P2P-DIET: A QUERY AND NOTIFICATION SERVICE BASED ON MOBILE AGENTS FOR RAPID IMPLEMENTATION OF P2P APPLICATIONS

by

Stratos Idreos

A thesis submitted in fulfillment of the
requirements for the degree of

Electronic and Computer Engineering

Technical University of Crete
Department of Electronic and Computer Engineering

Intelligent Systems Laboratory

Abstract

The paradigm of data sharing peer-to-peer systems has recently become a hot area of research due to the popularity of file sharing applications such as Napster, Gnutella and KazaA that adopt the "information pull" model. Peer-to-peer systems exhibit many interesting features like adaptivity, load balancing, self-organization, fault-tolerance and the ability to pool together large amounts of resources and put them at the disposal of a large community of users. In the last few years, the publish-subscribe paradigm has also emerged as a very promising one for the development of Internet applications concentrating on the "information push" model.

This dissertation presents the design and development of P2P-DIET, a peer-to-peer service that unifies the above two paradigms. The service has been implemented using the mobile agent system DIET Core developed in project DIET. Our service supports queries, subscriptions and notifications in a single unifying framework, and can be used for building various peer-to-peer applications like file sharing, e-commerce, and information dissemination over the Web.

P2P-DIET contains a fault-tolerance mechanism that guarantees connectivity, when nodes of the network fail or leave silently. The capability of location-independent addressing is supported, which enables the use of dynamic IP addresses. Because of this capability, nodes can disconnect and reconnect with a different address and at different parts of the network.

To demonstrate P2P-DIET, a file sharing application was implemented on top of it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Peer-to-peer systems have recently become a very active research area and very popular though file sharing applications such as Napster [49], Gnutella [28] and KazaA [40]. Much attention has also been focused on the copyright issues raised by these applications. Peer-to-peer systems exhibit many interesting properties like adaptivity, load balancing, self-organization, fault-tolerance and the ability to pool together large amount of resources.

The event notification or publish-subscribe paradigm has also emerged recently as a very promising way of building Internet-based systems. In an event notification service, users subscribe with their profile to events of interest and then asynchronously receive notifications on such related events. This dissertation presents the development of a peer-to-peer system that unifies both of the above paradigms.

1.1 Overview

In this dissertation we design and implement an Internet-scale, agent-based, peer-to-peer system which supports *queries*, *profiles* and *notifications*. The service itself can be used for building various peer-to-peer applications like file sharing, e-commerce, network management, stock market analysis and software administration. The system contains a fault-tolerance mechanism that guarantees connectivity, when nodes of the network fail or leave silently. The capability of location-independent addressing is supported, which enables the use of dynamic IP addresses. Because of this capability, nodes can disconnect and reconnect with a different address and at different parts of the network.

A file sharing application was implemented on top of the service. The users of the application can:

- publish their files so that other users may see and download them.
- query the system to search for files on the whole network.

- subscribe with a profile which will continue to produce notification on future resources too.
- receive notifications on files of interest owned by other users.
- receive stored notifications, which were produced at a time that the client was not online to receive them.
- arrange rendezvous with a file if the resource owner is not online.
- use dynamic IP address and different access point nodes of the network.
- download files directly from the resource owner user.
- locate other clients and possibly chat with them.

Peer-to-peer systems are currently a very popular alternative to centralized client-server systems. In its pure form, a peer-to-peer system has no servers and no functionality is centralized. All nodes of the network are equal peers. Adaptation, load-balancing, self-organization, fault-tolerance and the ability to pool together large amount of resources, are some of the benefits of peer-to-peer systems. In the last years, peer-to-peer systems have become the most popular way for users of the Internet to share huge amount of data. There are three main classes of peer-to-peer applications. In *parallelizable systems* a large task is broken into small subtasks and each one of the subtasks can be executed in parallel in different nodes of the network, for example, Seti@home [69]. Moreover, there are *file sharing* peer-to-peer systems like Napster [49], Gnutella [28], Freenet [25] and KazaA [40]. Another class of peer-to-peer applications are the collaborative systems, where the users collaborate in real time. Games are a type of collaborative system.

Event notification systems are systems that allow one to subscribe with a *profile* of interest or *long-standing query* so that he is notified when certain events of interest take place. In the file sharing scenario an event can be the action of a user to publish a file or subscribe with a profile. A profile is a long-standing query, that continues to produce results as time goes by and new resources are added to the system. Examples of event notification systems are intergrated development environments [45, 62, 65, 7], work-flow and process analysis systems [50, 35], graphical user interfaces [52], network management systems [46], software development systems [63] and security monitors [37, 8, 27].

Agent-based computing offers many desired characteristics for implementing a peer-to-peer system that supports queries, profiles and notifications. Software agents are dynamic, adaptable, computational entities that function continuously and autonomously in a particular environment, often inhabited by other agents or processes. They are able to inhabit and migrate to complicated, unpredictable, dynamic and heterogeneous environments [54, 70, 66]. Software agents carry out

activities in a flexible manner and are sensitive to changes in the environment, without requiring constant human guidance. From a design point of view, all these characteristics make the agents the right abstraction to represent peers in a peer-to-peer context, and their complex interactions can be the basis for a cooperative multi-agent system. Much research has been focused on agent-based systems and many agent development toolkits have been constructed in the last years, for example, JADE [22], OAA [20], RETSINA [38], ZEUS [29] and DIET Core [14]. The agent platform called DIET Core was used to implement P2P-DIET and will be briefly discussed in the next section.

1.2 The DIET Platform

DIET stands for Decentralized Information Ecosystem Technologies. DIET is a 5th Framework project funded by the European Commission under the Future and Emerging Technologies area [21, 16]. One of the goals of the project was the design of a multi-agent platform that is open, robust, adaptive and scalable based on an ecosystem-inspired approach. In the first year of DIET such a multi-agent platform has been developed called DIET Core [55, 14].

A basic concept in DIET Core is the *world*. There is one world per Java Virtual Machine (JVM). A world is an *environment* repository and can contain one or more environments. Environments in DIET Core provide a location for agents to inhabit and can host one or more agents. The DIET platform can contain more than one worlds. This is for instance the case when DIET Core runs on multiple computers. This can be viewed as a DIET *universe* that contains multiple planets (worlds). Each environment can have neighbor links to other environments, which are not necessarily in the same world. Each environment can be connected with other environments which are not necessarily in the same world, through *neighbor links*. These links allow agents to migrate to different environments and explore the DIET universe, without having any prior knowledge of the location of the environments. Those links are specified when the world is set-up but can change dynamically too, if the DIET universe changes.

The agents in DIET Core are very *lightweight*. They do not need a lot of memory to run while DIET ensures that agents give up their thread if they do not need it. The DIET Core will give a thread to an agent, when it needs it, for example, when it must handle an incoming message. The lightweight character of the agents allows several hundred thousands of agents to run in the same JVM on an ordinary desktop computer.

The support of the DIET Core for communication and agent migration is minimal. There is no specific protocol contained in the DIET platform, so there is no overhead and execution can be rapid. One the other hand, if one wants to use an agent communication protocol such as the ones defined by FIPA [24] then this protocol must be developed from scratch.

The DIET Core will directly expose agents to potential failure, which enhances the robust character of the platform. The core will satisfy a request, only if it can easily do so. For example local message delivery fails if the agent, who is supposed to receive the message can not be assigned a thread or his message buffer is full. In this way, no extra resources are spent, when the system is already overloaded. For more details on the DIET platform see [55, 14].

1.3 Contributions of the Dissertation

This dissertation presents the critical decisions, strategies, problems and solutions that occurred while designing and implementing P2P-DIET. There is a gap currently in the peer-to-peer systems literature since there are systems with only query capabilities, i.e., Gnutella [28] and systems with only profile/notification capabilities, i.e., SIENA [1]. We designed P2P-DIET in response to the obvious need to unify these paradigms. In this way, P2P-DIET unifies both queries and notifications.

We present the agents that inhabit the environments of P2P-DIET universe. There are two different types of environments. The *client peer* environment in the client nodes and the *super-peer* environment in the super-peer nodes. Each environment hosts different types of agents. We define their responsibilities and their goals. The goal of an agent is sometimes a part of a greater goal and he works together with other agents to satisfy that goal. Local communication is achieved by a direct connection between the two agents. Remote communication is based on mobile agents, the *messenger* type of agents. Those agents migrate to different environments to deliver messages and they keep travelling around the network for ever.

We define the routing strategy. The goal is to establish the appropriate routing paths in order to use the network efficiently loading it with the least possible overhead. We show how we build minimum weight spanning trees and shortest paths in the distributed environment and how we use them to satisfy the demands of the network in terms of routing messages. We present the fault-tolerance mechanisms and the reasons that make such mechanisms necessary. We allow the use of dynamic IP addresses and we assume that a node will not be connected for ever to the same part of the network. We define the additional issues to handle in order to support the previous capabilities. Moreover, we present in detail the process of adding or removing servers or clients from the network, in a way that guarantees stability and connectivity. We present the socket handling strategy, that enables the super-peer nodes to use the minimum number of resources.

We present the query and event notification service in detail. We give details for the basic concepts of our query and event service, which are *profiles*, *resources*, *notifications*, *queries*, *stored notifications* and *rendezvous*. We describe the propagating strategy, which concludes a profile hierarchy. We show how we

build and use the hierarchy and why it is so useful for the network. We present the language currently supported by P2P-DIET. This language is used in the profiles and queries in order for the clients to describe their interests and query the system.

The software agents must communicate to change information and cooperate. We present the communication protocols that the agents in P2P-DIET use for local and remote communication. There are four different protocols:

1. The *super-peer - super-peer* communication protocol, which is used by agents in a super-peer environment to communicate with agents in the local or a remote super-peer environment.

2. The *client peer - super-peer* communication protocol, which is used by agents in a client peer environment to communicate with agents in a remote super-peer environment.

3. The *super-peer - client peer* communication protocol, which is used by agents in a super-peer environment to communicate with agents in a remote client peer environment.

4. The *client peer - client peer* communication protocol, which is used by agents in a client peer environment to communicate with agents in the local or a remote client peer environment.

Note that remote communication is achieved by the Messenger agent. This means, that when an agent wants to communicate with a remote agent, he communicates with a local messenger agent and the messenger will implement the remote communication. Moreover, we present the different scenarios that take place in P2P-DIET, by showing in detail the conversations between agents, using UML sequence diagrams.

1.4 Organization of the Dissertation

This dissertation is organized as follows. In Chapter 2 we briefly discuss alternative peer-to-peer architectures supporting the query/answer paradigm or the profile/notification paradigm. Chapter 3 presents the architecture of P2P-DIET, the mobile agents implementing the required functionality and their responsibilities in the P2P-DIET universe. Moreover, in Chapter 3 we discuss the data model and language that the system currently supports for queries, profiles and notifications for a file sharing application. Chapter 4 presents the routing strategy in our system and the way different nodes are connected to or disconnected from the network. In Chapter 5 we discuss the event notification mechanisms and the propagating strategy. Chapter 6 describes in detail the different communication protocols used by the agents. In Chapter 7 we present our conclusions and future

work possibilities and finally Appendix A presents the different scenarios that take place in the network.

Chapter 2

Related Work and Alternative Architectures for P2P Systems

The process of designing a distributed peer-to-peer query and event notification service requires careful study of the different peer-to-peer architectures and their tradeoffs. In this chapter we discuss some alternative architectures for peer-to-peer systems. We briefly present some well-known architectures and informally discuss their advantages and disadvantages, in terms of fault-tolerance, routing, scalability and other fundamental network issues. Four basic architectures are considered: hierarchical, pure peer-to-peer, centralized peer-to-peer and super-peer based.

2.1 Some Distinctions

In the literature on peer-to-peer architectures one can easily distinguish between two classes of systems:

- Systems such as Napster [49] and Gnutella [28] where the primary mode of interaction is a user *query* (for example, "I am interested in songs by Jethro Tull") to which the system returns an *answer* which is a list of *matching* resources. We will call these systems *query systems*.

- Systems such as SIENA [1, 3] and DIAS [44] where the primary mode of interaction is a *user profile* that is input to the system. Later on, *events* at any node of the system can give rise to *notifications* that inform the users about the availability of resources matching their profiles.

These two classes of systems have been so far treated independently in the literature. Napster, Gnutella, KazaA and the like have monopolized the buzzword "peer-to-peer" while systems like SIENA, clearly based on peer-to-peer architectures, usually go by the name *event notification systems*. Examples of

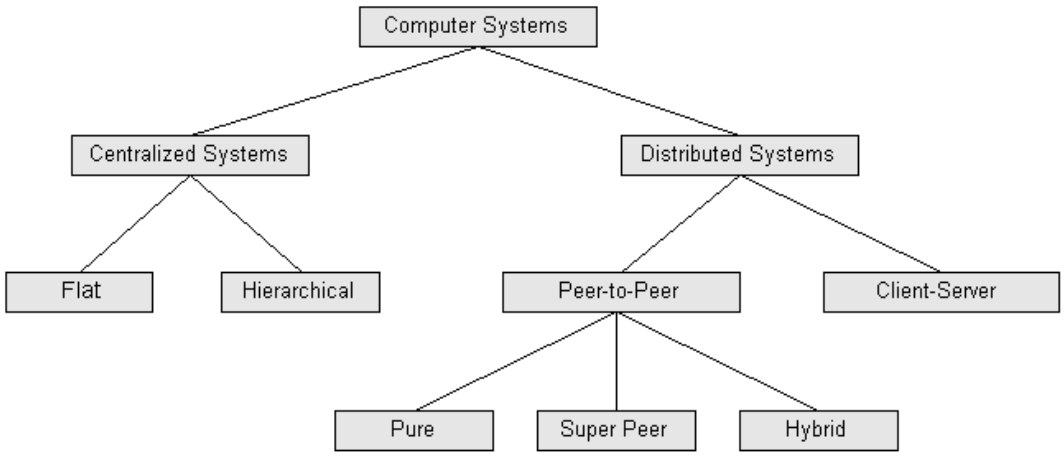

Figure 2.1: A Taxonomy of Computer Systems

event notification systems can be found in applications such as integrated development environments [45, 62, 65, 7], work-flow and process analysis systems [50, 35], graphical user interfaces [52], network management systems [46], software development systems [63] and security monitors [37, 8, 27].

We believe that the different functionalities offered by these two classes of systems are all very useful when offered as basic services for the development of peer-to-peer applications. Thus we made them the core services in our system P2P-DIET. Our presentation in the rest of this section owes a lot to [1, 3].

Now that we have made the above distinction and motivated our work, let us discuss various architectural alternatives that we could have followed in our implementation. We blur the distinction between query systems and event notification systems because all architectures we present can be used in both cases.

2.2 A Taxonomy of Computer Systems

Computer systems can be classified as *centralized* or *distributed*. Distributed systems can be further classified into systems that follow the *client-server* model and systems that follow the *peer-to-peer* model. The client server model may be *flat* or *hierarchical*. In the flat model all clients communicate with a single server only. In the hierarchical model servers form a hierarchy as discussed in Section 2.3. The peer-to-peer model can be *pure* or *centralized*.

In *pure* peer-to-peer systems there is no centralized functionality, while in *hybrid* systems the opposite is true. We adopt the hybrid terminology from [9, 10]. Moreover, there is an intermediate solution of *super-peer* systems, where the super-peer nodes form a pure peer-to-peer network but each super-peer is connected with clients in a centralized way. Those models will be discussed in detail later in the chapter. Figure 2.1 shows this taxonomy of computer systems.

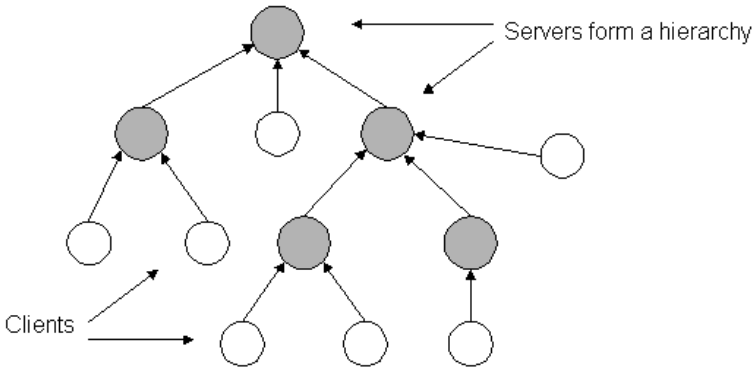

Figure 2.2: The hierarchical architecture

2.3 Hierarchical Architecture

The hierarchical topology is an extension of the simple client-server architecture. Any two servers are connected in a client-server way so that all servers form a tree as shown in Figure 2.2. Each server has one *master server* and one or more *child servers*. The server that has no master server is the *root*. It is understood that servers have two roles. They act both as clients and as servers in a network of servers, and as servers to their clients. This is the topology of the distributed implementation of the event dispatcher JEDI [26]. In this topology a server does not distinguish between other servers and clients, except of course his master server. In the case of an event notification system, this means that a server is able to receive notifications, subscriptions or resources, but it will be able to send only notifications to clients and servers. The network of servers forms a directed graph with the obvious advantage of the unique paths between the nodes. This is great in terms of routing, because the decision of how to send a message is trivial. On the other hand, there are problems such as the overloading of servers and fault-tolerance. Since there is no second path connecting two servers, it is reasonable to assume in large scale networks that some servers will be overloaded, because they will be the only way to connect other servers. Moreover, when a server fails, its child subnetworks are disconnected from the network and the network is cut into many parts. The higher in the hierarchy the problem occurs, the more important it is, because a bigger part of the network will be cut off.

2.4 Peer-to-Peer Systems

In peer-to-peer systems the nodes of the network have equal roles in the exchange of information and services. Various systems have been labelled as peer-to-peer systems in different application domains. Some well-known systems are those used for the exchange of music files, for example Napster [49], Gnutella [28],

Freenet [25] and KazaA [40]. Another well-known file sharing system is Pointera [58]. Then there are systems like SETI@Home [69], where users offer computing cycles or instant messaging systems like ICQ [33] to exchange personal messages. Finally, there are collaboration systems like Groove [56]. The big advantage of peer-to-peer systems is that they can create very large pools of information and computing power by gathering the resources and CPU cycles of their users. Additionally, the existing peer-to-peer systems are very strong in terms of load-balancing, self-organization, adaptation and fault-tolerance. The strengths of grass roots peer-to-peer systems have motivated many research projects to focus on understanding the problems in such systems and improve their performance, for example, [23, 34, 6]. Much research has been done in the area of improving the search efficiency in peer-to-peer environments by creating better search protocols, for example, Chord [32], Pastry [5], CAN [64] and Tapestry [12] by supporting point queries, and [11, 4] by supporting more expressive queries. For recent detailed surveys of peer-to-peer systems the reader is asked to consult [17, 47].

2.5 Pure Peer-to-Peer Networks

In pure peer-to-peer systems all peers have equal roles and responsibilities. This stands for all aspects such as query, download, publishing etc. No functionality is centralized in pure peer-to-peer systems. Every node is a *servent* (both a client and a server). Each node can equally communicate with any other connected node (neighbor). Gnutella [28] and Freenet [25] are good examples of pure peer-to-peer networks.

Gnutella is a file sharing protocol. The applications that are on top of Gnutella allow users to search and download files from other users. Gnutela users must know the IP address of a Gnutella node to connect to the network. When a user wants to search, he sends a query to all his neighbors. The neighbors may respond and they will forward the query to all their neighbors[1]. The message will actually travel only to a limited number of nodes using a time-to-live (TTL) mechanism. Gnutella does not contain a fault-tolerance mechanism. Various file sharing applications have been implemented using the Gnutella protocol, for example, [42, 72, 13]. The main advantage is that such systems distribute the main cost of sharing data (bandwidth and storage) across the peers, which means that the network can scale without the need for powerful servers. Despite their many strengths pure peer-to-peer networks that work as described tend to be inefficient. For example, Gnutella network uses flooding to publish query messages and this results in heavy network traffic. Another very important factor of inefficiency is the existence of peers with limited capabilities. The fact that all peers have equal roles causes bottlenecks. For example, Gnutella had low performance in August 2000 when some peers connected by dialup modems became overloaded

[1]This process of routing a message is called *flooding*.

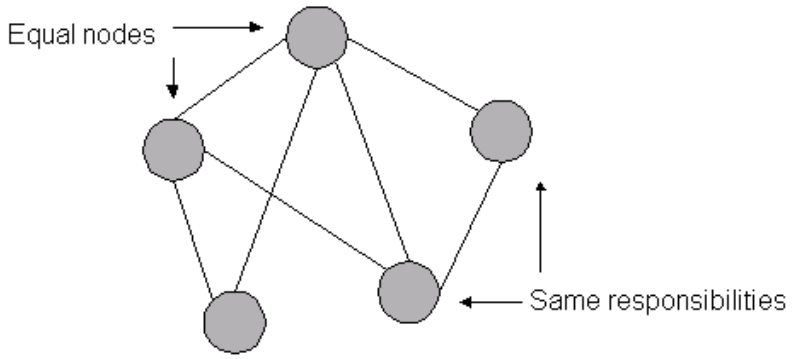

Figure 2.3: Pure peer-to-peer architecture

after a while as shown in study [39]. The obvious conclusion is that in order to build an efficient system one must take advantage of the heterogeneity among the peers. More capable peers with more resources (disk space, CPU and bandwidth) should have greater responsibilities than others. An example of a typical pure peer-to-peer network is shown in Figure 2.3.

Freenet is available in an Open Source reference implementation and it is a file sharing system based on the designs of [30, 31]. The main goal of Freenet is the ability of the clients to be anonymous. In this way, a user makes requests but no one knows his identity. Freenet is completely decentralized and its scalability has been studied in [31].

2.6 Centralized Peer-to-Peer Networks

Centralized peer-to-peer systems are a combination of pure peer-to-peer and ideas from centralized systems. In this case search takes place over a centralized directory but download still works in a pure peer-to-peer way, which means that peers are equal in download only. A well-known centralized system is Napster [49], where clients connect to a server that keeps a directory of the resources of all peers in the network. There are shortcomings in centralized peer-to-peer systems too. Although centralized search can sometimes be more efficient than distributed search, the cost of keeping an index for the resources on a single node is high, because the single node may become a scalability bottleneck and a single point of failure. Moreover, the fact that some peers are more important than others makes the system more vulnerable to attacks. Napster servers are organized in an *unchained architecture* as shown in Figure 2.4, which means that a user can see only files of users connected in the same server. Users in Napster cannot search for files globally, because they are restricted to searching on a single server that has indexed only a fraction of the available files in the network. The only way to achieve global search is for the user to connect to all servers (one at a time) and search. The advantage of this architecture is the ability of

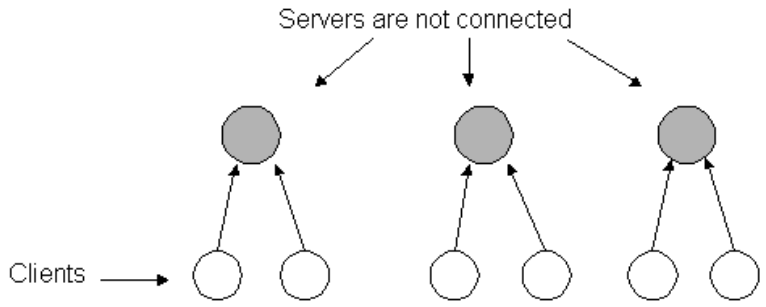

Figure 2.4: Napster unchained architecture

the network to scale, basically because of the more lightweight character of the servers. Another example of centralized peer-to-peer architecture is Groove [56]. Groove is a collaborative peer-to-peer system. The goal is to enable communication, content sharing and tools for joint activities [53] without relying on a server [56]. Moreover, Groove tries to guarantee security, privacy and flexibility.

The full *replication approach* is a solution to the global search problem of the unchained architecture of Napster. Each server replicates all information and in that way, it keeps a centralized index and its clients can search globally using this index. This approach is used in NTTP [7] and a variation in the Konspire [41] peer-to-peer system. Possible architectures for centralized peer-to-peer systems have been studied in detail in [10].

2.7 Super-Peer Networks

Super-peer systems are a cross between pure and centralized systems. A super-peer is a node of the network, that acts both as a server to a subset of clients, and as an equal in a network of super-peers. As discussed in [9], because super-peers act as centralized servers to their clients, the query process is more efficient than the one in Gnutella, where each individual client should handle queries. Moreover, the fact that there are many super-peers in the network, guarantees that no super-peer will need to handle a very large load and possibly become a single point of failure for the entire system. Clients are connected only to a single super-peer in a client-server way and are equal to each other in terms of download only. An example of super-peer system, is KazaA [40]. The super-peer nodes in the KazaA network are powerful nodes on fast connections that are generated automatically. Client peers connect to their local super-peer node to upload information on the files that they want to share and to send a query to the network. KazaA tries to improve download speed and to make downloading more reliable. A file will be downloaded from the fastest connection available. Moreover, it will be downloaded in parallel from different sources to speed up the download. Finally, a failed transfer will be resumed automatically.

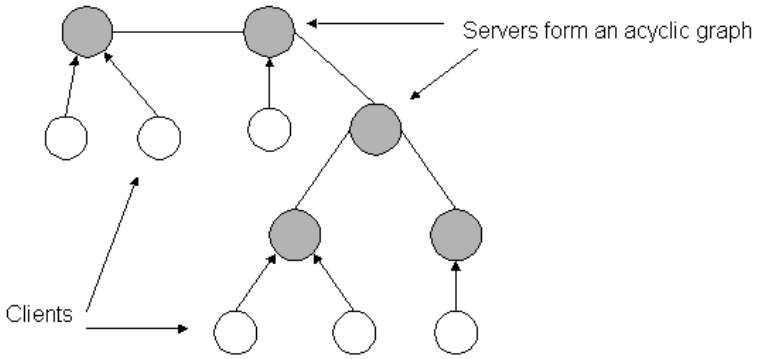

Figure 2.5: Acyclic peer-to-peer architecture

The interconnection topology of super-peers is of great importance for the network, because it affects critical functionalities such as the routing algorithms for efficient message delivery and fault-tolerance. In the next three subsections we will shortly present two basic topologies: *acyclic peer-to-peer* and *general peer-to-peer*.

2.7.1 Acyclic Peer-to-Peer Architecture

In the acyclic peer-to-peer architecture all servers are equal (peers). They communicate with each other symmetrically as peers, which means that information flows both ways. The graph composed by the servers, is an acyclic undirected graph as shown in Figure 2.5. Since there are no cycles, the communication path between two super-peers is unique. Again the routing problem has a trivial solution. We can either force super-peers to forward all messages to all their neighbors except the sender or store information on the paths for each super-peer and forward messages only to the appropriate neighbors each time. The main problem of the acyclic topology is fault-tolerance, since the failure of a single server can isolate a big subset of the network. Moreover, the network will not be flexible enough to handle critical situations, for example, the fact that one super-peer is overloaded because that super-peer will be in the single path of other super-peer messages. The USENET news system is an example of this architecture. The topology of the news servers is very similar to the acyclic peer-to-peer topology. The main protocol in USENET news is the NTTP [7]. NTTP provides both client-server and server-server commands. The selection mechanisms in USENET news are not very sophisticated, which limits its usability as an event service.

2.7.2 General Peer-to-Peer Architecture

If we remove the constraint of acyclicity from the acyclic peer-to-peer architecture, we obtain the general peer-to-peer architecture. The topology of the servers

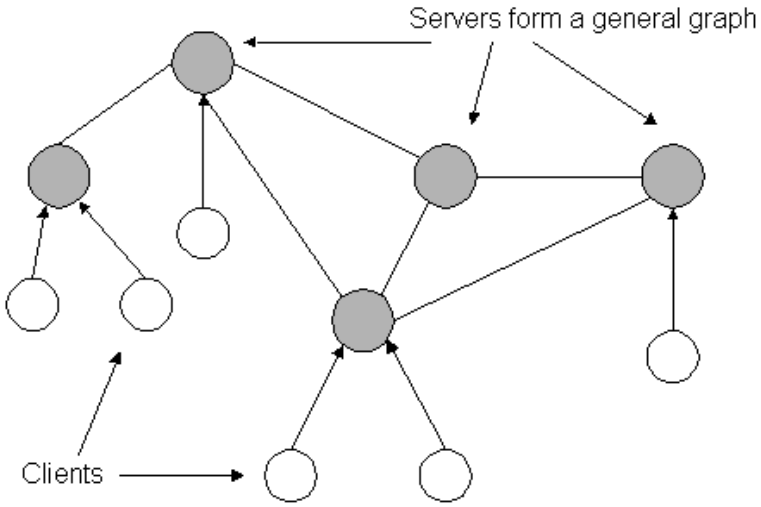

Figure 2.6: General peer-to-peer architecture

is a general undirected graph as we see in Figure 2.6. In this architecture, cycles are allowed. It is possible that one server can communicate with another through multiple paths and this is the difference between general peer-to-peer architecture and the previous ones. The communication of two connected servers is bidirectional as in the acyclic topology. The advantage is that the network is more robust. When a server fails its subnetworks can be reached through other paths, which means that unless that server had only one neighbor[2], the rest of the network can adapt to that situation and keep working properly. Intuitively, the more connected the servers are, the more robust the network will be. Of course, we should not expect real networks to be fully connected. The problem of having appropriate connectivity to achieve robustness is an important one.

A general peer-to-peer topology of servers has to face the problem of routing. Routing has no trivial solution in a general undirected graph. Each super-peer needs more information on the topology of the network to successfully deliver messages. This problem is discussed in detail in Chapter 4.

2.8 Summary

In this chapter we discussed alternative architectures for peer-to-peer systems. We briefly made the distinctions of typical query systems and event notification systems. We discussed the hierarchical architecture, the pure peer-to-peer architecture and continued with the centralized approach and the super-peer approach for building peer-to-peer networks. We considered the possible interconnection topologies of servers in a super-peer network: acyclic peer-to-peer and general

[2]A neighbor is a directly connected server.

peer-to-peer. In the rest of this dissertation we will present the architecture of P2P-DIET, we will analyze in detail the routing solutions (routing strategy) in P2P-DIET, the notification mechanism and the strategy for propagating notifications. Moreover, we will give details on the implementation such as the different agents that were created, the communication protocols that they use and the different scenarios that can take place in our peer-to-peer query and notification service.

Chapter 3

P2P-DIET Architecture and Agents

In the previous chapter we presented some alternative peer-to-peer architectures and some well-known related systems. This chapter presents in detail the architecture chosen for P2P-DIET. Additionally, we present the agents that were implemented, their responsibilities, their goals and how they interact with each other and with the users of the network. We describe the language that the clients use for queries and profiles. Note that in this chapter, we give a high level view of our work and many of the critical points will be explained in detail in the following chapters.

3.1 P2P-DIET Architecture

Our goal is to build a peer-to-peer system that supports queries, profiles and notifications. As we argued informally in the previous chapter, the most flexible architecture is that of a super-peer network, which allows super-peers to communicate through multiple paths and form a general undirected graph as shown in Figure 3.1. There are two kind of nodes in P2P-DIET. The *super-peer* nodes and the *client* nodes. Super-peers are connected in such a way that they form a general undirected graph. All super-peers are equal and have the same responsibilities, so the internal network consisting only of the super-peer nodes is a pure peer-to-peer network. Each super-peer serves a fraction of the clients and keeps indices on the resources of those clients. A client node is a computer of a real user of the system. Recourses - files that users want to share with others - will be kept in the computers of the users, although it is possible in special cases to store resources in a server[1]. Clients are equal to each other only in terms of download. When a client wants to actually download a resource, it downloads it in a pure peer-to-peer way from the resource owner client. A client node is

[1]This functionality is called rendezvous and will be explained in Chapter 5.

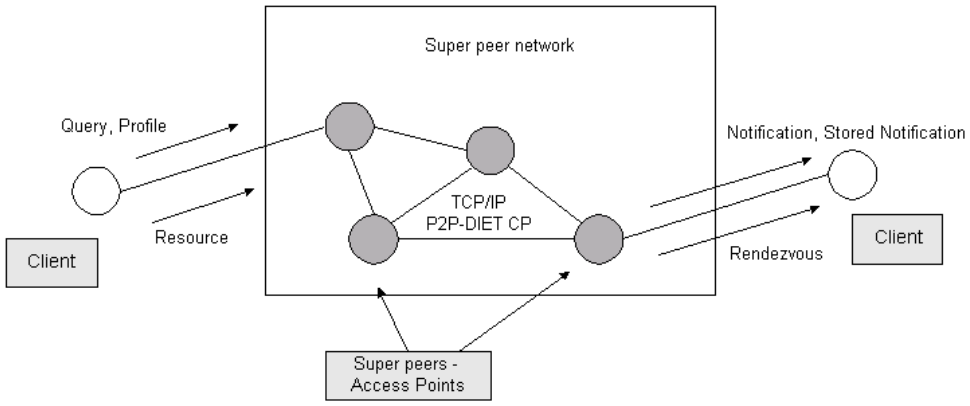

Figure 3.1: P2P-DIET architecture

connected to the network through a super-peer node, which is the *access point* of the client and as we will see, it is not necessary for the client to be connected to the same access point continuously. By adopting such topology, we can handle critical situations, like the failure of a node efficiently and develop a peer-to-peer service that supports query and notification scenarios.

Clients may *publish* a resource by sending metadata for the resource to their access point in order for other clients to see it. A resource is a file of any type, for example, music files or Microsoft Word documents. Moreover, clients may *subscribe* their profile which is a long-standing query. This means that by subscribing the profile, the user will expect to find all files that exist on the network and all files that will be published in the future. Additionally, clients receive *notifications* on a resource through their access point. A notification is a pointer to the real resource that is stored on the computer of the resource owner client. The client will use the notification to identify the resource owner and the resource name in order to be able to download it. Super-peers propagate the profiles of their clients to all other super-peers. In this way, when a new resource is added somewhere in the network, all profiles will be checked in case there is a match with the resource. This process guarantees, that the profiles will continue to produce notifications (long-standing query) for future resources. We could propagate resources instead, but we expect our clients to have numerous resources but only a few profiles. For example, in a file sharing application where users share music files, a user may have thousands of resources (music files), while he might be interested in a specific number of artists.

Each super-peer has a *profile hierarchy* to organize the process of profile forwarding. A profile is forwarded only if there is not any other more general profile already forwarded. The profiles that lie to the level one of the hierarchy are those that are actually forwarded, because there are not any other more general profiles. Those forwarded profiles have children in the hierarchy, which are profiles less general than their father (the forwarded profile), and those children may have

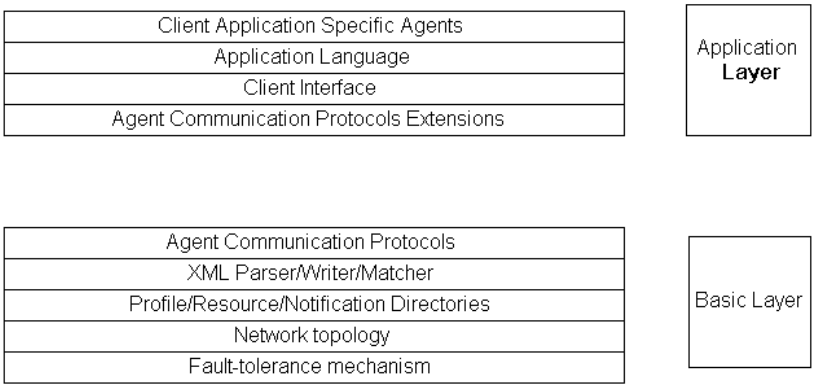

Figure 3.2: Layered View of P2P-DIET

other children etc. The depth of the hierarchy is not limited by the system. Using the hierarchy for the profiles, a super-peer receives less notification messages but continues to satisfy all the profiles of his clients. One notification message may satisfy a large number of profiles. Thus, when a notification arrives in a super-peer environment, the local super-peer agent must check if the notification must be delivered to more than one client agents. The clients to be checked are those who own a profile which is a child of the profile that gave rise to the notification.

Additionally the system supports the typical *ad-hoc query scenario*, where clients query the system to find matching resources anywhere in the network, without affecting their original profile, which will continue to produce notifications as new resources are published. Ad-hoc queries are answered by using essentially the same notification mechanism that is used for long-standing queries.

To enhance the general form of the network, we assume that clients (users) use dynamic IP addresses and can connect or disconnect anytime or even leave from the system silently. This brings new aspects in the notification character of our system. We need to handle situations where clients may loose notifications, because they are not connected at the moment that the notification is produced or cannot download a resource, because the resource owner is not connected. The answers to the previous problems are *stored notifications* and *rendezvous files*. The network will store notifications for a client, if he is not there to receive them and will try to upload a file from a client, so the interested client does not have to wait to be connected at the same time with the resource owner.

Profiles, resources and notifications are XML DOM trees. It is much cheaper to transmit the DOM trees than the actual XML file. Moreover, the DOM tree is ready to read or write anytime. On the other hand, an XML file has to be parsed. Of course, clients and super-peers keep backup XML files of the DOM trees. This is necessary because a client will not keep his computer on and the client application of P2P-DIET running. When the application is terminated

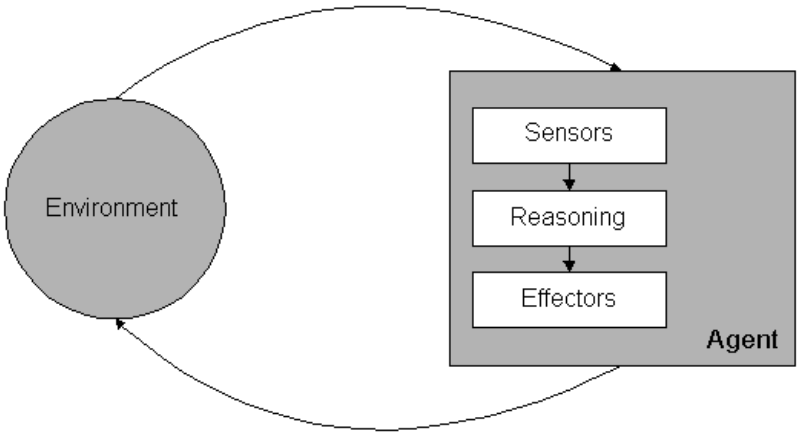

Figure 3.3: A software agent interacts with an environment through sensors and effectors

(exit client program) the DOM trees will be lost. Super-peers are servers which means that they are not going to be terminated, so they keep XML files just for backup. SAX technology was not an option because of the need to check for matches very often, so if we used SAX we would have to parse the XML files every time we wanted to make a check. A high-level view of the network is shown in Figure 3.1. A layered view of P2P-DIET is shown Figure 3.2. The agent communication protocols in Figure 3.2 are those that described in Chapter 6.

3.2 Agents

Until now we have not defined the concept of an agent. Software agents as described in [36] have certain characteristics. They are dynamic, adaptable, computational entities that function continuously and autonomously in a particular environment, often inhabited by other agents or processes. Software agents are able to inhabit complicated, unpredictable, dynamic and heterogeneous environments [54, 70, 66]. Our desire is a software agent to carry out activities in a flexible manner and to be sensitive to changes in the environment, without requiring constant human guidance as shown in Figure 3.3. A software agent should be able to learn from its experience, after a long period of time. Additionally, we expect agents that inhabit an environment with other agents or processes, to be able to communicate and work together with other agents to solve a problem or a set of problems. Moreover, software agents should be able to move-migrate to different environments. The goal of an agent can be very simple or very complicated. Moreover, the goal of a software agent can be an intermediate goal to achieve a final goal. In this way, more than one agents work independently to satisfy the final goal. An agent has to use reasoning to achieve his goals. In

general, agents have to perform three tasks:

1. *perception* of the conditions that dynamically change in the environment.
2. *reasoning* in order to understand the data being perceived and decide how to react.
3. *action* to change the conditions to the environment.

In analog to humans, agents have sensors to perceive changes in the environment and effectors to act.

3.2.1 The Mobile Agent System DIET Core

As we have already said in Section 1.2 a basic concept in DIET Core is the *world*. There is one world per Java Virtual Machine (JVM). A world is an *environment* repository and can contain one or more environments. Environments in DIET Core provide a location for agents to inhabit and can host one or more agents. The DIET platform can contain more than one worlds. This is for instance the case when DIET Core runs on multiple computers. This can be viewed as a DIET *universe* that contains multiple planets (worlds). Each environment can have neighbor links to other environments, which are not necessarily in the same world. Each environment can be connected with other environments which are not necessarily in the same world, through *neighbor links*. These links allow agents to migrate to different environments and explore the DIET universe, without having any prior knowledge of the location of the environments. Those links are specified when the world is set-up but can change dynamically, if the DIET universe changes.

