite mantener redes auto-organizables y tolerantes a fallos, empleando un conjunto de reglas inspiradas

en el crecimiento de los micelios. *RASupport* es multi-agente, debido a que *RAToolkit* permite llevar a cabo las fases de anunciamiento, selección, empatamiento y enlace, a partir de la interacción de un conjunto de agentes, los cuales tienen un objetivo bien definido.

Para llevar a cabo la fase de anunciamiento, los *peers normales* envían agentes a sus respectivos *super-peers*. Dichos agentes transportan documentos XML de especificación de recursos (RSs) con una complejidad de $O(1)$. Existen dos tipos de agentes de anunciamiento: agentes de anunciamiento inicial y agentes de anunciamiento de actualización. Los primeros se encargan de anunciar recursos la primera vez que un *peer normal* se une a un *super-peer*, mientras que los segundos se encargan de anunciar actualizaciones que se suscitan en los recursos dinámicos de un *peer normal*. La fase de descubrimiento se omite debido a que los recursos son anunciados explícitamente a los *super-peers*.

Las consultas de recursos se llevan a cabo haciendo uso de documentos XML, en donde se especifican los requerimientos mediante grupos de atributos requeridos. En cada uno de estos grupos se especifica el nombre, el número de *peers* requeridos y el conjunto de atributos que se necesitan. Para atributos numéricos (i.e., atributos enteros o flotantes), se especifica el valor mínimo requerido, el valor mínimo ideal, el valor máximo ideal, el valor máximo requerido y un valor de penalización por no satisfacer los valores requeridos. Por el contrario, para atributos de tipo *string* se especifica únicamente el valor requerido y el valor de la penalización. *RAToolkit* ofrece dos opciones para llevar a cabo la selección de recursos: *find_resources* y *performance*. La primera opción emplea una técnica basada en el protocolo de inundación, en la cual se inunda la red P2P con agentes de consulta. A pesar de que este procolo tiene la desventaja de poseer una complejidad exponencial y de causar tráfico en la red, garantiza que los agentes de consulta visiten a todos los *super-peers* únicamente una vez, por lo que aumenta la posibilidad de obtener resultados en la fase de selección. Por otro lado, la segunda opción emplea el protocolo de paseos aleatorios inteligentes (*iRandomWalks*), el cual está basado en el protocolo tradicional de paseos aleatorios. A diferencia de este último, el protocolo de paseos aleatorios inteligentes mantiene una lista de exclusión en los agentes de consulta, los cuales envían mensajes de prueba antes de visitar a un *super-peer*, evitando así llegar a *super-peers* que ya hayan sido o que serán visitados por otros agentes que transportan una misma consulta, por lo que de esta manera se reduce el ancho de banda en la fase de selección. De igual forma, los *super-peers* mantienen una lista de estadísticas, lo cual permite que los agentes de consulta lleguen a *super-peers* que probablemente regresarán resultados para un grupo especificado en una consulta. Por lo tanto, el protocolo de paseos aleatorios inteligentes se beneficia del aprendizaje de éxitos previos, aumentando así la probabilidad de éxito en una consulta en comparación con el protocolo tradicional de paseos aleatorios. *RASupport* ofrece la posibilidad de especificar consultas multi-atributo (*multi-attribute queries*), de un solo atributo (*single-attribute queries*) y de cero atributos (*zero-attribute queries*), haciendo uso de documentos XML.

El procesamiento de la fase de empatamiento de recursos se divide entre los agentes de

consulta envíados por un *super-peer* iniciador, un agente de empatamiento y un conjunto de subagentes de empatamiento. El agente de empatamiento espera los resultados de cada uno de los agentes de consulta y, una vez que recibe dichos resultados, inicia un proceso de dos etapas: empatamiento de *peers candidatos* y empatamiento de grupos candidatos, las cuales están basadas en algoritmos que utilizan programación dinámica. Para llevar a cabo la evaluación de las restricciones entre *peers* y entre grupos de atributos requeridos, se envían subagentes de empatamiento a los nodos involucrados. Una vez que un agente de empatamiento ha concluído su labor, comunica sus resultados a un conjunto de agentes de enlace, los cuales se encargan de llevar a cabo la fase de enlace, garantizando así la calidad de servicio deseada en sistemas P2P colaborativos.

Los resultados obtenidos a partir de la implementación de *RASupport* demuestran que dicho soporte es la única herramienta de *software* que existe actualmente, la cual facilita el desarrollo de sistemas P2P colaborativos y contempla los requerimientos y restricciones de una aplicación cliente, además de dar soporte a recursos distribuidos y heterogéneos caracterizados por múltiples atributos dinámicos y estáticos de una forma modular e innovadora.

