UNIVERSITÀ DEGLI STUDI DI FIRENZE
Dipartimento di Sistemi e Informatica
Dottorato di Ricerca in Informatica e Applicazioni
XXII Ciclo
Settore disciplinare INF/01

DYNAMIC NETWORKS: ALGORITHMS, SIMULATION, AND EXPERIMENTS

CARLO NOCENTINI

Supervisor: *Pierluigi Crescenzi*

PhD Coordinator: *prof. Rocco De Nicola*

April, 2010

Carlo Nocentini: *Dynamic Networks: algorithms, simulation, and experiments,* Dottorato di Ricerca in Informatica e Applicazioni, © April, 2010

ABSTRACT

A *dynamic network* is a network in which nodes and links vary with respect to time because of events different from failures. In this thesis we deal with algorithmic aspects of *dynamic networks*. In particular, we have chosen two specific kind of dynamic networks which have become very popular in the last years: peer-to-peer (in short, P2P) networks and mobile ad hoc networks (most commonly known as MANETs).

Contribution to the P2P field. P2P networks are implemented by means of the overlay network concept, which is a logical abstraction of the physical network in which a direct logical link between two nodes might correspond to a multi-hop physical path. One of the most challenging issue in P2P networking is how to organize and retrieve the resources shared by the overlay network: distributed hash tables (in short, DHT) are a common solution to this problem. On the other hand, the growing popularity of P2P applications has determined the need of environments helping the programmers to develop them. One of the most popular such environment is JXTA, a framework developed by Sun to give to developers an instrument to build their P2P applications. As a first result of this thesis, we propose the integration of a popular DHT algorithm (Chord) within the JXTA framework. Indeed, JXTA provide the developers with a resource sharing protocol producing a structure similar to a DHT, but not as good as a pure DHT. The resulting new JXTA implementation, called *JXTACh*, is experimentally compared with the original one: the results give strong evidence to the fact that the JXTA resource sharing protocol performances can be improved up to one order of magnitude by replacing it with a pure DHT.

Contribution to the MANET field. Mobile ad hoc networks are wireless networks where connectivity relies totally on wireless mobile devices. Two nodes are connected if they fall inside each other transmission (visibility) range. A possible way to build MANETs is using Bluetooth technology. Part of this thesis is dedicated to the *device discovery* Bluetooth protocol, according to which each device tries to connect to other devices within its visibility range in order to establish reliable communication channels yielding a connected topology. We report analytical and experimental proofs that when the transmission range of a node is a vanishing function of the number of devices, full connectivity can be obtained by letting each node to connect to a constant-size subset of visible neighbors. Finally, MANETs dynamic behavior raises up many issues due to their mobile feature: it is then important to have instruments to model such a feature. The last part of this thesis is dedicated to MOMOSE, an environment for the simulation of *mobility models*. MOMOSE already contains the most popular mobility models, but it also offers the possibility to add new mobility models

and to record data to evaluate the performances of the protocols executed during the simulation.

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Pierluigi Crescenzi who introduced me to the research giving me the possibility to appreciate it.

I would like also to thank Dr. Michele Loreti who supported and encouraged me during the whole PhD program period.

My thanks go also to all the people I met at Mathematics Departement of Tor Vergata University in Rome, which supported my research activity through the FP6 project AEOLUS. Thanks a lot to Prof. Miriam Di Ianni, Dr. Gianluca Rossi, Prof. Andrea Clementi, and finally to Dr. Francesco Pasquale with whom I shared very pleasant moments during the AEOLUS meetings and schools.

I thank very much Prof. Andrea Pietracaprina and Prof. Geppino Pucci, their contribution to my research activity was very important.

Many thanks to the reviewers of the thesis, Prof. Roger Wattenhofer and Prof. Giuseppe Persiano for their helpful and constructive comments.

I sincerely thank the Coordinator of my PhD program Prof. Rocco De Nicola for the encouragement and the suggestions he gave me during these years.

Furthermore, there has been a group of people which had a very important role during my period as a PhD student: people I met in the PhD student office. It is in general an hard problem finding real friends, my experience there has been a wonderful exception. Thanks Carlotta, Elena, Liliana, Lucia, Maddalena, Sara, Tania, Alessandro (both), Andrea, Francesco (the three of you), Leo, Massimiliano, Stefano, you know exactly how much I value your contribution and above all your Friendship.

Finally, thanks to my family for their encouragement and to Alicja, who is my first supporter and shares with me successes and failures.

CONTENTS

1 DYNAMIC NETWORKS 1
- 1.1 Wired Networks 2
- 1.2 Wireless Networks 5
  - 1.2.1 Infrastructure based networks 6
  - 1.2.2 Infrastructureless networks: Ad hoc networks 7
  - 1.2.3 Mobile Ad hoc networks 9

2 PEER TO PEER NETWORKS 11
- 2.1 Introduction to P2P overlay networks 11
- 2.2 Unstructured P2P Overlay Networks 13
  - 2.2.1 Communicating in an unstructured network 13
  - 2.2.2 Modeling unstructured networks 15
  - 2.2.3 Applications based on unstructured overlay networks 16
- 2.3 Structured P2P Overlay Networks 20
  - 2.3.1 A taxonomy for structured P2P overlay networks 20
  - 2.3.2 Examples of structured P2P overlay networks 23

3 JXTACH 47
- 3.1 Introduction 47
- 3.2 Related work 48
- 3.3 Understanding JXTA and Chord 49
  - 3.3.1 JXTA 49
  - 3.3.2 Chord 60
- 3.4 JXTA rendezvous service reverse engineering 67
  - 3.4.1 The Rendezvous Peer View 67
  - 3.4.2 The Shared Resource Distributed Index 70
  - 3.4.3 The Walker 72
- 3.5 JXTACh design and implementation 78
  - 3.5.1 The distributed hash table 78
  - 3.5.2 The distributed index 87
  - 3.5.3 The walker 90
  - 3.5.4 A simple utility class 92
- 3.6 Experimentation phase 92
  - 3.6.1 Static case 98
  - 3.6.2 Dynamic case with gentle disconnections 99
  - 3.6.3 Dynamic case with abrupt disconnections 100
  - 3.6.4 Incidence of negative queries 101
- 3.7 Conclusions 102