In P2P-DIET there are many worlds. Each super-peer and each client has a different world, which means that each different computer represents a different world. All the worlds together are the *P2P-DIET universe*. Each world has one environment. A world in a super-peer node has a *super-peer environment*, where 5 different types of agents live. The world in the client peer nodes has a *client peer environment*, where 3 different types of agents live. Each type of agent is explained in the following sections.

Each agent is assigned a unique *name tag*. This tag is randomly generated within its originating environment. It is used to distinguish a single agent throughout its lifetime and it is retained even when the agent migrates to a different environments. Additionally, each agent is assigned a *family tag*. This tag reflects the services that the agent offers and its responsibilities. Duplication of family tags is expected since more than one agents may offer the same services.

3.3 The Super-Peer Environment

In a super-peer environment there are five types of agents: a super-peer agent, an Are-You-Alive messenger, a clock agent, a build spanning tree scheduler and zero or more messenger agents. These agents are described below.

3.3.1 Super-Peer Agent

The super-peer agent is the most important agent that inhabits the super-peer environment. Each super-peer environment has one super-peer agent. This agent has the greatest number of responsibilities of any other type of agent in the P2P-DIET universe. It must work continuously and autonomously to serve the clients of this super-peer. It accepts profiles and resources and tries to find similarities between them. It arranges rendezvous, it stores notifications and rendezvous files. It assigns unique keys to the new clients. Moreover, it is responsible for the correct flow of messages in the network. It is responsible for forwarding each message to the correct super-peer agent, which depends on the root of the message if the message is broadcasted or the receiver in the case that the message is sent to a specific super-peer agent. The super-peer agents need a way to communicate with remote agents, for example, the client agent in the computer of a user or the super-peer agents of the neighbor super-peers. The way to achieve remote communication is to assign the message to a messenger agent as we will see in Section 3.3.5. The agents that inhabit in the super-peer environment and the way they interact with each other and with the P2P-DIET universe is shown in Figure 3.4.

3.3.2 Are-You-Alive Messenger

The Are-You-Alive messenger is an agent that works continuously in all super-peer environments. Each super-peer environment has one Are-You-Alive messenger. This agent is responsible for periodically checking the clients agents, that are supposed to be alive and are served by this super-peer. The super-peer agents of the neighbor super-peers are checked too, to guarantee connectivity. Both client agents and neighbor super-peer agents are in remote environments. The remote communication is achieved by the messenger agent, which means that the Are-You-Alive messenger is able to communicate with the messenger agent. Moreover, the Are-You-Alive messenger has to know which client agents and which super-peer agents it must check. This information is taken from the super-peer agent, who communicates with the are Are-You-Alive messenger when a client is connected or disconnected or when a new super-peer is added to the network. This means that the Are-You-Alive messenger is able to handle messages from the super-peer agent. Moreover, it is able to send messages to the

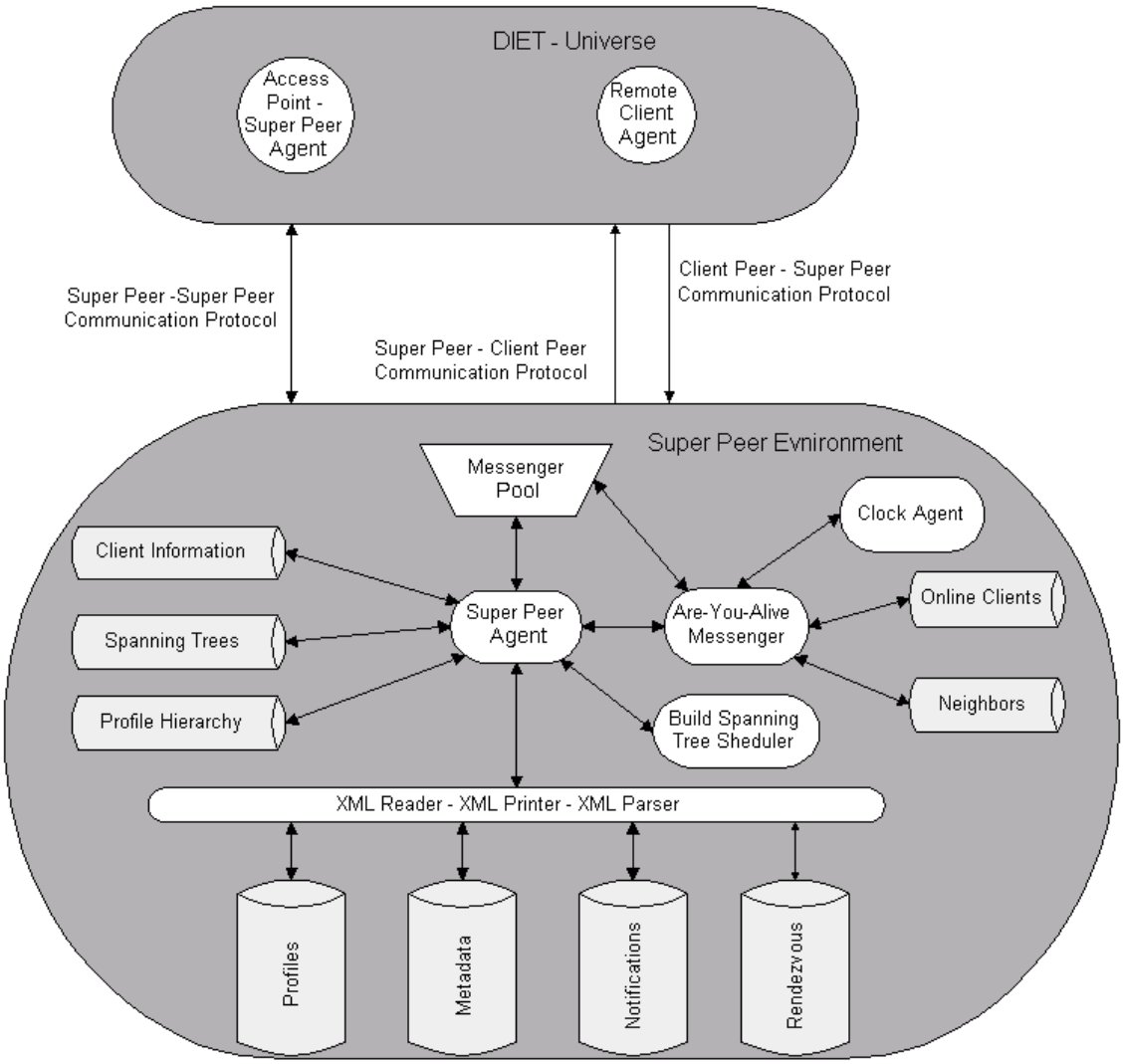

Figure 3.4: A Super Peer Environment

super-peer agent to inform him when a remote client agent or a super-peer agent has not replied to the are you alive messages.

3.3.3 Clock Agent

The clock agent is a scheduler for the Are-You-Alive messenger. It knows when it is the right time to send messages or to check for the replies. The clock agent sends four different types of messages to the Are-You-Alive messenger: `broadcast clients are you alive message`, `check alive clients`, `broadcast neighbors are you alive message` and `check alive neighbors`.

3.3.4 Build Spanning Tree Scheduler

The build spanning tree scheduler is a very lightweight agent. It is dormant until the super-peer agent sends him a `send me time to broadcast build spanning tree message`. Then it sleeps for the appropriate time, wakes up, sends the `time to broadcast build spanning tree message` to the super-peer agent and sleeps again until the super-peer agent sends a message. More details about spanning tree construction are given in Chapter 4.

3.3.5 Messenger

A messenger is the agent that implements remote communication between agents in different worlds. When an agent wants to send a message to an agent that inhabits a remote environment, instead of migrating, it assigns the job to a messenger. The messenger will migrate to the remote environment and deliver the message to the target agent. The agents that need remote communication are able to create messenger agents. The messenger does not need any information on the goal of the message, all it needs to know is the address of the remote environment, the family tag of the target agent and the message. Each environment has a messenger pool. When a messenger arrives at an environment, it delivers the message and stays in the pool, waiting for a local agent to assign him a job, a message to deliver. In this way, when an agent wants to send a remote message, it does not have to create a new messenger agent. It assigns the message to a messenger from the pool unless the pool is empty, in which case a new messenger will be created. Of course, there is an upper bound to the number of messengers in the pool, so if a messenger arrives to an environment, it delivers the message and the pool is full, it dies. This process is extra protection to prevent super-peer environments of running out of resources since each messenger needs a thread. From the time that a messenger agent is created, it migrates to different environments to deliver messages, so messengers travel all around the P2P-DIET universe until they arrive at an environment with a full messenger pool. This means that a messenger may live forever or may die after a few migrations.

3.4 Client Peer Environment

In a client peer environment there are three types of agents: a client agent, an interface agent and zero or more messenger agents. These agents are described below.

3.4.1 Client Agent

The client agent is the agent that connects the client peer environment with the rest of the P2P-DIET universe. It communicates, through messengers, with the

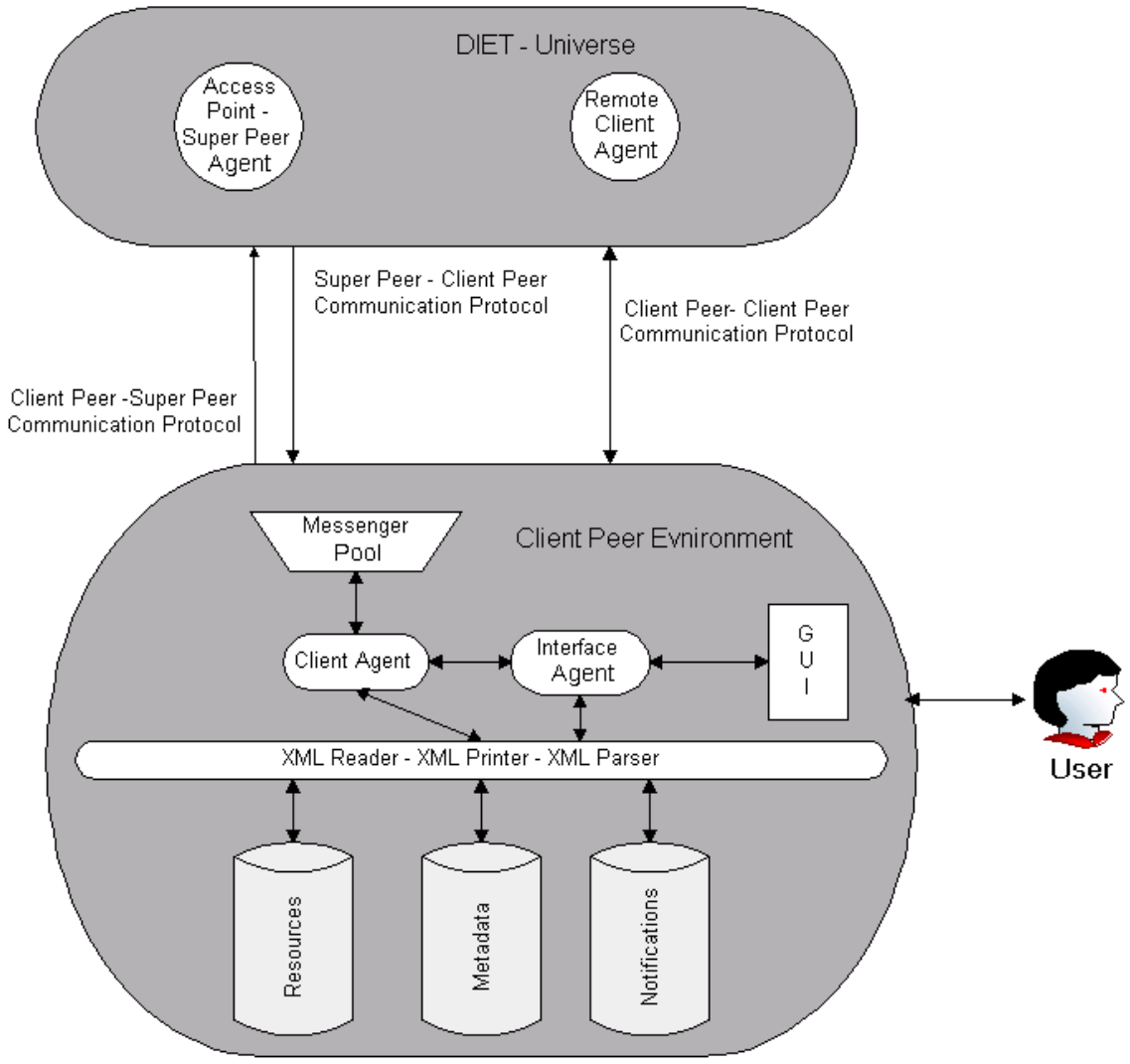

Figure 3.5: The Client Peer Environment

super-peer agent that is the access point or any other remote client agents for downloading or just for chatting. The client agent sends to the remote super-peer agent of the access point the profile of the client, the metadata of the resources, the queries, the requests for rendezvous, the finger requests etc. In Figure 3.5, we see all the agents in the client peer environment.

3.4.2 Interface Agent

The interface agent is responsible for forwarding the demands of the real user (person) to the client agent and of course messages from the client agent to the user. The interface agent is a listener to the GUI. For example, when the user chooses to publish a resource, or subscribe a profile the interface agent

must forward the message to the client agent, who is responsible to handle such events. Additionally, when there is some message from the network, for example, a notification, the interface agent must be informed by the client agent and fire the appropriate changes to the GUI, so that the user will understand that a new notification has arrived. In same cases, the interface agent does more than just forward commands. For example, in the case that a user chooses to download a file from a stored notification, the interface agent will listen to the GUI, it will understand which the desired resource is and it will parse the appropriate XML file to find out the file name of the desired resource and the key of the resource owner client. Then it will send the appropriate message to the client agent and the client agent will handle the rest of the downloading process.

3.4.3 Messenger

The messenger is exactly the same as the agent described in the subsection of the super-peer environment. Again there is a messenger pool in the client peer environment but the difference is that the size of the pool is limited, because of the limited requirement for messengers compared with the super-peer environment.

3.5 The Language for Metadata, Queries and Profiles

In this section we will describe the language that P2P-DIET currently offers to support a file sharing application. Note that the language is totally modular to the system. This means that a new language can be easily supported possibly to implement a different application scenario or to extend the current language.

3.5.1 Resource Metadata

Users of P2P-DIET can share their files (resources) with other users. A user may publish a resource in order to make it available. As we have already explained, the resource is not uploaded to a super-peer node. Instead, the resources are kept in the client nodes and users only send metadata for each resource. The metadata will be used for the matching between the resource and profiles of other clients. In this way, the metadata must contain enough information to describe the resource and we expect the user to fill in a form with the metadata. Those metadata are the same for all users and for all resources. Part of the metadata is initialized by the system, so if the user chooses not to fill the form there will still be some useful information on the resource. The metadata language currently used by P2P-DIET is a conjunction of `Atribute=Value` expressions where `Attribute` can be any of the following:

- *Name*. The name field gives information on the name of the resource. This field is initialized by the system and the user cannot change its value. Actually the value is the file name of the resource (without the type extension). For example, for the file `thesis.pdf` the name field would be initialized with the value `thesis`.

- *Type*. The type field gives information on the type of the resource. This field is initialized by the system and the user cannot change its value. The type value is the type extension of the full file name of the resource. For example, for the file `thesis.pdf` the type field would be initialized with the value `pdf`.

- *Size*. The size field gives information on the size of the resource. This field is initialized by the system and the user cannot change its value. The size value is the file size of the resource in bytes.

- *Path*. The size field gives information on the path location of the resource. This field is initialized by the system and the user cannot change its value. The path value is the full path location of the resource in the computer of the user and it will be used in the download process to locate the desired file.

- *Key*. The key field gives information on the key of the specific client. This key is initialized by the system and the user cannot change its value. The key value is taken from the XML log file of the client and it is the key that has been sent to the client by the super-peer agent of the access point, when the client connects for the very first time to the network. The key value will be used by super-peer agents and client agents to determine who is the owner of this resource. For example, when a user wants to download a resource, the first thing that he has to know is to identify the owner and locate his IP address. Details of how keys are formed and what role they play in various protocols are given in Chapter 4.

- *Title*. The title value gives information on the title of the resource. This field is initialized by the user. The title field may be null if the user chooses not to fill this field. The goal of this metadata field, is to give a better description of the resource than the name field. For example, for a file `thesis.pdf` the title field can be the title on the title page `P2P-DIET: A Query and Notification Service Based on Mobile Agents for Rapid Implementation of P2P Applications.`

- *Author*. The author field gives information on the author of the resource. This field is initialized by the client. The author field may be null if the user chooses not to fill this field. The goal of this field is obvious for document files but it can be used to describe the artist of an MP3 or an MPEG

```
<?xml version="1.0" encoding="UTF-8" ?>
<thesis.pdf> -
        <Auto_Keywords>
                <Client_Key Key="213" />
                <File_Type Type="pdf" />
                <File_Name Name="thesis" />
                <File_Path Path="C:\\Documents and Settings\\Stratos\\Desktop\\thesis.pdf">
                <File_Size Size="676318" />
        </Auto_Keywords>
        <Specified_Keywords>
                <Title>
                        P2P-DIET: A Query Dissemination Service using Mobile Agents
                </Title>
                <Author>
                        Stratos Idreos
                </Author>
        </Specified_Keywords>
</thesis.pdf>
```

Figure 3.6: A resource metadata example

resource file as well. For example, for a file `thesis.pdf` the author field can be `Stratos Idreos`. Note that there is no limitation to the number of the words and the title field can contain two or more authors.

An example of a possible resource metadata is shown in Table 3.6. The example is shown as an XML file.

The metadata fields that described are only there in the experimental version of the file sharing application of P2P-DIET so that we can experiment with textual and numeric values in matching queries with metadata as in SIENA [2]. In the next version of P2P-DIET to be targeted at a Digital Library application our metadata will be based on appropriate standard protocols such as Dublin Core [67, 68].

The interface agent in the client peer environment creates an XML DOM tree when the user fills the form with the metadata. The DOM tree is propagated to the client agent and from there to the super-peer agent of the access point that serves the client. Moreover, the interface agent uses the DOM tree to create an XML file and saves it to the XML_Metadata directory in the client environment. The XML file will be used only when the client migrates to a different access point, in which case the super-peer agent will ask for the metadata of all the resources of the client. The DOM tree exists only in main the memory and it is logical to assume, that the user will turn off his computer at some point or he will terminate the client application, so the DOM tree will be lost.

3.5.2 Queries and Profiles

Users of P2P-DIET can use the query or notification capabilities of the system by posing a query or subscribing with a profile. Queries and profiles are written using the same syntax because profiles are long-standing queries. This syntax is essentially a subset of the query language of DIAS [44].

A query (or profile) is conjunction of atomic formulas `Attribute Predicate Expression`. The attribute used determines the predicate and expression allowed.

The following attributes are supported in the current file sharing application for writing a query or profile:

- *Type*. Attribute Type can participate in atomic formulas of the form:

  Type = V1 OR V2 OR ... OR Vn

  where V1,V2,...,Vn are words.

  An example of such an expression is `Type = pdf OR doc`, which means that the user is interested in files of type `pdf` or `doc`. The type field gives information on the type of files for which the user wants to receive notifications. This is the type extension of the file name of a resource. The type field of the profile will be matched against the type field of the resource metadata.

- *Name*. Attribute Name can participate in atomic formulas of the form:

  Name $\sqsupseteq$ V1 OP V2 OP ... OP Vn

  where V1,V2,...,Vn are words

  and OP is either the OR or the AND operator.

  An example of such an expression is `Name` $\sqsupseteq$ `peer AND scale AND event` will produce notifications on resources that have on the name field all the words of the name field of the profile. Another example is `Name` $\sqsupseteq$ `peer OR event` will produce notifications on resources that have a name field either with the value `peer` or with the value `event`, for example, `peer-to-peer networks` or `event based systems`. The name profile gives information on the name of the resource that the user is interested on. The name field will be matched against the name field of the resource metadata.

- *Title*. Attribute Title can participate in atomic formulas of the form:

  Title $\sqsupseteq$ V1 OP V2 OP ... OP Vn

  where V1,V2,...,Vn are words

  and OP is either the OR or the AND operator.

  An example of such an expression is `Title` $\sqsupseteq$ `load AND network` will produce notifications on resources that have on the title field all the words of the title field of the profile. Another example is `Title` $\sqsupseteq$ `load OR event`

will produce notifications on resources that have a title field either with the value `load` or with the value `event`, for example, `load balance in peer-to-peer networks` or `event based systems`. The title field gives information on the title of the resource that the user is interested on. The title field will be matched against the title field of the resource metadata.

- *Author*. Attribute Author can participate in atomic formulas of the form:

  Author $\sqsupseteq$ V1 OP V2 OP ... OP Vn

  where V1,V2,...,Vn are words

  and OP is either the OR or the AND operator.

  An example of such an expression is `Author` $\sqsupseteq$ `stratos AND manolis` will produce notifications on resources that have on the author field all the words of the title field of the profile, for example, `Stratos Idreos, Manolis Koubarakis`. Another example is `Author` $\sqsupseteq$ `stratos OR manolis` will produce notifications on resources that have a author field either with the value `stratos` or with the value `manolis`, for example, `Stratos Idreos, Manolis Koubarakis` or `Manolis Koubarakis`. The author field gives information on the author of the resource that the user is interested in. The author field will be matched against the author field of the resource metadata.

- *Desired Client*. Attribute Desired Client can participate in atomic formulas of the form:

  Desired Client = V1 ; V2 ; ... ; Vn ;

  where V1,V2,...,Vn are keys of other clients.

  An example of such an expression is `Desired client = 234; 123; 546;` The Desired client field gives information on the clients that the user is interested in. The goal of this metadata field is that a user A may have noticed that resources from a specific user B are always interesting, so he wants to receive notification on every resource that user B publishes. The desired client field, will be matched against the key field of the resource metadata file.

- *Size*. Attribute Size can participate in atomic formulas of any of the following forms:

  1. Size symbol number

  where symbol can be one of the following =, < or >

  2. number symbol Size symbol number

  where symbol can be one of the following < or >

```
<?xml version="1.0" encoding="UTF-8" ?>
<Profile>
        <Auto_Keywords>
                <Key Key="213" />
        </Auto_Keywords>
         <Specified_Keywords>
                <Type>
                        doc
                </Type>
                <Name>
                        master thesis
                </Name>
                <Title>
                        peer + load + balancing
                </Title>
                <Author>
                        Idreos + Koubarakis
                </Author>
                <Desired_Client>
                        213
                </Desired_Client>
                <Desired_Client>
                        345
                </Desired_Client>
                <Size>
                        4 < size < 10
                </Size>
        </Specified_Keywords>
 </Profile>
```

Figure 3.7: A profile encoded in XML

An example of such an expression is `Size = 200 < size < 800`. The size field gives information on the size of the resources that the user is interested in. The goal of this metadata field is to find resources that their size satisfies the size field in respond to the fact that internet users have different types of internet access in terms of speed. The size field will be matched against the size field of the resource metadata.

- *Key.* The key field gives information on the key of the client. It is initialized by the system. The key value is taken from the XML log file of the client and it is the key, that has been sent to the client by the super-peer agent of the access point, when the client connected for the very first time to the network. This field will be used by super-peer agents to identify the client, who owns the profile. In this way, the key field is not a part of the query itself.

An example of a profile metadata file is shown in Table 3.7. The profile metadata values are transformed into an XML DOM tree by the interface agent. If the user chooses to send an empty query or profile, he will receive notifications on all existing and future resources. The DOM tree is forwarded to the client

agent and from there to the super-peer agent of the access point. The super-peer agent is responsible for placing the profile on the correct level of the profile hierarchy and if it is necessary to broadcast the profile to all super-peer agents of the network. The interface agent will save the DOM tree as an XML file to the Profile directory of the client in the client peer environment. Thus, the profile will be available for later use, even if the client application restarts.

A user may use the query capabilities of the system as well, by posing a query. A query supports the same language as the normal profile. This time, the super-peer agent of the access point will try to satisfy the query, and then it will broadcast it to the rest of the network, without saving it and without affecting the normal profile.

In the case that a match is found between a resource and a profile, the network must somehow inform the profile owner. The way to do this is to send a *notification* to the interested client. Additionally, in the case that a match is found between a resource and a query, the network must inform the user who sent the query. The way to do this is to send an *answer* to the interested client. Notifications and answers must contain all appropriate information on the resource in order for the client to be able to download it. This means that the notifications and answers hold information on the:

1. *Key* of the resource owner. The key will be used from the super-peer agent, which is the access point of the interested client, if the notification must be stored because the client is not online. The notification will be stored to the notifications directory of the client under a directory named from the key of the resource owner. In this way, notifications with same name from different resource owner will not be lost. Moreover, the key is necessary in the notification metadata, because a client will have to locate the resource when he wants to download it.

2. *Path* of the resource file. This field is necessary for the downloading process to locate the file in the client peer environment.

3. *Name* of the resource.

4. *Size* of the resource.

5. *Title* of the resource. This information is given by the user so it is possible that there will be no title information.

6. *Author* of the resource. This information is given by the user so it is possible that there will be no author information.

3.6 Summary

In this chapter we presented the language for queries and profiles in P2P-DIET, its architecture and the basic functionalities that the system offers. The details will be described in the following chapters. Moreover, we presented the way that the P2P-DIET universe is organized, the environments, the different agents and their goals. In the next chapter we present the routing strategy in P2P-DIET to achieve efficient message delivery.

Chapter 4

Routing Messages in P2P-DIET

In the previous chapter we presented the architecture of P2P-DIET. We described the different agents that were implemented and their responsibilities. Since our system is a super-peer network and the topology of the servers forms a general undirected graph, we have to deal with the routing problem, because as we discussed in Chapter 2, routing has no trivial solution in a general graph. We present the network functionality. Our chosen solution is to use shortest paths for unicasting and minimum spanning trees for multicasting and broadcasting in our network. Moreover, we present the way that our system handles the problem of dynamic IP addresses and the problem that a client might get disconnected while the system is in operation. Then, we continue by discussing the fault-tolerance problem and its solutions. We close this chapter with a section about socket handling, where we describe why we try to keep the socket connections open in many cases and how we do that.

4.1 Network Functionality

Let us now describe the functionality expected from our network. Our goal is to support the general character of the system and give the maximum flexibility to each node, so that it can communicate with the rest of the nodes in various ways. Thus, each super-peer must have the ability to perform the following operations:

- Unicast, i.e., send a message only to one super-peer (neighbor or not).
- Multicast, i.e., send a message to a subset of super-peers (neighbors or not).
- Broadcast, i.e., send messages which will be received by all other super-peers.

4.2 Techniques

The goal of a routing protocol is to establish appropriate routing paths and to use network resources as efficiently as possible loading the network with the less possible overhead. Routing in a network typically involves a rather complex collection of algorithms, that work more or less independently and yet support each other by exchanging information. The complexity of the problem is due to a number of reasons:

1. Routing requires coordination between all nodes of the network.
2. Routing must be able to cope with link or node failures.
3. To achieve high performance, routing algorithms may need to modify the adopted routes, when parts of the network become congested.

It is clear that routing depends on the underlying topology of the super-peers. In the case of the acyclic topology discussed in Section 2.7.1 the solution is trivial, because there are no paths to choose from. Each node is connected to all others through unique paths. On the other hand, the general peer-to-peer interconnection topology of the super-peers requires additional data structures, and rather complicated protocols to achieve routing. In the next sections we will present the techniques that we use for routing in the general graph topology that the super-peers form in the P2P-DIET network. The ideas we present have been known for a while in the area of routing for data networks [15]. In our case we apply these ideas to routing in the *overlay* network formed by the P2P-DIET super-peers.

4.3 Unicasting

The most basic need of the network is the ability for communication between two super-peers. If the super-peers are neighbors then the solution is simple, they just communicate through the arc that connects them. If the super-peers are not neighbors, then they have to communicate through other super-peers. To do that, we use the Bellmann-Ford *shortest path algorithm* [60, 43]. According to this algorithm, a shortest path from all the nodes of the network to a given destination node is made by finding first the one-arc shortest path from all nodes to the destination node, then by finding the two-arc shortest path and so on. For each node, the path with the smaller cost is considered to be the shortest path to the destination node. For a network with N nodes there will be *N-1* checks for each of the other nodes. In the worst case, the amount of computation is N^3 so the complexity of the algorithm is $O(N^3)$.

From here on, we consider the network formed by super-peers as a weighted graph. Each arc has a weight and this weight represents the communication cost

Node Key	Address	Weight	Receive Node
Key	Address	Weight >0	Address

Table 4.1: Routing Table for shortest paths

to send a message through this arc. In our case, the communication cost is a unity for each arc. In this way, the cost to send a message through a path is the number of hops.

4.3.1 Implementing Bellmann-Ford in P2P-DIET

Let us now describe how the Bellmann-Ford algorithm is implemented in the P2P-DIET network. The super-peer that wants all the other nodes to build shortest paths with it as a destination, sends a `build shortest path` message to all its neighbors using flooding along with the information, that it is the root of the message and that the weight is unity. From there on, the rest of the network should take the appropriate actions to correctly build shortest paths from all nodes to the destination node and this happens in the following way. The super-peer D who will be the destination node for the shortest paths sends a `build shortest path` message. When a super-peer SP receives such a message from a node S:

1. Adds 1 to the weight *W1*.
2. Forwards the message to all its neighbors except the sender S.
3. Stores the sender S as *receive node* in the shortest path to the destination node D.
4. When another `build shortest path` messages arrives from another sender X but for the same destination D and the weight *W2* of the new message is smaller than the old weight *W1*, then the super-peer SP:
   - Stores the new sender X as node of type *receive node* in the shortest path to the destination node D.
   - Updates the total weight (of the shortest path to D) to the new one *W2*.
   - Adds 1 to the weight *W2*.
   - Forwards the message to all its neighbors except the sender X.

Each super-peer has a routing table to hold information on the shortest paths from it to the rest of the nodes of the network. For each node D a super-peer S

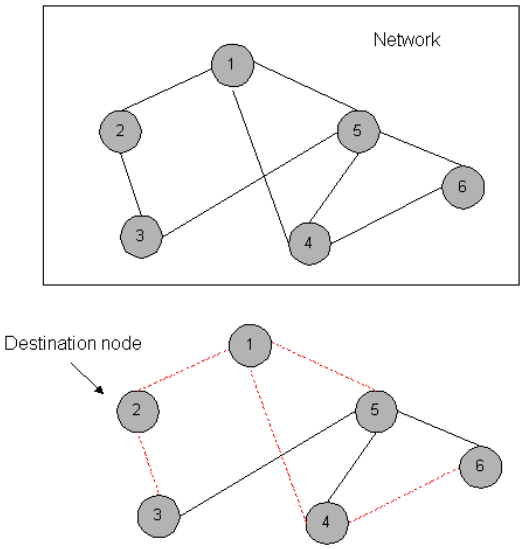

Figure 4.1: An example of a shortest path

has to know the receive node in the shortest path from S to D as shown in table 4.1. Moreover, the `build shortest path` messages are piggy-backed on `build spanning tree` messages when building minimum weight spanning trees as will be explained in Section 4.4.6.

4.3.2 Unicasting with Shortest Paths in P2P-DIET

Let us now describe how we implement this algorithm in the peer-to-peer environment of P2P-DIET. In the following protocol *sender* and *receiver* have the obvious meaning. The protocol for unicasting is the following:

- The sender S sends the message to the super-peer, who is its neighbor and is of type receive node in the shortest path to the receiver R along with the information, of who is the receiver R and that S is the sender of the message.
- Each super-peer forwards the message only to the super-peer, which is its neighbor and is node of type receive node in the shortest path to the receiver R.
- If the receiver address is equal to the super-peer address, the message has reached its destination.

It is obvious that if the super-peers are neighbors, then the sender, will have the receiver as a receive node (in the shortest path from the sender to the receiver) and the message will be delivered in only one step. Let us give a small example of the above procedure. Consider the case that node 6 in Figure 4.1 wants to

send a message to node 2. It is going to use the shortest path to node 2, so at first it sends the message to node 4. Node 4 understands that this is a message for node 2 and forwards it through the shortest path from node 4 to node 2, so the next node that receives the message is node 1. Node 1 understands that this is a message for node 2 and forwards it through the shortest path from node 1 to node 2, so nodes 2 receives the message. The message is delivered through the shortest path from the sender to the receiver with minimum number of hops.

4.4 Broadcasting

When broadcasting a message must be delivered to all the nodes of the network except the sender. We consider four basic solutions: *flooding*, *reverse path forwarding*, *spanning trees* and *minimum weight spanning trees* and justify our choice which is minimum weight spanning trees. Then, we present the way that we build and use minimum weight spanning trees in the peer-to-peer context of P2P-DIET.

4.4.1 Flooding

One common solution to the broadcasting problem in a general undirected graph is *flooding* [15]. When a node needs to send a message, it sends it to all its neighbors along with a unique key for the message. When flooding, each node transmits the message that received to all its neighbors except the sender, and stores the unique key of the message. If a node receives a message with the same key from another sender, it ignores it. Flooding is a simple protocol that works well. It is guaranteed that by using flooding all messages are received and one additional advantage is that there is no need for the nodes (super-peers) to store additional information on the network topology, in order to send messages. The obvious drawback is that we are not using network resources efficiently, because we actually transmit the message too many times and in that way we overload the network.

4.4.2 Reverse Path Forwarding

A well-known and widely used solution is *reverse path forwarding* [73]. It is a very simple technique with minimum storage requirements to the nodes of the network. According to reverse path forwarding, a node that receives a message will accept it only if the sender is part of the shortest path that connects this node with the node that generated and broadcasted the original message (the root of the message). Then, it will forward the message to all its neighbors except the sender. Thus, this method loads with less overhead the network than flooding does. In this way, the only information that a node X needs in order to forward

a message that was broadcasted by a node S is the shortest path from X to S. Thus each node needs a routing table as the one in Table 4.1. For example, SIENA uses reverse path forwarding to broadcast messages.

4.4.3 Spanning Trees

Another approach is to build a *spanning tree* [15] for *each of the nodes* of the network. The exact definition of a spanning tree is: A tree is a connected graph that contains no cycles. A spanning tree of a graph G is a subgraph of G that is a tree and that includes all nodes of G. This definition and more information on spanning trees can be found in [15]. A spanning tree is actually an acyclic picture of the general graph. It contains unique paths from the root node to all others. Thus, when a node wants to broadcast a message, it can use the edges of its spanning tree. It is understood that we must build a different spanning tree for each of the nodes of the network and that all super-peers must have knowledge of the spanning trees of the other nodes.

4.4.4 Minimum Weight Spanning Trees

A better approach is not to ignore the arc (links) weights and build *minimum weight spanning trees* [15]. A minimum weight spanning tree is a spanning tree with minimum sum of arc weights. The weights represent the communication cost to transmit a message from one node to another. In this way, the sum of the weights of all the arcs of the spanning tree is the actual cost of the broadcasting procedure when the root of the tree broadcasts a message to all the nodes of the network. We explain in detail how we build and use spanning trees in the next section. An example network and the spanning tree for 3 of its nodes is shown in Figure 4.2.

4.4.5 Minimum Weight Spanning Trees in P2P-DIET

The use of minimum weight spanning trees is a good solution given the static character of the pure peer-to-peer network consisted by the super-peers. Thus, we use weight minimum spanning trees to broadcast messages between the super-peers while the clients communicate only with their access point. In an unstable pure peer-to-peer network the spanning tree solution for broadcasting is not satisfactory, because each time a peer fails or is disconnected from the network all spanning trees must change. For the arc weights we adopt the most simple approach where all the weights are unity. In this way, the communication cost is the number of hops.

Some well-known systems such as SIENA [3] and Hermes [59] build shortest paths with Bellmann-Ford and use reverse path forwarding to broadcast messages as described in Section 4.4.2. We could adopt this policy and do not build

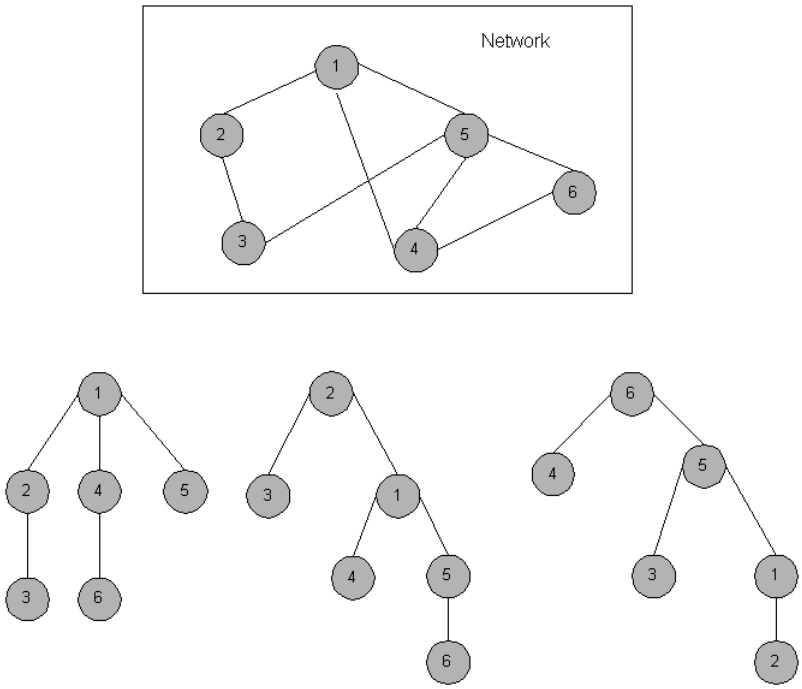

Figure 4.2: Examples of spanning trees

spanning trees. The tradeoffs are that in order to build minimum spanning trees the nodes of the network transmit more messages than the messages that are necessary for building the shortest paths. Moreover, the nodes must store more information (for the spanning tree). On the other hand, once the minimum weight spanning trees are ready the broadcasting procedure takes place with an optimal way by delivering the message only once to each node, while with reverse path forwarding this is not true. Given the static character of the pure peer-to-peer network that the super-peers form, we choose the solution of minimum weight spanning trees for broadcasting in P2P-DIET. In this way, we spent more messages to build minimum weight spanning trees, when a new super-peer is added or when a super-peer fails, and save messages when the network is properly working.

In the next subsection we describe the distributed algorithm, that we use to build minimum spanning trees. We are going to do this in steps, because various problems come up when we present the details of the solution. We are going to show in detail how we use spanning trees to broadcast messages, what information each node needs to efficiently use spanning trees, and when we need to build them.

4.4.6 Initial Algorithm

In order to build the minimum spanning tree for each super-peer we implement the algorithm proposed in [61]. First, we use flooding, because prior to building

the spanning tree, a super-peer has no knowledge for the network but its neighbors. The super-peer R that wants to build its spanning tree, sends a `build spanning tree` message to all its neighbors using flooding along with the information, that it is the root of the message and that the weight *W1* is unity. From there on, the rest of the network should take the appropriate actions to correctly build the spanning tree and this happens in the following way. When a super-peer X receives a `build spanning tree` message:

1. Adds 1 to the weight *W1*.
2. Forwards the message to all its neighbors except the sender S.
3. Sends confirmation to the sender S, that the super-peer X is a node of type *send node* in the spanning tree of the root R of the message.
4. When another `build spanning tree` messages arrives from another sender *NS* but for the same root R and the weight *W2* of the new message is smaller than the old weight *W1*, the super peer X will:
   - Send confirmation to the sender *NS*, that the super-peer X is a node of type *send node* in the spanning tree of the root R of the message.
   - Update the weight, from node R to node X, to the new weight *W2*.
   - Cancel the previous confirmation by sending a cancel message to the previous sender S.
   - Adds 1 to the weight *W2*.
   - Forwards the message to all its neighbors except the sender *NS*.
5. Waits for confirmation from one of the others and notes the sender or senders as nodes of type *send node* in the spanning tree of the root R.

A super-peer A knows one type of nodes for the spanning tree of another super-peer B, which is one or more send nodes. Those nodes are neighbors of the super-peer A. The send nodes will be used by super-peer A to forward any messages that were broadcasted by super-peer B. For example, in Figure 4.2 in the spanning tree of node 2, node 1 knows that nodes 4 and 5 are the send nodes. One could say that one spanning tree would be enough to satisfy the demands of our network, since all the nodes are connected with unique paths. This is not true. If we used only one spanning tree, then all the links[1], that are not included in the spanning tree would be wasted, while other links in the spanning tree would be overloaded, because of the non stop use. On the other hand, the use of a different spanning tree for each node, will force all links of the network to be used in the message delivery process and the load of the network traffic will

[1] A link is a direct connection between two super-peers.

Key	Address	IsNeighbor	Weight	Send nodes
Key	Address	True or False	Weight >=0	List of addresses

Table 4.2: Spanning Tree Table

not be incurred by only a subset of the links. Our decision to build one spanning tree for each node automatically means that each super-peer needs information on the spanning tree of all other nodes. But as we discuss in Section 4.4.7, each super-peer has to know only certain bits of information about the spanning trees of the others.