RASupport se empleó en el desarrollo de un sistema P2P colaborativo que tiene el objetivo de generar el conjunto de Mandelbrot, mediante la colaboración de los recursos de un conjunto de *peers* que satisfacen los requerimientos y las restricciones especificadas en una consulta de recursos. Los resultados del caso de estudio demuestran que el cliente de *RASupport* únicamente se centra en el propósito específico (i.e., fases de uso y liberación) del sistema, de manera que *RASupport*:

1. Facilita el desarrollo de sistemas P2P colaborativos, empleando el lenguaje Java.

2. Proporciona Calidad de Servicio, ya que ofrece cohesión entre las fases clave de la agregación de recursos: selección, empatamiento y enlace.

3. Oculta la complejidad que existe detrás de las fases de la agregación de recursos.

4. Agrega recursos distribuidos, dinámicos, estáticos, heterogéneos, multi-atributo y de un sólo atributo en una arquitectura P2P, mediante un conjunto de agentes que interactúan entre sí.

5. Permite realizar consultas multi-atributo, de un único atributo y de cero atributos, especificando requerimientos y restricciones en abstracciones de alto nivel, ie., en documentos XML.

6. Funciona apropiadamente incluso ante la presencia de *churn* en la red P2P.

7. Funciona adecuadamente en ambientes altamente dinámicos y en ambientes semi-dinámicos.

8. Permite la personalización de consultas de recursos.

7.3. Trabajo a futuro

A pesar de las contribuciones presentadas en esta tesis, aún queda un largo camino por recorrer en lo que respecta a la investigación sobre la agregación de recursos en sistemas P2P colaborativos. Una de las principales limitaciones de *RASupport* es la falta de un mecanismo robusto de balace de cargas, debido a que pocos nodos (únicamente *super-peers*) se encargan de resolver consultas y de indexar recursos. Por lo tanto, es posible diseñar mecanismos que nos ayuden a balancear cargas de una manera eficiente y robusta en una red P2P.

En la fase de selección, las consultas de recursos no siempre pueden resolverse, debido a la imprevisibilidad del protocolo de paseos aleatorios inteligentes y a que es posible que el valor del TTL no sea suficiente para inundar toda la red P2P, en caso de que se emplee el protocolo de inundación con agentes de consulta. Otro problema que se presenta en la fase de selección es que no existe tolerancia a fallos para mantener una consulta de recursos aún cuando un *super-peer* iniciador deja de estar disponible; con la finalidad de resolver esta problemática, es posible que los *peers normales* que solicitaron recursos envíen mensajes *stay alive* cada cierto tiempo para comprobar que la consulta de recursos continúa siendo atendida. Además, es posible definir condiciones lógicas *and*, *or* y *not* para cada atributo especificado en una consulta de recursos (e.g., se podría especificar una condición os_name="Linux" *or* "Windows"). Esta facilidad podría ser una contribución adicional para los lenguajes de consultas de recursos, debido a que ninguno de ellos permite seleccionar recursos empleando dichas condiciones lógicas.

RASupport utiliza el protocolo Myconet para gestionar la arquitectura P2P no estructurada basada en *super-peers*, por lo que se ofrece una gran resiliencia y auto-organización en caso de que un *peer* abandone la red. Aunque las arquitecturas P2P no estructuradas trabajan bien en ambientes altamente dinámicos, no son adecuadas para mantener sistemas de gran escala. Por esta razón, se espera extender *RASupport* de tal manera que dé soporte a arquitecturas estructuradas.

La seguridad es muy importante en sistemas P2P; como una forma de autenticación, los *peers* participantes podrían firmar reportes de actualización de recursos dinámicos. Con una autenticación eficiente, se podría asegurar que ningún *peer* consuma ancho de banda innecesariamente o gran capacidad de almacenamiento y/o procesamiento con la finalidad de perjudicar a otros *peers*. Además, los *super-peers* podrían solicitar firmas digitales para asegurarse de que los *peers normales* no encripten ni modifiquen sus especificaciones de recursos durante la fase de anunciamiento.

Así mismo, es posible crear comunidades de *super-peers* con similitud semántica, de manera que sea más fácil localizar recursos dentro de la red P2P. Por otro lado, el módulo de gestión de recursos *ResourceSim* se podría mejorar de tal manera que se generen recursos con características más parecidas a las que se presentan en escenarios del mundo real.