4 MOBILE AD HOC NETWORKS SIMULATION: MOMOSE 105
- 4.1 Introduction 105

4.2 Mobility models overview 107
4.2.1 Random based mobility models 107
4.2.2 Mobility models with temporal dependency 109
4.2.3 Mobility models with spatial dependency 111
4.2.4 Mobility models with geographical restrictions 114
4.3 A Description of MOMOSE Features 119
4.4 The Software Architecture and the Simulation Execution Flow 125
4.4.1 Extending MOMOSE 127
4.5 Java and C++ Performance Comparison 128
4.6 Related Work and Performance Comparison 129
4.7 Two Case Studies 132
4.8 Conclusions 134
5 BLUETOOTH AD HOC NETWORKS 135
5.1 Bluetooth overview 135
5.1.1 Bluetooth architecture 136
5.1.2 From piconets to scatternets, the Bluetooth topology 141
5.1.3 From constant r to $r(n)$ 159
5.2 Connectivity of $BT(r(n), c(n))$ 161
5.2.1 Case $\gamma_1 \sqrt{\ln n / n} \leqslant r(n) \leqslant n^{-\epsilon}$ 162
5.2.2 Case $n^{-\epsilon} < r(n) \leqslant 1$ 164
5.3 Achieving $c(n) = 3$ using a double choice protocol 166
5.4 Experiments 167
5.5 Conclusions 171
6 CONCLUSIONS 173

BIBLIOGRAPHY 179

1 DYNAMIC NETWORKS

In the course of the last century the need of information has become a major challenge in human society. As a consequence the necessity of having a mean to gather, process and distribute information as fast as possible and, in many cases, to the largest possible number of people, has arisen. At first, the response to this need was provided by the telephone network, which gave the possibility to communicate instantaneously even with people located thousand of kilometers away. This kind of network answered to the need of fast communication; the latter need, instead, was satisfied by radio and television networks, which were able to communicate with a huge number of people at the same time. Finally, thanks to the computer evolution, the mutual connection of many computers in order to form a *computer network*, was made possible.

Thus, we can say that this kind of network puts together the features of telephone, radio and television networks. Indeed, it takes the interactivity from the first one, while it takes the possibility to reach a lot of people at the same time (especially with the recent growth of personal computers technology and the explosion of Internet) from the others. Still, computer networks are not a mere intersection of old technologies; they opened up a new wide perspective which gave birth to an entire new field of computer science. It is also worth observing that, in the present days all the technologies are converging into a big internetwork (which is Internet). Indeed, classic telephone might be substituted by *Voice over IP* (VoIP) technology, whereas radio and television networks have their counterpart in streaming on the Internet.

A network is composed by nodes and links. As a dynamic network we consider a network in which nodes and links vary with respect to time. If we limit ourselves to this definition it is clear that it would be impossible to observe a non dynamic network in the real world. We then eliminate from the set of dynamic networks those in which the dynamism is due mostly to the failures. Specifically, we consider that a network is dynamic when the dynamism is caused by the nature of the network itself.

In the remainder of this chapter, we will give a brief overview of the computer networks, including those which are not dynamic according to the previous definition. For each network described, we will indicate if it can be considered dynamic or not, giving the reason of what was stated. We divided our description into two big subclasses, wired networks and wireless networks.

1.1 WIRED NETWORKS

There are many ways of classifying computer networks, one of the most popular is a classification based on their geographical distribution. When we talk about a *Personal Area Nertwork* (in short, PAN) we mean a small network limited to a single machine connected with its peripherals (i.e. printers, scanners).

If the network is limited to a single office or to a small company located in a single building then it is classified as a *Local Area Network* (in short, LAN). The purpose of such a network is mainly to connect personal computers and workstations in order to share resources and exchange information. The most popular architecture for LANs is Ethernet (IEEE 802.3), which is a bus-based broadcast network. In a bus-based broadcast network, the devices connected to it share the same channel to communicate, and each of them can send whenever it wants to. Conflicts are resolved in the following way: if two or more packets collide, each computer waits a random period of time after which it transmits again.

If the network is located in a city it is classified a *Metropolitan Area Network* (in short, MAN). An example of a metropolitan area network is the cable television network which is frequent in many cities in United States. This kind of architecture was intended to be used in places in which the antenna signal could not arrive. After the Internet explosion, cable TV operators added the possibility of having a bidirectional connection to the Web using a part of the spectrum which was unused by the TV signal. In such manner, the cable TV network was transformed into a metropolitan area network.

When a network spreads into a large geographical area we talk about a *Wide Area Network* (in short, WAN). A wide area network is composed of many LANs or of single machines interconnected via a communication subnet. Computers attached to the LANs are usually called hosts, while the communication subnet is composed of switching elements named routers, which are responsible for the routing of messages sent through the network by hosts.

Since there exist many different kinds of networks, and since a world wide communication is what is needed, therefore devices under a type of network shall be able to communicate with devices under a different type of network. To fulfill this need we have to connect different networks through machines called gateways which are in charge of the necessary translation. What we get is an *internetwork*, or more synthetically an *internet*. The most common configuration of an internet is formed as a set of LANs interconnected via a WAN. The most popular implementation of an internet is Internet (the convention to differ the two concepts is by writing the implementation with a capital letter).

In Figure 1.1 we represent the classification of the above described networks. All of those networks belong to the class of static networks; indeed, their dynamism is mostly due to the failures of the devices connected to them.

Distance	Area	
1 m	Square meter	PAN
10 m	Room	
100 m	Building	LAN
1 km	Campus	
10 km	City	MAN
100 km	Country	WAN
1000 km	Continent	
10000 km	Planet	Internet

Figure 1.1: Network classification according size.

As an example we can take Internet, intended as the physical internetwork composing it. When we talk about Internet, an activity which depends largely on the network structure is *routing*. In works like [1, 2, 3] there is the description of the dynamics of Internet routing through the analysis of the Autonomous Systems (AS) topology evolution. In particular, in [1] the authors describe a set of *routing pathologies*, and observe that the most likely pathology that occurs in end-to-end routing is an infrastructure failure or a temporary outage (at least 30 seconds).

A different case is seen when a logical network built over the networks we have just seen is provided. In such networks, called *overlay networks*, links interconnecting nodes at the logical level correspond to paths at the physical level. The World Wide Web is an example of an overlay network built on top of the Internet, where links between pages are the logical links which in the physical level become long paths crossing even continents.

Another example of an overlay network is constituted by peer-to-peer networks (in short, P2P). In this kind of networks the nodes are at the same time providing and using shared resources (i.e. CPU, memory, storage space) inside the network. The topology of a P2P network is built over the physical network, and the links between peers are logical. In this case as well, the proximity of two peers does not imply the geographic proximity between the machines hosting the peers in the physical network.

In Figure 1.2 we show a graphical representation of a simple overlay network, a dashed line means that the logical node is physically contained in the machine to which it is connected (e.g. logical node a is in the machine M1), while a thick line highlights the correspondence between the logical link connecting a to e

and the path $(M1, M2, M4, M5, M6)$ which is one of the possible physical paths connecting $M1$ with $M6$.