4.4.7 Distribute Spanning Tree Information

It is clear from what we discussed in the previous sections, that a super-peer needs information concerning all the spanning trees in the network, in order to send messages to each super-peer and to forward messages that started from other super-peers. By looking carefully at the algorithm and the process of sending and broadcasting a message, we understand that the information on a spanning tree is distributed and that no super-peer has full knowledge of a spanning tree of another node. Actually, no super-peer has full knowledge of its own spanning tree. All it knows is the send nodes and that there is no receive node. In this way, we achieved to build a different spanning tree for each node and moreover the information on the spanning tree of each of the nodes, is distributed in the network saving storage resources in the nodes of the network. This means, that the only extra information that a super-peer must store for a spanning tree is the send nodes. The spanning tree data structure is shown in Table 4.2. Each super-peer knows the key of other super-peers, their address, whether a super-peer is its neighbor or not and of course the send nodes for the spanning tree. In the spanning tree data structure there is one entry for each super-peer.

4.4.8 Overloading the Network

We have described the algorithm to build a spanning tree in Section 4.4.6. In this subsection we discuss why we had to slightly change this algorithm in our implementation and how we did that. When implementing the algorithm of Section 4.4.6, the obvious conclusion was that it is very heavy work for the network to build all the spanning trees at the same time, and actually what happened is that sometimes computers were running out of resources, for example, they were not able to handle any more sockets. The problem becomes bigger as the number of super-peers in the network grows. It is very critical that we do not lose any `build spanning tree` or `reply spanning tree` messages, because this would

mean that some of the spanning trees will not be correct, and that there will be problems in the communication of super-peers. To handle such situations and to guarantee the correct build of spanning trees, we cut the build spanning tree progress into pieces. The basic idea, is that spanning trees will be made one by one. When the first spanning tree is ready, then the network will start building the second one and so on. The trick is to make each super-peer understand, that this is a situation of network topology change, so it has to broadcast a `build spanning tree` message as a root, but not immediately. More precisely, it has to delay the message for a specific amount of time and that amount should be different for all super-peers otherwise they will delay for the same time and then they will start building the spanning trees all together, which of course is not what we want. Moreover, the delay time must be enough for the previous spanning tree to be ready.

When the administrator of the network starts up a new super-peer, he assigns a unique key for it starting with key `1` for the first node of the network, `2` for the second and so on. That key will be used to specify the delay time, the time to wait before broadcasting the build spanning tree message. The key multiplied by the *delay interval for spanning tree*, gives the delay time. The delay interval for spanning tree, is the same for all super-peers and it depends on the geographical topology of the super-peers. For the topology of the computers used in our lab, we found that a delay interval for spanning tree of 5 seconds is more than enough. Of course, when the super-peers are nodes in different cities, that interval must be a lot bigger. Thus, the very first super-peer, the super-peer that was added first in the network with key `1`, will wait for 5 seconds before broadcasting his build spanning tree message and in the meantime the spanning tree of the super-peer that triggered the change in the topology will have finished. The second super-peer, with key `2` will wait for 10 seconds, when the spanning tree of the first will be ready and so on. The exact algorithm is shown a flow-chart in Figure 4.3. The algorithm is fired when a new neighbor comes in the network, when a super-peer fails or when the message was received from another super-peer (forward build spanning tree message of other root). When the administrator wants to add a new super-peer, he assigns a unique key for the super-peer. The next thing that he has to do, is to assign the super-peer neighbors, which are the remote super-peers with direct link to the new one. Then the super-peer takes action and broadcasts a `new neighbor` message to its neighbors to inform them that it is in the neighborhood and that he wants to build a spanning tree.

Another detail in terms of overloading the network is that the messages for building minimum weight spanning trees and shortest paths are piggy-backed on normal P2P-DIET requests, which are forwarded using flooding when the network is in an unstable condition because of the arrival or fall of a super-peer.

The next logical step is to determine how the spanning trees are used to satisfy the functionality of our network, as we presented them in Section 4.1. In the next two subsections we are going to explain how a super-peer uses minimum

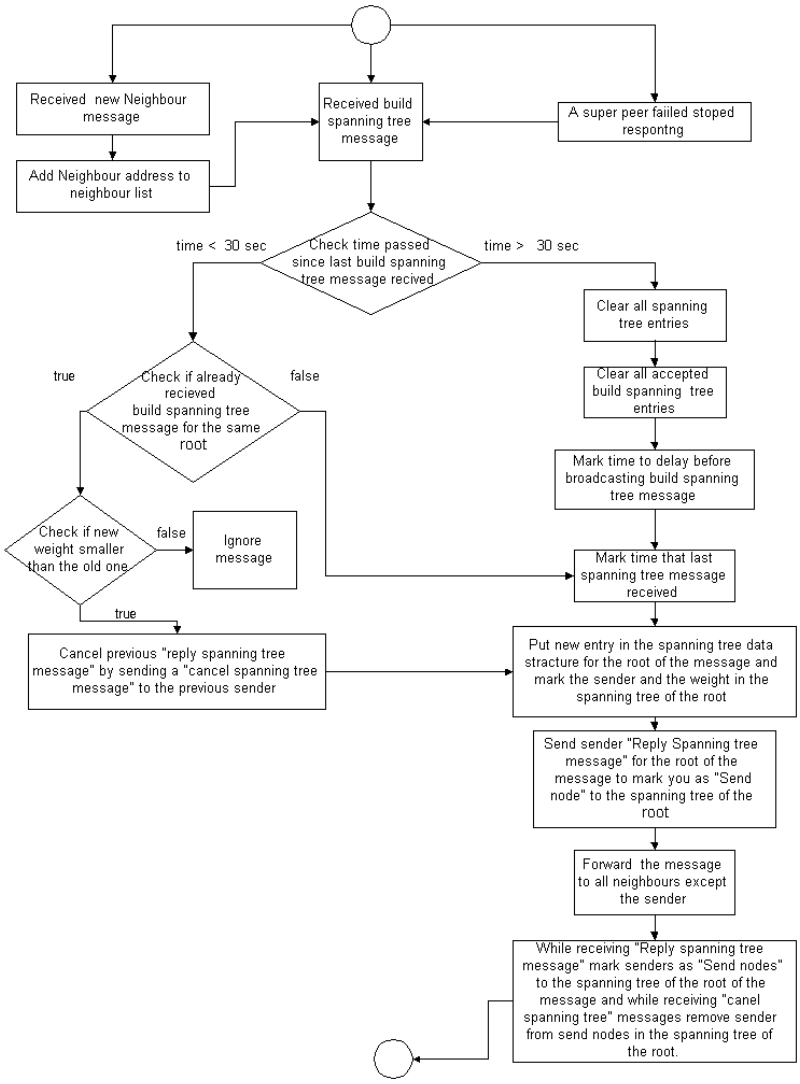

Figure 4.3: A distributed for building spanning trees algorithm.

spanning trees to multicast and broadcast messages. We will also give short examples based on Figure 4.2.

4.4.9 Broadcasting with Minimum Spanning Trees in P2P-DIET

In many cases a super-peer will need to send a message, that it will be received by all the super-peers of the network, for example, to broadcast a profile of a client. To do so, the super-peer will use its minimum spanning tree and somehow all the super-peers, that receive this message, will understand that they must forward the message through the spanning tree of the original sender, the root of the message. The steps for broadcasting are:

- The root sends the message to the nodes, which are its neighbors and are nodes of type send node to its spanning tree.
- Each super-peer forwards the message only to those super-peers, which are its neighbors and are nodes of type send node to the spanning tree of the root.
- When there are no send nodes (deepest level of spanning tree), do nothing.

Consider the case that node 2 in Figure 4.2 wants to broadcast a message. At first node 2 sends the message to nodes 3 and 1. Node 3 has no children so it takes no action. Node 1 sends the message to nodes 4 and 5. Node 4 has no children, so it takes no action while node 5 sends the message to node 6 and node 6 takes no action. The message has been delivered to all nodes of the network only one time. Actually, the message is forwarded downstream the spanning tree of the root with minimum number of hops.

4.5 Multicasting

In many cases, a super-peer will need to send a message, that it will be received by a subset of super-peers. In that case, we adopt a simple approach by using the minimum weight spanning tree of the super-peer that wants to multicast a message. The message is sent along with the information of the addresses of the super-peers that belong to the multicast group. When a super-peer receives a multicast message it forwards the message to the send nodes in the minimum weight spanning tree of the root and if its address is concluded in the multicast group it accepts the message. Note that the current P2P-DIET file sharing application does not use the multicast capabilities of the basic layer but the feature is there to support future applications.

4.6 Updating Spanning Trees and Shortest Paths when Topology of the Super-Peer Network Changes

Spanning trees and shortest paths are sensitive to the topology of the network. When the topology changes, some of the links in a spanning tree and in a shortest path will be inaccessible and possibly there will be other links to use. The existence of inaccessible links, means that there will be inaccessible super-peers since all the paths in a spanning tree and in a shortest path are unique. There are three situations that change the topology of the network:

- A new super-peer is added in the network.
- A super-peer fails.
- A super-peer was frozen for a long time and recovers.

We will describe the ideas by using the spanning tree case but the some observations stand for shortest paths too.

Every time a new super-peer is added to the network, it must broadcast a `build spanning tree` message to build its own spanning tree. Of course, all other super-peers must include the new one to their spanning trees. If they do not, they will not be able to broadcast messages, which will be received by the new super-peer. Every time a super-peer fails or freezes for a long time, the network must recover and build spanning trees again, because all the links involving the super-peer that failed will be inaccessible. The first node that realizes the situation, broadcasts a `build spanning tree` message. All other super-peers must understand that they cannot ignore that message. When a super-peer receives a `build spanning tree` message, it must decide if it is a message to ignore, because it has already received such a message from the same root or if the network builds spanning trees again. This is mostly a timing problem.

The way to solve this problem, is to put time stamps when the network is stable, which is when all spanning trees are ready. When a super-peer receives a new `build spanning tree` message, if the network is stable, it must accept it no matter what, erase all spanning trees and all accepted messages for building a spanning tree and finally broadcast a `build spanning tree` message for its own spanning tree. In the case that the network is not stable, the super-peer must check if it has received a `build spanning tree` message from the same root. The real way to put those time stamps, is to know when the process of building spanning trees will have finished. For the experiments in our lab, where maximum 12 super-peers were used, that time was max 30 seconds. The additional check in the algorithm is:

Every time a super-peer receives a `build spanning tree` message, he checks if 30 seconds have passed since the last `build spanning tree` message that received. If those seconds have passed, the super-peer must broadcast a `build spanning tree` message and erase all previous spanning tree information. In both cases, the super-peer marks the time, that a `build spanning tree` message was received to use it for the next check.

Using the above simple checks, a super-peer always knows if the network is stable or not. Again the time of 30 seconds depends on the geographical topology of the super-peers and in fact it is the same as the delay interval (time to wait before broadcasting a build spanning tree message), because it is the time for one spanning tree to be ready. Those time intervals can be easily defined in the source code of P2P-DIET by the administrator.

4.7 Clients

Super-peers are the basic nodes of our network, but clients are the nodes who actually use the network. A client node is not equal to a super-peer node. Clients and super-peers communicate in a hierarchical (client-server model) way. Each client is connected to only one super-peer, which is the *access point* of the client to the network. Clients are equal to each other and can directly communicate for downloading or just for chatting. A client uses the network to:

- Pose a query to the network.
- Subscribe with a profile to its access point.
- Publish a resource to its access point.
- Receive a notification from its access point.
- Receive a stored notification from its access point or from its previous access point.
- Download a file from another client.
- Download a rendezvous file from his access point.

All these functionalities will be analyzed in detail in the next chapter. We are going to give some details from the network point of view, so the reader can understand the next subsections. Profiles are forwarded using a sophisticated profile hierarchy mechanism. Each super-peer stores the profiles of his clients and the profiles that have been forwarded by other super-peers. Resource metadata are not forwarded, so each super-peer stores the resource metadata of his clients. A super-peer sends notifications to its clients to inform them, that there is a

Key	Address	Access Point	MyClient	Connected	Profile
Key	IP address of client	Access Point IP address	True or false	True or false	XML DOM tree

Table 4.3: Client Information Data structure

match between their profile and a resource of another client. If the client is not connected the super-peer stores the notification and sends it the next time that the client connects as a stored notification. If a client wants to download a resource but the resource owner is not online, it can arrange a rendezvous. The access point of the resource owner, has the responsibility to inform its client, that it must send the specific resource, to the access point of the client who asked for the rendezvous.

4.7.1 Dynamic Addresses

Users are real Internet users. This means that most of them have dynamic IP addresses, and in that way, when a user disconnects and reconnects, he will have a different IP address. His resources will be inaccessible to other users, who have already received notifications for a resource of a disconnected user, because even if the user connects again, he will have a different IP address. It is obvious that we need a way to identify the clients and we cannot use their address to do that. The way to do so, is to assign each user a unique key and identify the user from the key every time he connects to the network. The key is assigned by the super-peer, that the user connects to for the first time. The network guarantees that each user has a unique key in the following way: Each super-peer has a unique key (assigned by the administrator when he sets-up the super-peer) and it uses it to create and assign client keys. Consider the case of a super-peer with key "4". The first new client will be assigned the key "41", the second "42" and so on. Since all super-peers have different keys, all clients will have different keys too.

When a client wants to download a resource using a notification, it must first find the address of the resource owner. To do so, it sends to its access point a `request other client address` message. It is understood, that each super-peer must know at all times, if a client is connected to the network and his IP address, so that he can produce the correct pointers to the resources of each client. The client information data structure is shown in Table 4.3. There will be one entry for each client.

4.7.2 New Client

When a client connects for the first time, the whole network must understand that a new client is connected and give him a unique key. The client-user chooses one of the available super-peers and sends a `new client` message. The super-peer immediately assigns a new key to the client and sends it back. The client uses this super-peer as access point to the network from there on. The steps to add a new client are:

- The super-peer assigns a new key for the client and sends it to the client.
- The super-peer adds the client to the client list and stores its address, its access point, the fact that it is connected and that it is its client. Then broadcasts through its spanning tree a `new client` message to all other super-peers.
- Each super-peer that receives that message, will add the client to the client list and store its address, its access point, the fact that it is connected and that it is not its client. Then it forwards the message to all the send nodes of the spanning tree of the root of the message.

4.7.3 Connecting

Every time a client wants to connect to the network, it must use its key. All super-peers must be aware of the fact, that the client has connected and know the new address of the client. This happens in the following way:

- The client sends a `connect` message along with its key.
- The super-peer updates the client entry with that key in the client list: stores its address, its access point, the fact that it is connected and that it is its client. Then, broadcasts through its spanning tree a `client connected` message to all other super-peers. Finally, sends to the client any stored notifications and rendezvous requests.
- Each super-peer that receives that message checks if the client was its client the previous time it was connected. If this is the case, it sends to the client any stored notifications and rendezvous requests. Then, it updates the client entry with that key in the client list: stores its address, its access point, the fact that it is connected and that it is not its client. Then, it forwards the message to all the send nodes of the spanning tree of the root of the message.

4.7.4 Disconnecting

When a user disconnects or equivalently terminates the client application program, the network must adapt and all super-peers need that information to inform other clients that they cannot download any resources from this one.

- The client sends a `disconnect` message along with its key to its access point.
- The super-peer updates the client entry with that key in the client list: stores its address as null, its access point remains the same, the fact that it is disconnected and that it is its client. Then broadcasts through its spanning tree a `client disconnected` message to all other super-peers.
- Each super-peer that receives that message, updates the client entry with that key in the client list: stores its address as null, its access point remains the same, the fact that it is disconnected and that it is not its client. Then it forwards the message to all the send nodes of the spanning tree of the root of the message.

4.7.5 Client Migration

Until now, we have assumed that a client always connects to the same access point. This is not true. P2P-DIET fully supports client migration to different super-peers. A user may want to change access point for his own reasons. For example, he may not be satisfied by the performance of the super-peer or it is possible that the super-peer does not respond for a long time to the messages of the client. Moreover, in a future version of P2P-DIET, where a load balancing mechanism will be included, the client migration process will be very useful to move clients from one super-peer with more clients (more resources to handle) to another with less clients.

When a client migrates, we must be very careful not to lose any useful information regarding the client. Consider the case that a client chooses to connect to an access point which is different than the previous access point, the super-peer that was its access point last time he was connected to the network. The previous access point will hold all possible information on stored notifications, rendezvous arranged, rendezvous files uploaded and of course the metadata of the resources of the specific client. The obvious solution is that the previous access point of the client must send all this information to the client. The new access point will need the profile and the metadata of the resources, while the old one does not need them any more. The additional steps in the connect procedure to support client migration are:

- The client sends a `connect` message along with its key.

- The super-peer checks if the client was its client the previous time it was connected. If not, the super-peer requests all the metadata of the resources from the client and stores them. It is possible that the super-peer does not have the profile of the client, either because it was not forwarded (because of the hierarchy) or because the client has not subscribed a profile. In this case, the super-peer requests from the client to send its profile, if it has one. Then, it updates the client entry with the key of the client in the client list: stores its address, its access point, the fact that it is connected and that it is its client. Then, broadcasts through its spanning tree a `client connected` message to all other super-peers.

- Each super-peer that receives that message checks, if the client was its client the previous time it was connected. If it was, the super-peer sends any stored notification, rendezvous notifications and rendezvous obligations directly to the client. Then, updates the client entry with that key in the client list: stores its address, its access point, the fact that it is connected and that it is not its client. Then, it forwards the message to all the send nodes of the spanning tree of the root of the message.

4.7.6 Adding a Super-Peer to a Working Network

An important need in an Internet scale peer-to-peer network, is to add super-peers because the number of clients is growing all the time. You need more super-peers to support more clients or equivalently to support clients in a different city or country. We have already described the process of adding a new super-peer in a network free of clients. The only difference is that, if there are already clients in the network the new super-peer needs information on the clients. It needs to know the address of each client, its access point, its key and the profiles but only those profiles that have been forwarded. Without all these information the new super-peer will not be able to serve its clients and to find a match between the resources of its clients and the profiles of other super-peer clients. All this information is stored in the client data structure of each super-peer. When the new super-peer connects, all its new neighbors will send their client data structure with a `clients` message. The new super-peer accepts only the first `clients` message. It changes all `MyClient` fields to false and it erases the profiles of the clients that are not forwarded, which are a subset of the profiles of the clients of the super-peer, that sent the `clients` message. Now it has all the information on the clients of the network and forwarded profiles from all super-peers.

4.8 Fault-tolerance

In the next two subsections we are going to discuss the fault-tolerance mechanism and why it is important to have such a mechanism. Naturally we cannot be sure

that a client or a super-peer will always be online working properly.

4.8.1 Super-Peers

Super-peers may fail in any way. For example, a crash on the system or even running out of resources (overloaded - cannot handle any more sockets), are two basic reasons for a super-peer to fail or freeze temporally. Each super-peer may be a receive node or a send node to one or more spanning trees or shortest paths of other super-peers, so the fact that a super-peer fails is very critical. If the super-peer that failed is a receive node in a shortest path, then the destination node of this shortest path will not be able to receive any more messages. If the super-peer that failed is a send node, the root of this spanning tree will not be able to correctly broadcast. Moreover, one simple super-peer is usually participating in more than one spanning trees or shortest paths with different roles, so the super-peer that failed, will cause serious problems to all others.

The proper way to solve that problem, is to make every super-peer periodically check whether its neighbor super-peers are alive and are working properly. This is done in the following way:

- Periodically send to all neighbors an `are you alive` message and wait for a specific amount of time for their answers.
- When the time has passed, check if all super-peers have answered with an `I am alive` message.
- If at least one super-peer has not answered, broadcast a `build spanning tree` message to rebuild all spanning trees and shortest paths in the network.

4.8.2 Clients

Clients may face system failures too. Additionally, clients may leave from the network silently. A user will not always use the disconnect option in the GUI, so he may turn off his computer or disconnect from the Web. In both cases his is not connected to the network and all super-peers, must be aware of that, so they do not think that his resources are on line or that he is able to receive notifications on other client resources. Again the super-peer periodically checks, if all the clients that it serves are alive.

- Periodically send to all clients that you serve an `are you alive` message and wait for a specific amount of time for their answers.
- When the time has passed, check if all clients have answered with an `I am alive` message.

- For each client that has not answered handle the situation as if you have received a `disconnect` message from the specific client.

4.9 Overloading Socket Handling

One must be very careful when handling too many sockets (connections) at a time. There is the possibility, that the computer will run out of resources and cannot handle any more open connections, which means that some messages will be lost and some messages will not be sent. In situations like this, a computer might freeze and may recover on its own after some time. We cannot rely on the possibility of recovery. Even then, many nodes of the network will be unreachable, in the sense that they will not receive messages from some super-peers, for all the time that the super-peer is down. It is clear that we need a careful control of the connections that a super-peer opens to prevent problem situations. The next three subsections present the techniques that are used to handle connections and resources.

4.9.1 Socket Handling

The basic thing that we have to be careful about is the idle time of the sockets. Idle time is the time to wait before terminating a connection, so if a connection is setup, the message is sent and no other message is sent to the same destination or received from there, the socket is closed. DIET Core ensures that when a message is sent to a node of the network, a socket will be opened. If the same node wants to send one more message to the same destination before the idle time has passed, the same socket will be used. The same thing will happen, when the destination node wants to reply or send another message to the previous node. Only when both nodes want to send each other messages at the same time, two sockets will be opened. This socket handling mechanism guarantees that a minimum number of sockets will be opened, which means that minimum resources are used and more clients or super-peers can be connected at a time.

4.9.2 Constant Connections

We are interested in avoiding the useless process of opening and closing sockets for the same destination nodes. We try to keep the sockets (connections) open, which are sure to be used for a little time. The way to keep a socket open is simple. Either start it with a rather big idle timeout or send a message before the timeout has passed using the same socket. The first solution is very unwise, because all sockets will remain open for a long time. We need only the connections, that we are going to use again for sure. The second way can be easily implemented. All we have to do is send a message to the sockets that we want to keep open,

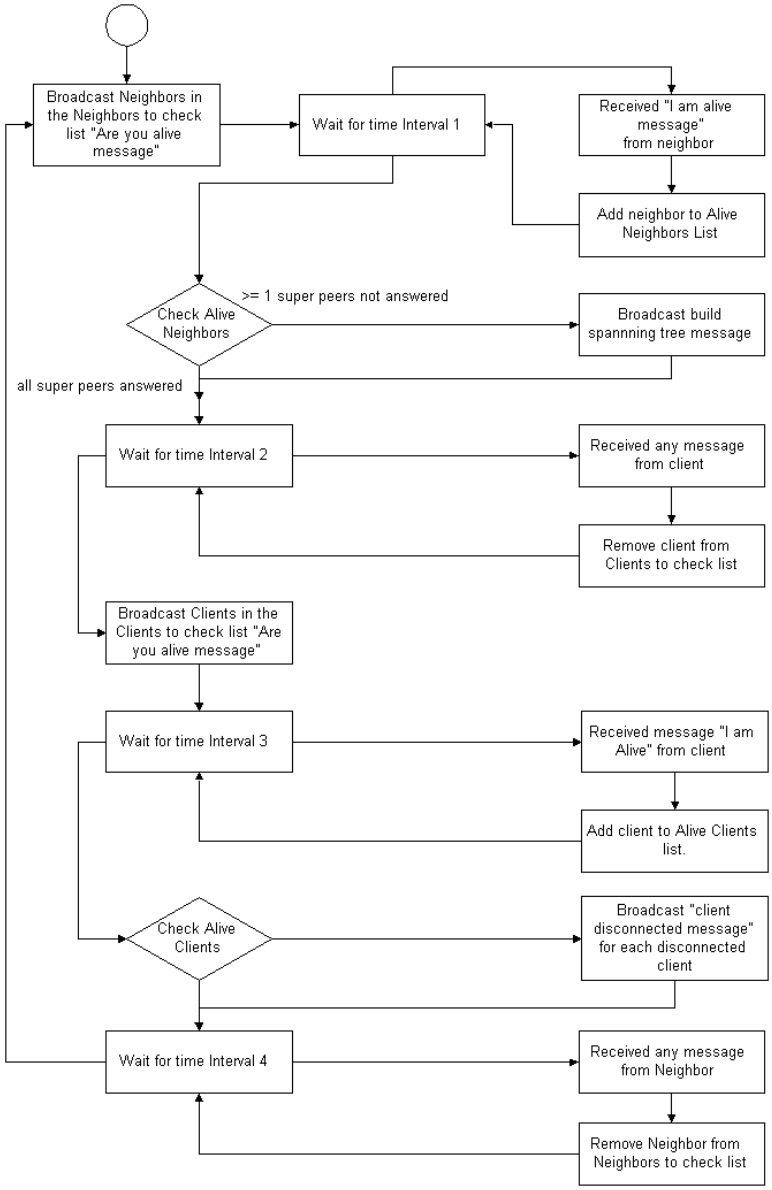

Figure 4.4: Fault-tolerance mechanism in P2P-DIET

before the timeout exceeds. We already have the mechanism that periodically sends `are you alive` messages to super-peers and clients. Thus, the only thing to do, is carefully set the idle timeout, so the super-peer will have sent for sure an `are you alive` message before the end of timeout to all neighbors and clients. In our network, super-peers send `are you alive` messages to super-peers every 3 minutes, so we set the socket idle timeout to be 3.5 minutes. In this way, a super-peer has constant connections with all its neighbors, so it does not have to setup a new connection each time it wants to send a message to a super-peer node. We do not keep open connections with clients, because clients are too many and a super-peer cannot afford to have all these connections open all the time. It is useless to do so, because clients do not communicate all the time. We check clients periodically, but with period three times the period we use to check super-peers, which is a logical number.

4.9.3 Avoiding Useless Are You Alive Messages

Another useful tactic, is not to send an `are you alive` message if the node has sent a message in the past few minutes. There is no need to ensure that a node is alive. It definitely is alive, otherwise it would not have sent a message. In this way, we achieved to avoid a great number of messages, because in a scaled network a super-peer may serve 1000 or more clients. If we can avoid, for example, 800 `are you alive` messages and 800 `I am alive` messages in reply every 10 minutes, this is a great relief for the network. The complete algorithm of keeping the sockets open and being sensitive in node failures (clients and super-peers) is shown in Figure 4.4.

4.10 Summary

In this chapter we discussed routing in P2P-DIET. We showed how we build spanning trees being careful enough not to lose any messages during this process. We explained how minimum weight spanning trees are used by a super-peer to broadcast or multicast a message and how shortest paths are used by super-peers to unicast messages. We discussed the problem of dynamic IP addresses and the solution of the unique keys. We continued by describing the way that clients connect to or disconnect from the network and how the network reacts to those events. We showed how a client may migrate to a different access point and how a super-peer may be added to the network, even after many clients have used it. In the last sections we described the fault-tolerance mechanism and our socket handling strategy to avoid overload situations. In the next chapter we present our query and event notification mechanism by describing the propagating strategy and the profile hierarchy.

Chapter 5

Query and Event Notification Service

In the previous chapter we presented the routing strategies of P2P-DIET. Our strategy is based on building spanning trees for the general graph and uses those spanning trees to deliver messages efficiently among super-peers. We showed how clients come and go, and we presented a fault-tolerance mechanism. From here on, when we say that a super-peer broadcasts a message, we mean that this happens through its minimum spanning tree as shown in Section 4.4.9 and when we say that a super-peer sends a message to another, we mean that this happens through the shortest path from the sender to the receiver as shown in Section 4.3.2.

This chapter presents the query and event notification service in detail. We give details for the basic concepts of our event service such as profiles, resources, notifications etc. We describe the strategy for propagating the profiles, which includes a profile hierarchy and we show how we build and use the hierarchy and why it is so useful for the network.

5.1 Routing Strategies

In the next subsections we describe the way that the network handles the profiles, resources, notifications, stored notifications, rendezvous and hierarchies.

5.1.1 Resources

A resource is a file, that a user wants to publish on the network, so other users may see it and download it. The super-peers are busy enough trying to find matches and keep the network in a stable situation. If the files were uploaded to super-peers, then the super-peers would have to serve download requests for all the files that their clients own. This would be a drawback in the scale character

of the network. Clients do not upload their resources. Instead, they keep them in their computers and if another client wants to download a file, they serve him. This means that clients can directly communicate regardless of whether they are connected to the same or different super-peers. Moreover, super-peers have to know at all times the address of a connected client and if it is connected or disconnected as described in Section 4.7, because when a client wants to download a resource, it needs the address of the target client, who owns the resource and the way to find that address is to ask its access point.

We have already explained why it is a good choice to give the responsibility of downloading to clients, but super-peers have to find matches. This means that they need a way to see the files of the users that they serve and we have already said that files are not uploaded. This problem is solved, by making clients send metadata for their resources to their access points. The super-peers use those metadata to find a match between the metadata and incoming profiles. The metadata describe a resource and part of them is initialized by the system, for example, the size or the name of the file, while the rest of them by the user. We believe that if a user wants to publish a file, he will spend a few seconds to give a value to the metadata fields. The exact metadata that currently the system supports are the fields of the language presented in Chapter 3. For now all we need to know is that a metadata file concludes the key of the resource owner and the file name.

5.1.2 Propagate Profiles

The fundamental characteristic of an event notification service is the profile (long-standing query), which will produce notifications, even if files are published to the network after the profile was subscribed. A user subscribes with his profile to his access point. The access point saves the profile of the client and changes it only if the client sends a new profile. Each super-peer knows the resources of the clients that it serves and can find matches between the resources of its clients and incoming profiles. This means that a client can see files of all other clients only if its profile is broadcasted to all super-peers or if the resources of all clients are sent to its access point. The critical decision, is that of propagating profiles or resources. We expect our network to scale and support many clients. Each client will have a few profiles, but may have a large number of resources. If, for example, users share mp3s, it will be logical to assume that each user might share a thousand or more files in which case the user will have published a thousand resources. It is obvious that the process of propagating profiles is much cheaper for the network. The solution of propagating profiles guarantees the long-standing query ability of the profile, because when the profile arrives for the first time to a super-peer it is saved in the client information data structure as shown in Figure 4.3, and the super-peer will try to find matches between the profile and the metadata files of its clients. If one of its clients adds a resource,

this resource will be tested with all the profiles stored in the super-peer, which are all the profiles of its clients and all the profiles that are broadcasted by other super-peers. It is possible to avoid a lot of messages using a hierarchy to handle the forwarding of the profiles. The hierarchy will be explained in detail in Section 5.2.

5.1.3 Notifications

Clients subscribe profiles, pose queries, publish resources and wait for notifications. A notification is actually a pointer to a resource of another client. In fact, it is the resource metadata file, which is sent all the way to the client, who has the profile that caused the match. Thus, the client knows all the metadata of the file and the resource owner key, so if it wants to download the file all it has to do, is to find the address of the resource owner with the key concluded in the notification and contact it to ask the file. But how the notification arrives to the interested client? The notification is produced by the access point that serves the resource owner client, either when the forwarded profile of the client came or when the resource was added and the profile preexisted. In both cases, the notification message, not the notification itself, concludes the key of the client, who owns the matching profile. It would be possible to force that super-peer to contact the interested client and deliver the notification, but this would have brought an anarchy to the system, because all super-peers would communicate with all clients. Moreover, as we will see in the next subsection, having communication only with the access point gives solutions to problem situations and it is necessary for the profiles hierarchy to work. In this way, when a super-peer finds a match between one of its resources and a profile, which belongs to a client of another super-peer, it sends the notification to that super-peer (using as we described in the previous chapter the spanning tree of the receiver). The super-peer that receives the notification forwards it to its client.

5.1.4 Stored Notifications

We have already discussed that clients are real users that connect to or disconnect from the network at any time. This means that clients are not online all the time. When they subscribe a profile they receive notifications on preexisting resources in just a few seconds. No one can guarantee that the client will be always online, the time that a matching resource is added to the network to receive any notifications. We cannot ignore this situation and lose the notification. Actually, we do not expect clients to wait online for other clients to put resources that match their profiles. The notifications are stored in the Stored Notifications directory in the super-peer environment of the access point of the client, and delivered to the client the next time that it connects to the network. This means that when a super-peer realizes that it must send a notification to a client, it has to check

if the client is connected or not. If the client is connected the notification is delivered otherwise it is stored. As we said in the previous chapter the access point sends the stored notification even if the client connects to a different super-peer in order to avoid losing any notifications that are produced during the time that the client was disconnected.

5.1.5 Rendezvous

Again we will deal with the problem that a client is not connected all the time to the network. In the previous sub section we showed how we store the notifications if a client is not online to receive them. There is one more similar situation: a client wants to download a file for which it has already received a notification, but the resource owner client is not online so the resource is unreachable. If we do nothing about that, the client will have to wait or to be lucky, to be online at the same time with the resource owner. We can solve this by uploading the file and give it to the client who wants it. The network will always be online so it is easy enough to upload the file from the resource owner the next time that it connects and give it to the interested client, when it asks for it. Let us now describe all the process of a rendezvous request. First of all the rendezvous is arranged when a client cannot download a resource, because the other client is not online and sends to his access point an `arrange rendezvous` message along with its key, the resource owner key and the file name. The access point will send the message to the access point of the resource owner, where the rendezvous data (XML DOM tree) will be updated. When the resource owner client reconnects, the super-peer will check the rendezvous tree and send a `send file to server` message to the client. This means that the client has to send the specific file to the access point of the client, who asked for the resource. In this way, the file is uploaded to the access point of the interested client and stored to the Rendezvous directory in the super-peer environment of the access point of he client who made the rendezvous request.

5.1.6 Queries

P2P-DIET works as an event notification service and as a typical search system too, giving maximum flexibility to users to find files that are interested on. To achieve the search ability we use *queries*. A query is in fact the same as the typical profile. Thus, it supports the same attributes and the same language. The only difference is that the system handles it in a different way. A query is not stored and it is disappeared after all super-peers have seen it and tried to find a match. When a client sends a query, it is broadcasted by the access point to all super-peers. The answers that are produced by queries are the same with the ones that produced by the typical profiles. A client will use a query to find anything that was added to the network up to this moment, without affecting its

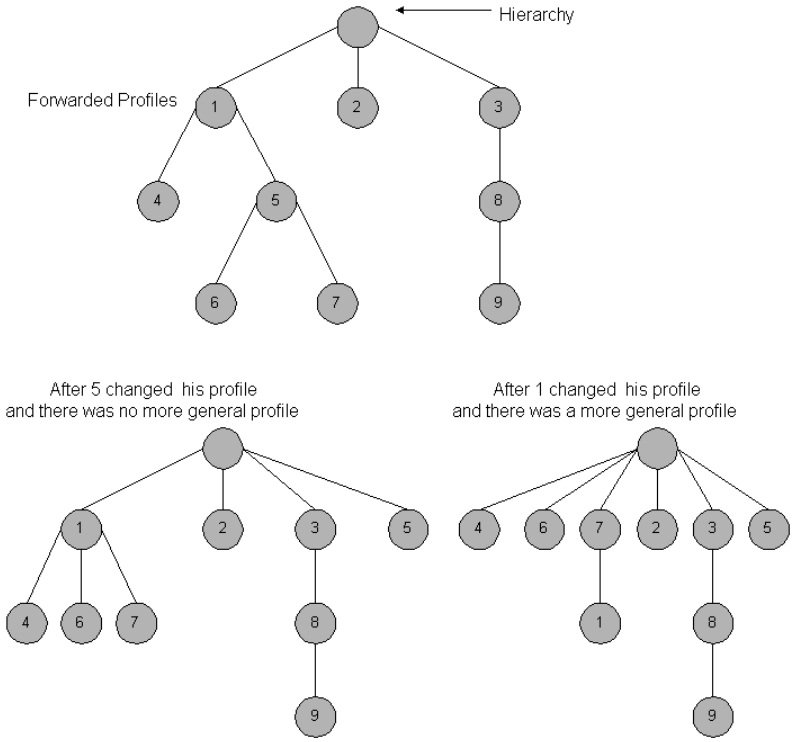

Figure 5.1: Hierarchy Examples

original profile, which will continue to produce notifications as new resources are published. A user may use a query without any limitations.

5.2 Profile Hierarchy

Each client may subscribe a profile. We expect our network to scale so each super-peer will serve a large number of clients. It is more than possible that some of the clients will be interested in the same type of files, which means that they will subscribe with overlapping profiles. We will try to take advantage of that, to load the network with fewer messages. This section explains in detail why the profile hierarchy is important in an event based peer-to-peer service. We will show how we build and use the hierarchy. Moreover, we will discuss all critical situations and we will give examples for each one.

We have already explained that we choose to forward profiles. The hierarchy tries to take advantage of the similarities between different profiles in order to forward a profile only if we really need to do so. First of all, we need a way to compare two profiles and realize if the first one is *more general* than the second. The definition more general means that the more general profile will find matches (resources), that satisfy the less general profile too. The next reasonable step is

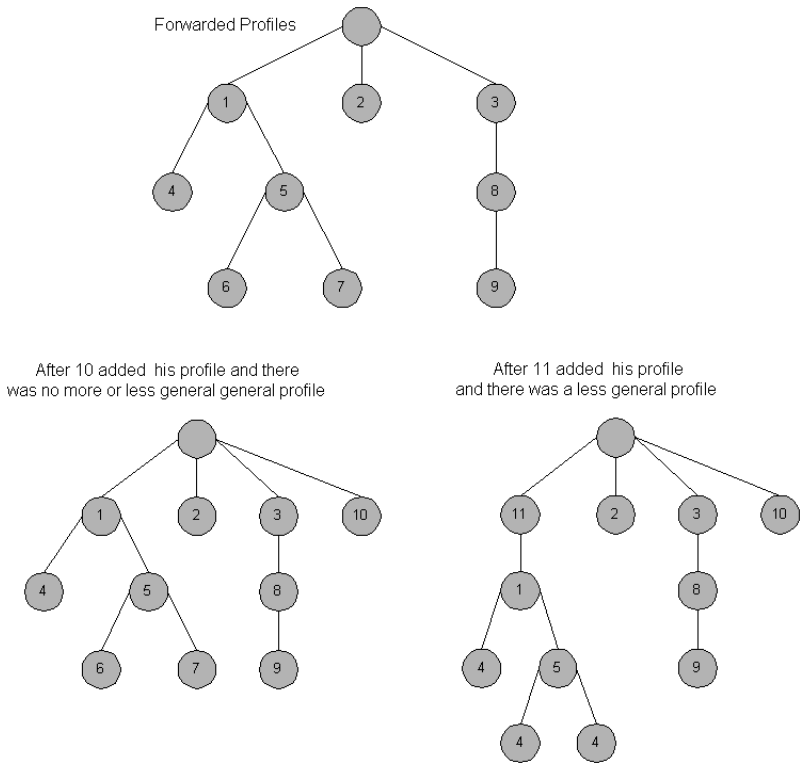

Figure 5.2: Hierarchy Examples

to decide not to forward the less general profile since the more general will bring all the notifications that the less general would bring and even more. One could say that we do not need a hierarchy but as we will explain all the details in these section, it will be clear that we need the hierarchy structure otherwise we would have to repeat the same checks many times.

We forward only the more general profiles and we put all the profiles in a tree as shown in Figure 5.1. The profiles that lie to the level one of the tree are the profiles that actually forwarded, because there is no more general profile than them. Those forwarded profiles have children in the tree, which are profiles less general than their father (the forwarded profile), and those children may have other children etc. The depth of the hierarchy is not limited by the system. An example of a profile with two less general profiles is shown in Figure 5.3.

Let us now give an example based on the Figure 5.1. The numbers in the nodes of the tree indicate the key of the client, who owns the profile and the time sequence that the profile was subscribed. The clients are connected to the same access point, that has the hierarchy. At first client 1 sends its profile. There are no other profiles, so the profile is forwarded and goes to the first level of the hierarchy. Then client 2 sends its profile but there is no more general profile, so it is forwarded and the profile goes to the first level. The same things happen with profile 3. Until now profiles 1, 2 and 3 are forwarded. Then client 4 subscribes

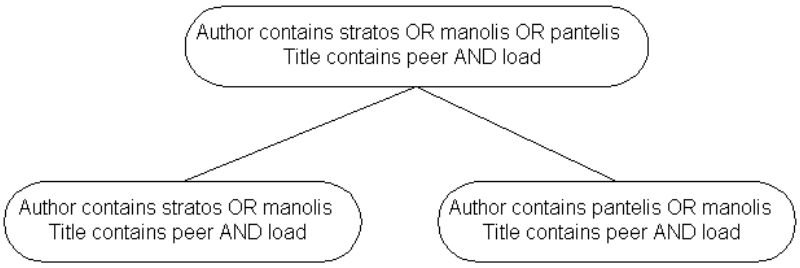

Figure 5.3: An example of profiles in hierarchy

its profile, which is less general than the profile 1, so it is not forwarded and goes to the second level of the hierarchy as a child of the profile1. The same things happen with profile 5. When client 6 sends its profile the system realizes, that it is less general then the profile 1 but the check is repeated in the second level as well. The profile 6 is compared with both profile 4 and 5. It is less general than the profile 5, so we put it in the third level of the hierarchy as a child of profile 5. We continue in the same way for each profile that comes.