Con la finalidad de mejorar el validador de documentos XML de consulta de recursos, se podría desarrollar un subsistema de validaciones, de esta manera se podrían cambiar y agregar fácilmente reglas dentro de *RASupport*. Los identificadores de las consultas de recursos actualmente se encuentran delimitados por la función *hashCode* de Java, sin embargo, es posible desarrollar una función más robusta que minimice las colisiones en los identificadores de dichas consultas.

La agregación de recursos en sistemas P2P colaborativos es un tema de investigación relativamente nuevo, de manera que muchos problemas aún permanecen abiertos. Estos sistemas serán de suma importancia en un futuro cercano, debido a su amplio dominio de aplicación y a la constante evolución de los dispositivos móviles. Por lo tanto, la agregación de recursos en dispositivos móviles puede llegar a ser un tema de investigación relevante, lo cual podría proporcionar beneficios significativos, e.g., explotar y aprovechar los recursos de los dispositivos que pertenecen a toda la humanidad, con la finalidad de llevar a cabo supercómputo científico.

Publicación del autor

[Arellanes et al., 2014] D. Arellanes, S. Mendoza, and D. Decouchant, *Support for resource aggregation in collaborative P2P systems*, In 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE '14), IEEE Computer Society, Campeche, Mexico, October 2014.

Apéndice A

A.1. Glosario

Adaptación emergente. Es la capacidad de un sistema de cambiar de comportamiento ante la presencia de una perturbación dentro de su entorno. Esta propiedad típicamente se presenta en sistemas complejos, e.g., las colonias de hormigas.

Agente activo. En un sistema multi-agente, un agente activo es aquel que tiene una meta simple, e.g., un pájaro en una parvada.

Agente de anunciamiento. En la presente tesis, un agente de anunciamiento es aquel que transporta un documento XML de especificación de recursos con un costo de $O(1)$. Existen dos tipos de agentes de anunciamiento: el inicial y el de actualización.

Agente de consulta. En la presente tesis, un agente de consulta es aquel que transporta un documento XML de consulta de recursos y su comportamiento varía en función de los requerimientos de un usuario, debido a que este último puede requerir desempeño o encontrar recursos imperativamente. Para ello, emplea alguno de los dos protocolos de consulta propuestos en la presente tesis: paseos aleatorios inteligentes (*iRandomWalks*) o inundación con agentes de consulta.

Agente de empatamiento. En la presente tesis, un agente de empatamiento es aquel que se encarga de llevar a cabo el empatamiento de *peers* y el empatamiento de grupos, cuya finalidad es determinar cuáles *peers* satisfacen los requerimientos y las restricciones de una consulta de recursos.

Agente de enlace. En la presente tesis, un agente de enlace es aquel que determina si los recursos seleccionados y empatados están disponibles para su uso.

Agregación de recursos. Es un subconjunto de la colaboración de recursos que comprende las fases de anunciamiento, descubrimiento, selección, empatamiento y enlace. Las fases clave de la agregación de recursos son la selección, el empatamiento y el enlace de recursos.

Agregado. En GENI, un agregado (*aggregate*) es una colección de recursos similares gestionados por un administrador que proporciona sus recursos a los experimentadores de GENI.

Atributo. En la presente tesis, el término atributo hace referencia a una propiedad que caracteriza a un recurso, e.g., espacio total de almacenamiento en el disco duro.

Auto-organización. Es el proceso de un sistema en el cual un patrón a nivel global (macroscópico) emerge a partir de interacciones entre componentes de menor nivel.

Calidad de servicio. Es una métrica que mide la calidad de los servicios proporcionados, en términos de tasas de errores, ancho de banda, rendimiento, retraso en la transmisión, disponibilidad, *jitter*, etc.

Churn. Es el efecto colectivo que se produce como consecuencia de la llegada y partida de miles o millones de *peers*.

Clearinghouse. En GENI, un *clearinghouse* es un repositorio central que responde a usuarios agregando un conjunto de recursos propagados a través de múltiples dominios de administración.

Cohesión. En la presente tesis, este término hace referencia a la relación que existe entre dos fases de la colaboración de recursos. Existe un alto grado de cohesión entre dos fases cuando los resultados de una fase son las entradas de la otra fase.

Colaboración de recursos. Es el proceso en el que los recursos de diversos nodos de un sistema distribuido, son compartidos e interactúan entre sí para lograr grandes y complejas tareas, difíciles de conseguir utilizando un solo nodo. Este proceso comprende siete fases: anunciamiento, descubrimiento, selección, empatamiento, enlace, uso y liberación.