Overlay networks dynamism is due not only to the underlying physical network failures. Indeed, there are many others causes which usually depends on the purposes of the logical network. Let us consider the World Wide Web: in [4] the authors present a classification of Web dynamics that should be taken into consideration by search engine designers. Three of these dynamics are the following ones.

- *Dynamics of Web size*: the need of information presentation and information exchange in the commercial, government and educational sectors, causes the growth of the Web size at an exponential rate.
- *Dynamics of Web pages*: the increasing Web usage and the need of keeping the information on the web pages valuable, have the effect of creating a situation in which new pages come into existence, others are removed for a short time or forever, some, instead, are modified or migrated to a different URL.
- *Dynamics of Web link structures*: links between Web pages are continuously appearing and disappearing because of various reasons.

In P2P systems a source of dynamism could be generated by the intrinsic nature of these systems, which are usually used by peers to cooperate in order to provide a service which should result in an improved benefit (with respect to the same service implemented by a single node) to the whole network. This would be true in a perfect world: in a real P2P application, however, there exists a phenomenon called *free riding*. A *free rider* is a node which uses the P2P network to get the maximum benefit without giving support to the network in return. This can be described as a peer which remains connected to the network just for the time necessary to get the service and, when finished, disconnects from the P2P network. This kind of nodes are, unfortunately, a vast majority in a P2P network. In [5] it was observed that in 2006 more than 70% of nodes participating to the eDonkey P2P network were free riders. This kind of percentage results in various problems, such as the high dynamism introduced by the high churn of free riders connection and disconnection. The problem of how to limit the incidence of the free riders phenomenon is not the object of this thesis: we limit ourselves to observe that one of the most popular solutions is to create incentive mechanisms in order to persuade the free riders to cooperate (a short review can be found in [6]).

When we have to deal with a high dynamism, it is important to find proper protocols and data structures to face it. In P2P networks, a largely studied and used instrument to manage the resource sharing task is the *distributed hash table* (in short DHT). This kind of distributed data structures defines a way to

decentralize the resource storage in a distributed system and a way to retrieve it. Indeed, the basic operations that a DHT provide are a *store* operation and a *retrieve* operation. There exist many DHT protocols such as Chord, CAN, Kademlia, Pastry, Tapestry. All the DHT protocols are based on the idea of mapping resource identifiers and nodes identifiers into the same space through an hash function $\mathcal{H}$. In this way we can define a relation $(\mathcal{H}(n), \mathcal{H}(id))$ where node n is said to be *responsible* for the resource identified by the identifier id.

In Chapter 2 we will delve deeper into the case of P2P networks, while in Chapter 3 we will show how the application of a DHT algorithm like Chord, can improve the performances in a resource discovery protocol of a P2P platform.

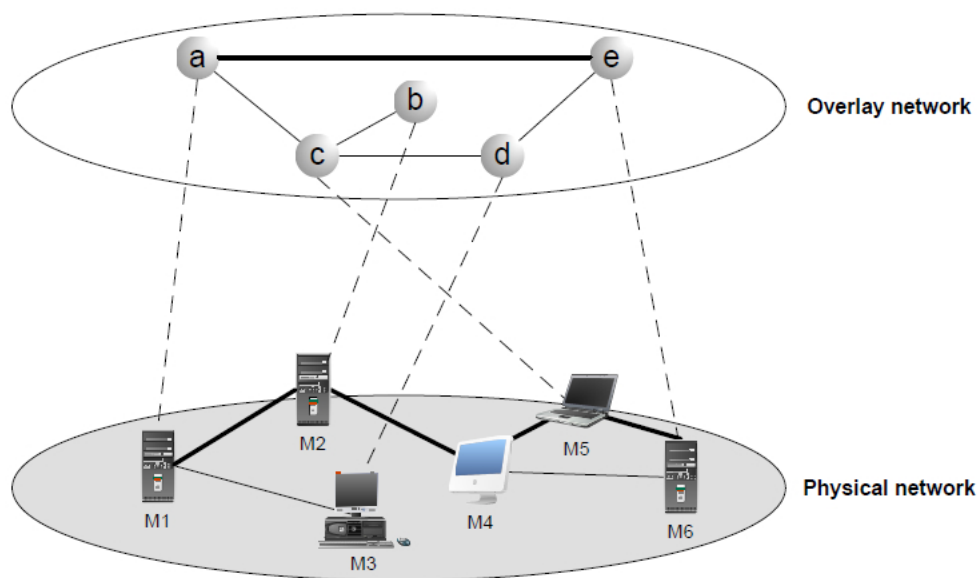


Figure 1.2: A small example of overlay network.

1.2 WIRELESS NETWORKS

In the last few years there was an increase of the use of wireless technology, based on infra-red or radio signals, substantially different from networks described in the previous section, in which links are obtained by cables.[1] This is due to several reasons: the fast development and the price reduction of increasingly small devices equipped with wireless capability (i.e. mobile phones, PDAs, laptops, netbooks, sensors); the fact that a wireless network is easier to implant in places in which it would be difficult or impossible to install cables; finally, wireless networks are a perfect way to provide connectivity to portable devices in public places like airports, coffee bars, campus and so on.

1 Digital wireless communication is not a new idea, in fact Guglielmo Marconi's wireless telegraph used the Morse code.

We can apply to the wireless networks the same classification we have seen in the previous section: *Wireless Personal Area Network* (WPAN), *Wireless Local Area Network* (WLAN), *Wireless Metropolitan Area Network* (WMAN) and *Wireless Wide Area Network* (WWAN). However, the wireless nature of the devices give to these networks different characteristics with respect to their wired versions, such as:

- Interference: infra-red signals suffer from interference of sunlight and heat sources, radio signals are more difficult to interfere with, but there are some possibilities of interference with other electronic devices or with other wireless devices using the same frequency.
- Slower transfer rate due to a smaller bandwidth.
- Considerable changes of network conditions, because of for example:
  - Higher data loss due to interference.
  - Frequent disconnections which can be caused by users' movement.
- Limited computing and energy resources.
- Limited service coverage.
- Limited transmission resources.
- Weaker security, because of an easier interception of radio signals on behalf of attackers, which results in a much more difficult implementation of security.

However, there exists another way of classifying wireless networks worth mentioning: by network formation and underlying architecture. Following this classification we can divide wireless networks into two big classes: *infrastructure based* network and *infrastructureless* network.

1.2.1 *Infrastructure based networks*

This first class of wireless networks is composed by wired devices forming the network backbone providing services to the wireless devices. An example of this network is a wireless local area network where a wireless access point connected via cable provide wireless connectivity, this could be the case of a university department, in which the wireless network provides the possibility to have a connection to the visiting people (without being forced to use a cable). This kind of network falls into our definition of static network, in fact here we just add the failures due to the wireless nature of the devices. Figure 1.3 shows a simple graphical representation of a WLAN.