Until now we showed how we decide if a new profile must be forwarded or not, based on the profiles that are already forwarded. There is the possibility that the new profile is more general than a profile that is forwarded. We could forward the new profile too, but in that way we would receive almost double the number of notifications since we will have forwarded two similar profiles. We adapt to this situation by always making double checks. When a new profile comes and we compare it with another we do the check in the following way: consider a case that the new profile is profile A and we want to compare it to profile B. At first we check if profile B is more general than the profile A and if it is not, we check if the profile A is more general then profile B. The first case is the one that we have already described in the previous example. To understand the second one we could look the example of the Figure 5.2. The first tree shows the hierarchy. When client 11 sends its profile, the system compares it to the profile 1 and finds that profile 1 is not more general than 11. Then, the check is reversed and it is found that profile 11 is more general than profile 1, so the hierarchy changes as shown in the figure. Of course, the previous checks do not happen only in the first level of the hierarchy. In fact, by always making double checks between two profiles, we do not care for the sequence that the profiles come to the super-peer. For example, in Figure 5.2 in the first tree if profiles 6 and 7 come earlier than the profile 5, then they would be in the second level as children of profile 1. Later, when profile 5 comes, we see that it is less general than profile 1 and in the next step we see that it is more general than profiles 5 and 6 so the hierarchy looks like the first tree in Figure 5.2.

Based on the previous discussion on double checks, we can see the need for extra messages that cancel a previous broadcast of a profile. When we move a profile from the first level of the hierarchy to the second, because a more general

profile came, we forward the more general one but we also have to cancel the profile that was already forwarded. If we do not do that we will receive many notifications more than one time. The super-peer broadcasts a `remove profile` message through its spanning tree to inform all other super-peers that it does not want to receive any more notifications for that profile. The rest of the super-peers just erase the profile from their client information data structure.

Another thing that we have to be careful about, is no to lose any information because of the hierarchy. When a new profile is subscribed but not forwarded because of a more general profile, the client will receive only notifications on future resources and will lose notifications on preexisting resources. This is an easy problem to bypass by firing the search scenario for the new profile. In this way, the client will receive all notifications on resources that preexisted the profile and the profile will not be forwarded saving from the network the notifications on future resources.

The previous observation eliminates the thoughts that the hierarchy removes from the network the load of forwarding profiles. The profiles are not forwarded but search messages are broadcasted so it is exactly the same thing. The load is saved by the limited notification messages. One notification will satisfy more than one profile. The notification is transmitted along with the information of the profile that caused the match. Of course, the notification is not delivered only to the client, who owns the forwarded profile, but to all clients that their profiles are children of the forwarded profile in the hierarchy or children of its children and so on. For example, if a notification arrives for the client 1 in Figure 5.2 then the notification is sent to client 1,4,5,6,7. Now it is better understood why use the hierarchy instead of just knowing the profiles that we forward. For example, when a notification would arrive we would have to check all the profiles for match and then deliver notifications.

5.3 Summary

In this chapter we discussed the query and event notification mechanism in P2P-DIET. We explained the basic concepts of the service such as profiles, queries, notifications, resources and we gave details for the propagating strategy. We presented the hierarchy mechanism used to minimize network traffic, by reducing the number of notifications that are produced. In the following chapter we will describe the communication protocols, that the agents use to contact other agents, exchange information and cooperate to achieve a final goal.

Chapter 6

Agent Communication Protocols

In the previous chapter we presented the event notification mechanisms of P2P-DIET. We described the strategy used to achieve both the notification ability and the typical query scenario. In this chapter describe in detail the protocols that agents use in P2P-DIET universe to communicate. There are four different protocols:

1. The *super-peer - super-peer* communication protocol, which is used by agents in a super-peer environment to communicate with agents in the local or a remote super-peer environment.

2. The *client peer - super-peer* communication protocol, which is used by agents in a client peer environment to communicate with agents in a remote super-peer environment.

3. The *super-peer - client peer* communication protocol, which is used by agents in a super-peer environment to communicate with agents in a remote client peer environment.

4. The *client peer - client peer* communication protocol, which is used by agents in a client peer environment to communicate with agents in the local or a remote client peer environment.

Note that remote communication is achieved by the Messenger agent. This means that when an agent wants to communicate with a remote agent, he communicates with a local messenger agent and the messenger will implement the remote communication. When we describe the messages in the next sections we will refer to remote communication directly between the sender and the receiver agent to avoid repeating the following three steps:

1. The sender connects to a local messenger and gives him the message, the environment address and the target agent.

2. The messenger migrates to the remote environment.

3. The messenger connects to the receiver agent and deliver the message.

6.1 Super-Peer - Super-Peer Communication Protocol

6.1.1 New Neighbor message

When a super-peer is about to set up, the administrator gives as input the neighbors of the super-peer. Those neighbors must be informed that a new super-peer is added in their neighborhood. The new neighbor message is sent by a super-peer agent to another super-peer agent to inform him, that he is a new neighbor. The new super-peer agent sends his environment address along with the message, so all his neighbors know his address and can communicate with any agent in his environment. The existence of the new super-peer changes the topology of the network, which means that all super-peers must rebuild their spanning trees. The `new neighbor` message is handled both as a `new neighbor` message and as a `build spanning tree` message and it practically fires the process of rebuilding all spanning trees. When a super-peer agent broadcasts a new neighbor message to all his neighbors, he expects to receive after a while reply messages for his spanning tree and `build spanning tree` messages from other super-peers. When a super-peer agent receives such a message, he adds the address of the new super-peer to the neighbors list and sends an `add super-peer` message to the Are-You-Alive messenger to check the new super-peer too.

6.1.2 Build Spanning Tree message

The `build spanning tree` message is a request from a super-peer to build his spanning tree. The message is sent from a super-peer agent to another super-peer agent. A super-peer agent broadcasts this message to all his neighbors, when he wants to build his spanning tree or when he wants to forward a same message from another root. In the second case the message is sent to all the neighbors of the super-peer, except the sender and the only information changed in the packet is the sender address. When a super-peer agent receives such a message, he has to decide if he will accept it or not. The decision is based on the accepted build spanning tree messages list and on the stability of the network. If the network is stable the message is accepted anyway and the super-peer has to rebuild his spanning tree, because the topology has changed, so his sends a `send me time to broadcast build spanning tree` message to the spanning tree scheduler. If the message is not stable the message is accepted only if the address of the root of the message is not in the accepted build spanning tree messages list. If the message is accepted the time is marked as the time that the last build spanning tree message accepted and the sender of the message is marked as a

receive node in the spanning tree of the root of the message. The address of the root of the message is put in the accepted build spanning tree messages list. Moreover, the super-peer agent must reply to the sender super-peer agent with a `reply spanning tree` message.

6.1.3 Reply Spanning Tree message

The `reply spanning tree` message is sent by a super-peer as a reply to a `build spanning tree` message. The message must contain the root of the `build spanning tree` message and the sender of the `reply spanning tree` message. The receiver will mark the sender as a send node in the spanning tree of the root.

6.1.4 New Client message

The `new client` message is sent by a super-peer to a remote super-peer to inform him that a new client is connected to the network. The message is broadcasted by a super-peer agent, when he receives a new client message from a client. It will contain the address of the root, the sender and information on the client: the client key, his address and his access point. When a super-peer agent receives a `new client` message from another super-peer agent, he puts the client to the client information data structure and forwards the message to all the send nodes of the spanning tree of the root of the message by changing only the sender information.

6.1.5 Client connected message

The `client connected` message is sent by a super-peer to a remote super-peer to inform him that a client is connected. The client is not a new one. He is a registered client, he has a key, and now he is connected to the network again. The message is broadcasted by his access point because all super-peers need to know that the client is connected again, his address and his access point. When a super-peer receives a `client connected` message he updates the client entry in the client information data structure. Moreover, ha has to check if the client was his client the previous time he was connected, in which case he has to send the client any notifications or rendezvous requests. Then the super-peer agent forwards the message to all the send nodes in the spanning tree of the root of the message.

6.1.6 Client Disconnected message

The `client disconnected` message is sent from one super-peer to another to indicate that a client is disconnected from the network. The super-peer agent who is the access point of the client will broadcast the message. The message

must contain the key of the client and of course the root and the sender of the message. When a super-peer agent receives a `client disconnected` message will update the client information data structure by marking the address of the client as null and the boolean connected as false. The access point information will not be updated until the next time the client is connected in order to send any notifications for the client to his access point. Additionally the super-peer agent that receives such a message must forward it to all the send nodes in the spanning tree of the root of the message by changing only the sender address.

6.1.7 New Profile message

The `new profile` message is sent by a super-peer to another to inform that a client subscribed a new profile. The message is broadcasted by the access point of the client and it will contain the key of the client, the DOM tree of the profile and the root and sender of the message. When a super-peer agent receives a `new profile` message updates the client information data structure. The profile will be checked with all the resources of the clients of this super-peer. If there is any match a notification will be sent to the root of the message, which is the access point of the client with the new profile. Then the super-peer will forward the message to all the send nodes in the spanning tree of the root of the message by changing only the sender address.

6.1.8 Notification message

The `notification` message is sent from one super-peer A to a super-peer B when a notification is produced and must be delivered to one of the clients that the super-peer B serves. The resource owner is one of the clients that super-peer A serves. The message is sent from super-peer A to super-peer B through the spanning tree of super-peer B. The message will contain the key of the key of the client whose profile fired the notification process, the receiver which is the access point of the client and the sender. If the super-peers are not neighbors the message will have to pass through other super-peers. When a super-peer receives a `notification` message, he checks if he is the receiver. If he is not he forwards the message to the receive node of the spanning tree of the receiver of the message by changing only the sender address. If the super-peer is the receiver of the message then he has to check if the client is online to receive the notification. If he is online the notification is sent. If he is not online the notification is stored in the notifications directory of the client. The notifications are stored to hard disk for protection in a possible failure of the super-peer. Then the super-peer agent will repeat the process for all the clients that are children of the client in the profile hierarchy. The profiles of those clients ar less general than that of the client who fired the notification, so they are interested in the notification too.

6.1.9 Client List message

The `client list` message is sent by a super-peer to another to give him the client information data structure. This message is sent when a new super-peer is added to the network or when a super-peer recovers. In both cases the super-peer needs to know all about the clients that are registered to the network so he can serve his clients and send notifications to clients of other super-peers. The message contains only the client data structure. The super-peer that receives such a message updates all the MyClient fields with the value false and creates a clone of the client information data structure.

6.1.10 Arrange Rendezvous message

The `arrange rendezvous` message is sent from super-peer A to super-peer B to inform him that one of the clients of super-peer A wants to have a rendezvous with a file of one of the clients of super-peer B. The super-peer agent of the client who wants the rendezvous sends this message through the spanning tree of the access point of the client, who is the resource owner. The message must contain the key of the client who is interested on the resource, the key of the client who is the resource owner, the file name of the resource, the receiver of the message and the sender. When super-peer agent receives an `arrange rendezvous` message, he checks if he is the receiver and if he is not, he forwards the message to the receive node of the spanning tree of the receiver of the message by changing only the sender address. If the super-peer agent is the receiver, then he updates the DOM tree of the rendezvous of the resource owner to contain the information that a client wants a rendezvous with one of his files. The key of the client is stored and the name of the file.

6.1.11 Search message

The `search` message is a request for search. The message is sent by a super-peer agent when one of his clients sends a query as a query to the network. The super-peer agent will try to find any matches with resources of the clients that he serves and if there is a match the answer is sent immediately. Then the super-peer agent broadcasts the message to all super-peers. The message must contain the key of the client, the query, the root and the sender of the message. When a super-peer agent receives a search message tries to find any matches between the query and the resources of the clients that he serves. If there is a match a search notification message is sent to the access point of the client who sent the query. Then the super-peer agent will forward the message to all the send nodes in the spanning tree of the root of the message by changing only the sender address.

Message	Packet
New Neighbor	Address of the sender
Build Spanning Tree	Address of the root, Address of the sender
Reply Spanning Tree	Address of the root, Address of the sender
New Client	Address of the root, Address of the sender, Key of the client, Address of the client, Address of the Access Point of the client
Client connected	Address of the root, Address of the sender, Key of the client, Address of the client, Address of the Access Point of the client
Client Disconnected	Address of the root, Address of the sender, Key of the client
New Profile	Address of the root, Address of the sender, Key of the client, DOM tree of the profile
Notification	Address of the root, Address of the sender, Key of the client, DOM tree of the notification
Client L ist	Client list
Arrange Rendezvous	Address of the root, Address of the sender, Key of the client, Key of the resource owner, Name of the resource file
Search	Address of the root, Address of the sender, Key of the client, DOM tree of the search profile
Notification Search	Address of the root, Address of the sender, Key of the client, DOM tree of the search notification
Child Profile	Address of the root, Address of the sender, Key of the client, DOM tree of the child profile
Remove Profile	Address of the root, Address of the sender, Key of the client

Table 6.1: Super-Peer agent to Super-Peer agent messages

6.1.12 Answer message

The `answer` message is sent by super-peer A to super-peer B to inform him that one of the clients of super-peer A has a resource that matched the query of one of the clients of super-peer B. The message is sent by super-peer A through the spanning tree of super-peer B. The packet of the message must contain the key of the client whose query caused the answer, the DOM tree of the answer, the receiver and the sender of the message. When a super-peer agent receives such a message, he checks if he is the receiver and if he is not, he forwards the message to the receive node of the spanning tree of the receiver of the message by changing only the sender address. If the super-peer agent is the receiver of the message, he sends the answer to the client. An answer will not be saved if the client is not online to receive it. We assume that a client will send a query and then wait for a few seconds to receive any answers.

6.1.13 Child Profile message

The `child profile` message is sent in order to forward the child profile. The child profile was a child in the profile hierarchy of a profile that was forwarded. If the forwarded profile is cancelled for any reason, for example the client subscribes a new profile which belongs in a different position in the hierarchy. The children of the profile in the hierarchy must be forwarded so their clients will receive any notifications. The difference when forwarding a child profile is that there is no need to match the child profile with any preexisting resources in the network because any notifications will have been received because of the previous forwarded profile. When a super-peer agent receives a `child profile` message will update the client information data structure to conclude the child profile but he will not try to find any matches with the resources of his clients. The message will contain the child profile, the key of the client who owns the profile, the root and the sender of the message. The super-peer agent who is the access point of the client will broadcast the message through his spanning tree and all other super-peer agents will forward the message to the send nodes of the root by changing only the sender address.

6.1.14 Remove Profile message

The `remove profile` message is sent from one super-peer to another to inform him that he must delete the specific profile from the client information data structure. The message must contain the key of the client whose profile must be removed, the root and the sender of the message. This message is sent by the access point of the client when the profile of the client has been moved from the first level of the hierarchy. This is the case when a more general profile is subscribed and the profile goes to the second level under the new profile or when the client sends a new profile which is less general than one or more profiles in the hierarchy. The profile must not produce any more notification which means that all super-peer agents must remove it from the client information. When a super-peer agent receives a `remove profile` message, he removes the profile and forwards the message to to the send nodes of the root by changing only the sender address.

6.1.15 Broadcast Neighbors Are You Alive message

The `broadcast clients are you alive message` is sent by the clock agent to the Are-You-Alive messenger in the same super-peer environment to inform him that it is time to broadcast all neighbor super-peer agent of the super-peer an are you alive message. The message will not contain any objects. The clock agent does not expect any replies. The Are-You-Alive messenger will start sending are you alive messages to all the super-peer agents of the super-peer that are

supposed to be online.

6.1.16 Check Alive Clients message

The `check alive clients` message is sent by the clock agent to the Are-You-Alive messenger in the same super-peer environment to inform him that it is time to check the clients that replied to the are you alive message. The message will not contain any objects. The clock agent does not expect any replies. The Are-You-Alive messenger will check the Alive Client list and for any client that not answered he will sent the super-peer agent a `client disconnected` message.

6.1.17 Check Alive Neighbors message

The `check alive neighbors` message is sent by the clock agent to the Are-You-Alive messenger in the same super-peer environment to inform him that it is time to check the neighbors that replied to the are you alive message. The message will not contain any objects. The clock agent does not expect any replies. The Are-You-Alive messenger will check the Alive Neighbors list and for any neighbor that not answered he will sent the super-peer agent a `super-peer disconnected` message.

6.1.18 Send Me Time To Broadcast Spanning Tree message

The `send me time to broadcast build spanning tree` message is sent by the super-peer agent to the build spanning tree scheduler to the same super-peer environment to inform him that he must send a time to broadcast spanning tree message after a specific amount of time. The message will not contain any objects. The build spanning tree scheduler will found the time T to wait before sending the message by multiplying the key of the super-peer with the interval for spanning tree. The scheduler will sleep for T time and then will send the message.

6.1.19 Time To Broadcast Spanning Tree message

The `time to broadcast build spanning tree` message is sent by the build spanning tree scheduler to the super-peer agent in the same super-peer environment to inform him that it is time to broadcast a build spanning tree message. The message will not contain any objects. The super-peer agent that receives the message will broadcast a `build spanning tree` message to all his neighbors.

6.1.20 Messenger Send message

The `messenger send` message is sent by the super-peer agent or the Are-You-Alive messenger to a messenger in the messenger pool of the same super-peer environment to inform him that he must migrate to a remote environment and deliver a message to an agent. The message will contain the address of the remote environment, the family tag of the receiver agent and of course the message to deliver. The messenger will migrate to the remote environment and try to deliver the message to the target agent. If there is no messenger in the pool the super-peer agent or the Are-You-Alive messenger will create a new messenger to deliver the message.

6.1.21 Destroy Yourself message

The `destroy yourself` message is sent by the super-peer agent or the Are-You-Alive messenger to a messenger of the same super-peer environment to inform him that he must commit suicide because there is no room in the messenger pool in this environment. This happens when a messenger arrives in the super-peer environment, deliver the message and the super-peer agent or the Are-You-Alive messenger realizes that there are too many messengers in the pool. When the messenger receives the destroy yourself message will give up his thread and die. This is a procedure for extra protection on the danger of running a super-peer out of resources.

6.1.22 Add Client message

The `add client` message is sent by the super-peer agent to the Are-You-Alive messenger to the same super-peer environment to inform him that a client is connected, so the Are-You-Alive messenger will periodically check this client too, if he is alive. The message must contain the address of the client only. The Are-You-Alive messenger will put the client to his client list and from here on the client will receive are you alive messages periodically.

6.1.23 Remove Client message

The `remove client` message is sent by the super-peer agent to the Are-You-Alive messenger to the same super-peer environment to inform him that a client is disconnected, so the Are-You-Alive messenger will stop checking this client. The message must contain the address of the client only. The Are-You-Alive messenger will remove the client from his client list.

6.1.24 Add Super-Peer message

The `add super-peer` message is sent by the super-peer agent to the Are-You-Alive messenger to the same super-peer environment to inform him that a super-peer is added to the network as a neighbor, so the Are-You-Alive messenger will periodically check this super-peer too, if he is alive. The message must contain the address of the super-peer agent only. The Are-You-Alive messenger will put the super-peer agent to his Neighbors list and from here on the super-peer agent will receive are you alive messages periodically.

6.1.25 I Am Alive message

The `I am alive` message is sent from a super-peer agent to a remote Are-You-Alive messenger as a reply to the message are you alive that the remote Are-You-Alive messenger sent. The message must contain the address of the super-peer agent that replies to the are you alive message. The remote Are-You-Alive messenger will understand that the super-peer agent is still alive.

6.1.26 super-peer Disconnected message

The `super-peer disconnected` message is sent by an Are-You-Alive messenger to the super-peer agent to the same super-peer environment to inform him that one of his neighbors is disconnected or failed. The message will contain the address of the super-peer agent that failed. The super-peer agent will understand that the interconnection topology of super-peer has changed and he will broadcast a `build spanning tree` message.

6.1.27 Client Disconnected message

The `client disconnected` message is sent by an Are-You-Alive messenger to the super-peer agent to the same super-peer environment to inform him that one of his clients is disconnected or failed. The message will contain the address of the client agent that failed. The super-peer agent will update the client information to mark that the client is not online and he will broadcast through his spanning tree a `client disconnected` message to all super-peers.

6.1.28 Are You Alive message

The `Are You Alive` message is sent by an Are-You-Alive messenger to the super-peer agent to a remote super-peer environment to check him if he is alive. The message will contain the address of the remote Are-You-Alive messenger. The super-peer agent will reply to the Are-You-Alive messenger with an I am alive message.

Message	Packet
From super peer agent to are you alive messenger	
Add Client	The address of the client agent
Remove Client	The address of the client agent
Add Super Peer	The address of the super peer agent
I A m Alive	The address of the super peer agent
From are you alive messenger to super peer agent	
Super Peer Disconnected	The address of the super peer agent
Client Disconnected	The address of the client agent
Are You Alive	The address of the Are you alive messenger

Table 6.2: Super-peer agent to Are You alive messenger messages

6.2 Client Peer - Super-Peer Communication Protocol

6.2.1 New Client message

The `new client` message is sent by the client agent to a remote super agent to inform him that a new client wants to connect to the network with him as an access point. The message will contain the environment address of the client agent only. The super-peer agent will build anew key for the client. He will send the key to the client with a Key message and then he will create the folders of the client in the hard disk. Those folders are the Rendezvous folder, the notifications folder, the profiles folder and the resource folder. Then the super-peer agent will broadcast a `new client` message through his spanning tree to all super-peers.

6.2.2 Connect message

The `connect` message is sent by the client agent to a remote super agent to inform him that a new client wants to connect to the network with him as an access point. The message will contain the environment address of the client agent and the key. The super-peer agent will check if this client was his client the previous time he was connected to the network. If he was the super-peer agent will send him any stored notifications. If he was not the super-peer agent asks from the client the metadata of his resources with a request resources message. The super-peer agent needs the resources to satisfy profiles of other clients. Then the super-peer agent will broadcast a `client connected` message through his spanning tree to all super-peers.

6.2.3 Disconnect message

The `disconnect` message is sent by the client agent to a remote super agent to inform him that the client is disconnected from the network. The message must contain only the key of the client. The super-peer agent will update the client information to mark the client as disconnected and his address as null. Then he will broadcast a `client disconnected` message through his spanning tree to all super-peers.

6.2.4 New Profile message

The `new profile` message is sent by the client agent to a remote super-peer agent to subscribe a new profile for the client. The message must contain the key of the client and the DOM tree of the profile. The super-peer agent will update the client information to conclude the new profile of the client. There is no difference if this is the first time that the client subscribes a profile or if there was already a subscribed profile from this client. In the second case the old profile will be erased. The profile must take his place in the profile hierarchy. If there is no older profile the profile will be forwarded unless there is a more general profile already forwarded. If there is a previous profile of the same client in the hierarchy then carefully the profile must be replaced. The older profile will be removed and then the new one will take his place.

6.2.5 Query message

The `query` message is sent by the client agent to the remote super-peer agent as a query for the network. The message must contain the query which is the query and the key of the client. The super-peer agent will try to match the query with the resources of this clients and sends any answers. Then the super-peer agent will broadcast a query message to all super-peers through his spanning tree, so if there are any matching resources somewhere in the network the appropriate answers will produced.

6.2.6 New Resource message

The `new resource` message is sent by a client agent to a remote super-peer agent to inform him that the client wants to publish a new resource to the network. The message must contain the key of the client and the DOM tree of the resource. The super-peer agent will try to find any matches between the new resource and the profiles of his clients and send any notifications to directly to his clients with a notification message. If a client is not connected his notification will be stored. Then the super-peer agent will check all the forwarded profiles from other super-peers. If there is a match between the new resource and one of the

forwarded profiles, he sends any notifications to the access point of the client who owns the forwarded profile through the spanning tree of that access point with a notification message.

6.2.7 Remove Resource message

The `remove resource` message is sent by the client agent to a remote super-peer agent to inform him that the client wants to unpublish one of his resources. The message must contain the key of the client and the name of the resource. The super-peer agent will erase the DOM tree from the memory with the name of the resource and the XML file from the hard disk.

6.2.8 Resources message

The `resources` message is sent by the client agent to a remote super-peer agent to send all metadata for the resources of the client. This message is sent as a reply ro the request resources message of the super-peer agent when the client connects to a different access point that his previous one. The message must contain the key of the client and the DOM tree of each of the resources of the client. The super-peer agent will store the DOM trees as XML files to the resources directory of the client. There is no need to check for matches because those metadata were online to the previous access point of the client so any notifications will have been produced.

6.2.9 Request Other Client Address message

The `request other client address` is sent by the client agent to a remote super-peer agent to ask for the address of another client in the network based on the key of that client. The message must contain the key of the client who sends the message and the key of the client to find the address. The request other client address message is used in many cases. A client agent uses this message, when the user wants to download a resource from a remote client and needs his address for a direct connection. The message is also used when a user wants to chat with another client so the address of the remote client is necessary. The super-peer agent will check his client information to see if client is connected. If the client is not connected, the super-peer agent will send a requested address not available message to the client. If the client is online the super-peer agent sends a requested address message along with the address of the client.

6.2.10 Request Access Point List message

The `request access point list` message is sent by a client agent to a remote super-peer agent to ask for the list with all the super-peers in the network. The

client agent needs that list to give the choice to the user to connect to different access points. The message must contain only the key of the client. The super-peer agent sends all the addresses of the super-peer agents in the spanning tree data structure with an access point list message.

6.2.11 Arrange Rendezvous message

The `arrange rendezvous` message is sent by a client agent to a remote super-peer agent as a request for a rendezvous with a resource. The message is sent when a user wants to download a resource but the resource owner is not online so the client decides that he wants a rendezvous. The message must contain the key of the client, the key of the resource owner and the name of the resource. The super-peer agent will check if the resource owner is one of his clients and this is the case, he updates the rendezvous DOM tree of the client to conclude the new rendezvous. If the client is served by another super-peer then a `arrange rendezvous` message is sent to the remote super-peer agent through his spanning tree.

6.2.12 Request Rendezvous Condition message

The `request rendezvous condition` message is sent by a client agent to a remote super-peer agent to ask the condition of the rendezvous requested by this client. The message must contain the key of the client. The super-peer agent will sent any rendezvous notifications to the client.

6.2.13 Request Rendezvous File message

The `request rendezvous file` message is sent by client agent to a remote super-peer agent to request a rendezvous file for download. The message must contain the key of the client and the name of the file to download. The super-peer agent will send the file to the client.

6.2.14 Rendezvous File message

The `rendezvous file` message is sent by a client agent to a remote super-peer agent to send a file that was requested for rendezvous from one of the clients of the super-peer. The client that sends the message is the client who owns the resource while the super-peer is the access point of the client who asked for the resource. The message must contain the key of the resource owner client, the key of the client who asked for the rendezvous and the resource file. The super-peer agent will check if there is space in the rendezvous directory of the client and if there is he will store the file.

Message	Packet
From client agent to super peer agent	
NewClient	Address of the client agent
Connect	Address of the client agent, Key of the client
Disconnect	Key of the client
New Profile	Key of the client, DOM tree of the profile
Search Profile	**Key**of the client, DOM tree of the search profile
New Resource	Key of the client, DOM tree of the resource
Remove Resource	Key of the client, the name of the resource
Resources	Key of the client, a list with the DOM tree of each of the resource of the clie **nt**
Request Other Client Address	Key of the client, Key of the desired client
Request Access Point List	Key of the client
Arrange Rendezvous	Key of the client, Key of the resource owner, the name of the resource
Request Rendezvous Condition	Key of the c lient
Request Rendezvous File	Key of the client, the name of the resource
Rendezvous File	Key of the client, the resource file
Finger Client	Key of the client, Key of the desired client
From client agent to are you alive messenger	
I am alive	The en vironment address of the client agent

Table 6.3: Client Peer - Super-Peer Communication Protocol messages

6.2.15 Finger Client message

The `finger client` message is sent by a client agent to a remote super-peer agent to finger another client based on the key. The message must contain the key of the client who sends the message and the key of the client to finger. The super-peer agent will check his client information data structure, and if the client is connected, he will send a `finger client connected` message or else, he will send a `finger client not connected` message to the client who asked for the finger process.

6.2.16 I Am Alive message

The `I am alive` message is sent from a client agent to a remote Are-You-Alive messenger as a reply to the message `are you alive` that the remote Are-You-Alive messenger sent. The message must contain the address of the client agent that replies to the are you alive message. The remote Are-You-Alive messenger will understand that the client agent is still alive.

6.3 Super-peer - Client Peer Communication Protocol

6.3.1 Key message

The `key` message is sent by a super-peer agent to a client agent to give him the key of the client. The message is sent as a reply to the `new client` message from the client agent. The packet of the message must contain the key. The client agent that receives that message must update the XML file of the client to conclude the key. Note that this scenario takes place only once for each client. After the first connection, the client will use that key to identify himself to the network.

6.3.2 Notification message

The `notification` message is sent by a super-peer agent to a client agent to give him a notification. The message must contain the DOM tree of the notification. The client agent will store the notification as a XML file to the notification directory of the client. Then he will inform the interface agent with a `interface notification` message.

6.3.3 Stored Notification message

The `stored notification` message is sent by a super-peer agent to a client agent. This message is sent when the client connects and there are stored notifications in the notification directory of the client in the super-peer environment. Those notifications arrived in the super-peer environment a time that the client was no connected. The super-peer agent sends this message as a reply to the connect message which was sent by the client agent. He checked the key of the client and realized that there are stored notifications. He sends the notifications with the stored notification message and then the notifications are removed from the directory. The notifications are stored to hard disk as XML files so the message contains the XML file of the notification. The super-peer agent could parse the XML file and send the DOM tree but this would mean extra work for the super-peer agent which is not necessary. The client agent can parse the XML file when it is needed. The super-peer agent knows the correct address of the client because the address sent along with the connect message from the client agent.

6.3.4 Notification Number message

The `notification number` message is sent from a super-peer agent to a client agent to inform him on the number of the notifications that are sent. This message is necessary because the notifications are sent one by one and there is no

way to guarantee that all notifications will arrive in the client peer environment. The message must contain only the number of the notifications. The client agent that accepts such a message checks if he has received the correct number of notifications and sends to the interface agent a `stored notifications number` message.

6.3.5 Requested Address message

The `requested address` message is sent by a super-peer agent to a client agent to deliver an address of a remote client. This message is sent as a reply to the "request other client address" message which is sent by the client agent when the user wants to download a resource from a notification and he needs the address of the remote client. The super-peer agent will reply with a requested address message if the requested client exists and he is online. There is no difference if the requested client is one of the clients that this super-peer serves or not. The message must contain the environment address of the client agent of the remote client.

6.3.6 Requested Address Not Available message

The `requested address not available` message is sent by a super-peer agent to a remote client agent as a reply to a `request other client address` message. The super-peer agent found out that either the desired client does not exist(there is not a client in the network with the desired key) or the client is disconnected at the moment which means that there is no way to communicate with him now and his address in the client information is null. The message will not contain any objects.

6.3.7 Access Point List message

The `access point list` message is sent by a super-peer agent to a remote client agent as a reply to a `request access pint list` message, which was sent by the client agent. The super-peer agent will build a list with addresses of all the super-peer environment in the DIET universe and send it to the client. The message will contain only the list with the addresses. The client agent that receives such a message will update the log XML file on the local directory of the client(in the client peer environment) to conclude the addresses of the super-peer environment. If there were any super-peer environment addresses from a previous access point list message, they will be erased because it is possible that one of the super-peers may have failed. The next time that the user will choose to assign a new access point from the list the GUI will see the super-peer environment addresses from the XML file.

6.3.8 Rendezvous Notification message

The `rendezvous notification` message is sent by a super-peer agent to a client agent to deliver a rendezvous notification. This message is sent as a reply to the `request rendezvous condition` message, which was sent by the client agent. The super-peer agent sends this message only if there is a notification in the rendezvous directory of the client in the super-peer environment. The notifications are stored to hard disk as XML files so the message contains the XML file of the notification. The super-peer agent could parse the XML file and send the DOM tree but this would mean extra work for the super-peer agent, which is not necessary. When a client agent receives a rendezvous notification message stores the XML file of the notification to rendezvous notification directory of the client in the client peer environment. Then the client agent will wait for the `rendezvous notification number` message from the super-peer agent. The client agent can parse the XML file when it is needed.

6.3.9 Rendezvous Notification Number message

The `rendezvous notification number` message is sent from a super-peer agent to a client agent to inform him on the number of the rendezvous notifications that are sent. This message is necessary because the notifications are sent one by one and there is no way to guarantee that all notifications will arrive in the client peer environment. The message must contain only the number of the notifications. The client agent that accepts such a message checks if he has received the correct number of rendezvous notifications and sends to the interface agent a `rendezvous notifications number` message.

6.3.10 Send File To Server message

The `send file to server` message is sent from a super-peer agent to a remote client agent as a command. The client agent must send a file to a remote super-peer agent(not the one that sent him the message). The super-peer agent sends this message when the client agent connects to the network and there are rendezvous obligations for the client which means that another client has asked for a rendezvous with one of the resources of this client. The super-peer agent will send a `send file to Server` message to inform the client agent that he must send the resource to a specific super-peer agent for a specific client that asked for the resource. The message must contain the name of the resource, the super-peer environment address of the receiver and the key of the client, who asked for the resource. The super-peer agent will find all these information in the Rendezvous XML file of the client which is checked every time a client connects. to the network(not necessarily to the same super-peer). When the client agent receives this message will immediately send the file to the desired super-peer with

Message	Packet
From super peer agent to client agent	
Key	The key of the client
Notification	The DOM tree of the notification
Stored Notification	The DOM tree of the notification
Notification Number	The number of the stored notifications
Requested A ddress	The environment address of the requested client
Requested Address Not Available	The key of the requested client
Access Point List	The list with environment addresses of the access points
Rendezvous Notification	The DOM tree of the notification
Rendezvous Notification Number	The number of the rendezvous notifications
Send File To Server	The key of the client who asked the file, the name of the file, the environment address of the super peer to send the file
Finger Client Connected	The key of th e client, the environment address of the client agent
Finger Client Not Connected	Key of the client
Fromare you alive messenger to client agent	
Are You Alive	The environment address of the super peer where the are you alive messenger inhabits

Table 6.4: Super-Peer - Client Peer Communication Protocol messages

a `rendezvous file` message. This process is abstract to the real user.

6.3.11 Finger Client Connected message

The `finger client connected` message is sent by a super-peer agent to a client agent to deliver an address of a remote client. This message is sent as a reply to the `finger client` message, which is sent by the client agent when the user to finger a remote client. The super-peer agent will reply with a finger client connected message, if the requested client exists and he is online. There is no difference if the requested client is one of the clients that this super-peer serves or not. The message must contain the environment address of the client agent of the remote client. The client agent will send to the interface agent a `finger client is online` message so the real user will understand that the client is online.

6.3.12 Finger Client Not Connected message

The `finger client not connected` message is sent by a super-peer agent to a client agent. This message is sent as a reply to the `finger client` message, which

is sent by the client agent when the user to finger a remote client. The super-peer agent will reply with a finger client not connected message, if the requested client not exists or he is not online. There is no difference if the requested client is one of the clients that this super-peer serves or not. The message will contain the key of the desired client. The client agent will send to the interface agent a `finger client is not online` message so the real user will understand that the client is not online.

6.3.13 Are You Alive message

The `are you alive` message is sent by an Are-You-Alive messenger to a client agent. The purpose of the message is to check the client agent, if he is alive. The message must contain the super-peer environment address of the Are-You-Alive messenger so the client agent can reply. When a client agent receives an are you alive message, immediately replies with an `I am alive` message.

6.4 Client Peer - Client Peer Communication Protocol

6.4.1 Request Resource message

The `request resource` message is sent by a client agent to a remote client agent as a request for a download. The message is sent when the client agent receives a `download interface` message from the interface agent and has found through the access point the environment address of the client agent of the resource owner. The message must contain the environment address of the client that wants to download and the file name. The two client agents communicate directly to each other. The client agent that receives a request resource message will check if the resource with the desired name exists and if it does, he will reply with a `send resource` message to the remote client agent. If the resource does not exist he will reply with a `resource does not exist` message. All this process is abstract to the real user in the resource owner client node.

6.4.2 Send Resource message

The send resource message is sent by a client agent A to a remote client agent B as a reply to a `request resource` message that the client agent B sent to the client agent A. The message is sent only if the requested resource exists and must contain the resource file. The client agent that receives a send resource message will store the resource file to the download directory in the local client peer environment, using the same name. Then, when the download is complete,

he will send a `interface download complete` message to the interface agent, so the real user will understand that the resource is downloaded.

6.4.3 Resource Does Not Exist message

The `resource does not exist` message is sent by a client agent A to a remote client agent B as a reply to a `request resource` message that the client agent B sent to the client agent A. The message is sent only if the requested resource does not exists. Note that there is no possibility that the name in the `request resource` message was wrong because the name was produced by a notification that the client agent B had received by his access point super-peer agent. This means that the only possibility is that the client A has unpublished the resource after a user request. The message will not contain the name of the desired resource. The client agent that receives a resource does not exist message will send a `interface file does not exist` message to the local interface agent, so the real user will be informed that the resource is unpublished by the resource owner.

6.4.4 Chat Request message

The `chat request` message is sent by a client agent A to a remote client agent B as a request for chat. The message is fired by the `chat interface` message received by the local interface agent and after the client has verified through the access point super-peer agent that the desired client exists and is online at the moment. The message must contain the environment address of the client. The client agent that receives a chat message will check if the user is already chatting with another client. In that case he will reply to the client agent with a `chat request denied` message. If the user is not in a chat mode the request is accepted and the client agent will reply with a `chat request accepted` message. Moreover, he will send a `interface connected chat` message to the local interface agent, so the real user will know that he is in a chat mode with another user.

6.4.5 Chat Request Accepted message

The `chat request accepted` message is sent by a client agent A to a remote client agent B as a reply to the `chat request` message that the client agent B sent. The message is sent only if the request is accepted and must contain the environment address of the client agent. The client agent that receives a chat request accepted message will send to the local interface agent a `interface connected chat` message, so the real user will know that he is in a chat mode with another user.

6.4.6 Chat Request Denied message

The `chat request denied` message is sent by a client agent A to a remote client agent B as a reply to the `chat request` message that the client agent B sent. The message is sent only if the request is denied and must contain the environment address of the client agent. The request is denied only if the client is already in a chat mode with another client. The client agent that receives a chat request denied message will send to the local interface agent a `interface connected chat failed` message, so the real user will know that his request for chat was denied by the remote client.

6.4.7 Chat Message message

The `chat message` message is sent by a client agent to a remote client agent when the two clients are in a chat mode. The message is fired by the interface agent that sends a `chat message interface` message to the client agent. The message must contain the environment address of the client agent and the string of the chat message. When a client agent receives a `chat message` message will forward the message to the local interface agent with an `interface chat message` message, so the real user may see the message that the remote user sent.

6.4.8 Chat Exit message

The `chat exit` message is sent by a client agent A to a remote client agent B when the user of the client agent A wants to exit chat mode. The message is fired when the local interface agent sends a `chat disconnect interface` message or a `chat exit interface` message. The message must contain the environment address of the client agent. When a client agent receives a `chat exit` will exit from chat mode and send to local interface agent a `interface chat disconnected` message, so the real user will understand that the remote user has quit chatting with him.

6.4.9 Messenger Send message

The `messenger send` message is sent by the client agent to a messenger in the messenger pool of the same client peer environment to inform him that he must migrate to a remote environment and deliver a message to an agent. The message will contain the address of the remote environment, the family tag of the receiver agent and of course the message to deliver. The messenger will migrate to the remote environment and try to deliver the message to the target agent. If there is no messenger in the pool the client agent or the Are-You-Alive messenger will create a new messenger to deliver the message.

Message	Packet
From client agent to remote client agent	
Request Resource	Environment address of the client agent, the name of the resource
Send Resource	The resource file
Resource Does Not Exist	The resource name
Chat Request	Environment address of th e client agent, the key of the client
Chat Request Accepted	Environment address of the client agent, the key of the client
Chat Request Denied	The key of the client
Chat Message	The message string
Chat Exit	The key of the client
From client agent t o local messenger	
Messenger Send	The message, the environment address to migrate and the target agent
Destroy Yourself	Nothing

Table 6.5: Client Peer-Client Peer Communication Protocol messages (1)

6.4.10 Destroy Yourself message

The `destroy yourself` message is sent by the client agent to a messenger of the same client peer environment to inform him that he must commit suicide, because there is no room in the messenger pool in this environment. This happens, when a messenger arrives in the client peer environment, deliver the message and the client agent realizes that there are too many messengers in the pool. When the messenger receives the destroy yourself message will give up his thread and die. This is a procedure for extra protection on the danger of running a client peer out of resources.