Componente de software. Es el elemento de un sistema de software que ofrece un conjunto de servicios, o funcionalidades, a través de interfaces bien definidas.

Condor ClassAd. Es un lenguaje que sirve para representar las características y restricciones de máquinas y trabajos en un sistema Condor.

Consulta de cero atributos (*zero-attribute query*). Son aquellas consultas de recursos que únicamente especifican el número de recursos requeridos sin declarar qué atributos son necesarios para cada recurso. Este tipo de consultas son las que se llevan a cabo en *grids*, nubes y nubes P2P, en donde los usuarios están interesados en encontrar un conjunto de nodos independientemente de sus capacidades.

Consulta de recursos. Es el método que se utiliza para obtener un conjunto de recursos que satisfacen los requerimientos y las restricciones de una aplicación en particular. Para llevar a cabo la consulta de recursos, es necesario especificar el conjunto de atributos y sus respectivos rangos de valores, así como expresiones booleanas que sirvan como restricciones de búsqueda. En la práctica, las consultas de recursos se pueden especificar usando *Extended Backus–Naur Form* (EBNF), *Virtual Grid Description Language* (vgDL) o documentos XML.

Consulta de un solo atributo (*single-attribute query*). Son aquellas consultas de recursos que unicamente especifican un atributo para un recurso en particular.

Consulta multi-atributo (*multi-attribute query*). Son aquellas consultas de recursos que especifican mas de un atributo para un recurso en particular.

Especificación de recursos. La especificación de recursos o RS (*Resource Specification*) es el documento mediante el cual un nodo anuncia los recursos que posee, detallando sus respectivos atributos y restricciones de uso.

Grupo de atributos. En la presente tesis, un grupo de atributos es un conjunto de atributos que describen a un recurso específico, e.g., el espacio libre en el disco duro y el espacio total en el disco duro forman parte de un grupo de atributos que describe a un recurso de almacenamiento.

Indice de cobertura. Es aquel que cubre todas las columnas de una tabla, incluyendo la llave primaria.

Inundación. Es un protocolo que consiste en enviar mensajes a todos los nodos vecinos (los cuales actúan de manera similar), en donde el proceso es detenido por un valor TTL con la finalidad de evitar el envío de peticiones. Esta técnica tiene la desventaja de causar tráfico en la red y de poseer una complejidad exponencial.

Mensaje de solicitud de recursos. En la presente tesis, un mensaje de solicitud de recursos es aquel que especifica los recursos que se pretenden utilizar de un *peer* proveedor, de acuerdo a los grupos óptimos en los que se desea que dicho *peer* trabaje.

Micelio. Es una masa de hifas que constituye el cuerpo vegetativo de un hongo.

Módulo de software. Un módulo de software, también conocido como paquete, es un contenedor cuyos elementos pueden ser otros módulos, componentes de software, interfaces y/o clases.

Nodo de reporte. En SWORD, un nodo de reporte es un nodo proveedor que anuncia sus recursos, mediante un documento XML de especificación de recursos, a una base de datos centralizada.

Nube P2P. Es una infraestructura de cómputo en la nube que ofrece una arquitectura distribuida, en donde los *peers* contribuyen con sus recursos no utilizados, mientras se lidia con fluctuaciones de recursos y una buena escalabilidad.

Orquestación de agentes. En la presente tesis se introduce este concepto, el cual se refiere al proceso en el que cada uno de los agentes de un sistema multi-agente interactúan entre sí para lograr un fin común, de manera que dichos agentes poseen

diferentes objetivos particulares y un mismo fin global. Este concepto está basado en la colaboración que existe entre los músicos de una orquesta sinfónica, los cuales tienen diferentes objetivos particulares (la ejecución de sus respectivos instrumentos) y un mismo fin global (el ensamble de una misma pieza musical).

Paseos aleatorios inteligentes. Es un protocolo propuesto en la presente tesis que está basado en el protocolo tradicional de paseos aleatorios, el cual propone reducir el ancho de banda en una red P2P, beneficiándose del aprendizaje de éxitos previos. Este protocolo garantiza que una consulta de recursos llegue a aquellos *super-peers* que anteriormente obtuvieron resultados para una consulta similar a la solicitada.

Peer normal. En la presente tesis, un *peer* normal es aquel que solicita recursos a su *super-peer* y recibe respuesta de este.

Peer. Son aquellos que son tratados de la misma manera en una red P2P y actúan como clientes y servidores simultáneamente. También se conocen como nodos.