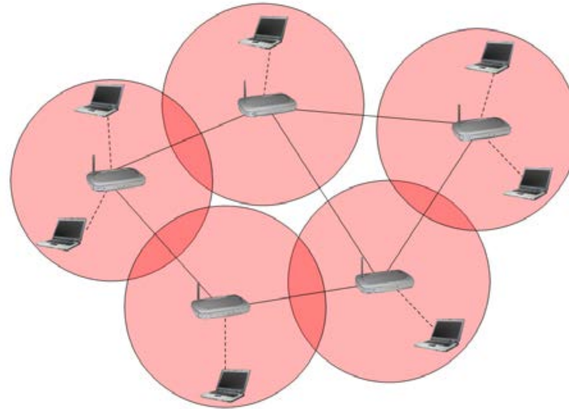


Figure 1.3: An infrastructured WLAN, continuous lines represent wired links, dashed ones represent wireless links.

1.2.2 *Infrastructureless networks: Ad hoc networks*

An infrastructureless network lacks the wired backbone. Connectivity is demanded to the wireless devices in a point-to-point manner only. These networks are called *ad hoc networks* and they are formed dynamically through the cooperation of an arbitrary set of independent nodes, where two nodes communicate if they are within each other transmission range. A technology building ad hoc networks is the Bluetooth technology. Ad hoc networks are designed to have an existence limited on time for extemporaneous services or applications.

In this kind of network there are a lot of sources of dynamism. Let us take the example of a sensor network: we can enumerate at least two principal case of dynamism,

- *Battery consumption*: sensing devices are equipped with a battery and the energy saving policy has a direct effect on the topology created by the devices. Indeed, to decrease energy consumption of sensors, they are usually put in a sleeping state, in which they stop to communicate, therefore it is obviously translated into a high degree of dynamism.

- *External events*: sensor networks are deployed in many different environments. A classical example could be a fire sensing network in buildings or in open-air (e.g. forests). In these environments there is a high possibility of damages leading to the destruction of sensors, therefore, the sensor network have to deal with this kind of dynamism.

The two cases shall be enough to classify ad hoc networks as a part of the class of dynamic networks. These networks raise a larger number of challenges with respect to their wired counterpart, which are, as well, directly connected to

the two above mentioned issues. We list below the most popular, as it shall be clear, they are not completely independent one from another.

- *Energy Saving*: as we have already said, nodes are equipped with batteries and every algorithm designed for this kind of network should take it into account. The main objective is to save as much energy as possible. In other words, algorithm must be energy-efficient.
- *Coverage*: this is the problem of determining the area covered by either the sensors or the signal of the nodes.
- *Localization*: in many applications it is important for a node to know its geometric position in the network area.
- *Node Placement*: in some application we have to take into consideration the places in which to put nodes, mostly because of environmental reasons or need of a thorough coverage.
- *Neighbor discovery*: this is the first operation to be performed in order to build a network and it highly influences the network performance. It is important for a node to know its neighborhood, hence, a routing protocol can benefit from a well designed neighbor discovery algorithm.
- *Density Control*: this is directly connected with energy saving, in fact a density control algorithm should manage the process which determines when a node should be operable (awake) and when inoperable (asleep). This should be done while keeping the network connectivity and the coverage.
- *Security*: when we talk about wireless networks, security is a much more challenging problem, since radio signals are easier to be intercepted by attackers.
- *Topology control*: an ad hoc network lacks of a central infrastructure, therefore its topology is not fixed. Thus, the network needs to be able to self configure in an appropriate topology for which routing protocols are implemented. The quality of the topology can be measured in terms of connectivity, energy-efficiency and throughput. The first two are connected to the number of edges and to the maximum degree of any node (for this concept we will see more in Chapter 5). The last one is a measure that says how much information can be transported over the network.
- *Routing*: Routing is directly connected to topology control, indeed, routing algorithm should take advantage of the connectivity, the energy-efficiency and the throughput provided by the underlying topology.

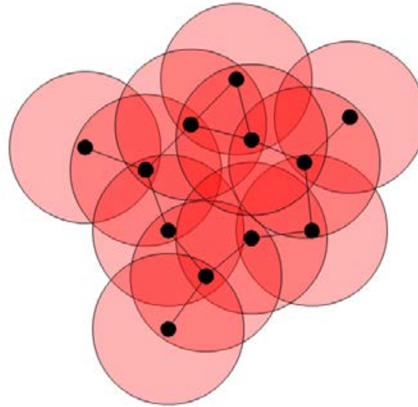


Figure 1.4: A schematic representation of an ad hoc network.

1.2.3 *Mobile Ad hoc networks*

A great possibility given by the wireless devices, which can not be provided by the wired ones, is mobility. Wireless technology, in fact, is directly connected to portable technology which is, by definition, capable of movement. Ad hoc networks in which the devices are moving are named Mobile Ad hoc networks (in short MANET).

Mobility adds far more dynamism to a wireless network. Indeed, the properties and challenges described above are still valid in this context although they have also to take into consideration the mobility. Furthermore, the mobility property adds new possible applications.

A motionless ad hoc network is quite limited from a practical point of view. It could be used just to connect devices which are not intended to change their location, like for example a workstation connected to other workstations or to its peripherals.

When we add mobility, the number of ways a network can be deployed increases dramatically. As an example, we can list four different applications for MANETs

- *Community Networks*: a MANET could be deployed to form a communication channel between group of friends traveling together in a touristic site.
- *Emergency response networks*: a MANET could be deployed in areas in which a natural disaster has destroyed the wired infrastructure, like an earthquake or a fire, to establish connection between rescue forces.
- *Vehicle networks*: a MANET could be created through the installation of wireless devices on vehicles like cars or trucks to implement a distributed traffic monitoring system which dynamically informs the drivers about

the situation along their routes (this kind of network is known as VANET, *Vehicular Ad Hoc Network*).

- *Sensor networks*: sensors networks can be composed by mobile devices, the motion of the sensors can be determined either by the environment (e.g. if we deploy a sensor network in water, sensors are subject to the water flow), or by the fact that the sensors themselves might be capable of movement.

All the challenges are affected by the mobility, but those which are directly connected to the network topology have now to deal with its dynamic nature of it. The topology of MANET is in fact, because of its intrinsic nature, continuously changing. There exists a lot of protocols facing the issues raised up by the mobility, it is then important to have both instruments to model the behavior of MANETs and good metrics to measure the most important parameters influencing the system. Furthermore, it is useful to have the possibility to test the correctness or efficiency of protocols using these instruments, through a simulation environment. In Chapter 4 we will go deeper into the field of simulation of MANETs, showing an overview of the field and presenting a simulation tool as an example.