6.4.11 Connect Interface message

The `connect interface` message is sent by an interface agent to the local client agent as a command to connect to the P2P-DIET network. The message is fired by a change in the GUI when the user enables the connect choice and will not contain any objects. The client agent that receives such a message will try to connect to the super-peer agent in the access point that is already assigned. The client agent always uses the super-peer environment address that is in the XML log file of the client. This address changes only if the real user wants to assign a new access point. The client agent will send a `connect` message to the remote super-peer agent.

6.4.12 Disconnect Interface message

The `disconnect interface` message is sent by an interface agent to the local client agent as a command to disconnect from the P2P-DIET network. The message is fired by a change in the GUI when the user enables the disconnect choice and will not contain any objects. The client agent that receives such a message will try to disconnect by sending a `disconnect` message to the super-peer agent in the remote super-peer environment of the access point.

6.4.13 Exit Interface message

The `exit interface` message is sent by an interface agent to the local client agent as a command to exit the client application. The message is fired by a change in the GUI when the user enables the exit choice and will not contain any objects. When the client agent receives an `exit interface` message will check if he is connected to the network and if he is he will send a `disconnect` message to the super-peer agent in the remote super-peer environment of the access point. Then the client agent will exit.

6.4.14 Send Profile Interface message

The `send profile interface` message is sent by an interface agent to the local client agent as a command to subscribe a profile. The message is fired by a change in the GUI when the user enables the subscribe profile choice and will contain the DOM tree of the profile. When the client agent receives a send profile interface message will send to the super-peer agent in the remote super-peer environment of the access point a `new profile` message.

6.4.15 Publish Resource Interface message

The `publish resource interface` message is sent by an interface agent to the local client agent as a command to publish a resource. The message is fired by a change in the GUI when the user enables the publish resource choice and will contain the DOM tree of the metadata of the resource. When the client agent receives a publish resource interface message will send to the super-peer agent in the remote super-peer environment of the access point a `new resource` message.

6.4.16 Remove Resource Interface message

The `remove resource interface` message is sent by an interface agent to the local client agent as a command to remove a resource. The message is fired by a change in the GUI when the user enables the remove resource choice and will contain the name of the resource. When the client agent receives a remove

resource interface message will send to the super-peer agent in the remote super-peer environment of the access point a `remove resource` message and then will remove from the Resources directory the resource and the XML file with the metadata.

6.4.17 Assign Access Point Interface message

The `assign access point interface` message is sent by an interface agent to the local client agent as a notification on the current access point super-peer environment address. The message is fired by a change in GUI when the real user enables the choice `assign access point` or `assign access point from list`. The packet of the message must contain the environment address of the new super-peer. When a client agent accepts an `assign access point interface` message will update the variable access point which shows the current access point environment address. Moreover, the client agent will update the access point element in the client XML log file with the current value. The client agent will not try to connect to the super-peer agent until the interface agent sends a `connect` message. This means that the access point environment address may change several times while the client is disconnected from the network but the last value will be used as an access point environment address.

6.4.18 Request Access Point List Interface message

The `request access point list interface` message is sent by an interface agent to the local client agent as a command to request the current access point list from the super-peer agent in the super-peer environment of the access point. The message is fired by a change in GUI when the real user enables the choice `request access point list`. The packet of the message will not contain any objects. When the client agent receives the request access point list interface message will send a `request access point list` message to the super-peer agent in the super-peer environment of the access point.

6.4.19 Request Rendezvous Condition Interface message

The `request rendezvous condition interface` message is sent by an interface agent to the local client agent as a command to request the condition of the requested rendezvous from the access point. The message is fired by a change in GUI when the real user enables the choice `request rendezvous condition`. The packet of the message will not contain any objects. When the client agent receives such a message will send a `request rendezvous condition` message to the super-peer agent in the super-peer environment of the access point.

6.4.20 Download Rendezvous File Interface message

The `download rendezvous file interface` message is sent by an interface agent to the local client agent as a command to download a rendezvous file from the access point. The message is fired by a change in GUI when the real user enables the choice `download rendezvous file`. The packet of the message will contain the name of the desired resource. When the client agent receives such a message will send a `request rendezvous file` message to the super-peer agent in the super-peer environment of the access point.

6.4.21 Finger Interface message

The `finger interface` message is sent by an interface agent to the local client agent as a command to finger another client. The message is fired by a change in GUI when the real user enables the choice `finger`. The packet of the message will contain the key of the desired client. When the client agent receives such a message, he will send a `finger client` message to the super-peer agent in the super-peer environment of the access point.

6.4.22 Download Interface message

The `download interface` message is sent by an interface agent to the local client agent as a command to download a resource. The message is fired by a change in GUI when the real user enables the choice `download notification`. The interface agent parses the notification XML file and finds the key of the resource owner and the file name of the resource. The packet of the message will contain those information. When the client agent receives such a message, he will update the variable resource to download with the file name of the resource and he will send a `request other client address` message to the super-peer agent in the super-peer environment of the access point. The environment address of the desired client agent is not known and because clients use dynamic IP address if the resource owner is connected, he will have a different address than the previous one. This means that the client agent has to be informed if the resource owner is connected to any access point in the network at this time and of course his address, so he sends the request other client address to the super-peer agent. Note that the fact that the resource owner may has changed access point does not affect at all this process because his new access point will have informed the rest of the network on this change.

6.4.23 Start Download Interface message

The `start download interface` message is sent by an interface agent to the local client agent as a command to start downloading a resource. This message

is sent as a reply to the `interface start download question` message sent by the local client agent. We have already described the `download interface` message which fires the download process and that the client agent will send a `request other client address` message to the super-peer agent. If the super-peer agent replies to the client agent with a `requested address` message, which means that the client exists and that he is online, then the client agent will send the `interface start download question` message to the interface agent. The `start download interface` message is a verify from the interface agent (and from the real user, who selects the OK choice and fires the message) to start downloading the resource. The message will contain the name of the resource and the environment address of the recourse owner client agent. The interface agent will have these information from the packet of the `interface start download question` message sent by the local client agent.

6.4.24 Search Interface message

The `search interface` message is sent by an interface agent to the local client agent as a command to query the network using the query. The message is fired by a change in GUI when the real user enables the choice `search`. The packet of the message will contain DOM tree of the query. When the client agent receives such a message will send a `query` message to the super-peer agent in the super-peer environment of the access point. The query DOM tree will not be used again and it will not be saves as an XML file.

6.4.25 Chat Interface message

The chat interface message is sent by an interface agent to the local client agent as a command to chat with a remote client. The message is fired by a change in GUI when the real user enables the choice `chat` from the menu or after the finger selection. The packet of the message will contain key of the desired client. When the client agent receives such a message will send a `request other client address` message to the super-peer agent in the super-peer environment of the access point. In order to chat the remote client must be connected to the network at this time.

6.4.26 Chat Message Interface message

The `chat message interface` message is sent by an interface agent to the local client agent as a command to send a chat message to a remote client. The message is fired by a change in GUI when the real user writes a message in the chat interface. The packet of the message will contain string of the message to send. When the client agent receives such a message will send a `chat message` message to the client agent in the client peer environment of the remote client.

Message	Packet
From interface agent to local client agent	
Connect Interface	Nothing
Disconnect Interface	Nothing
Exit Interface	Nothing
Send Profile Interface	The DOM tree of the profile
Publish Resource Interface	The DOM tree of the resource
Remov e Resource Interface	The name of the resource
Assign Access Point Interface	The environment address
Request Access Point List Interface	Nothing
Request Rendezvous Condition Interface	Nothing
Download Rendezvous File Interface	The name of the resource
Finger Interface	The key of the client
Download Interface	The key of the resource owner, the name of the resource
Start Download Interface	The key of the resource owner, the name of the resource, the environment address of the client
Search Interface	The DOM tree of the search profile
Chat Interface	The key of the desired client
Chat Message Interface	The message string
Chat Disconnect Interface	Nothing
Chat Exit Interface	Nothing
Get File From Chat Client Interface	The name of the resource

Table 6.6: Client Peer-Client Peer Communication Protocol messages (2)

This is the basic message that implements the communication between two clients that are in a chat mode because it transfers their messages.

6.4.27 Chat Disconnect Interface message

The `chat disconnected` interface message is sent by an interface agent to the local client agent as a command to disconnect from the remote client and stop chatting with him. The message is fired by a change in GUI when the real user writes the disconnect command to the chat interface. The packet of the message will not contain any objects. When the client agent receives such a message, he will send a `chat disconnected` message to the client agent in the client peer environment of the remote client and he will update the variable chatting to false.

6.4.28 Chat Exit Interface message

The `chat exit interface` message is sent by an interface agent to the local client agent as a command to exit from chat mode. The message is fired by a change in GUI when the real user writes the exit command to the chat interface. The packet of the message will not contain any objects. When the client agent

receives such a message, he will send a `chat disconnected` message to the client agent in the client peer environment of the remote client and he will update the variable chatting to false.

6.4.29 Get File From Chat Client Interface message

The `get file from chat client interface` is sent by an interface agent to the local client agent as a command to download a resource from the remote client agent with whom the local client is chatting. The message is fired by a change in GUI when the real user writes the get command followed by the name of the resource to the chat interface. The packet of the message will contain the resource file name. When the client agent receives this message, he will send a `request resource message` to the client agent in the client peer environment of the remote client. The rest of the download process does not change which means that the remote client agent will reply with the messages from the standard download process.

6.4.30 Interface Notification message

The `interface notification` message is sent by a client agent to the local interface agent to inform him that a new notification has arrived in the local client peer environment. The message is fired by the `notification` message which was sent by the super-peer agent in the super-peer environment of the access point. The packet of the message will contain the name and the size of the resource. When the interface agent receives the interface notification message will fire a change to the GUI to inform the user on the new notification.

6.4.31 Interface Stored Notifications Number message

The `interface stored notifications number` message is sent by a client agent to the local interface agent to inform him on the number of the stored notifications downloaded from the access point. The message is fired by the `stored notifications number` message, which was sent by the super-peer agent in the super-peer environment of the access point. The message will contain the number of the stored notifications. The interface agent that receives this message will fire a change to the interface to show the client this information.

6.4.32 Interface Start Download Question message

The `interface start download question` is sent by a client agent to the local interface agent. This message is a part of the download process and is fired by the `requested client address` message, which was sent by the super-peer agent in the super-peer environment of the access point. This means that the

resource owner is online and now the user must verify that he wants to download the resource. The message must contain the environment address of the resource owner and the resource name. The interface agent that receives such a message will fire a change in the GUI by showing the dialog to choose between starting the download or cancel it.

6.4.33 Interface Start Download Not Possible message

The `interface start download not possible` message is sent by a client agent to a local interface agent to inform him that is is not possible to download a resource. This is due to the fact that the resource owner client is not online at the moment. The message is fired by the `requested address not available` message, which was sent by the super-peer agent in the super-peer environment of the access point. This means that the resource owner is not online and that it is not possible to continue the download process. The message will contain the key of the resource owner and the name of the resource. When an interface agent receives an `interface start download not possible` message will fire a change in the GUI to inform the user and will show a dialog for the user to choose if he wants to arrange a rendezvous with this file or not.

6.4.34 Interface Download Complete message

The `interface download complete` message is sent by a client agent to the local interface agent to inform him that the download of a resource is completed. This message is the last part of the download process. The message is fired by the `send resource` message, when the resource file is received. The packet of the message will contain the name of the resource. When an interface agent receives such a message will fire a change in the GUI to inform the user.

6.4.35 Interface File Does Not Exists message

The `interface file does not exist` message is sent by a client agent to the local interface agent to inform him that the requested file does not exist. This can happen when the resource owner has unpublished the resource from the network. The message is fired by the `resource does not exists` message, which was sent to the client agent by the client agent of the resource owner client. The message must contain the name of the resource. The interface agent that receives the message will fire a change to the GUI so the user will understand that the file does not exist. In such a case nothing can be done. If the resource owner chooses to republish the resource in the future and this client has not changed his profile, then he will receive again a notification.

6.4.36 Interface Rendezvous Notification Number message

The `interface rendezvous notification number` message is sent by a client agent to the local interface agent to inform him on the number of the rendezvous that are uploaded to the access point. The message is fired by the `rendezvous notification number` message, which was sent by the super-peer agent in the super-peer environment of the access point. The message will contain the number of the stored notifications. The interface agent that receives this message will fire a change to the interface to show the client this information.

6.4.37 Interface Connected Chat message

The `interface connected chat` message is sent by a client agent to the local interface agent to inform him that he is in chat mode with a remote client. The message is fired by the `chat request accepted` message, which was sent to the client agent by the remote client agent with whom the user wants to chat. The packet of the message will contain the key of the remote client. When the interface agent receives this message will print a message to the chat interface to inform the client that the remote user has accepted his request and that he can start chatting.

6.4.38 Interface Connected Chat Failed message

The `interface connected chat` message is sent by a client agent to the local interface agent to inform him that his request for chat failed. This can happen due to the fact that the remote client is not online at the moment or because he is already chatting with another client, so he did not accept the request for chat. The message is fired by the `chat request not accepted` message, which was sent to the client agent by the remote client agent with whom the user wants to chat or by the `requested address not available` message , which was sent by the super-peer agent in the super-peer environment of the access point. The packet of the message will contain the key of the remote client. When the interface agent receives this message will print a message to the chat interface to inform the client that the chat request failed.

6.4.39 Interface Chat Disconnected message

The `interface chat disconnected` message is sent by a client agent to the local interface agent to inform him that the remote client is not in chat mode anymore. This means that they cannot chat and can happen either because the remote client chooses to disconnect or because the remote client chooses to exit chat mode. The message must contain the key of the remote client. When the

Message	Packet
From client agent to local interface agent	
Interface Notification	The name of the resource, the size of the resource
Interface Stored Notifications Number	The number
Interface Start Download Question	The name of the resource, the addres s of the client agent
Interface Start Download Not Possible	The name of the resource
Interface Download Complete	The name of the resource
Interface File Does Not Exists	The name of the resource
Interface Rendezvous Notification Number	The number
Interface Connected Chat	The of the remote client
Interface Connected Chat Failed	The of the remote client
Interface Chat Disconnected	The of the remote client
Interface Chat Message	The message string
Interface Finger Client Is Online	The of the remote cli ent, the address of the client agent
Interface Finger Client Is Not Online	The of the remote client

Table 6.7: Client Peer-Client Peer Communication Protocol messages (3)

interface agent receives this message, he will print the appropriate message to the chat interface so the user will understand that he is not chatting anymore.

6.4.40 Interface Chat Message message

The `interface chat` message is sent by a client agent to the local interface agent to inform him that a new chat message has arrived. The message is fired by the `chat message` message, which was sent to the client agent by the remote client agent with whom the user is in chat mode. The packet of the message must contain the string of the chat message. When an interface agent receives a `chat message` message, he will print the message to the chat interface so the user will see it.

6.4.41 Interface Finger Client Is Online message

The `interface finger client is online` message is sent by a client agent to the local interface agent to inform him that the requested client is online. The message is fired by the `finger client connected` message, which was sent by the super-peer agent to the remote super-peer environment of the access point. The message must contain the key of the remote client and his environment address. The interface agent that receives such a message must fire a change to GUI to inform the user on the fact that the client is online and show a dialog to

give him the choice to chat with the remote user.

6.4.42 Interface Finger Client Is Not Online message

The `interface finger client is not online` message is sent by a client agent to the local interface agent to inform him that the requested client is not online. The message is fired by the `finger client not connected` message, which was sent by the super-peer agent to the remote super-peer environment of the access point. The message must contain the key of the remote client. The interface agent that receives such a message must fire a change to GUI to inform the user on the fact that the client is not online.

6.5 Summary

In this chapter we described in detail the different protocols that the agents in the P2P-DIET universe use to communicate. We showed the header of each message, the scenario that the message is used for and how the receiver agent must react and probably reply. In the next Appendix A we give extensive examples of the different scenarios, that take place in the system by describing the conversations between the agents.

Chapter 7

Concluding Remarks

Let us now summarize what we have done in this dissertation. In this work we dealt with the problem of designing and implementing an agent-based query and event notification service in the peer-to-peer context. The main goal was to build a system to work in the real world, the internet, so we had to focus on scalability, fault-tolerance and location independent addressing. On top of the service we implemented a file sharing application, which gives maximum flexibility to its users, since it supports both the query and the dissemination scenario.

Initially we surveyed the area of distributed systems. The process of designing a distributed, peer-to-peer, event notification service requires careful study of the different peer-to-peer architectures and their tradeoffs. We discussed some alternative architectures for peer-to-peer systems. We briefly presented some well-known architectures and informally discussed their advantages and disadvantages, in terms of fault-tolerance, routing, scalability and other fundamental network issues. We considered four basic architectures: hierarchical, pure peer-to-peer, centralized peer-to-peer and super-peer.

Then, we presented in detail the architecture chosen for P2P-DIET. Moreover, we presented the agents that were implemented, their responsibilities, their goals and how they interact with each other and with the users of the network. In P2P-DIET there are many worlds. Each super-peer and each client has a different world, which means that each different computer represents a different world. All the worlds together are the P2P-DIET universe. Each world has one environment. The world in the super-peer nodes has one *super-peer environment*, where 5 different types of agents inhabit. One *super-peer agent*, one *Are-You-Alive messenger*, one *clock agent*, one *build spanning tree scheduler* and the *messenger pool*, which contains zero or more *messenger agents*. The world in the client peer nodes has one *client peer environment*, where 3 different types of agents inhabit. One *client agent*, one *interface agent*, and the *messenger pool*, which contains zero or more *messenger agents*.

We described the language, that the clients use to query the system. The metadata that are currently supported are: File name, File type, File size, Title,

Author and Desired clients. Note that the language is totally abstract to the system. This means, that a new language can be easily supported possibly to implement a different application scenario or to extend the current language.

Then, we dealt with the problem of routing messages in the peer-to-peer context of our service. Since our system is a super-peer network and the topology of the servers forms a general undirected graph, we have to deal with the routing problem, because as we discussed in Chapter 2, routing has no trivial solution in a general graph. We presented the network demands and we considered some basic solutions. We explained how we use minimum weight spanning trees and shortest paths to satisfy the demands of our network and how we build them. We discussed about clients, the real users of the network, and presented the way that the system handles the problem of dynamic IP addresses and the problem that a client might be disconnected. We continued by discussing the fault-tolerance problem and its solutions. We discussed the problem of socket handling, where we described why we try to keep open the connections in many cases and how we do that.

We presented the event notification service in detail. We gave details for the basic concepts of our event service such as profiles, resources, notifications, stored notifications, queries, rendezvous. We described the propagating strategy, which concludes a profile hierarchy and we showed how we build and use the hierarchy and why it is so useful for the network, since it allows the super-peer agents to forward one notification that interests many clients.

We described in detail the protocols that agents use in P2P-DIET universe to communicate. There are four different protocols:

1. The *super-peer - super-peer* communication protocol, which is used by agents in a super-peer environment to communicate with agents in the local or a remote super-peer environment.

2. The *client peer - super-peer* communication protocol, which is used by agents in a client peer environment to communicate with agents in a remote super-peer environment.

3. The *super-peer - client peer* communication protocol, which is used by agents in a super-peer environment to communicate with agents in a remote client peer environment.

4. The *client peer - client peer* communication protocol, which is used by agents in a client peer environment to communicate with agents in the local or a remote client peer environment.

Note that remote communication is achieved by the messenger agent. This means that when an agent wants to communicate with a remote agent, it communicates with a local messenger agent and the messenger will implement the remote communication.

Bibliography

[1] A. Carzaniga. *Architectures for an Event Notification Service Scalable to a Wide Area Networks*. Phd thesis, Politecnico di Milano, Italy, 1998.

[2] A. Carzaniga and D.S. Rosenblum and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'2000)*, pages 219–227, 2000.

[3] A. Carzaniga and D.S. Rosenblum and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[4] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 28th International Conference on Distributed Systems*, July 2002.

[5] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale- peer-to-peer storage utility. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Paltforms (Middleware 2001)*, November 2001.

[6] A. Rowstron and P. Druschel. Storage managment and caching in past, a large scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[7] B. Kantor and P. Lapsley. Network news transfer protocol. *RFC 977*, February 1986.

[8] B. Mukherjee and L. Heberlein and K. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, May 1994.

[9] B. Yang and H. Carcia-Molina. Designing a super-peer network. In *Proceedings ICDE*, 2003.

[10] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proceedings ICDE of the 27th International Conference on Very Large Databases*, September 2001.

[11] B. Yang and H. Garcia-Mollina. Improving efficiency of peer-to-peer search. In *Proceedings of the 28th International Conference on Distributed Systems*, July 2002.

[12] B. Zhao and J. Kubiatowicz and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide area location and routing. Technical report ucb/scd-0101141, Computer Science Division, U.C.Berkley, April 2001. Available at http://www.cs.berkley.edu/ravenben/publications.CSD-01-1141.pdf.

[13] Bearshare Home page. http://www.bearshare.com.

[14] C. Hoile and F. Wang and E. Bonsma and P. Marrow. Core specification and experiments in diet: a decentralised ecosystem-inspired mobile agent system. In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002)*, pages 623–630, July 15–19 2002.

[15] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[16] DIET Home page. http://www.dfki.de/diet.

[17] D.S. Milojisic and V. Kalogeraki and R. Lukose and K. Nagaraja and J. Pruyne and B. Richard and S. Rollins and Z. Xu. Peer-to-peer computing. Technical Report HP-2002-57, HP Laboratories, Palo Alto, 2001.

[18] E. Bonsma and C. Hoile. A distributed implementation of the swan look-up system using mobile agents. In *Proceedings of the AAMAS 2002 Workshop on Agents and Peer-to-Peer Computing*, 2002. Forthcoming volume in Lecture Notes in Computer Science, G. Moro and M. Koubarakis (eds.).

[19] E.A. Kendall and C.V. Pathak and P.V.M. Krishna and C.B. Suresh. The layered agent pattern language. In *Proceedings of the Conference on Pattern Languages of Programms (PLoP'97)*, 1997.

[20] E.A. Kendall and P.V.M. Krishna and C.V. Pathak and C.B. Suresh. Patterns of intelligent and mobile agents. pages 92–99.

[21] European Commission IST Future and Emerging Tecnologies. Universal information ecosystems proactive initiative, 1999.

[22] F. Bellifemine and A. Poggi and G. Rimassa. Jade — a fipa-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, London, UK, 1999. The Practical Application Company Ltd. Available at http://sharon.cselt.it/projects/jade/PAAM.pdf.

[23] F. Dabek and M.F. Kaashoek and D. Karger and R. Morris and I. Stoica. Wide area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[24] FIPA Home page. http://www.fipa.org.

[25] Freenet website. http://freenet.sourceforge.net.

[26] G. Gugola and E. DI Nitto and A. Fuggeta. Exploiting an event based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering ICSE 98*, Kyoto, Japan, April 1998.

[27] G. Vigna and R. Kemmerer. Netstat: A network based intrusion detection approach. In *Proceedings of the 14th Annual Computer Security Application Conference*, Scottsdale AZ, USA, December 1998.

[28] Gnutella website. http://gnutella.wego.com.

[29] H.S. Nwana and D. TNdumu. A perspective on software agents. *The Knowledge Engineering Review*, 14(2):1–18, 1999.

[30] I. Clark. A distributed decentralized information storage and retrieval system. Division of Informatics, Universiry of Edinburg, 1999.

[31] I. Clark and O. Miller and T.T. Hong. Freenet: A destributed anonymous information retrieval system. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity an Unobservability, LNCS 2009*, New York. Springer.

[32] I. Stoica and R. Morris and D. Karger and M.F. Kaashoek and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[33] ICQ Home Page. http://www.icq.com.

[34] J. Kubiatowicz and D. Bindel and Y. Chen and S. Czerwinski and P. Eaton and D. Geels and R. Gummadi and S. Thea and H. Weather-Spoon and W. Weimer and C. Wells and B. Zhao. Oceanstore: An architecture for global scale persistent storage. In *Proceedings ASPLOS*, pages 190–201, November 2000.

[35] J.E. Cook and A.L. Wolf. Discovering models of software processes from event based data. *ACM Transactions on software Engineering and Methodology*, 7(3):191–230, July 1998.

[36] J.M. Bradshaw. *Software Agents*. AAAI Press, Mento Prak, USA, 1997.

[37] K. Ilgun and R. Kemmerer and P. Porras. Stare transition analysis: A rule based intrusion detection system. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[38] K. Sycara and A. Pannu and M. Williamson and D. Zeng. Distributed intelligent agents.

[39] K. Truelove. To the bandwidth barrier and beyond. Originally an online document available through DSS Clip2, now unavailable because the company has gone out of business. Please contact the primary author of this paper for a copy of the document.

[40] KazaA Home Page. http://www.kazaa.com.

[41] Konspire Home Page. http://konspire.sourceforge.com.

[42] Limewire Home page. http://www.limewire.com.

[43] L.R. Ford and D.R Fukkerson. *Flows in Networks*. Princeton University Press, 1963.

[44] M. Koubarakis and T. Koutris and P. Raftopoulou and C. Tryfonopoulos. Information alert in distributed digital libraries: The models, languages and architecture of dias. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2002)*, volume 2458 of *Lecture Notes in Computer Science*, pages 527–542, September 2002.

[45] M.R. Cagan. The hp softbench environment: An architecture for a new generation of software tools. *Hewlett Packard Journal: Technical Information from the Laboratories of Hewlett Packard Company*, 41(3):36–67, June 1990.

[46] M.T. Rose. *The Simple Book*. Prentice Hall, Englewood Cliffs, 5th edition edition, 1991.

[47] N. Daswani and H. Garcia-Molina and B. Yang. Open problems in data sharing peer-to-peer systems. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *Lecture Notes in Computer Science*, pages 1–15. Springer, January 2003.

[48] N. Jennings and M. Wooldridge. Software agents. *IEE Review*, pages 17–20, 1996.

[49] Napster website. http://www.napster.com.

[50] N.S. Bargouti and B. Krishnamurthy. Using event contexts and matching constraints to monitor software processes enginnering. *IEEE Computer Society*, May 1995.

[51] OpenNap Home Page. http://opennap.sourceforge.com.

[52] OSF. *OSF Motif Programmers Quide*. Prentice Hall, Englewood Cliffs, 5th edition edition, 1991.

[53] P. Leigh and P. Benyola. Future developments in peer networking. Technical report, Raymond James and Associates, INC, 2001.

[54] P. Maes. Modeling adaptive autonomous agents. *Artificial Life Journal*, 1, 1994.

[55] P. Marrow and M. Koubarakis and R.H. van Lengen and F. Valverde-Albacete and E. Bonsma and J. Cid-Suerio and A.R. Figueiras-Vidal and A. Gallardo-Antolin and C. Hoile and T. Koutris and H. Molina-Bulla and A. Navia-Vazquez and P. Raftopoulou and N. Skarmeas and C. Tryfonopoulos and F. Wang and C. Xiruhaki. Agents in Decentralised Information Ecosystems: The DIET Approach. In M. Schroeder and K. Stathis, editors, *Proceedings of the AISB'01 Symposium on Information Agents for Electronic Commerce, AISB'01 Convention*, pages 109–117, University of York, United Kingdom, March 2001.

[56] P. Suthar and J. Ozzie. The groove platform architecture. Available at Groove Networks Presentation. devzone.groove.net/library/Presentations/GrooveApplicationArchitecture.ppt.

[57] P.D. O'Brien and M.E. Wiegand. Agents of change in business process management. *Lecture Notes in Artificial Intelligence*, 1198:132–145, 1997.

[58] Pointera Home Page. http://www.pointera.com.

[59] P.R. Pietzuch and J.M. Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, July 2002.

[60] R. Bellmann. *Dynamic Programming*. Princeton University Press, 1957.

[61] R.G. Gallager and P.A. Humblet and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[62] R.O. Hart and G. Lupton. Dec fuse: Building a graphical software development from unix tools. *Digital Technical Journal of Digital Equipment Corporation*, 7(2):5–19, Spring 1995.

[63] R.S. Hall and D. Hembigner and A. van der Hoek and A.L. Wolf. An architecture for post-development configuration management in a wide area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, May 1997.

[64] S. Ratnasamy and P. Francis and M. Handley and R. Karp and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[65] S. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, pages 57–66, July 1990.

[66] S. Russel and P. Norvig. *Arificial Intelligence: A Modern Approach*. Prentice Hall Inc, 1995.

[67] S. Weibel. The dublin core: A simple content description model for electronic resources. *Nfais Newsletter*, July 1998.

[68] S. Weibel and K. Traugott. The dublin core metadata initiative: Mission, current activities, and future directions. *Dlib Magazine*, December 2000.

[69] SETI@home Home Page. http://setiathome.ssl.berkley.edu.

[70] S.R. Hedberg. The first harvest of softbots looks promising. *IEEE Expert*, pages 6–9, August 1995.

[71] T. Barners-Lee. Universal resources identifiers in www, a unifying syntax for the epression of names and addresses of obhects on the network as used in the world wide web. Technical report, RFC 1630, June 1994.

[72] Toadnode Home page. http://www.toadnode.com.

[73] Y.K. Dalal and R.M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, December 1978.

Appendix A

Scenarios

In Chapter 6 we presented the communication protocols. Each agent understands a fraction of those protocols. This depends on the type of his environment and the type of the environment that the target agents inhabit. This chapter presents the possible conversations between agents. Each conversation is a scenario that happens while the system works. By reading a scenario, the reader understands which agents are involved, what protocols and messages use to communicate and exchange information and the time sequence of those messages. Actually this chapter completes Chapter 6 by describing the use of each message and the way that an agent responds to it and handles the situation. In many cases we give more details on how an agent reacts in order for the scenario to be more readable. As we have already described, remote communication is implemented by the messenger agent. In each scenario we use one messenger agent to represent the type of those agents and we do that to save space on the paper and have more readable scenarios. For example, when a messenger migrates to a remote environment to deliver a message, the target agent will need to reply or to send a message to another remote agent. To do so a messenger agent from the pool is used and this means that it is more possible that a different messenger is used than the one that brought the message. When describing the scenarios we use only one messenger to make the scenario more easy to read. Moreover, the scenario hides many details, for example, in the remote communication between super-peer agents, we hide the fact the sender has to find the IP address of the receiver node in the spanning tree of the receiver agent and that he builds the appropriate communication packet to contain the useful information of the message. The word "migrate" represents the fact the messenger has migrated to a remote environment and again hides all the details that a socket was opened or an already opened socket with the same remote computer is used. By describing the scenarios we indent to present the situations that a message is used, the respond of the receiver (probably with another message), the agents and their role in each scenario and the time sequence of the messages.