Recurso de un único atributo (*single-attribute*). Es aquel recurso que está caracterizado por un único atributo.

Recurso dinámico. Es aquel recurso que cambia constantemente, e.g., el ancho de banda de la red.

Recurso estático. Es aquel recurso que nunca o muy difícilmente cambia, e.g., el número de núcleos de un procesador.

Recurso multi-atributo (*multi-attribute*). Es aquel recurso que está caracterizado por más de un atributo. Son los más populares en el mundo real.

Recurso. En la presente tesis, el término recurso hace referencia a un componente físico o virtual (e.g., capacidad de almacenamiento) que está caracterizado por un conjunto de atributos.

Red de sensores. Es aquella que está compuesta por cientos o miles de pequeños nodos, cada uno de ellos equipado con un sensor. La mayoría de estas redes utilizan una comunicación inalámbrica y los nodos cuentan comúnmente con recursos y capacidades de comunicación limitadas. En la mayoría de los casos, estas redes son usadas para llevar a cabo procesamiento de información.

Resiliencia. Es la capacidad de proveer y mantener un nivel aceptable de servicio ante la presencia de fallos que pudiesen alterar la operación normal de la red P2P.

Sistema multi-agente. Es un sistema en donde un conjunto de agentes interactúan entre sí con la finalidad de lograr un mismo objetivo.

Sistemas DCAS. Son aquellos sistemas que sensan constantemente el mundo físico utilizando sensores, actuadores y nodos de procesamiento. Comúnmente se utilizan para llevar a cabo la detección y predicción del clima.

Sistemas P2P colaborativos. Son aquellos que agregan un grupo de diversos recursos (e.g., hardware, software, servicios y datos) para lograr una tarea de gran escala, mediante las fases de la colaboración de recursos.

Sistemas P2P. Son sistemas distribuidos sin un control central, cuyos nodos son autónomos y equivalentes en funcionalidad.

Slice. En GENI, un *slice* es un conjunto de recursos virtualizados que se asignan a un experimento en particular.

Subagente de empatamiento. En la presente tesis, un subagente de empatamiento es aquel que se encarga de determinar la relación que existe entre dos *peers*, e.g., la latencia y el ancho banda.

Super-peer. En la presente tesis, un *super-peer* es aquel que efectúa peticiones en nombre de los *peers* que se encuentran bajo su responsabilidad, de manera que se encarga de seleccionar, empatar y enlazar recursos.

Toolkit. Es un conjunto de clases reusables y relacionadas, diseñadas para proporcionar funcionalidad de propósito específico y útil, e.g., un conjunto de clases que permite construir listas, tablas asociativas, pilas y otras estructuras similares.

Topología de anillo. Son utilizadas en DHTs para mantener un conjunto de *peers* que se encuentran distribuidos en un anillo (e.g., el diseño utilizado por el protocolo Chord), asignando un identificador a cada *peer*.

Topología. En la presente tesis, la topología es una red P2P no estructurada basada en *super-peers*, la cual puede variar dependiendo del módulo de agregación de recursos que se esté usando, e.g., es posible añadir un módulo que utilice una topología centralizada.

Red superpuesta (*overlay network*). Es una red que se construye encima de otra red. En sistemas P2P dinámicos los *peers* posiblemente poseen pocos vecinos (i.e., los *peers* de los cuales tienen conocimiento), a esta relación se le denomina *overlay* y se construye en la capa de aplicación.

Xenosearch. Es un mecanismo de búsqueda de recursos desarrollado para la plataforma abierta denominada XenoServer.