The remainder of this thesis is structured in the following way

- Chapter 2 gives an overview of P2P overlay networks, describing the most popular protocols developed to organize such networks.
- Chapter 3 describes how the injection of Chord, a well known DHT protocol, to JXTA, a popular framework for the implementation of P2P applications, have resulted into a remarkable increase of performances in terms of lookup of resources in the system.
- Chapter 4 move from wired networks to wireless networks, specifically to the MANETs context. It gives, in the first part, an overview of the literature produced in the mobility models field. Then, in the second part, describes MOMOSE, a highly flexible and easily extensible environment for the simulation of mobility models.
- Chapter 5 describe a technology which is used to build mobile ad hoc networks: the Bluetooth technology. Specifically, it will be considered the problem of the device discovery phase, which in mobile context, acquires particular importance.

2 PEER TO PEER NETWORKS

In this Chapter we will give an overview of P2P overlay networks and their inner organization, which can be *structured* or *unstructured*. Unstructured network does not have any topology constraint, they evolve as general graphs, while the structured P2P networks are organized through a special distributed data structure called *distributed hash tables* (in short DHT), which give to the overlay network a specific topology.

2.1 INTRODUCTION TO P2P OVERLAY NETWORKS

The most popular architecture in network systems has been, in the first years of development, the client-server model, a model in which the roles of the service provider and of the service user were clearly defined and distinguishable. In the last years the structure of the networks has become more complex. This phenomenon has resulted in many consequences, one of which is that the client-server model has turned to be less effective when by itself. That's why many different kinds of architectures have been created in order to support the old model in a way which allow exploiting the increased complexity of the networks. P2P is one of these new architectures. Sometimes people make a mistake and watch P2P systems as alternatives to the client-server based systems. This is not a correct way of perceiving it, P2P should be watched as a technology intended to exist in parallel to the old model.

In P2P networks nodes are spread in a flat mode (almost) without any hierarchical structure. The peers are not only server or client, the two figures are merged into the same node, called *peer*. Peers are at the same time providers and users of services available on the network, furthermore peers cooperate to provide services with a higher performance taking advantage of their distributed nature.

In [7] the authors give an abstraction of a P2P overlay network architecture, we (see in Figure 2.1 for a graphical representation of this scheme). Five layers are used to describe the components of the P2P overlay network structure

- *Network Communication Layer*: describes the hardware and software concerning the network of the devices attached to the P2P network. Such devices might be PCs, PDAs, cellphones, sensors. Namely it threats the network interface of each physical node.

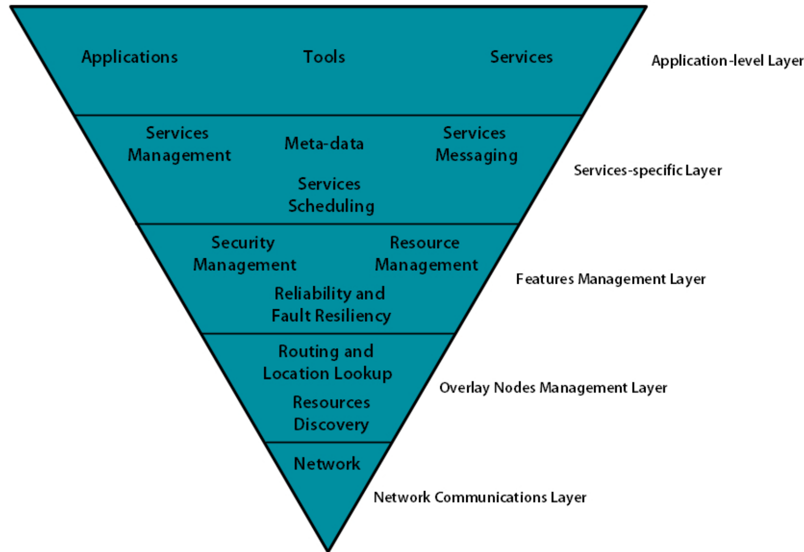


Figure 2.1: Abstraction of the P2P overlay network architecture.

- *Overlay Nodes Management Layer*: in this layer we already talk about *peers*, here reside mechanisms which enable to find peers and to route messages among peers in the overlay network.
- *Features Management Layer*: deals with the robustness and security of the system (availability, reliability, fault resilience).
- *Services-Specific Layer*: is the layer acting as an interface between the applications and the P2P's low level infrastructure, hiding to the upper level the management of messages, task scheduling, and real organization of data (through meta-data).
- *Application-level Layer*: in this layer reside all the applications, tools and services exploiting the underneath layers.

P2P overlay networks can be subdivided into two subclasses, according to the fact if they have or not some sort of internal organization among peers. When the peers are spread over the physical network without a determined topology and the contents are placed without a specific policy, the overlay network is said *Unstructured*. If, otherwise, the overlay network formed by peers, has a predetermined topology and there exists rules for the content storage, it is called *Structured*. We will give a brief description of the first type, and then we will concentrate on the second type as it contains what we have called distributed hash tables.

2.2 UNSTRUCTURED P2P OVERLAY NETWORKS

In this kind of overlay networks, the peers does not have a deterministic topology, in Figure 2.2 we show a graphical representation of an unstructured topology. This means that there are no particular rules to establish a connection between two peers.

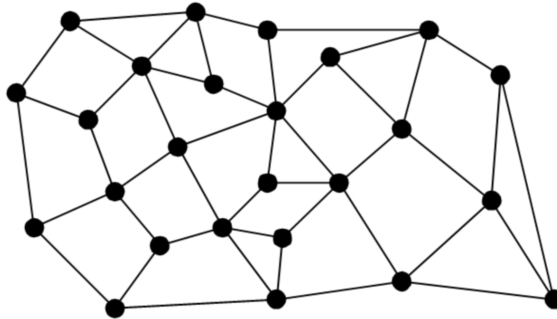


Figure 2.2: Unstructured topology of an overlay network.

2.2.1 *Communicating in an unstructured network*

In an unstructured topology, peers have informations only about the peers which are directly connected to them, namely about their *neighbors*. Then, when an information has to be propagated through the network (e.g. a query), the simplest way to do it is by using the *flooding* approach. The node that originates the information to be spread, sends it to each of its neighbors, which, in turn, propagate the information to their neighborhood (Figure 2.3). Each message used during the flood is associated with a time to live. When the time to live expires, the propagation of the message ceases as well. The time to live is used to avoid an unnecessary propagation of the flooded message. When we think about a search algorithm, a way to alleviate the problem of redundant messages is to assign a low time to live to each message. If the search is successful, then the algorithm terminates. If it is not, the search is restarted with a greater time to live (of course there is an upper bound over which the search is considered failed). This strategy is known as *iterative deepening* or *expanding ring*.