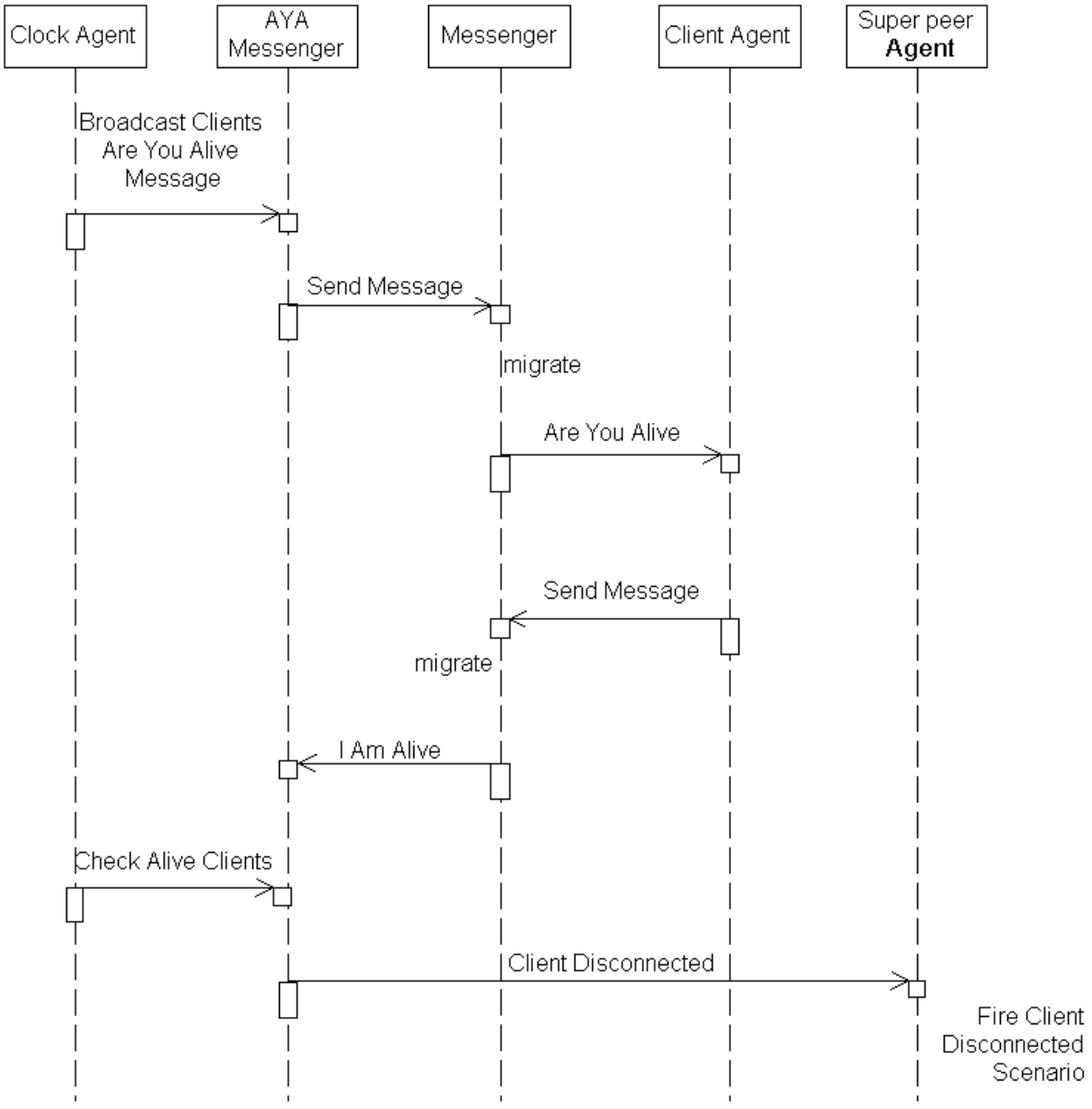

Figure A.1: Are You Alive Client scenario

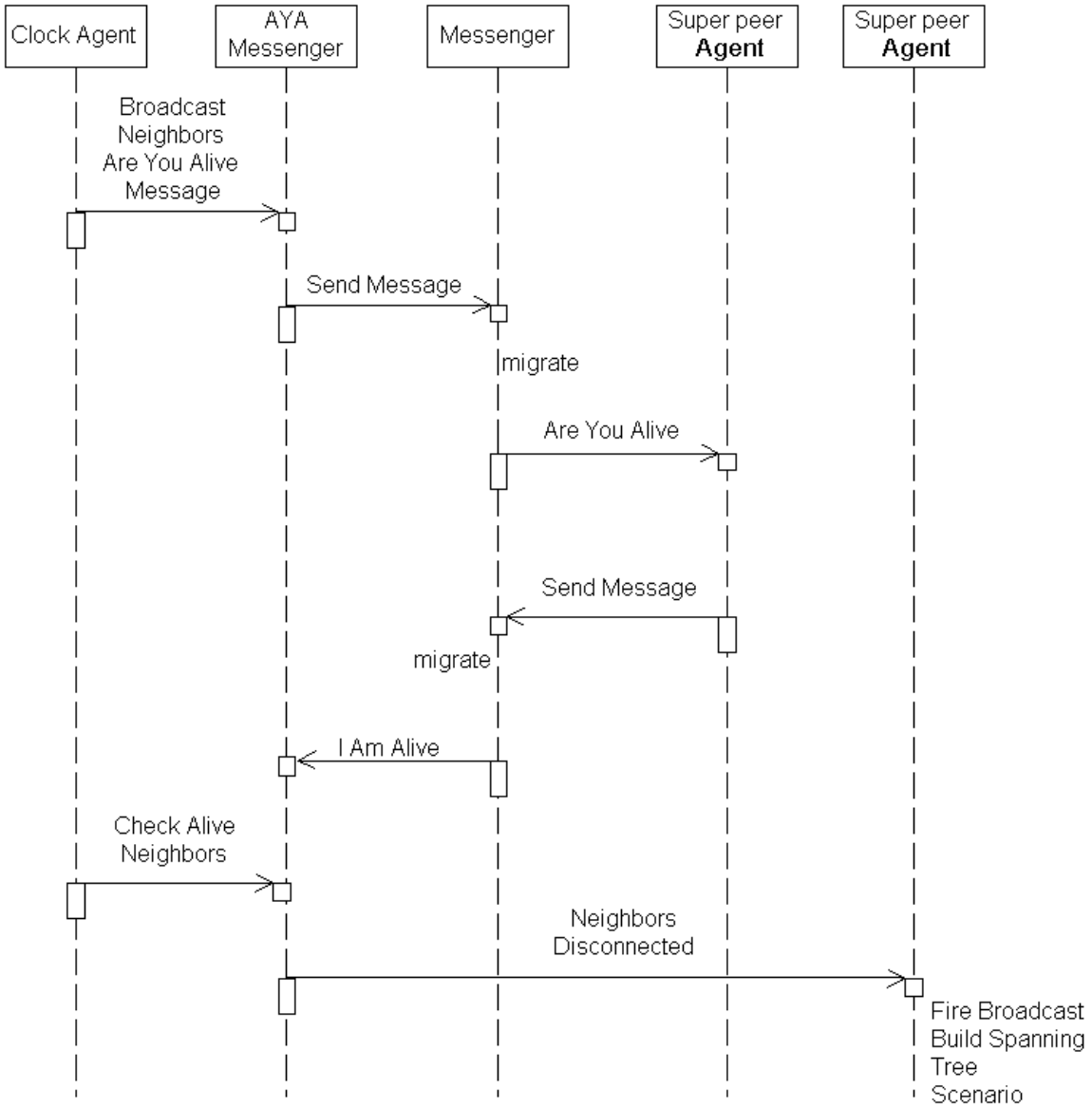

Figure A.2: Are You Alive Neighbor scenario

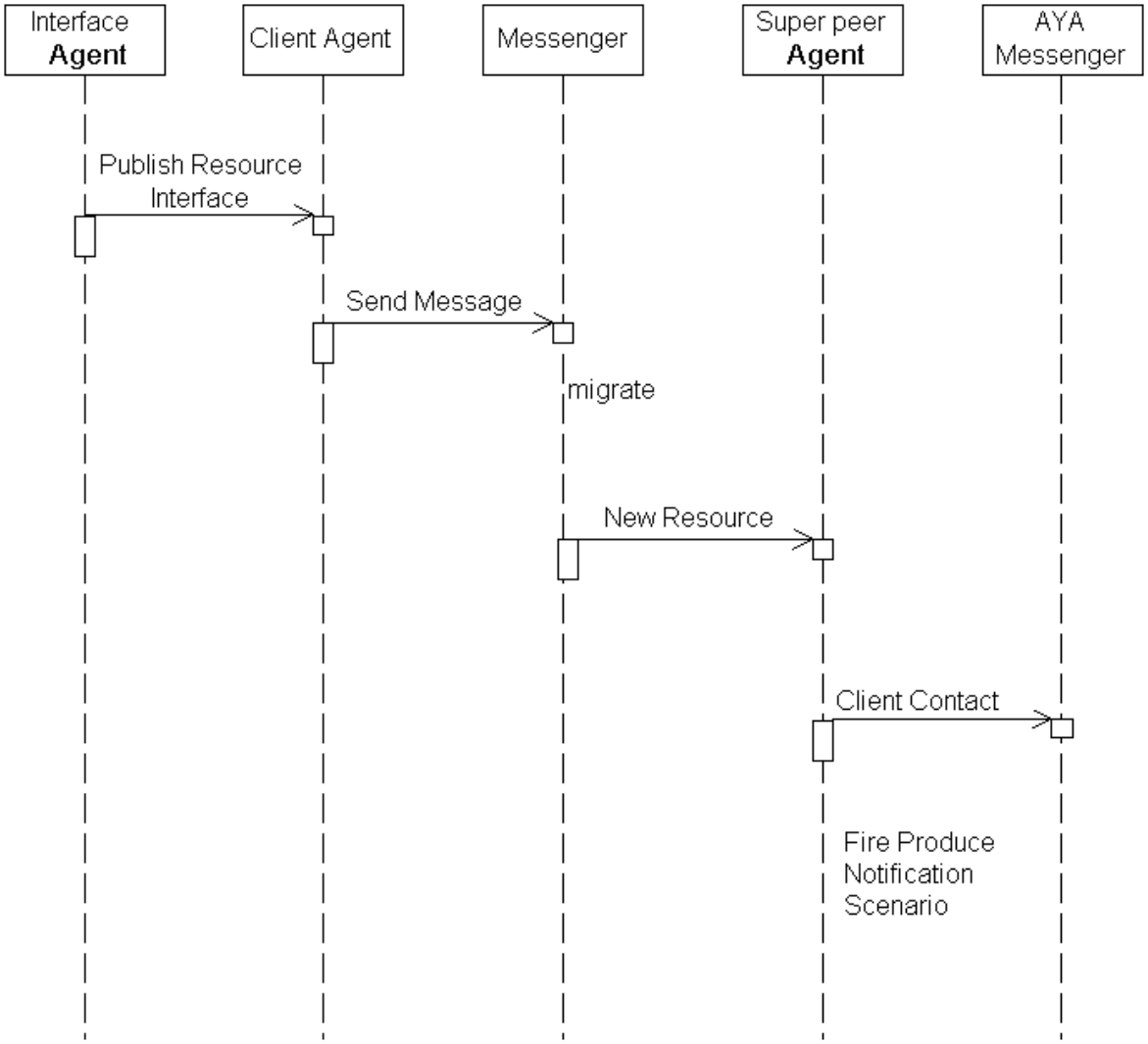

Figure A.3: Publish Resource scenario

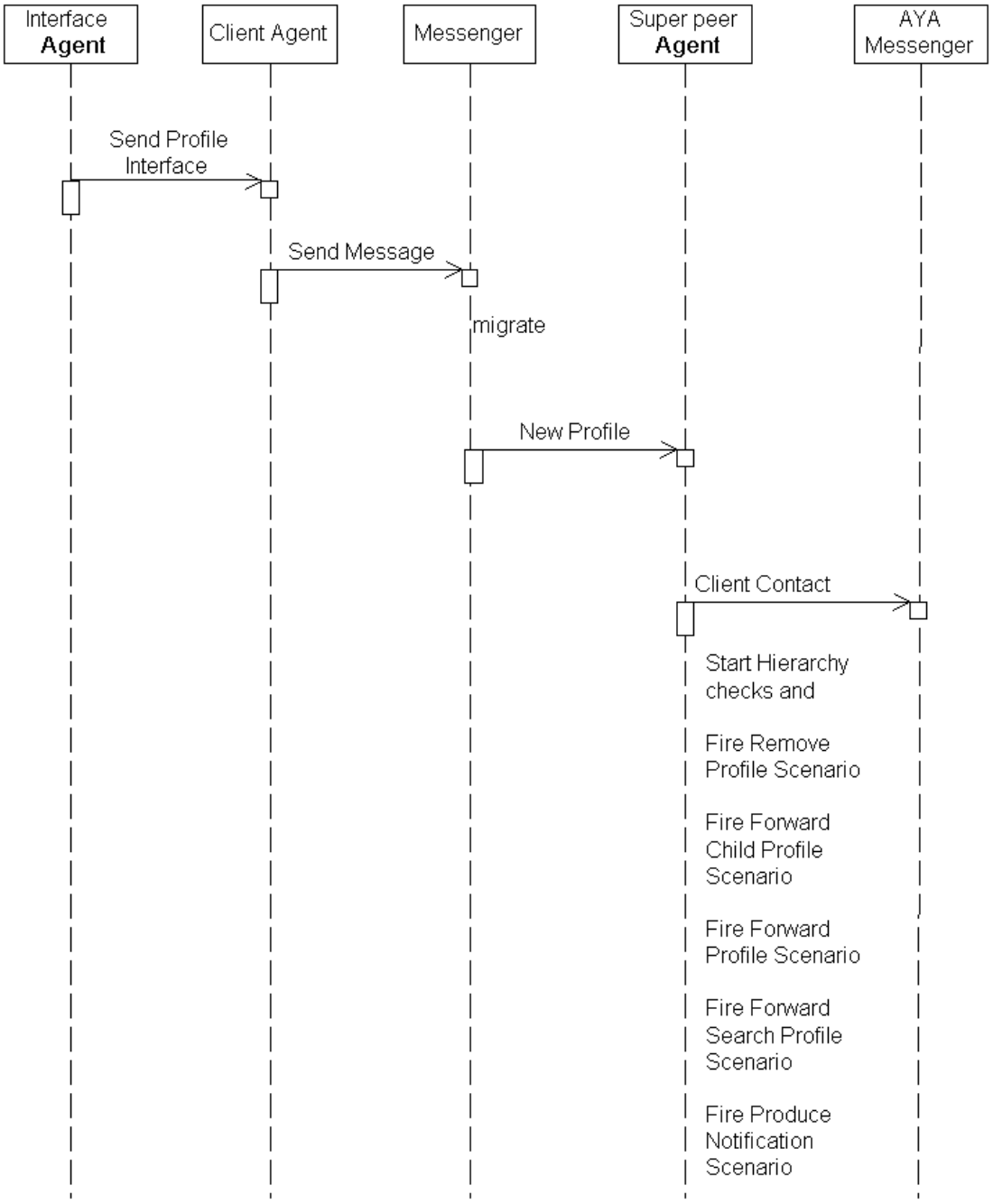

Figure A.4: Subscribe Profile scenario

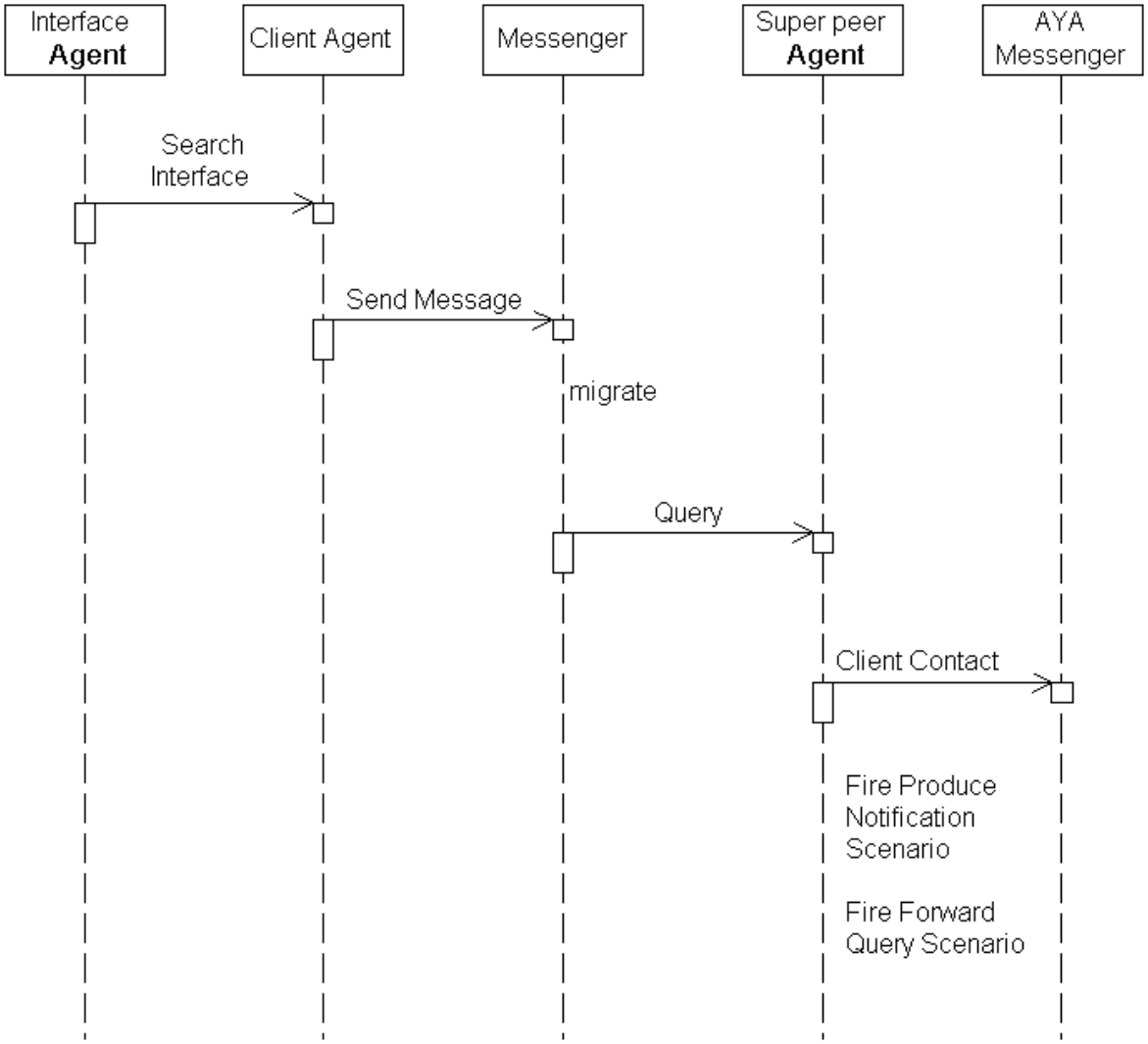

Figure A.5: Query scenario

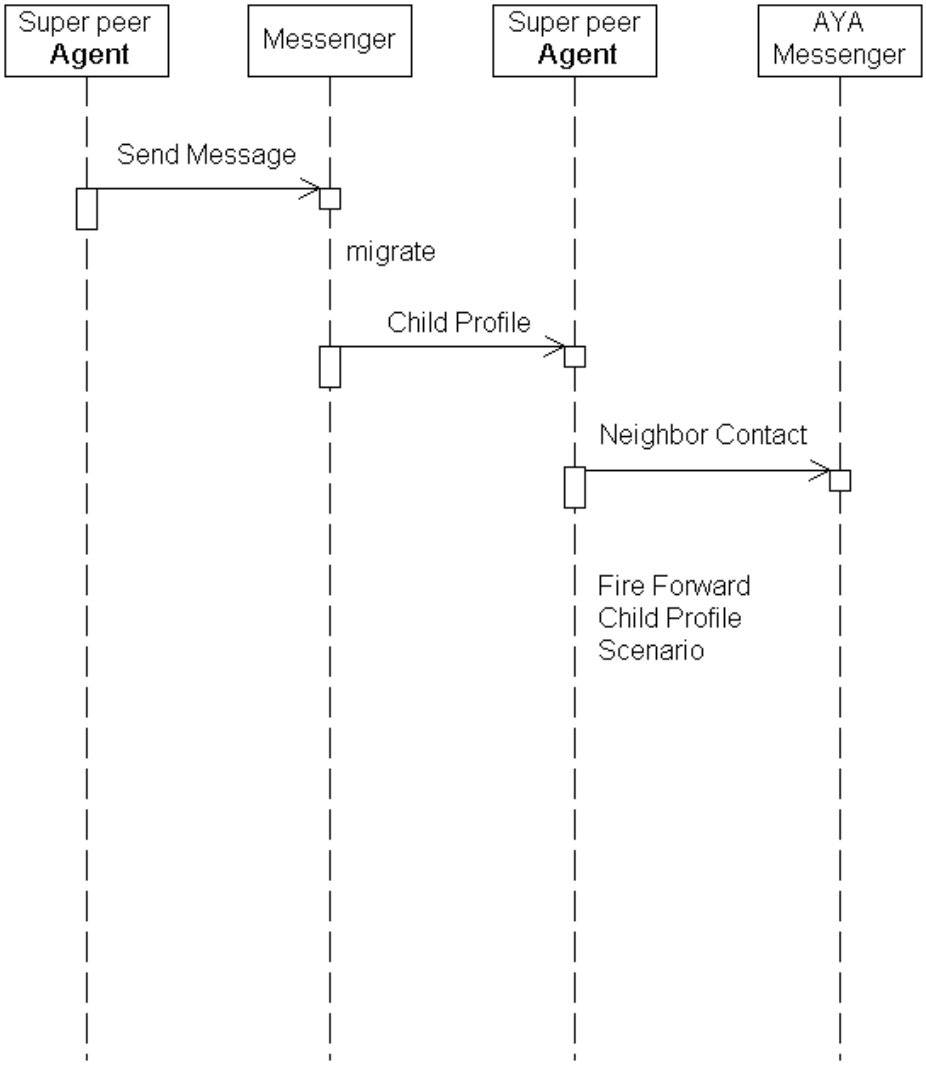

Figure A.6: Forward Child Profile Scenario

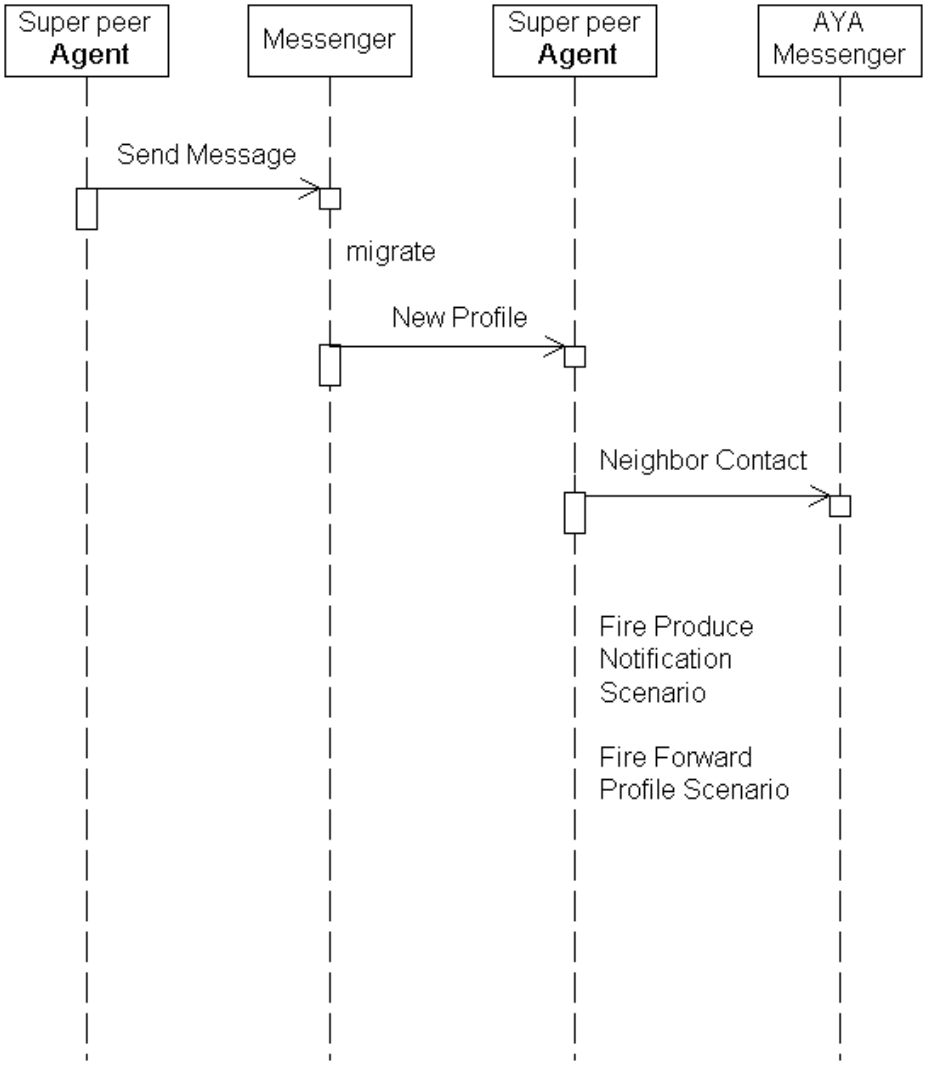

Figure A.7: Forward Profile Scenario

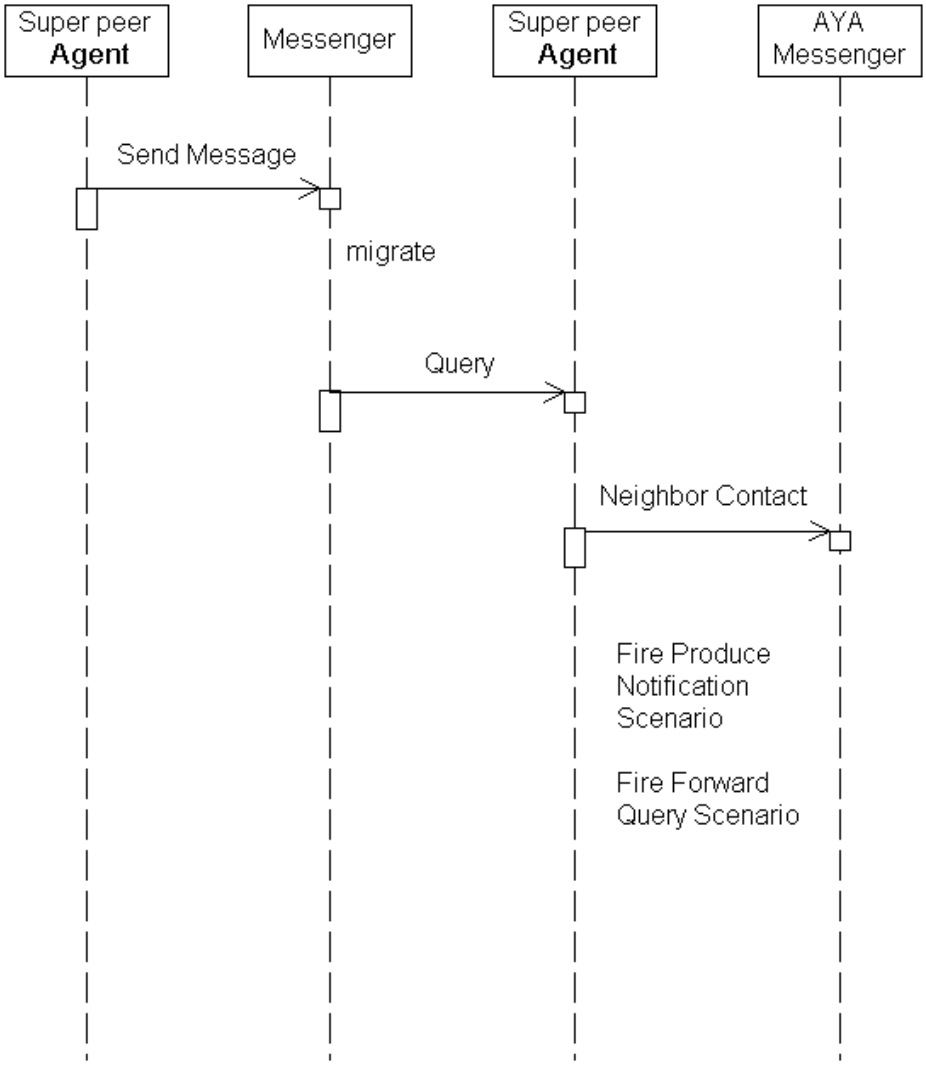

Figure A.8: Forward Query Scenario

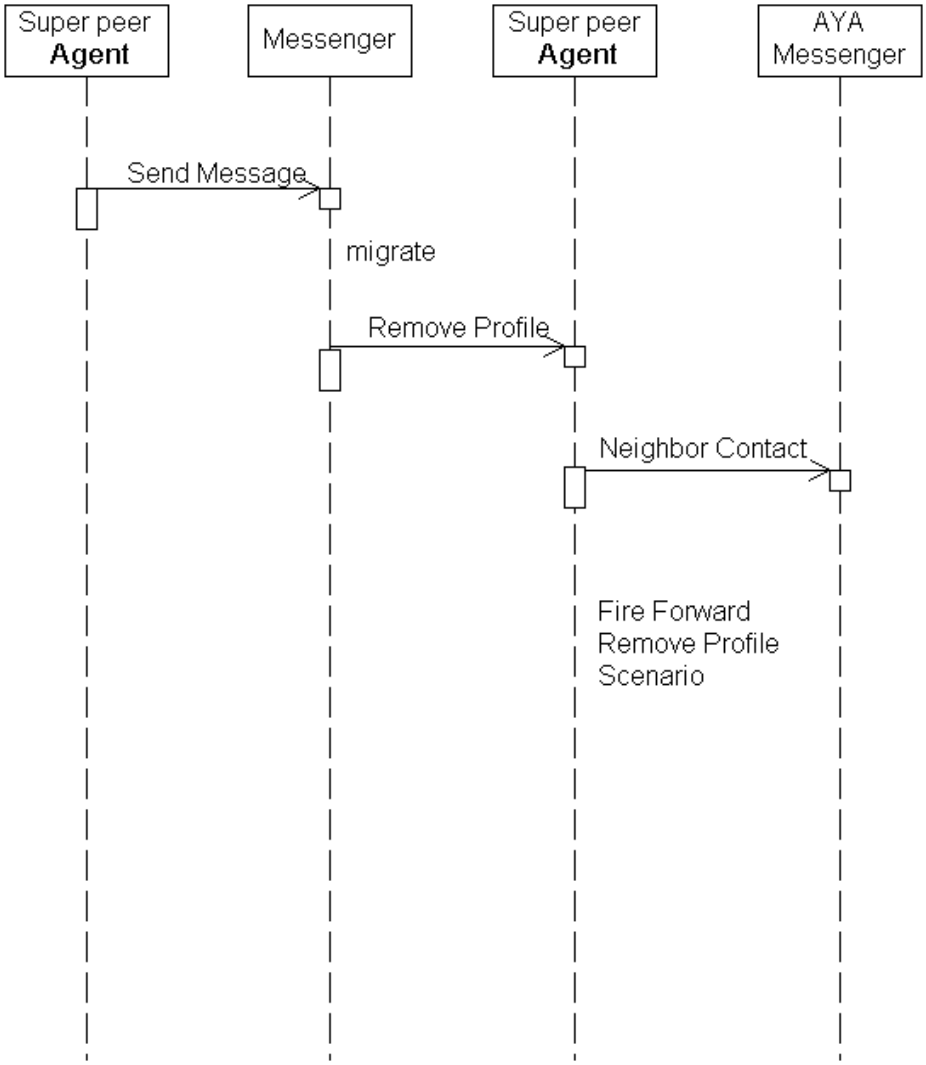

Figure A.9: Forward Remove Profile Scenario

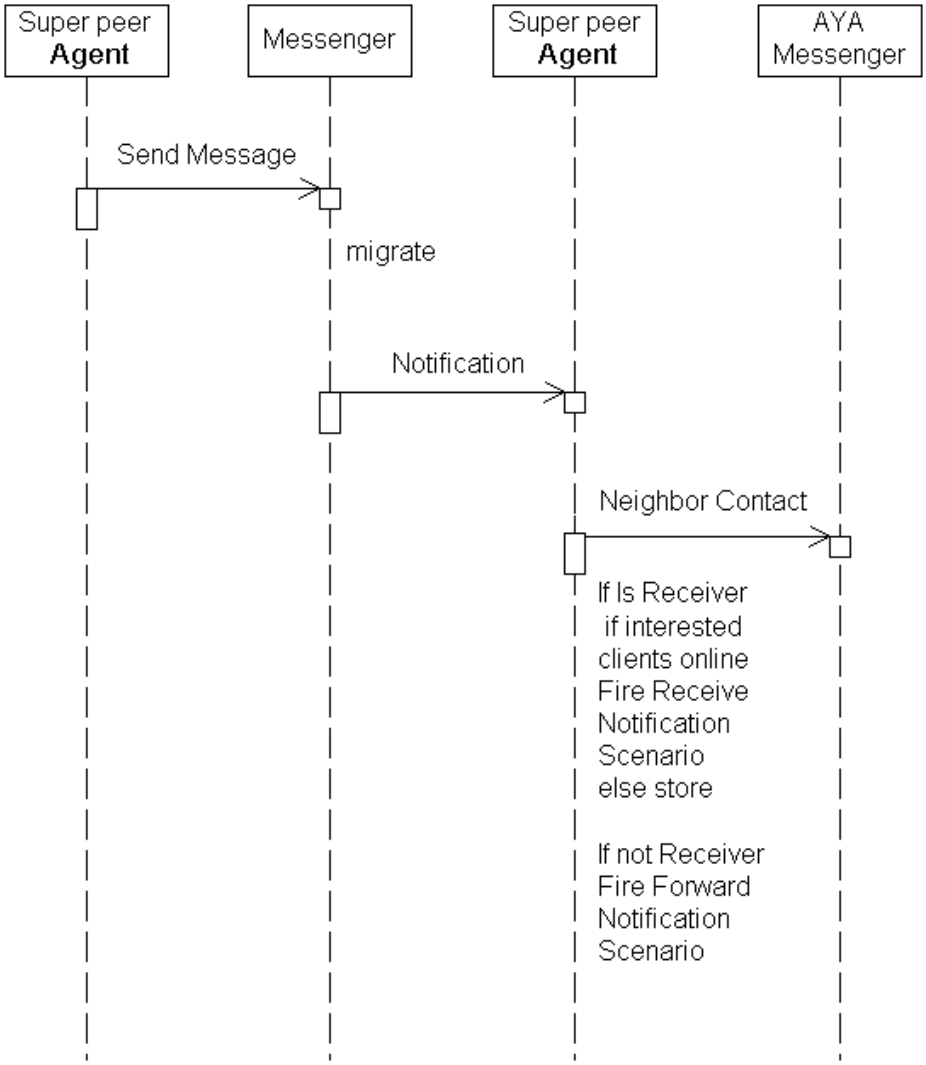

Figure A.10: Forward Notification Scenario

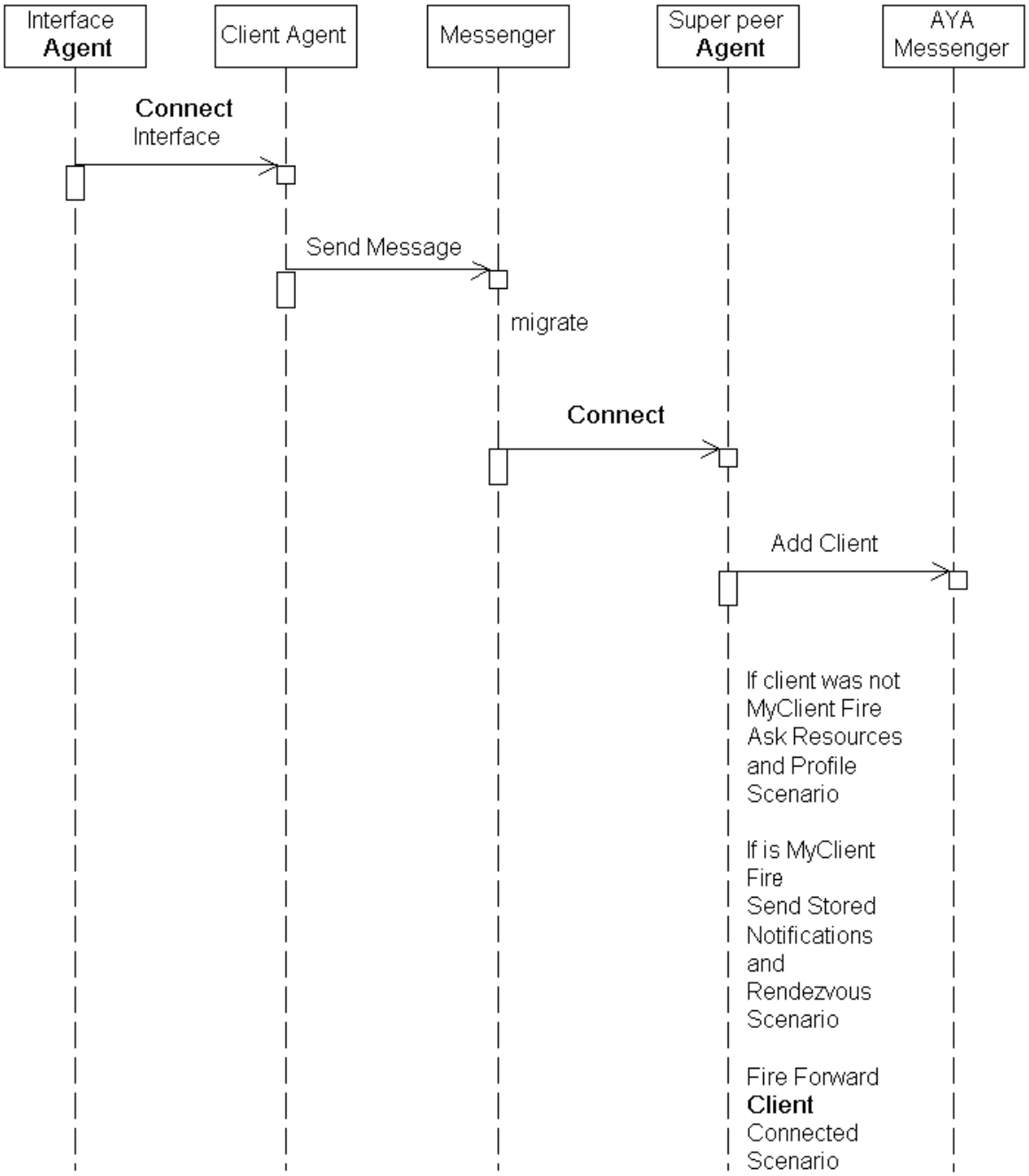

Figure A.11: Connect Scenario

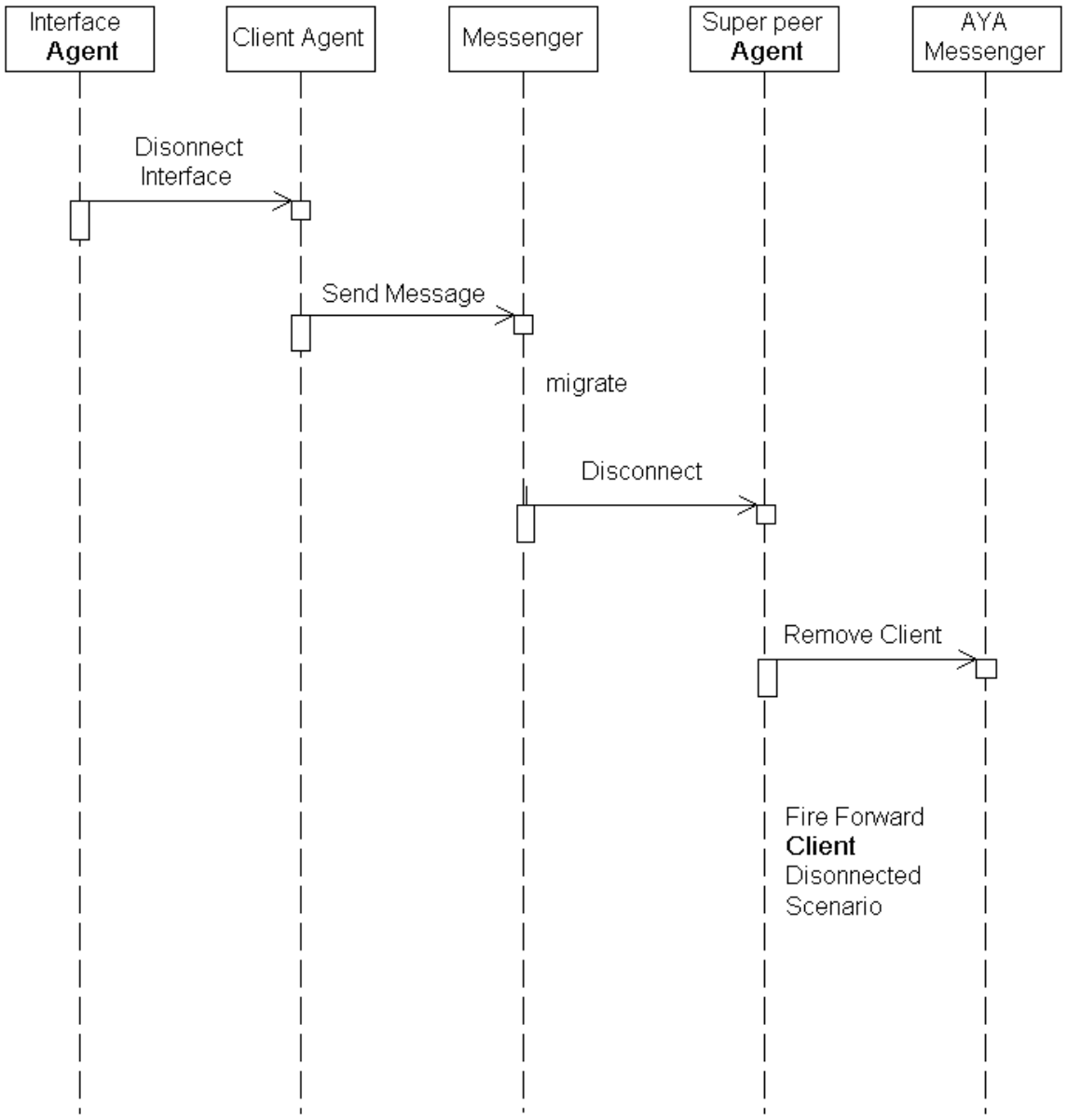

Figure A.12: Disconnect Scenario

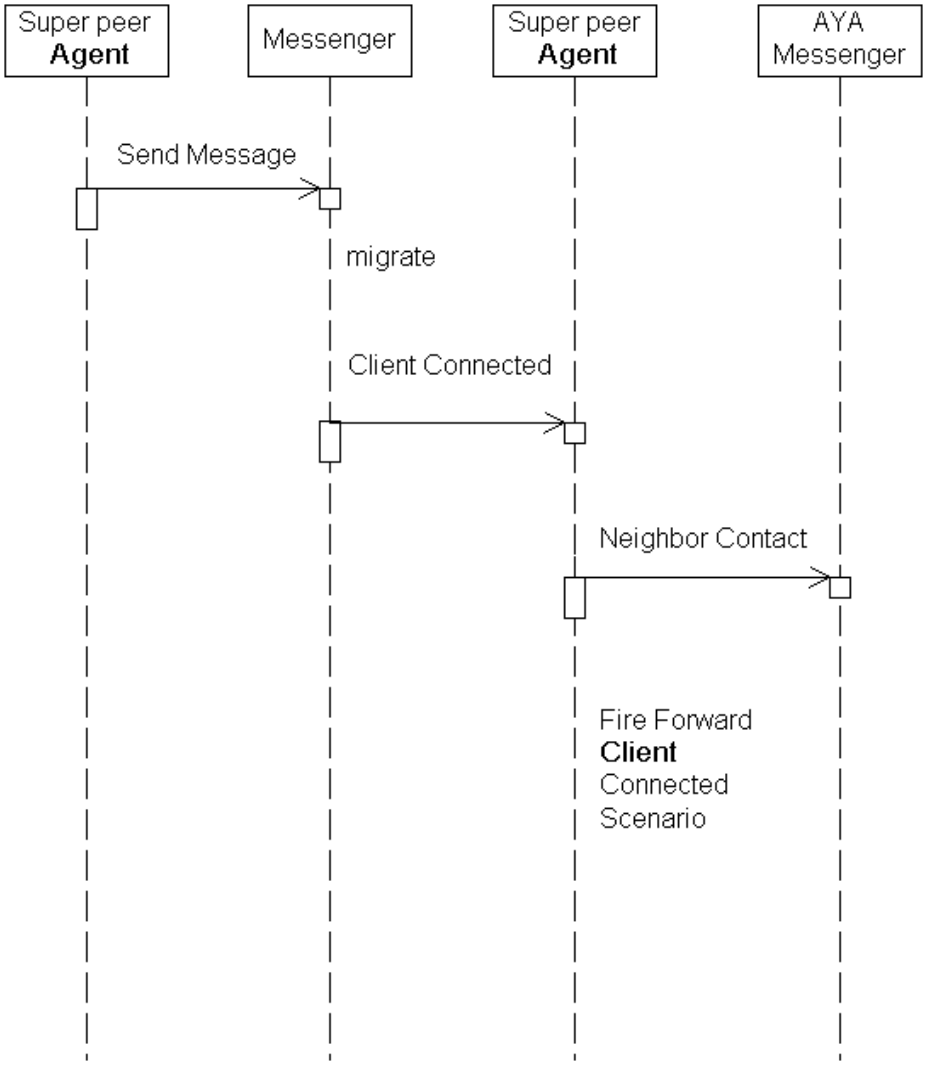