A.2. Siglas

ACM *Association for Computing Machinery*

API *Application Programming Interface*

CLI *Command-Line Interface*

CSC *Computing Classification System*

DCAS *Distributed Collaborative Adaptive Sensing*

DHT *Distributed Hash Table*

DICLOUD *Data-Intensive Cloud*

DOM *Document Object Model*

FOSS *Free and Open Source Software*

FPGA *Field Programmable Gate Array*

GENI *Global Environment for Network Innovations*

GPU *Graphics Processing Unit*

GUI *Graphical User Interface*

IaaS *Infrastructure as a Service*

IDE *Integrated Development Environment*

JDK *Java Development Kit*

JVM *Java Virtual Machine*

NP *Normal peer*

P2P *Peer-to-Peer*

PaaS *Platform as a Service*

QoS *Quality of Service*

RASupport *Resource Aggregation Support*

RPC *Remote Procedure Call*

RS *Resource Specification*

SA Sistema Autónomo

SaaS *Software as a Service*

SAX *Simple API for XML*

SDAQ *Single Attribute Dominated Queries*

SLA *Service Level Agreement*

SP *Super-peer*

SQL *Structured Query Language*

SSL *Secure Sockets Layer*

StAX *Streaming API for XML*

SWORD *Scalable Wide-Area Resource Discovery*

TTL *Time To Live*

UML *Unified Modeling Language*

VLAN *Virtual Local Area Network*

VM *Virtual Machine*

XML *Extensible Markup Language*

Bibliografía

[1] [Bandara and Jayasumana, 2013] H. M. N. Dilum Bandara and A. P. Jayasumana, *Collaborative Applications over Peer-to-Peer Systems - Challenges and Solutions*, Peer-to-Peer Networking and Applications, Vol. 6, No. 3 , pp 257-276, 2013.

[2] [Stoica, Morris, Karger, Kaashoek and Balakrishnan, 2001] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 149-160 , New York, USA, 2001.

[3] [Moca and Silaghi, 2009] M. Moca and G. C. Silaghi, *Resource Aggregation Effectiveness in Peer-to-Peer Architectures*, In Advances in Grid and Pervasive Computing, pp. 388-399, Geneva, Switzerland, May 4-8, 2009.

[4] [Bandara and Jayasumana, 2012] H. M. N. Dilum Bandara and A. P. Jayasumana, *Evaluation of P2P resource discovery architectures using real-life multi-attribute resource and query characteristics*, In Consumer Communications and Networking Conference (CCNC '12), pp. 634-639, Las Vegas, Nevada, USA, June 14-17, 2012.

[5] [Bandara and Jayasumana, 2011] H. M. N. Dilum Bandara and A. P. Jayasumana, *Characteristics of multi-attribute resources/queries and implications on P2P resource discovery*, In Proc 9th ACS/IEEE Int. Conf. on Computer Systems and Applications, pp. 173 - 180, Sharm El-Sheikh, Egipt, December 27-30, 2011.

[6] [Anderson, Cobb, Korpela and et.al., 2002] D. P. Anderson, J. Cobb, E. Korpela and et. al., *SETI@home: An Experiment in Public-Resource Computing*, In Communications of the ACM, Vol. 45 No. 11, pp.56-61, November, 2002.

[7] [Sullivan, Werthimer, Bowyer and et. al., 1997] W. T. Sullivan, D. Werthimer, S. Bowyer and et. al, *A new major SETI project based on Project Serendip data and 100,000 personal computers*, In Astronomical and Biochemical Origins and the Search for Life in the Universe, IAU Colloquium 161, pp. 729, Bologna, Italy, 1997.

[8] [Anderson, 2004] D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, In Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, pp. 4-10, 2004.

[9] [Bandara and Jayasumana, 2012] H. M. N. Dilum Bandara and A. P. Jayasumana, *Adaptive Sensing of Atmosphere: State of the art and research challenges*, In Globecom Workshops (GC Wkshps), pp. 1378-1383, Anaheim, CA, USA, December 3-7, 2012.

[10] [Bandara and Jayasumana, 2011] H. M. N. Dilum Bandara and A. P. Jayasumana, *On Characteristics and Modeling of P2P Resources with Correlated Static and Dynamic Attributes*, In Global Telecommunications Conference (GLOBECOM 2011), PP. 1-6, Houston, TX, USA, December 5-9, 2011.

[11] [Gómez, 2009] V. A. Gómez Pérez, *Soporte para la interacción de usuarios nómadas con áreas autónomas*, Master's thesis, Centro de Investigación y de Estudios Avanzados del IPN, Departament of Computer Science, December 2009.

[12] [Tanenbaum, 2007] A. S. Tanenbaum and M. V. Steen, *Distributed Systems Principles and Paradigms*, 2nd Edition, Prentice Hall, USA, 2002.

[13] [Gallardo, 2008] O. A. Gallardo Pérez, *Especificación de un framework para la construcción de un Sistema Distribuido Peer-to-Peer (P2P)*, Master's thesis, Universidad de los Andes, Departament of Compuer Engineering, February 2008.

[14] [Gamma, Helm, Johnson and et. al., 1995] E. Gamma, R. Helm, R. Johnson and et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st Edition, Addison-Wesley Pusblishing Company, pp. 26-28, USA, 1995.

[15] [Mandelbrot, 1982] B. Mandelbrot, *The fractal geometry of Nature*, 1st Edition, W. H. Freeman and Company, pp. 1-19, USA, 1982.