A different approach is the *random walk* policy. A peer initiating the propagation chooses at random a neighbor to which the message is sent, the process is repeated for each peer until a time to live expires. If we are performing a search operation, the search is successful if we encounter the object of the research along the random walk, otherwise the last node receiving the query has to send a failure message (see Figure 2.4). To raise the probability of having a success, more than one random walk messages can be sent through the network.

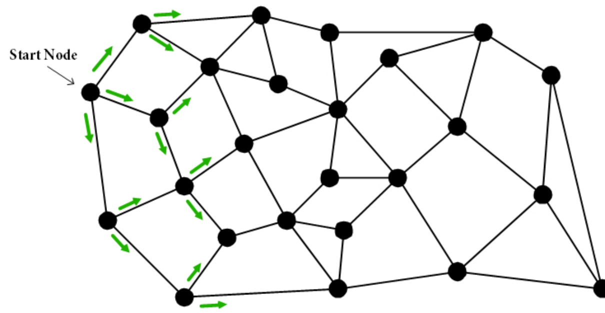


Figure 2.3: Flooding over unstructured topology.

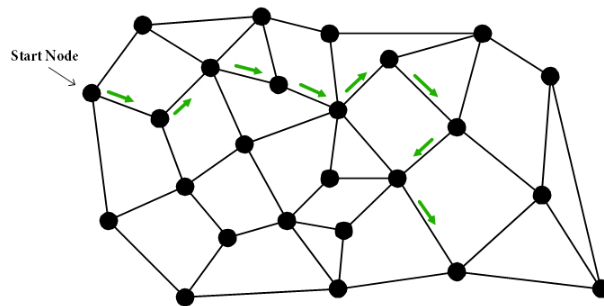


Figure 2.4: Random walk over unstructured topology.

We presented just two of the most simple examples of message routing policy in unstructured overlay networks, focusing on the search problem. However, we can cite other strategies, such as *guided search* which is based on the concept of "goodness" of a neighbor, which is measured in various ways keeping track of network statistics.

The unstructured P2P overlay networks are good if we have to search contents which are highly replicated. Indeed, if a resource is rare, a lot of peers will have to be contacted. In this case, then, the unstructured approach looses in performances.

2.2.2 *Modeling unstructured networks*

We give a brief description of the ways used to model the unstructured topologies. The natural structure used to model a network is a graph. In case of an overlay network, peers are the vertices of the graph while the logical connection between them are the edges.

When we consider the graph modeling an overlay network, we are interested in many of its properties, among which we cite the *degree* distribution and the *diameter*. The degree of a vertex in a graph is the number of vertices connected to it through the edges, while the diameter is the maximum among the lengths of shortest paths between pairs of vertices of the graph. The degree affects the load distribution, while the diameter is connected to the message routing.

The most popular graph used to model an unstructured network is the *random graph*, in which the edges are generated uniformly at random given the set of vertices, each edge exists with a given probability p. The advantage of these graph are is in its exceptional analytical properties which make them good instruments to measure some important characteristics of the network. However, they are not able to capture two important aspects of a real network, the *clustering* property and the real distribution of node degrees.

The clustering is the property of a network such that two nodes are more likely to be connected if they have a common neighbor. This property is measured by the *clustering coefficient*. If we think about the definition of a random graph we see that this property is not modeled.

If we study the real distribution of the nodes' degrees we find that they follow a *power-law* distribution. In case of a random graph, instead, we observe a node distribution following a *Poisson* distribution.

It is mainly because of these two reasons that there has been an effort to find graphs capturing these two important properties. Power-law random graphs and scale-free graphs are solutions which goes this direction ([8, 9]).

2.2.3 *Applications based on unstructured overlay networks*

The reason why peer-to-peer has become so popular is surely the use of this approach to implement file sharing applications. The majority of this kind of instruments are based on an unstructured overlay network. In the followings we describe shortly the main characteristics of the most popular applications.

Gnutella

Gnutella was the first file-sharing system implementing a real P2P architecture. In this application peers are embedded in an unstructured network. Peers, which are called *servent*, communicate with each other using a pure flooding approach limited to a certain radius and the data placement does not follow any particular rule. This structure suffers from the drawbacks we have seen in the flooding description. To help with the improvement of the routing performance and with the scalability of the system, the latest versions of Gnutella implement the concept of *ultra-peers* which are peers with a higher bandwidth used to process query on behalf of normal peers called *leaf* (see Figure 2.5). In this approach the queries travel through the ultra-peers network and the leaf peers are merely a starting or an arrival point.

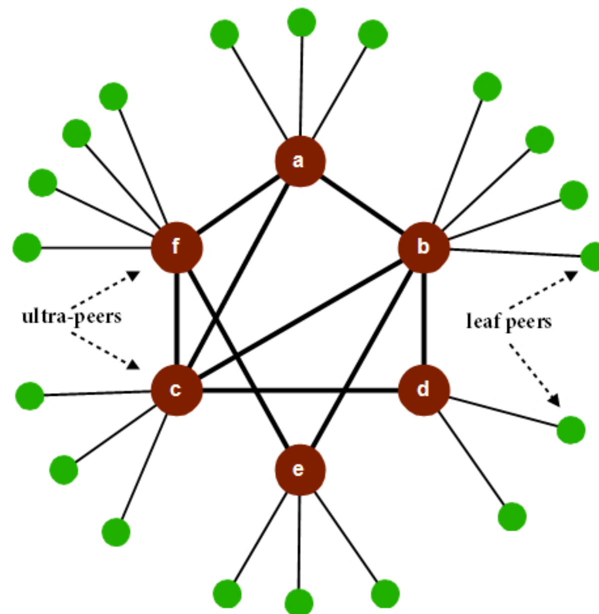


Figure 2.5: Gnutella ultra-peers structure.

Freenet

Freenet was proposed by Clark in [10] as a file-sharing application with security, anonymity and deniability features. Each user shares a part of its storage space, therefore the system can be considered as a cooperative distributed file system incorporating location independence and transparent lazy replication. The routing algorithm is designed in an adaptive way in order to gradually improve over time the route with a specified content using only local knowledge of the network. Each query is routed through a chain of proxy peers in a similar way to the IP's routing mechanism. A message is given an hop-to-live counter to avoid its indefinite propagation. A mechanism to avoid loops is provided.

The search for a key follows a steepest-ascent hill climbing with backtracking algorithm. We give an example of a Freenet search query propagation in Figure 2.6. To enhance the lookup response time, Freenet implements the following replication mechanism: every time a content is searched and found, it is replicated along the path returning to the peer who generated the request (if we consider Figure 2.6, the content will be replicated at node c).