Figure A.13: Forward Client Connected Scenario

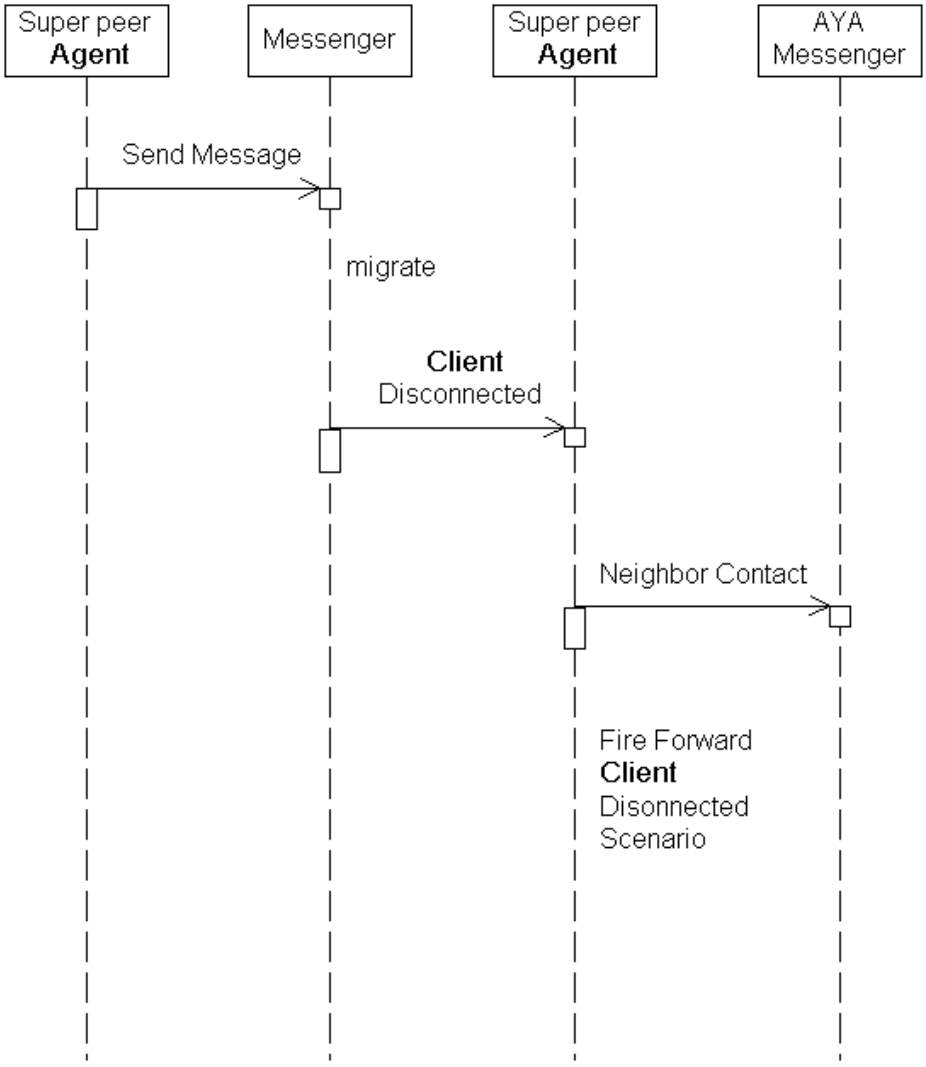

Figure A.14: Forward Client Disconnected Scenario

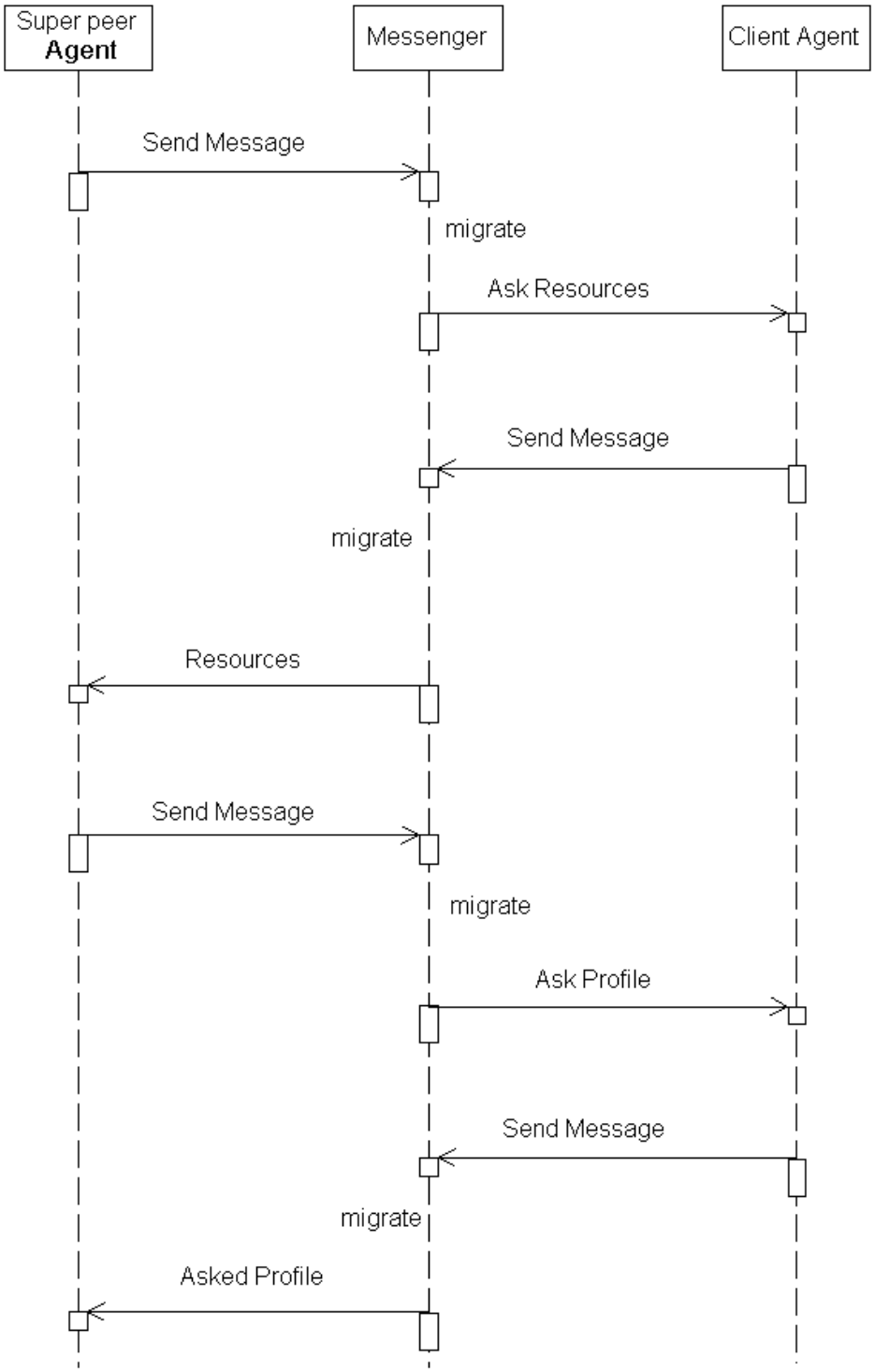

Figure A.15: Ask Resources And Profile Scenario

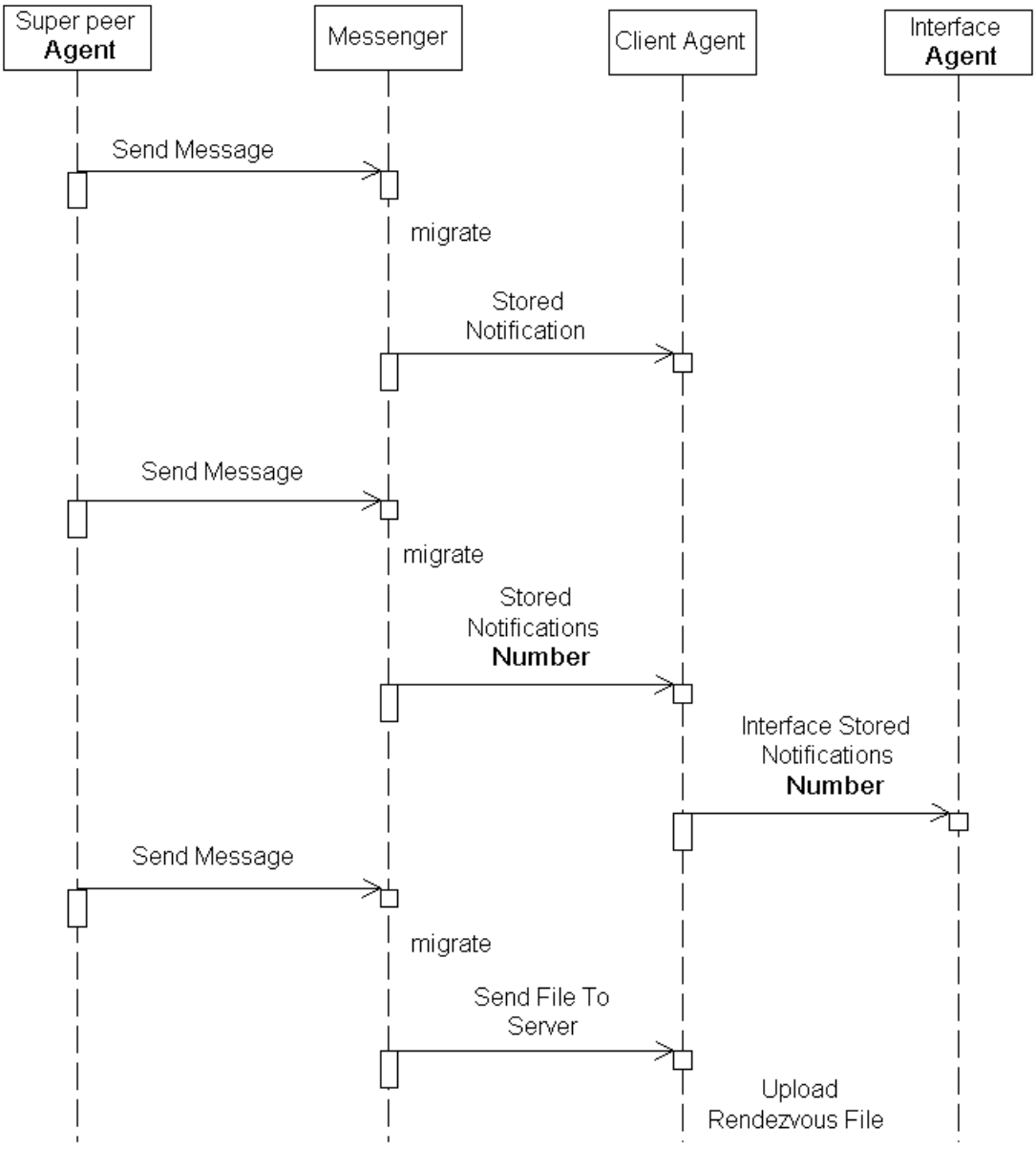

Figure A.16: Send Stored Notifications And Rendezvous Scenario

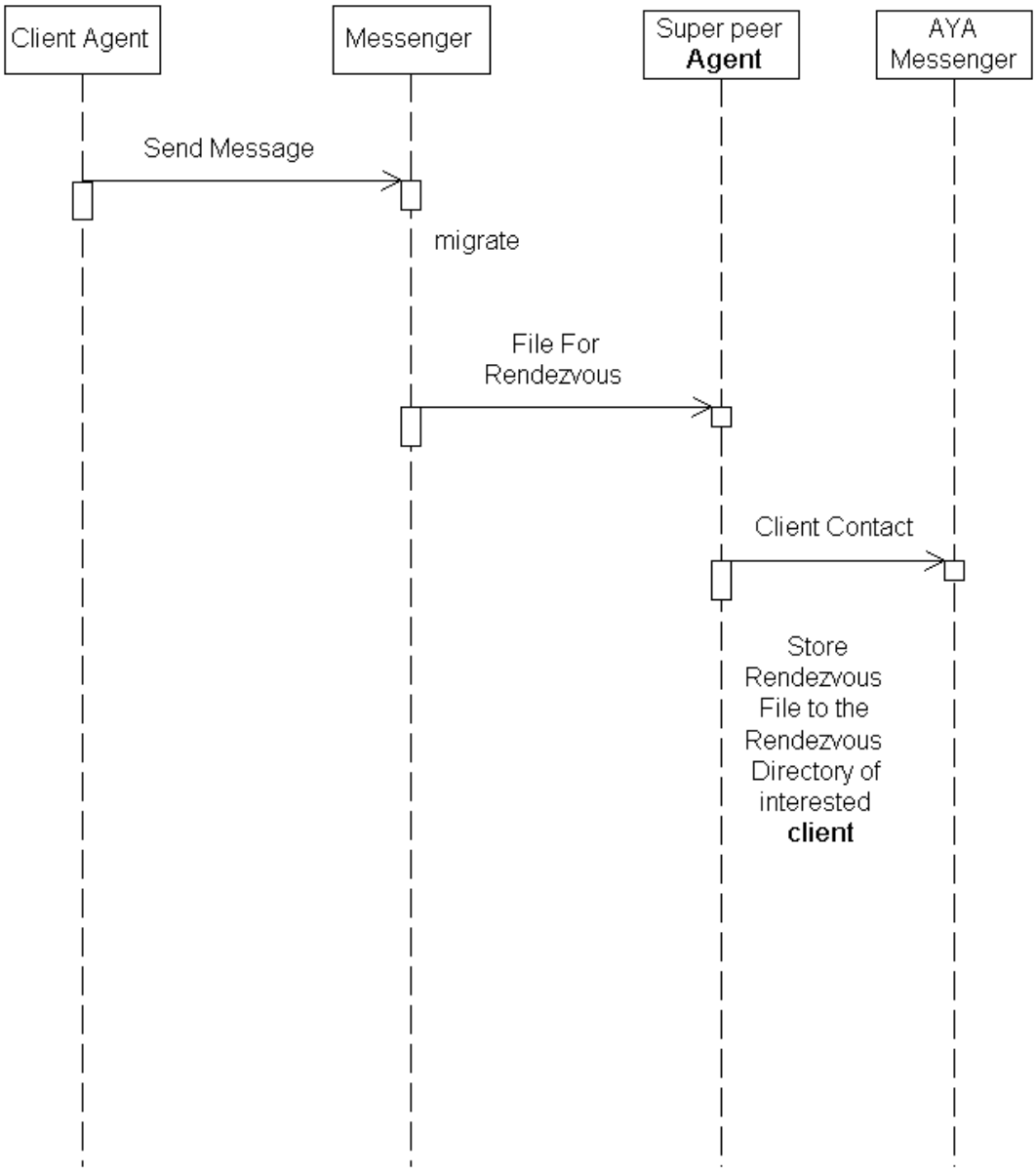

Figure A.17: Upload Rendezvous File Scenario

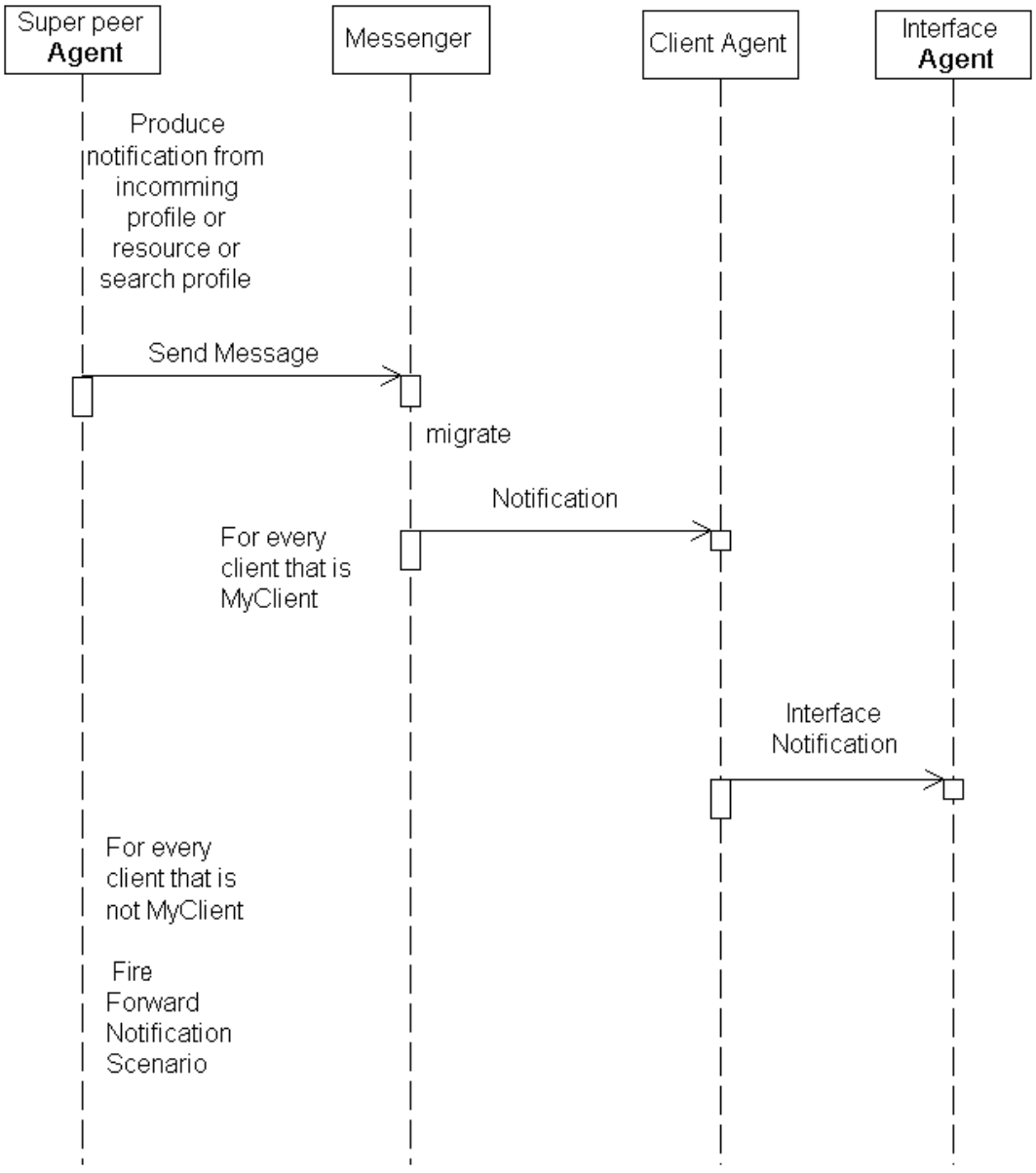

Figure A.18: Produce Notification Scenario

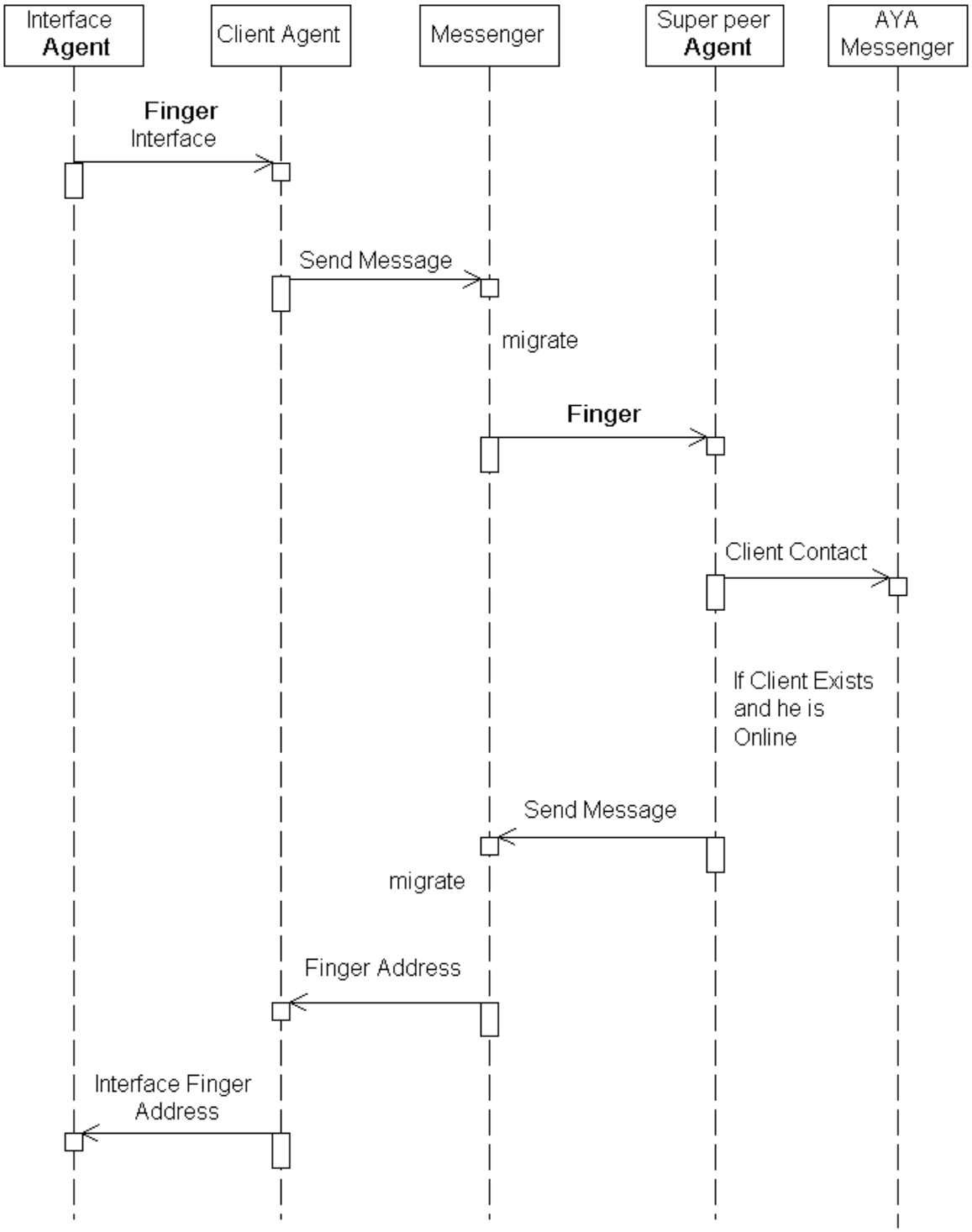

Figure A.19: Finger Scenario

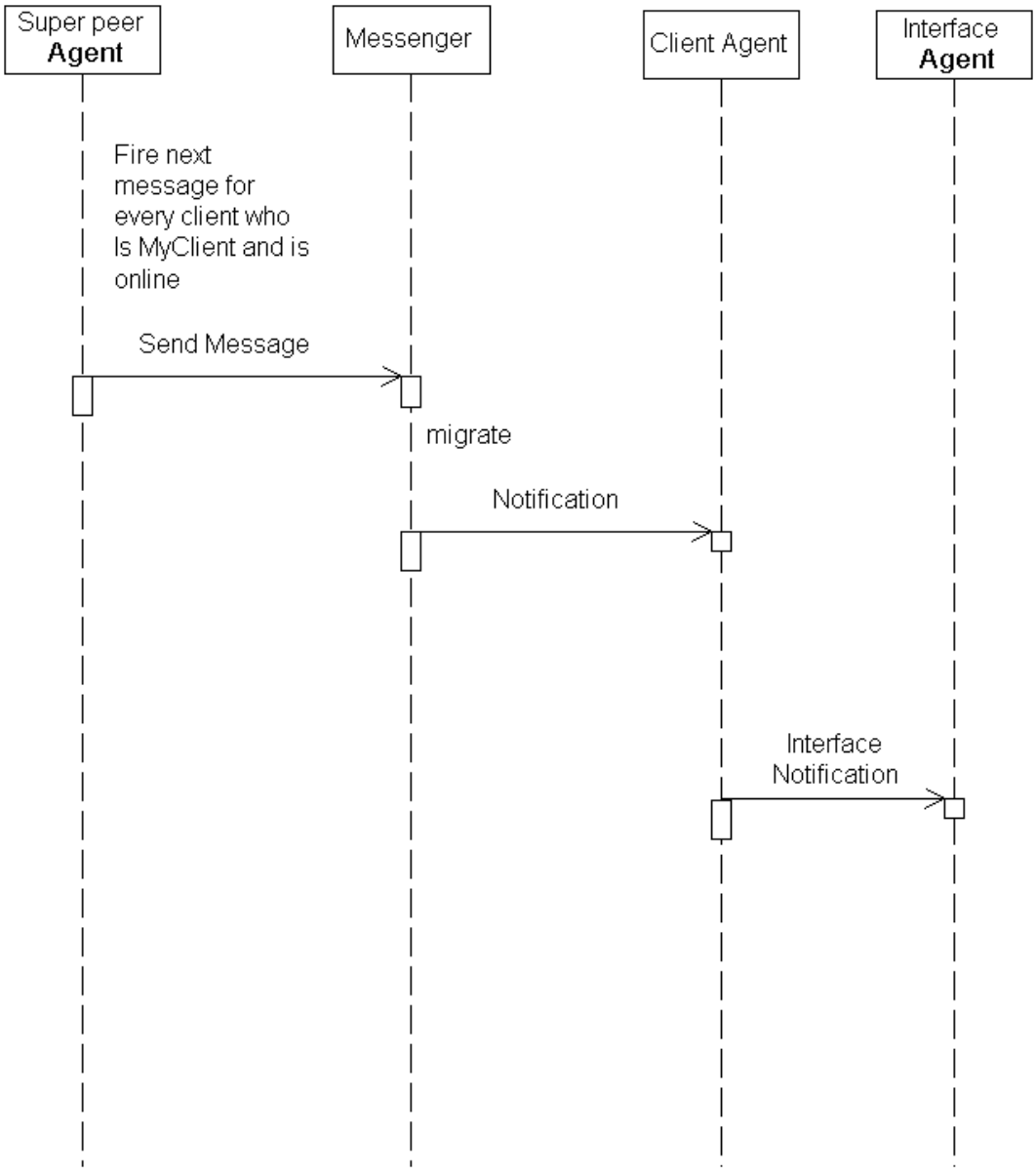

Figure A.20: Send Notification To Clients Scenario

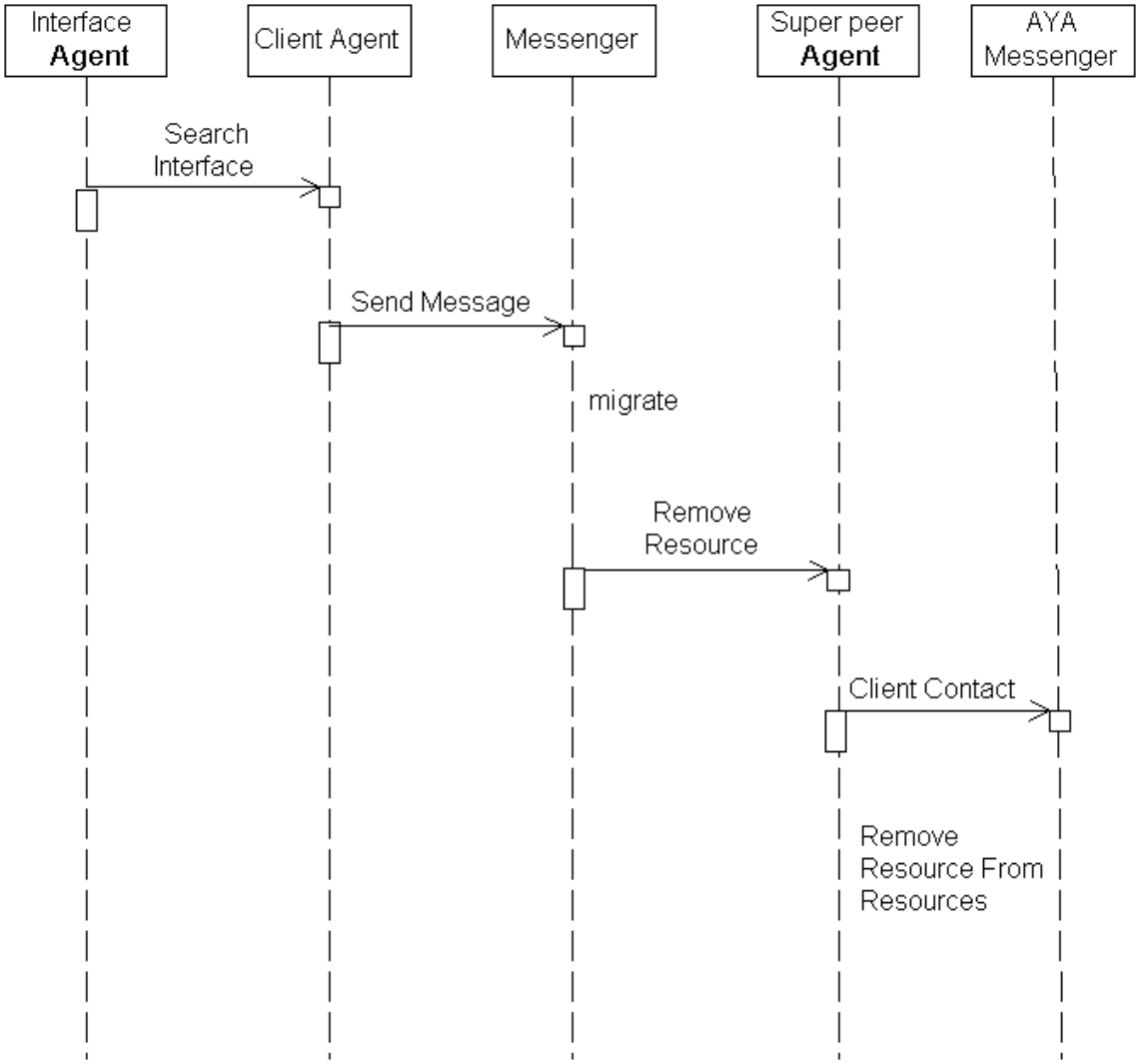

Figure A.21: Remove Resource Scenario

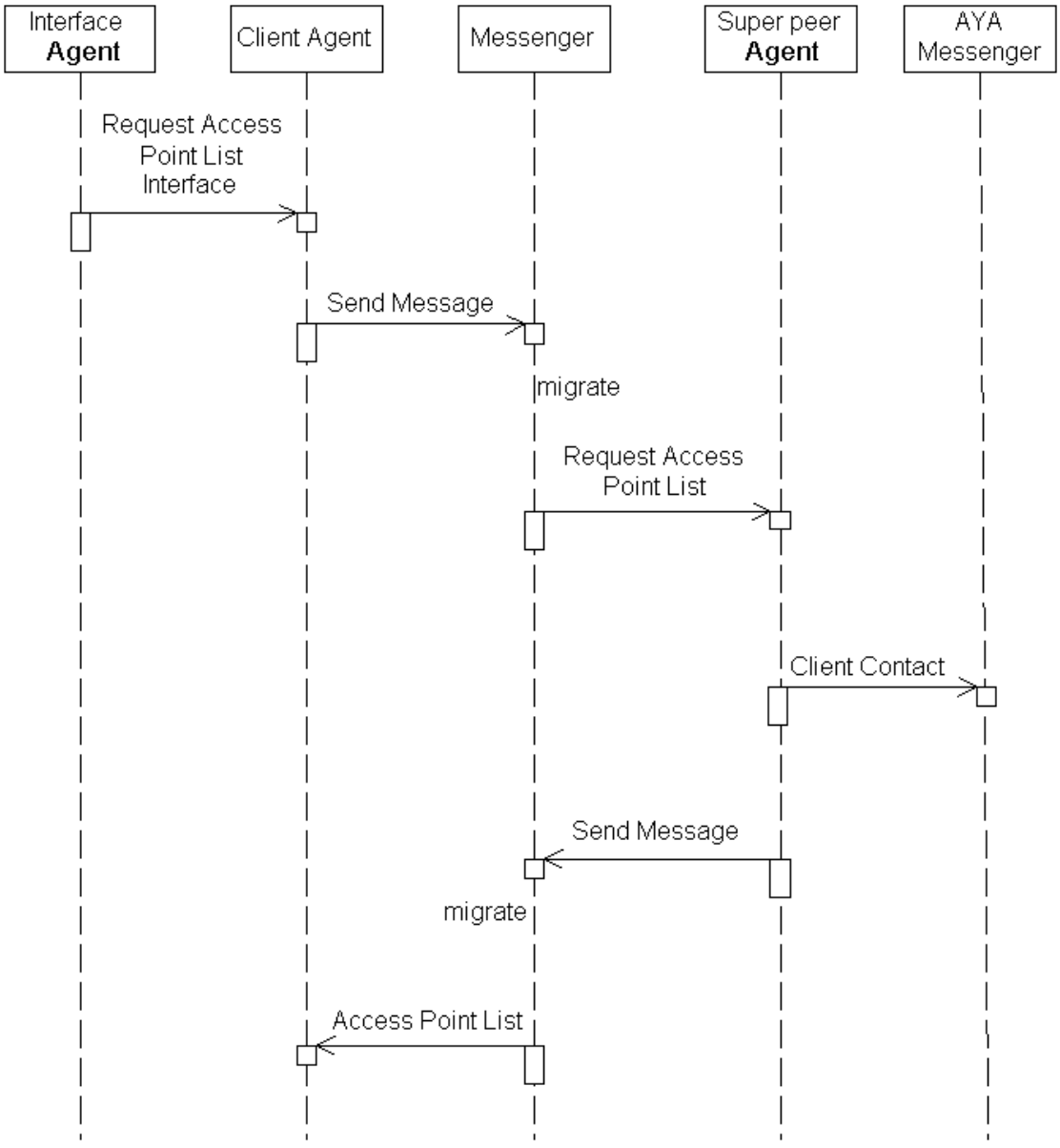

Figure A.22: Request Access Point List Scenario

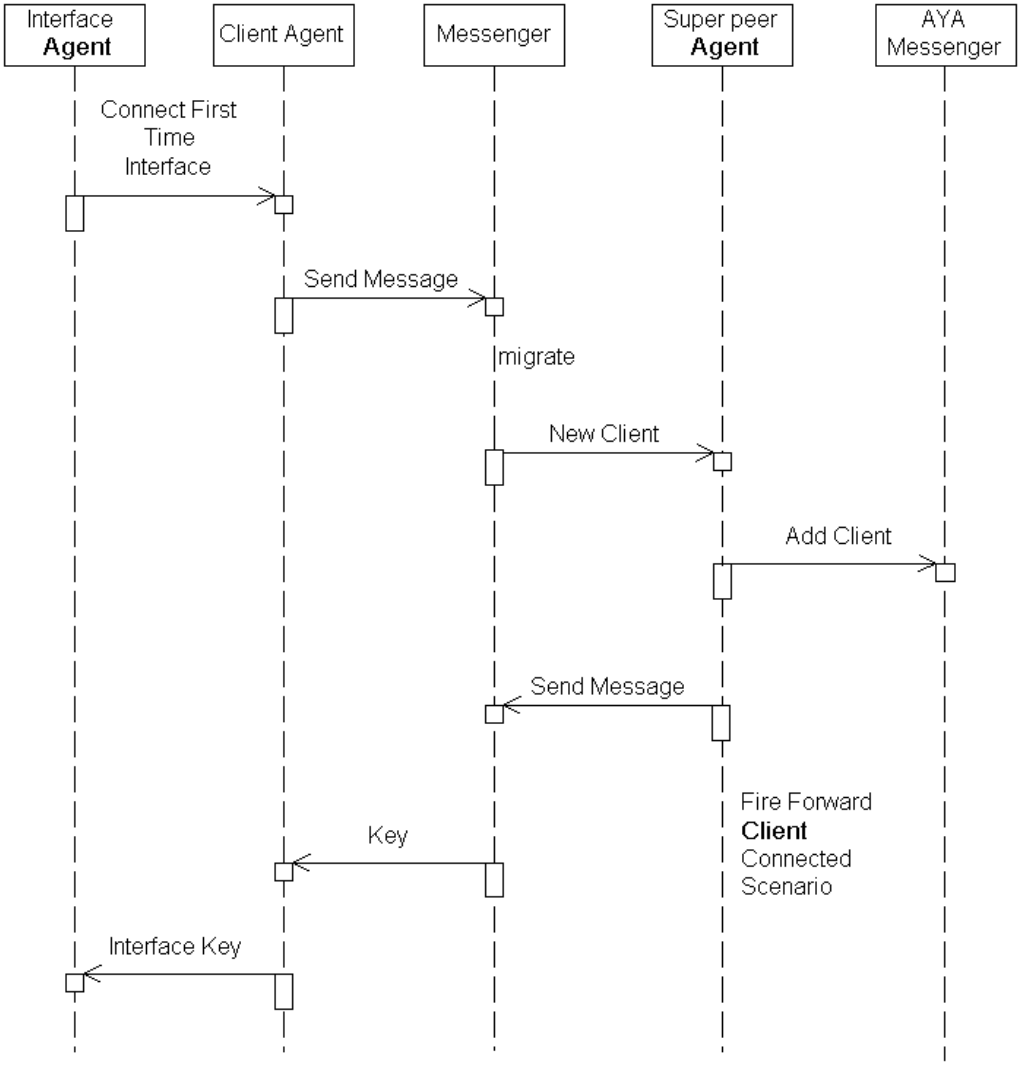

Figure A.23: New Client Scenario

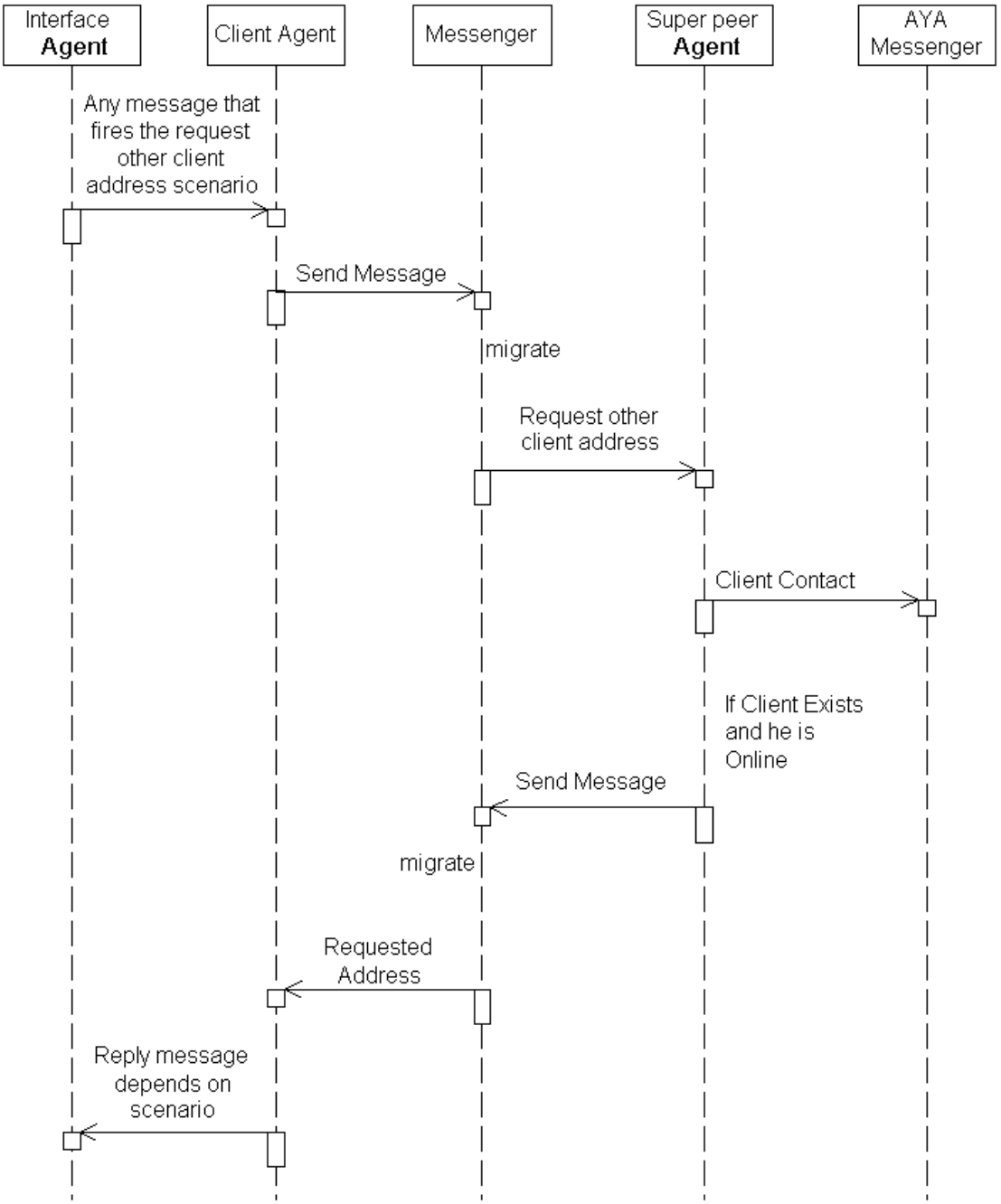

Figure A.24: Request Other Client Address Scenario