[16] [Álvarez, 2007] J. A. Álvarez Cedillo, *Algoritmo MPI para crear un fractal en el conjunto Mandelbrot*, Magazine from Universidad Cristóbal Colón, No. 19, pp. 157-166, 2007.

[17] [Jayasumana, 2012] A. P. Jayasumana, *File Sharing to Resource Sharing – Evolution of P2P Networking*, In IEEE Consumer Communications and Networking Conf. Tutorials (CCNC '12), 2012.

[18] [Bandara, 2012] H. M. N. Dilum Bandara, *Enhacing collaborative Peer-to-Peer systems using resource aggregation and caching: a multi-attribute resource and query aware approach*, Doctoral Thesis, Colorado State University, Fort Collins, Colorado, Department of Electrical and Computer Engineering, Fall 2012.

[19] [Berman et al., 2014] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhurif, R. Ricci and I. Seskar, *GENI: A federated testbed for innovative network experiments*, In Computer Networks, Vol. 61, pp.5-23, March 14, 2014.

[20] [Foster and Kesselman, 2004] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, 2nd Edition, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.

[21] [Karger et al., 1997] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web*, In Proc. 29th Annual ACM Symposium on Theory of Computing, pp. 654-663, May, 1997.

[22] [Ratnasamy et al., 2001] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content addressable network*, In Proc. ACM Special Interest Group on Data Communication (SIGCOMM '01), August, 2001.

[23] [Rowstron and Druschel, 2001] A. I. T. Rowstron and P. Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, In Proc. IFIP/ACM Int. Conf. on Distributed Systems Platforms, pp. 329–350, November, 2001.

[24] [Bharambe, Agrawal and Seshan, 2004] A. R. Bharambe, M. Agrawal, and S. Seshan, *Mercury: Supporting scalable multi-attribute range queries*, In Proc. ACM Special Interest Group on Data Communication (SIGCOMM' 04), August/September, 2004.

[25] [Shen, Apon and Xu, 2007] H. Shen, A. Apon, and C. Xu, *LORM: Supporting low-overhead P2P-based range-query and multi-attribute resource management in grids*, In Proc. 13th Int. Conf. on Parallel and Distributed Systems, December, 2007.

[26] [Cai, Frank, Chen and Szekely, 2004] M. Cai, M. Frank, J. Chen and P. Szekely, *MAAN: A multi-attribute addressable network for grid information services*, Journal of Grid Computing, January, 2004.

[27] [Zahn and Schiller, 2005] T. Zahn and J. Schiller, *MADPastry: A DHT substrate for practicably sized MANETs*, In Proc. 5th Workshop on Applications and Services in Wireless Networks, June/July, 2005.

[28] [Ganesan, Yang and Garcia-Molina, 2004] P. Ganesan, B. Yang and H. Garcia-Molina, *One torus to rule them all: Multi-dimensional queries in P2P systems*, In Proc. 7th Int. Workshop on the Web and Databases(WebDB '04), June, 2004.

[29] [Maymounkov and Mazières, 2002] P. Maymounkov and D. Mazières, *Kademlia: A peer-to-peer information system based on the XOR metric*, In Proc. 1st Int. Workshop on Peer-to-peer Systems (IPTPS '02), pp. 53–65, February, 2002.

[30] [Costa et al., 2009] P. Costa, J. Napper, G. Pierre and M. Steen, *Autonomous resource selection for decentralized utility computing*, In Proc. 29th Int. Conf. on Distributed Computing Systems, June, 2009.

[31] [Ranjan, Harwood and Buyya, 2008] R. Ranjan, A. Harwood and R. Buyya, *Peer-to-peer based resource discovery in global grids: A tutorial*, IEEE Communication Surveys, vol. 10, no. 2, 2008.

[32] [Dekar and Kheddouci, 2009] L. Dekar and H. Kheddouci, *A resource discovery scheme for large scale ad hoc networks using a hypercube-based backbone*, In Proc Int. Conf. on Advanced Information Networking and Applications, pp.293–300, May, 2009.

[33] [Tan, Han and Lu, 2008] Y. Tan, J. Han and Y. Lu, *Agent-based intelligent resource discovery scheme in P2P networks*, In Proc. Pacific-Asia Workshop on Computational Intelligence and Industrial Application, pp.752–756, December, 2008.

[34] [Kwan and Muppala, 2010] S. Kwan and J. K. Muppala, *Bag-of-tasks applications scheduling on volunteer desktop grids with adaptive information dissemination*, In Proc. 35th IEEE Conf. on Local Computer Networks (LCN '10), pp.560–567, October, 2010.