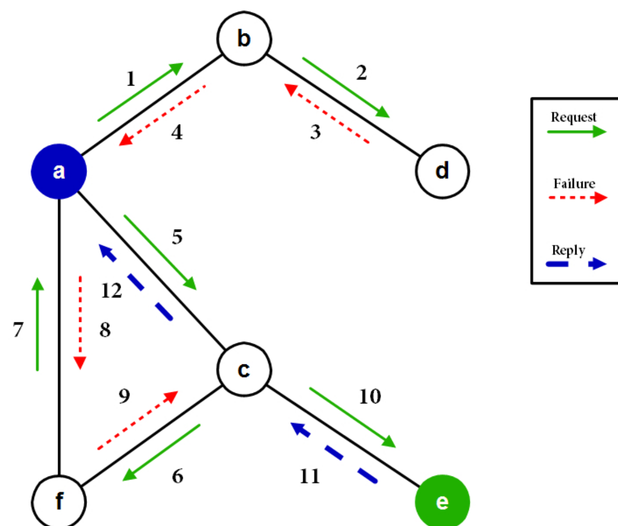


Figure 2.6: Routing of a search query in Freenet.

BitTorrent

BitTorrent is a centralized P2P system, it is mainly used for file-sharing purposes and it is based on an unstructured overlay network. However, it relies on some powerful servers called *trackers*, which manage the communication among peers. In a file-sharing context, the communication is formed by part of files, thus, in BitTorrent files are sliced into pieces of fixed size. When a peer is searching for a file, it needs to know a tracker URL and some other information about the file.

These data are contained inside a file *.torrent* which the peer has to download from a website. The tracker owns information (literally keeps track) about peers containing the file, both those who have the entire file, called *seeds*, and those who have only some part of it, called *downloaders*.

The tracker serves as a landmark, sending to peers asking for downloads a random list of peers containing the file. At this point the communication is managed directly by peers in a flat manner. Each peer connects via TCP to the peers contained in the list received from the tracker, but it is allowed to upload only up to five peers at the same time, choking the other connections (the upload operation is called *unchoke*). The connections are bidirectional. The limited number of communication channels is set to let the peers use a *tit-for-tat* algorithm to achieve a consistent download rate adjusting the set of peers to which it is uploading. Moreover, the use of a tit-for-tat approach is useful to limit the free-riders problem. In Figure 2.7 we represent the main phases of the peer request for a file *b.torrent* from peer p1.

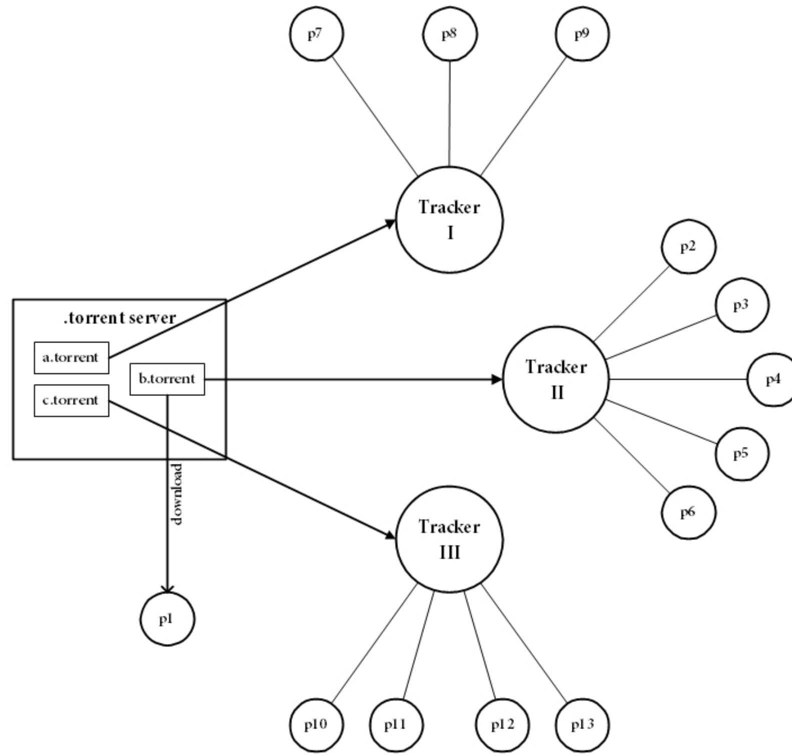
It is clear that this system has a bottleneck and a single point of failure into the tracker servers system. As a first step to solve this problem, the latest version of some BitTorrent clients (Vuze, µTorrent) have added the possibility to use a DHT protocol (Kademlia) to avoid the use of the trackers. It is important to highlight the fact that it does not lead to a decentralized version of BitTorrent, but it could be a path to follow.

eDonkey2000/Overnet

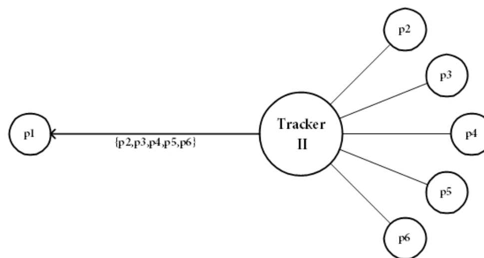
eDonkey2000 is probably the most popular file-sharing P2P overlay network, on which the well know client *e-mule* is based. It uses an hybrid approach, in fact it is both a client-server application and a P2P application.

It is a client-server application in the way it manages the lookup of a file, the network is in fact constituted by a small number of peers which have indexing responsibilities (the *servers*) and by a large number of peers sharing and searching resources (the *clients*). A client firstly connects to a server and then sends to it all the information about itself and about the contents it shares. It is important to state that servers do not contain any data, they just serve as dictionaries for the resources stored inside the clients. When a client peer performs a lookup for a file, it contacts the server which replies with a list of peers owning that file. Then, as it happens in BitTorrent, the communication continues in a P2P manner.

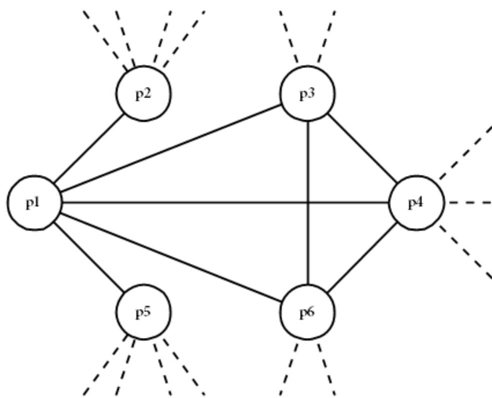
In fact eDonkey2000 network is a P2P application from the moment in which peers start sending to each other files or parts of files. Similarly to BitTorrent, servers are bottlenecks and single points of failures, for that reason, the developers added in this protocol as well, the possibility to decentralize the network through a DHT (e-mule implements a personal version of Kademlia, called KAD).