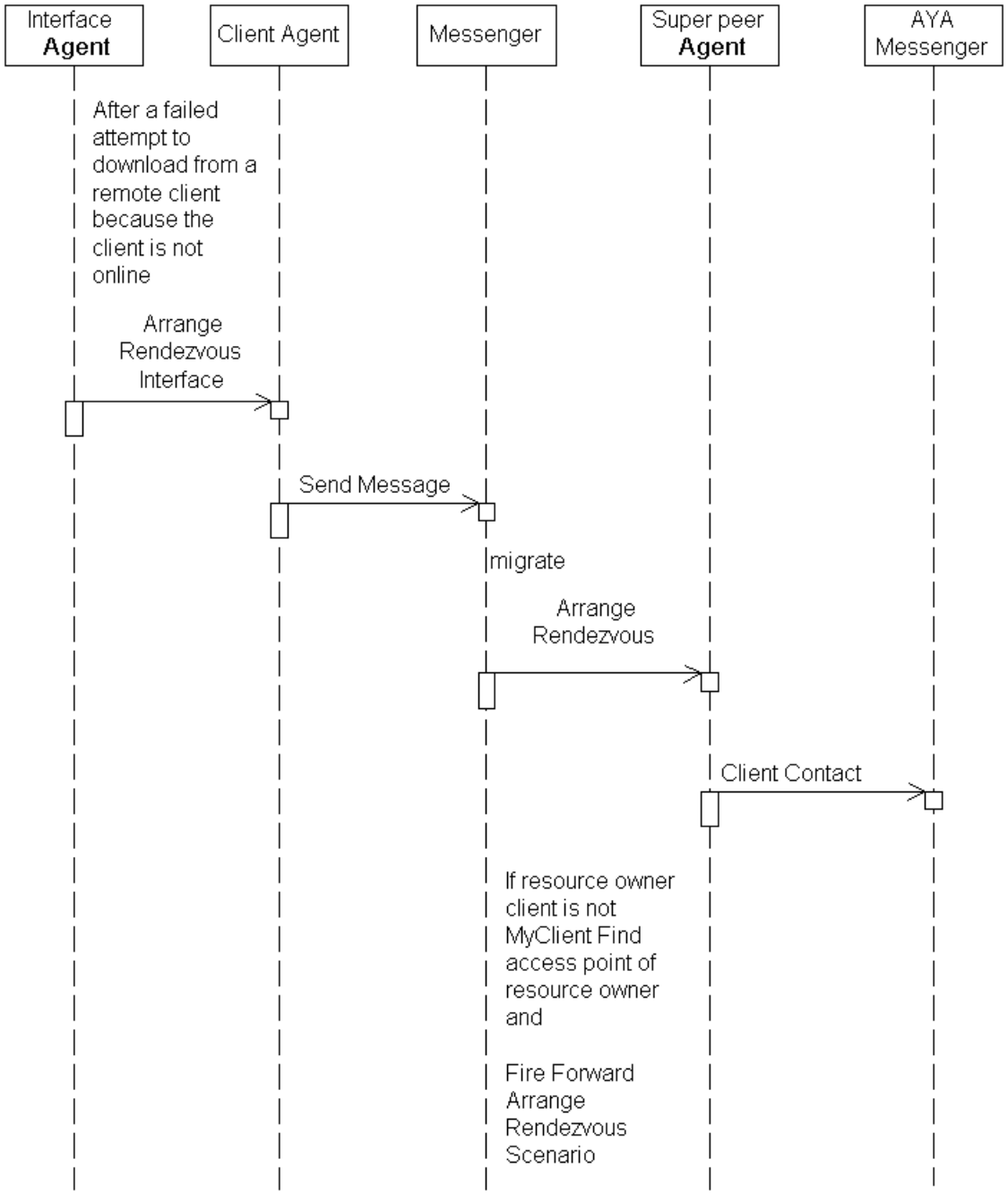

Figure A.25: Arrange Rendezvous Scenario

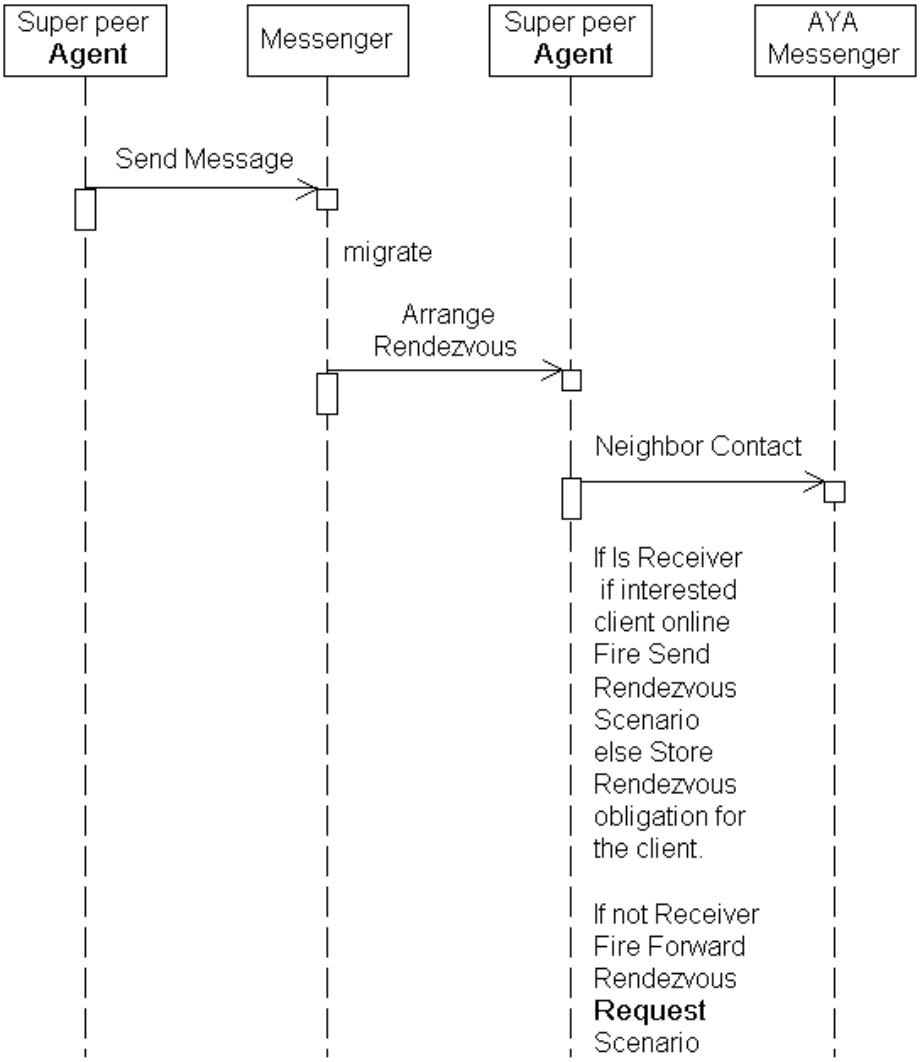

Figure A.26: Forward Rendezvous Request Scenario

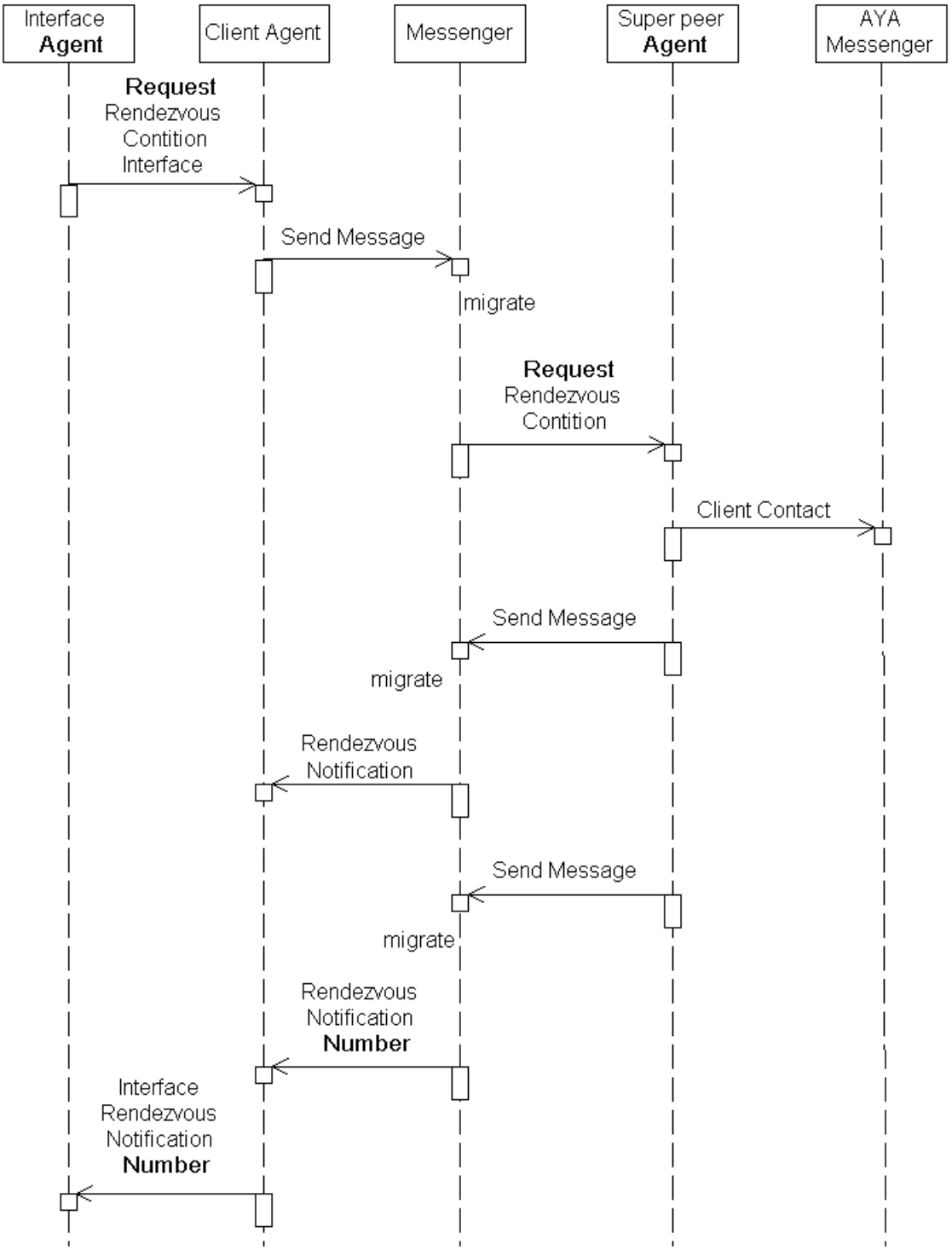

Figure A.27: Request Rendezvous Condition Scenario

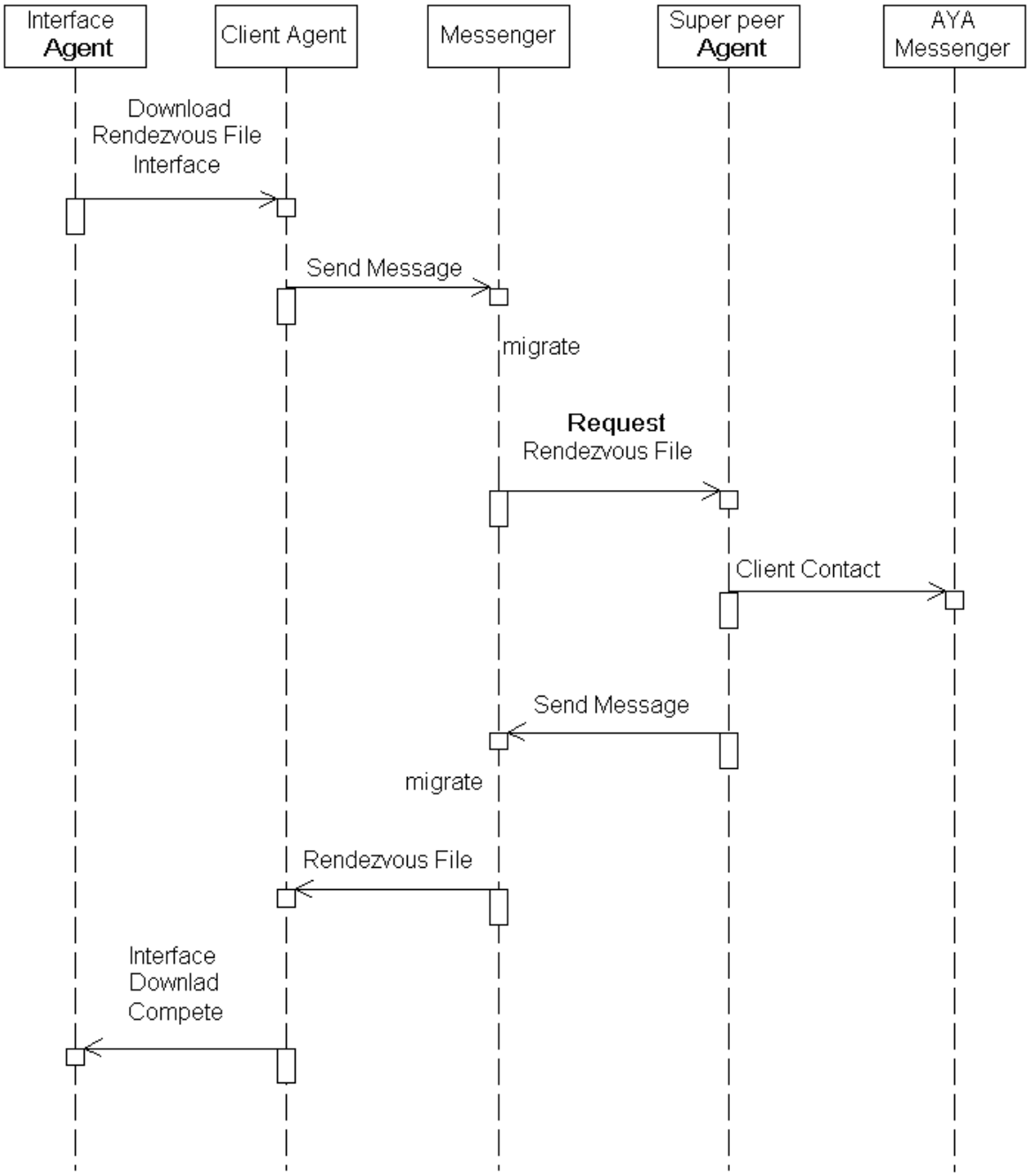

Figure A.28: Request Rendezvous File Scenario

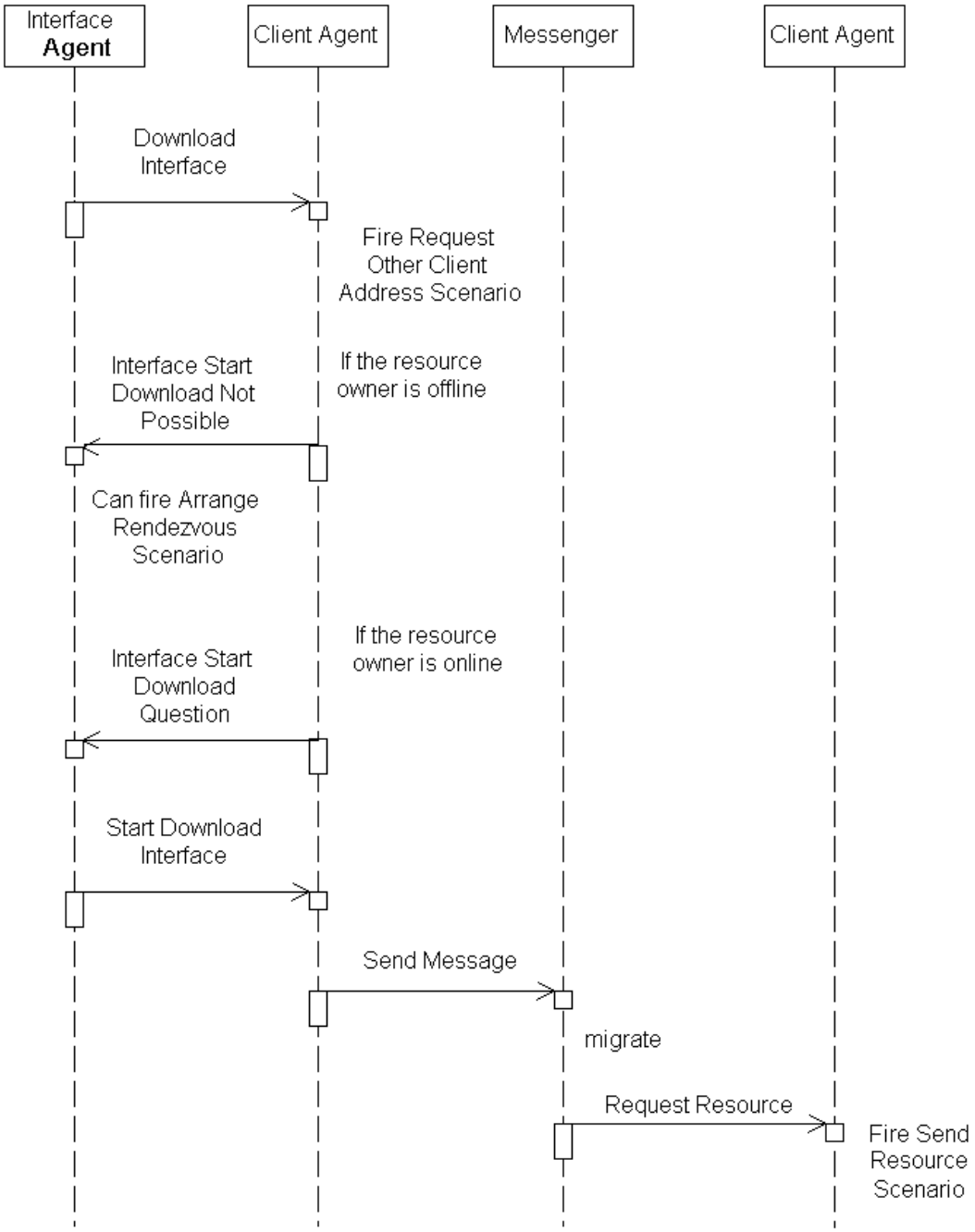

Figure A.29: Download Scenario

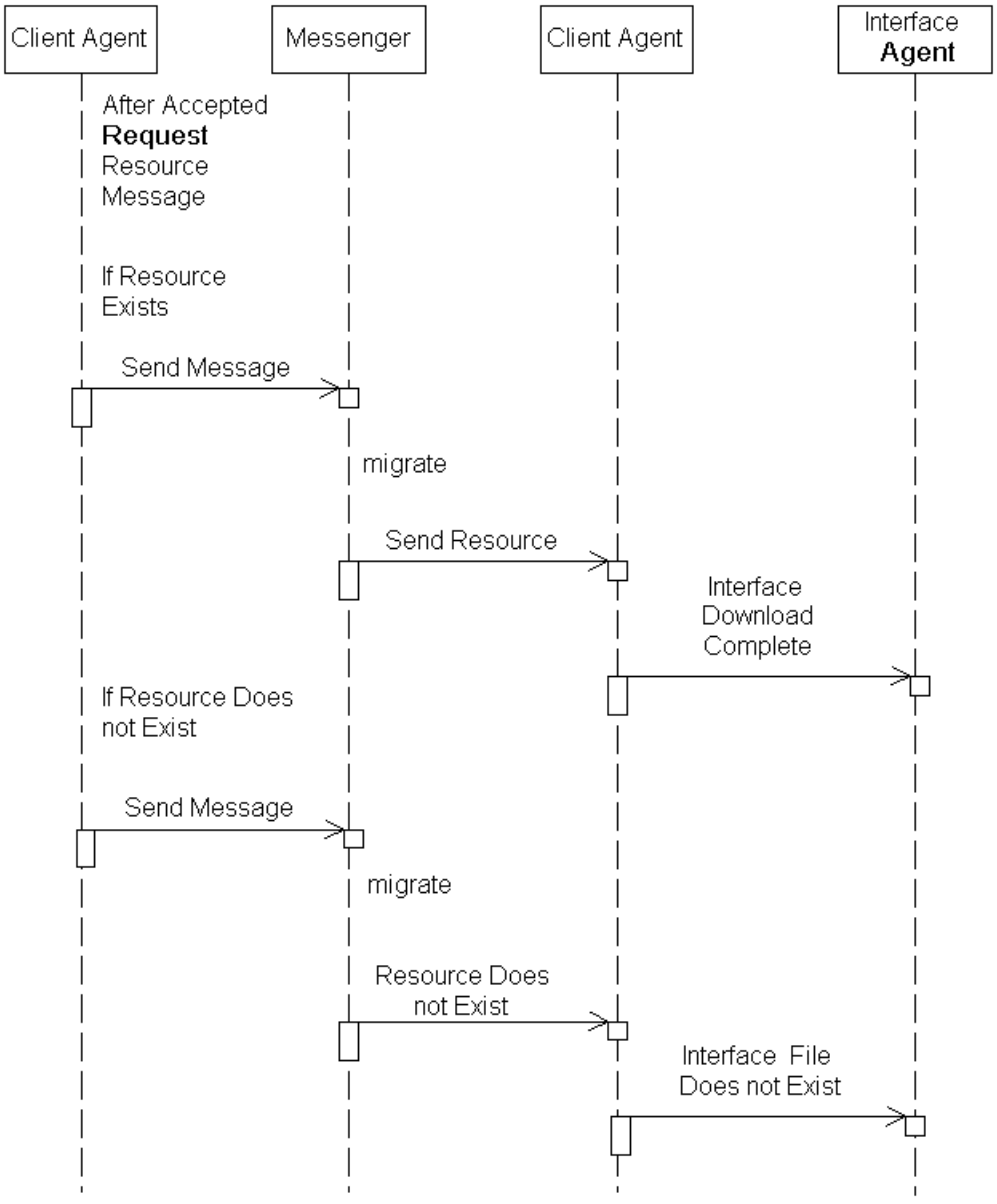

Figure A.30: Send Resource Scenario

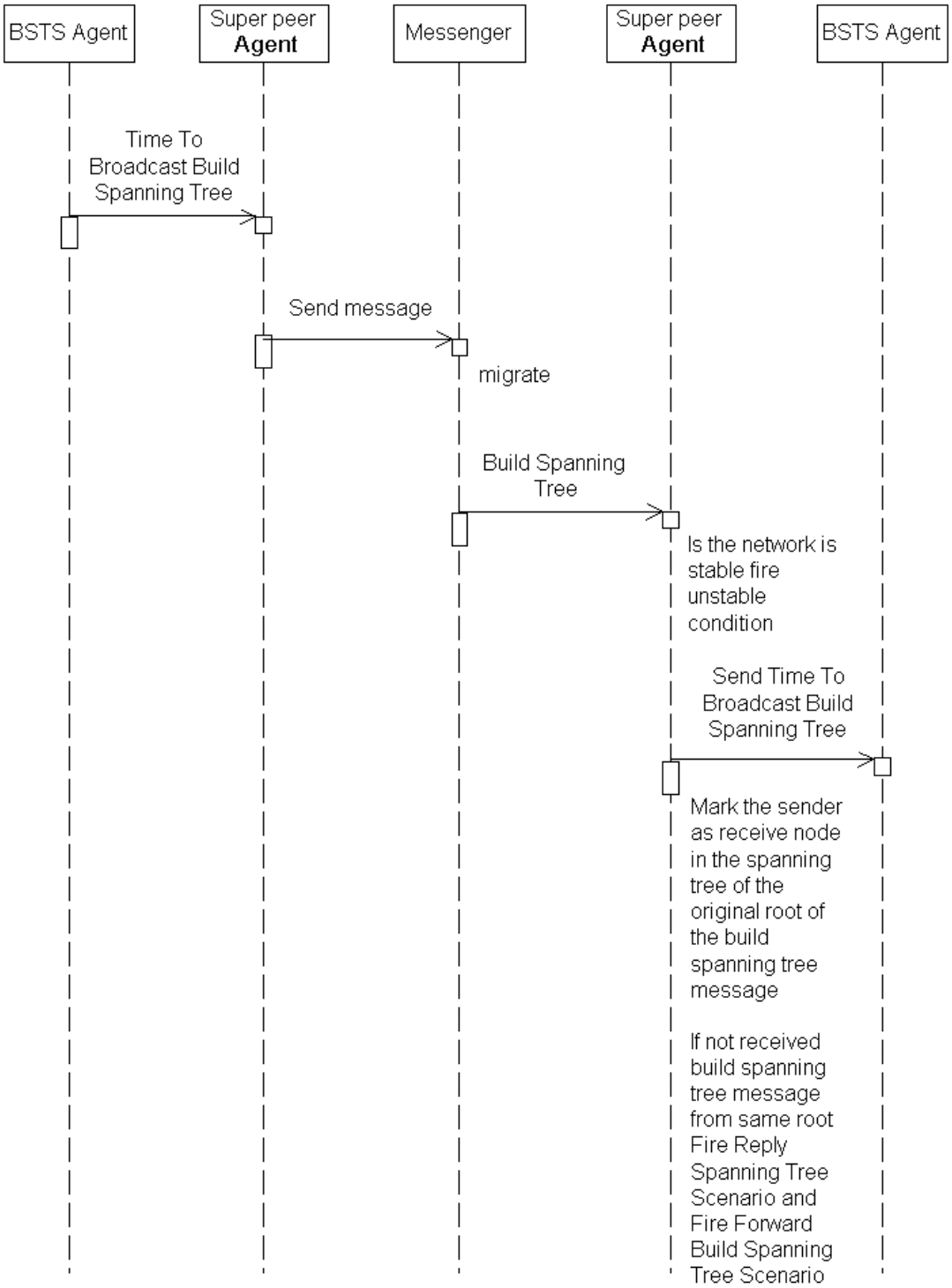

Figure A.31: Build Spanning Tree Scenario

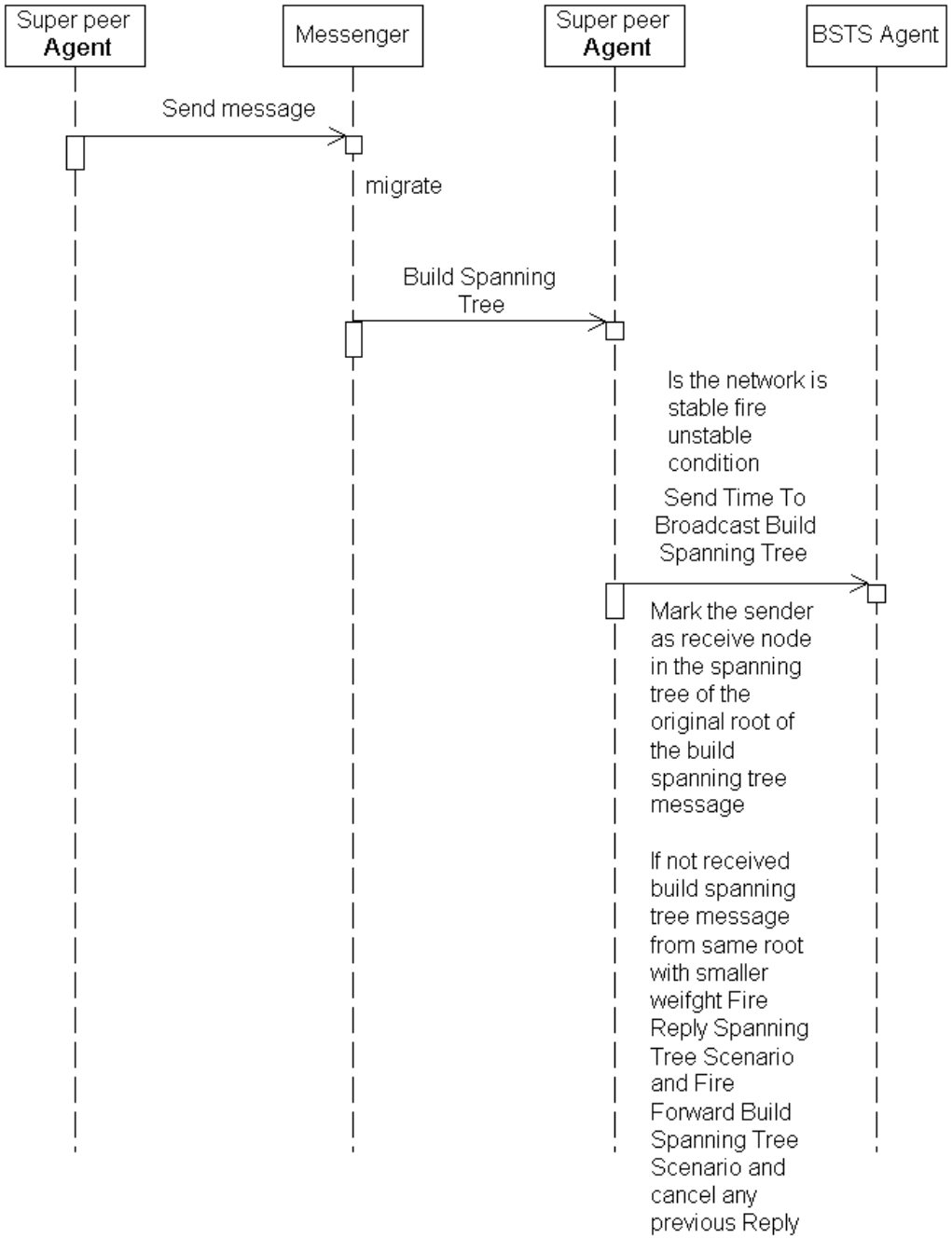

Figure A.32: Forward Build Spanning Tree Scenario

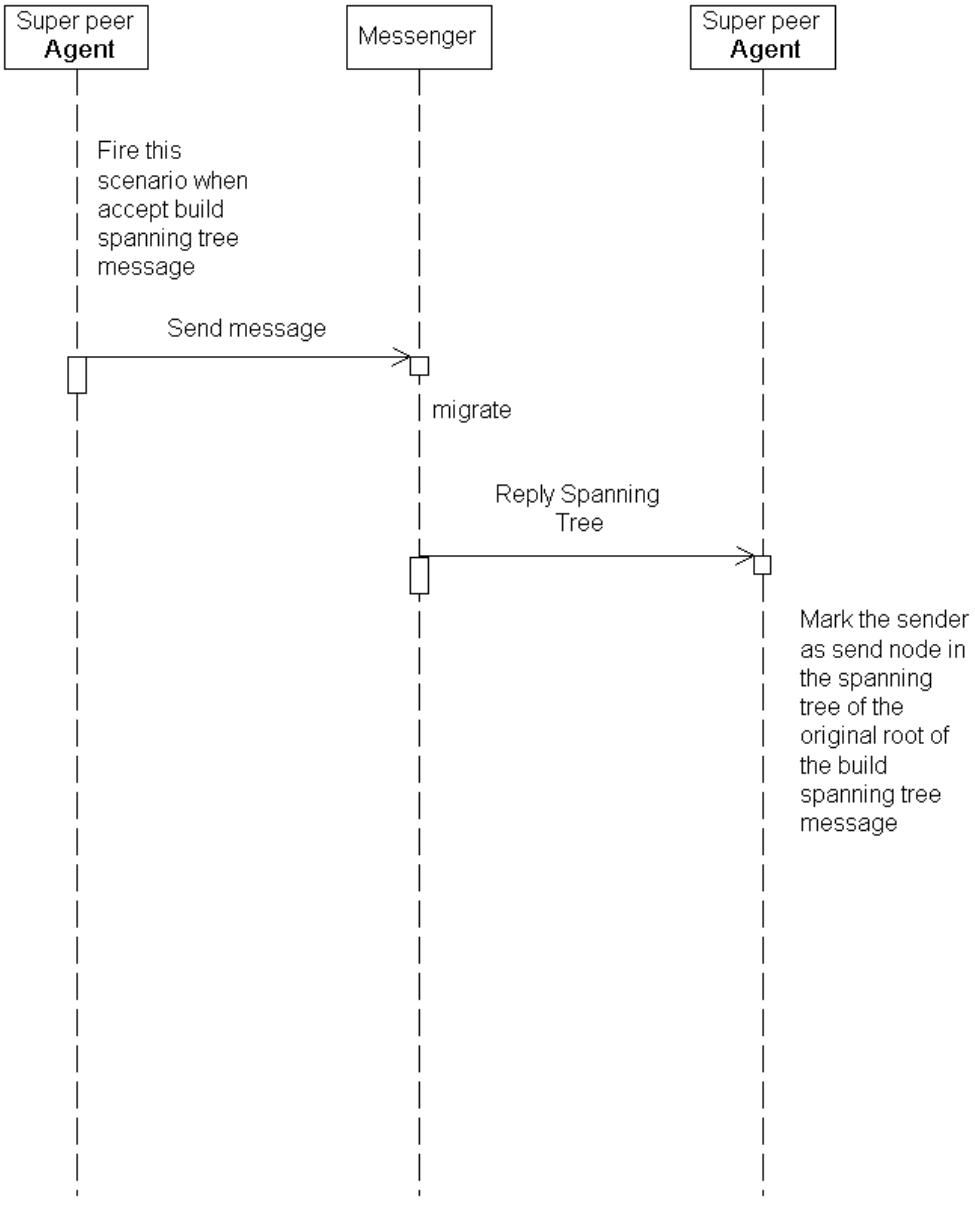

Figure A.33: Reply Spanning Tree Scenario

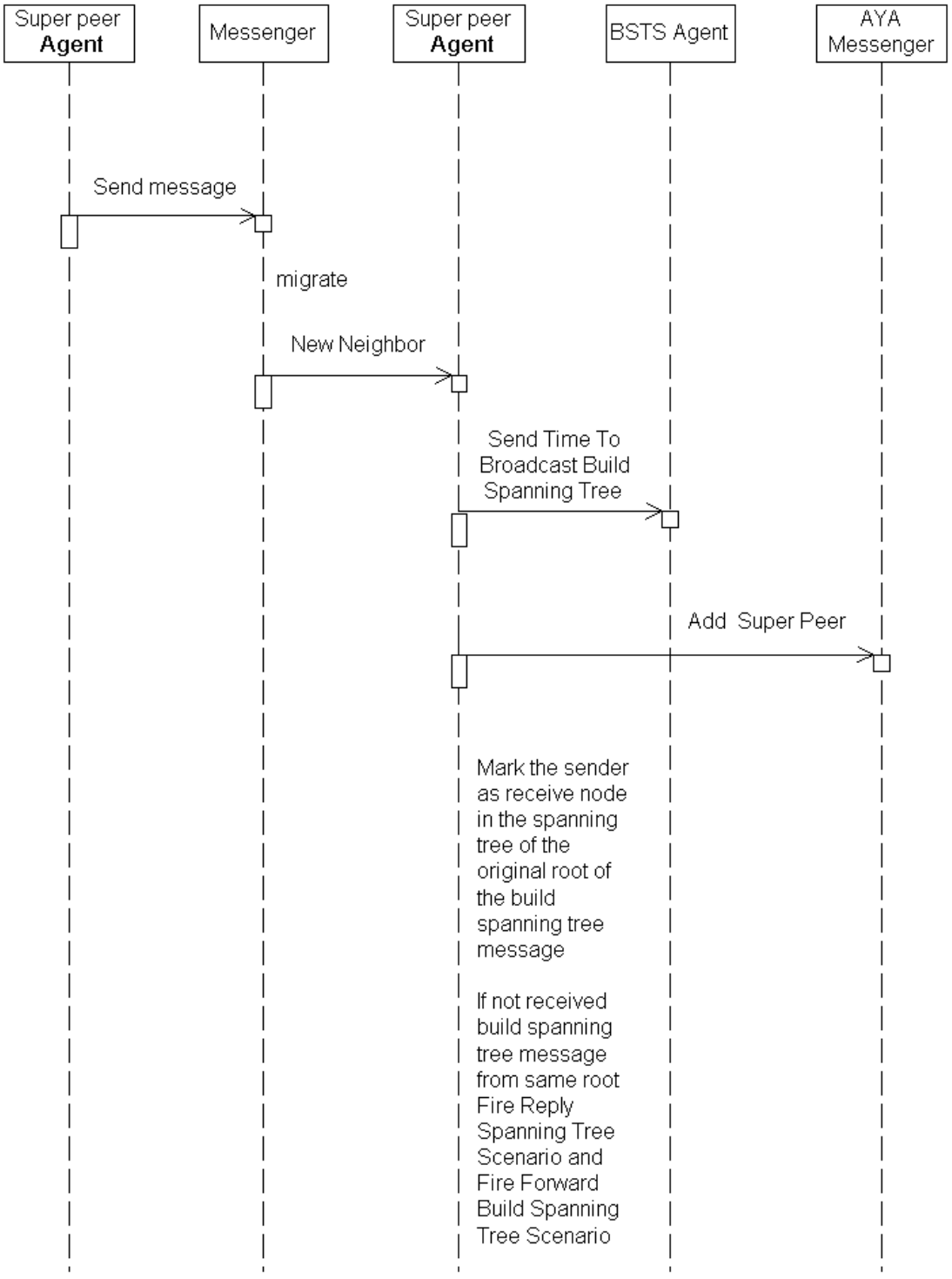

Figure A.34: New Neighbor Scenario

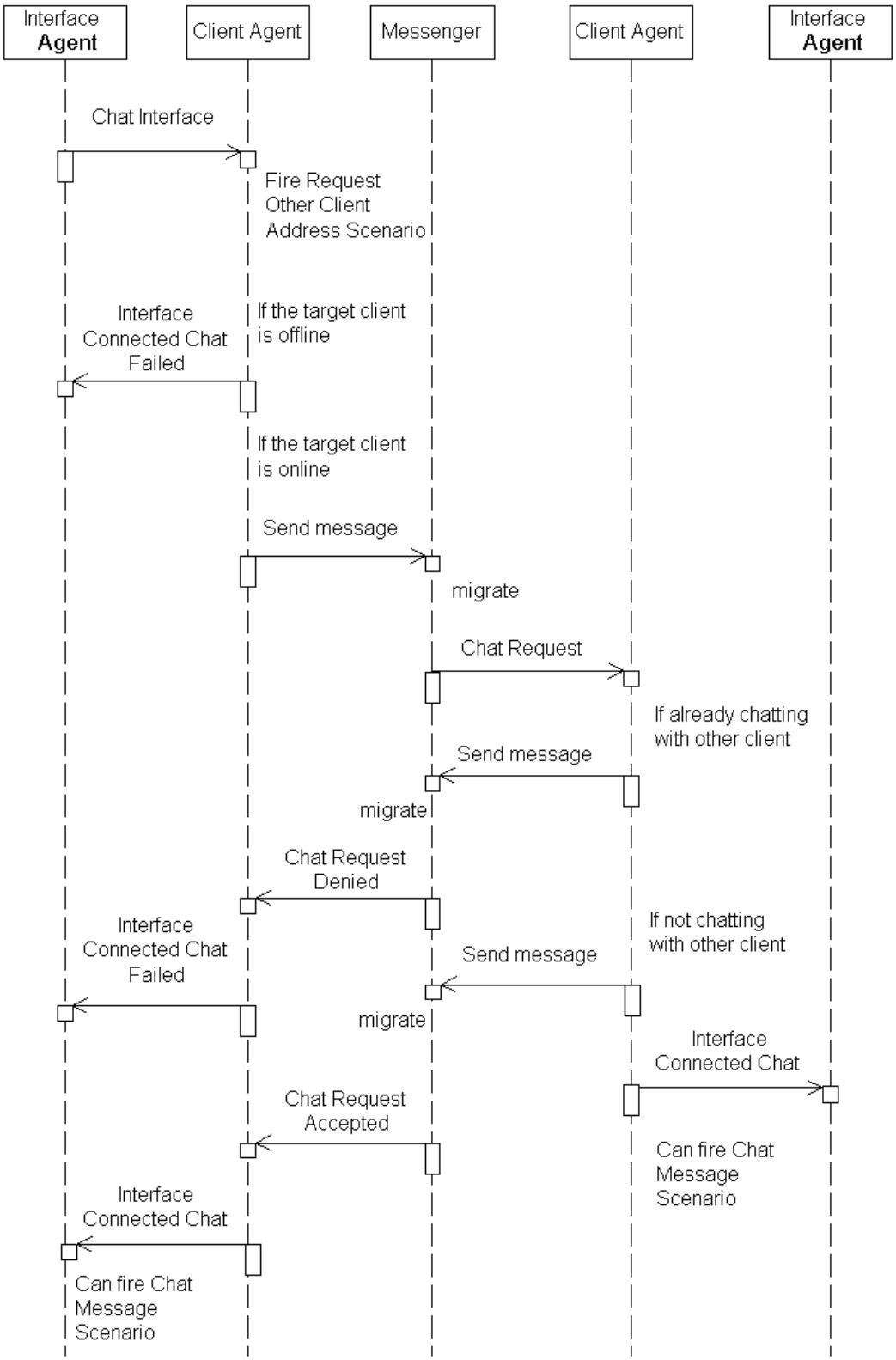

Figure A.35: Connect Chat Scenario

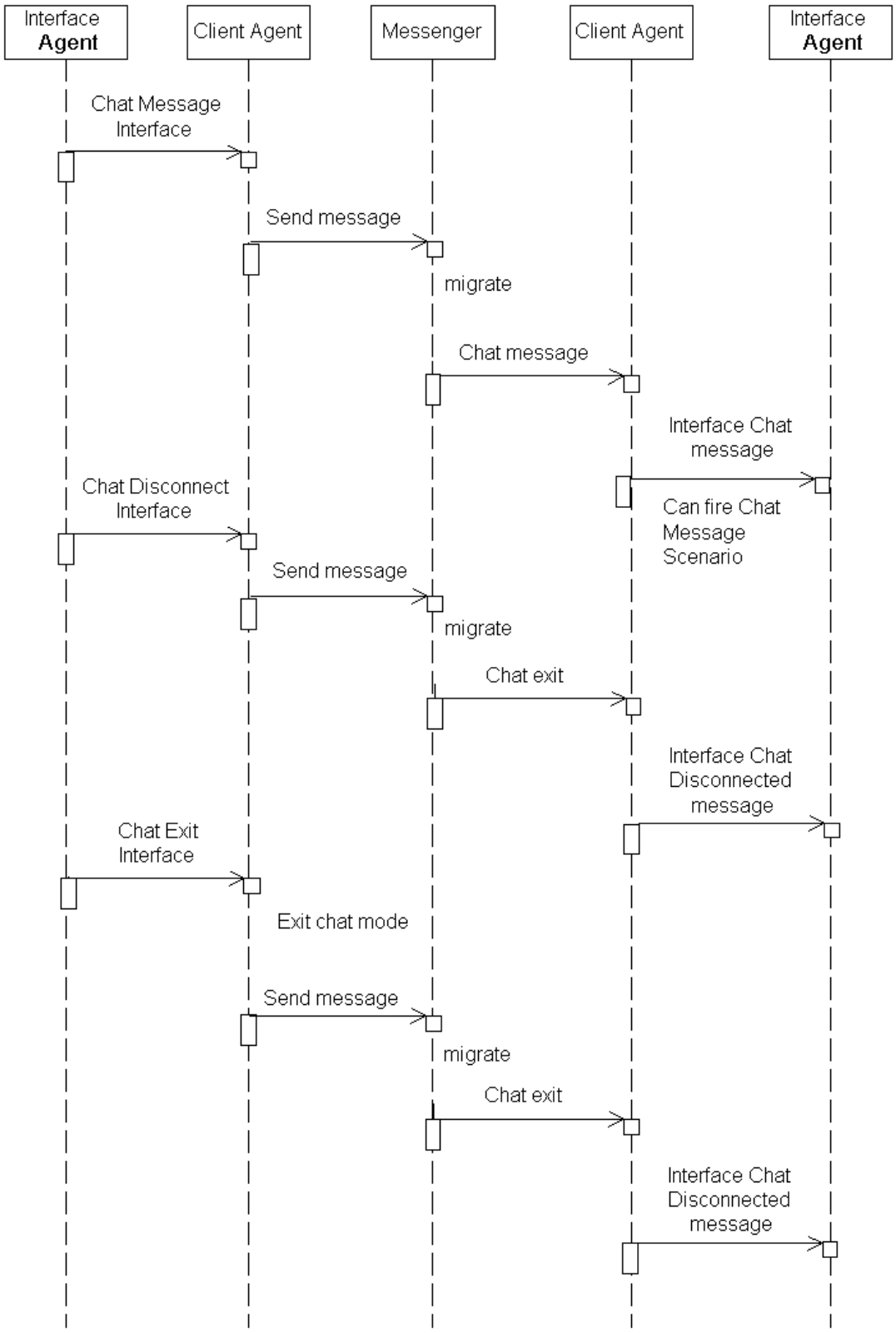

Figure A.36: Chat Message Scenario