[35] [Lee et al, 2012] P. Lee, A. P. Jayasumana, H. M. N. D. Bandara, S. Lim and V. Chandrasekar, *A peer-to-peer collaboration framework for multi-sensor data fusion*, J. of Network and Computer Applications, vol. 35, no. 3, pp.1052–1066, May, 2012.

[36] [Oppenheimer et. al, 2008] D. Oppenheimer, J. Albrecht, D. Patterson and A. Vahdat, *Design and implementation tradeoffs for wide-area resource discovery*, ACM Transactions on Internet Technology (TOIT), vol. 8, no. 4, September, 2008.

[37] [Lo et al, 2005] V. Lo, D. Zhou, Y. Liu, C. GauthierDickey and J. Li, *Scalable Supernode Selection in Peer-to-Peer Overlay Networks*, Second International Workshop on Hot Topics in Peer-to-Peer Systems, pp. 18-25, 2005.

[38] [Liu et al, 2011] M. Liu, T. Koskela, Z. Ou, J. Zhou, J. Riekki and M. Ylianttila, *Super-peer-based coordinated service provision*, In J. Network and Computer Applications, vol. 34, pp. 1210–1224.

[39] [Sun et al, 2008] X. Sun, Y. Tian, Y. Liu, and Y. He, *An unstructured P2P network model for efficient resource discovery*, In Proc. Int. Conf. on Applications of Digital Information and Web Technologies, pp. 156–161, August, 2008.

[40] [Snyder, Greenstadt and Valetto, 2009] P.Snyder, R. Greenstadt and G. Valetto, *Myconet: A Fungi-Inspired Model for Superpeer-Based Peer-to-Peer Overlay Topologies*, In Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO '09), pp. 40-50, September 14-18, 2009.

[41] [Shen, Xu and Chen, 2006] H. Shen, C. Xu, and G. Chen, *Cycloid: A constant-degree and lookup-efficient P2P overlay network*, Performance Evaluation, vol. 63, no. 3, pp. 195-216, March, 2006.

[42] [Chun et al, 2003] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, *PlanetLab: an overlay testbed for broad-coverage services*, In Computer Communication Review, vol. 33, no. 3, pp. 3-12, 2003.

[43] [Spence and Harris, 2003] D. Spence and T. Harris, *Xenosearch: Distributed resource discovery in the xenoserver open platform*, In Proceedings of High Performance Distributed Computing, pp. 216-225, June 22-24, 2003.

[44] [Voulgaris, Jelasity, and van Steen, 2005] S. Voulgaris, M. Jelasity, and M. van Steen, *A Robust and Scalable Peer-to-Peer Gossiping Protocol*, In Second International Workshop, AP2PC 2003, pp. 47-58, July 14, 2005.

[45] [Snyder, Osmanlioglu, and Valetto, 2011] P. Snyder, Y. Osmanlioglu, and G. Valetto, *Biologically inspired attack detection in superpeer-based P2P overlay networks*, In Proc. Bionetics 2011, pp. 99-114, 2011.

[46] [Oliveira, Tiago, and et al, 2011] T. de Oliveira, T. Pessoa, A. Cardoso, and J. Júnior, *WChord: a Hybrid and Bio-Inspired Architecture to Peer to Peer Networks*, In Nature and Biologically Inspired Computing (NaBIC), pp. 353-358, Salamanca, Spain, October 19-21, 2011.

[47] [Levenshtein, 1966] V. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, In Soviet Physics Doklady, Vol. 10, No. 8, pp. 845-848, February, 1966.

[48] [Winkler, 1990] W. Winkler, *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*, In Proceedings of the Section on Survey Research, American Statistical Association, pp. 354-359, Anaheim, CA, 1990.

[49] [Lucia, 2013] D. Lucia, *MycoCloud Improving QoS by managing elasticity of services in decentralized clouds*, Master's Thesis, Politecnico di Milano, Milan, Department of Electronics and Information, Spring 2013.

[50] [Snyder, 2013] P. Snyder, *Modeling and Engineering Self-Organization in Complex Software Systems* , Doctoral Thesis, Drexel University, USA, Department of Computer Science, Winter 2013.

[51] [Pacitti, Akbarinia and El-Dick, 2012] E. Pacitti, R. Akbarinia and M. El-Dick, *P2P Techniques for Decentralized Applications*, Morgan & Claypool Publishers, April, 2012.

[52] [Oliver, 1994] I. Oliver, *Programming Classics: Implementing the World's Best Algorithms*, Pearson Education, April, 1994.