(a) p1 downloads *b.torrent* which points to Tracker II



(b) Tracker II sends the list of peers owning the file or a part of it



(c) The overlay network is created

Figure 2.7: Phases of connection of BitTorrent.

2.3 STRUCTURED P2P OVERLAY NETWORKS

The first P2P systems followed the unstructured approach. This approach, as we already stated, is not efficient in case of rare resources retrieval, because of the use of flooding. Furthermore, as we have seen in the cases of BitTorrent and eDonkey2000 networks, when there is a trial to solve the problem of flooding introducing a level of hierarchy, the system looses in robustness because of the presence of single point of failures (trackers in BitTorrent, servers in eDonkey2000).

We have already seen that the solution adopted by those two systems is the application of a DHT approach, which is one of the most popular implementations of a *structured P2P overlay network*.

The structured P2P overlay networks are born to address the problems of unstructured overlays, providing the network topology with a particular geometry in order to take advantage from it for routing and maintenance. The word structured, indeed, states a tightly controlled topology. Furthermore, differently from the unstructured overlays, they support *key-based routing* in which object identifiers and node identifiers are mapped into the same address space. The routing of the lookup query for an object toward a node which should contain it, is guided by a structure defined on the address space.

2.3.1 *A taxonomy for structured P2P overlay networks*

The structured overlay P2P networks can be classified according to many different dimensions. We use and report the list of dimensions enumerated in [11]

- **Maximum number of hops taken by a request given an overlay of N nodes**: there exist *multi-hop*, *one-hop* and *variable-hop*. The first category encloses the overlays where the number of hops performed by a request message is usually $O(\log N)$. The second category contains overlay networks where the number of hops is $O(1)$. It is obtained at expense of memory used to maintain the routing structure and it is feasible (and reasonable) at certain conditions concerning bandwidth, churn rate and number of peers. In [12] there is an evaluation of these parameters which enables to compare the two approaches. To give an idea, we report in Figure 2.8 a graphical representation of the zones formed by the values of the parameters where one-hop and multi-hop have their own application.[1]

  The last category contains overlay networks usually used in highly dynamic environments, such as systems where nodes are mobile. In these kind of overlays, each node adapts its behavior in response to the dynamic change of bandwidth, essentially a peer dynamically changes its routing

1 The figure is similar to the one in [12].

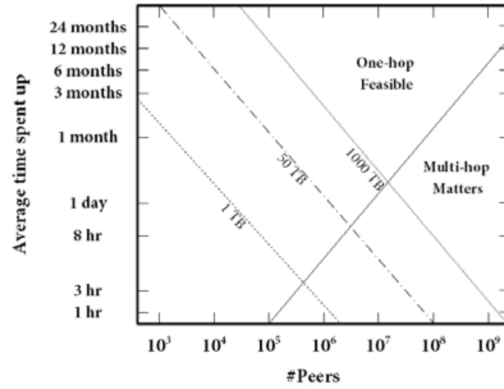


Figure 2.8: Regions where one-hop and multi-hop approaches should be used ([12]).

table size. This results in a *variable* number of hops to be performed by a message in order to reach its destination.

- **Organization of the peer address space**: there exist two categories for the way the address space is organized, *flat* and *hierarchical*. The first one contains the overlay networks that have a single flat address space where nodes are spread in an almost uniform way. The latter category [13] contains networks which organize themselves in two layers. At the bottom level peers are divided into disjoints groups. Each group is organized in an intra-group DHT, while at the top level the groups are organized in an inter-group DHT. Hence, to find a peer, firstly a top-level DHT is used to determine the right group and then the right peer is found through the bottom level DHT (see Figure 2.9).

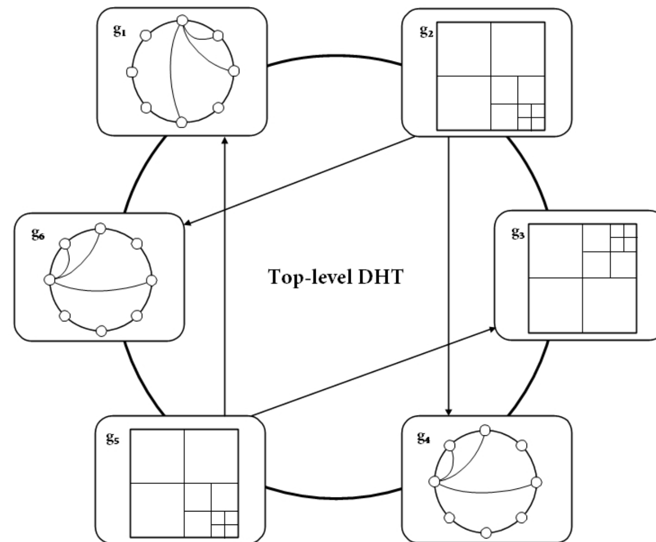


Figure 2.9: Hierachical overlay P2P network. The g_i are the bottom-level groups

- **Next-hop decision criteria**: the criteria of where to route the message from a peer to another is defined by the *distance metric* used in the overlay network. At each step of the routing process, the distance from the target peer has to decrease. Then a metric which converge is needed. Converging metrics used in DHTs include prefix matching, XOR metric, Euclidean distance in a d-dimensioned space, linear distance in a ring and modulo bit shifting in De Bruijn graphs.

- **Geometry of the overlay**: the overlay network topology is a very important characteristic. Each choice influences the properties of the system, such as scalability and performances in terms of number of hops. The main topologies are the following(as listed in [14]):
  - Ring
  - Tree
  - Multi-dimensional Cartesian space
  - XOR-ring
  - Butterfly
  - De Bruijn graphs

  Another important feature concerning the overlay geometry is the node degree variation as the the network size increases. Among the most important categories we can cite the *logarithmic degree graphs* (see examples further in the chapter) and the *constant degree graphs* like butterflies and De Bruijn graphs. Both of the mentioned properties determine the dimension and, in some cases, the growth of the routing tables, as well as the way the overlay maintenance is managed.

- **Overlay maintenance**: a P2P overlay network has to deal with the problem of joining and unjoining peers, known as peer *churn*. A strategy that allows to manage this dynamism in order to keep the overlay topology, has to be defined. There exist many different approaches to overlay maintenance. We can group them into three classes, *active*, *opportunistic* and *passive*. In a system following the active strategy, when a peer detects a failure or a departure of one of its neighbors, it sends to all the peers the updated list of its neighbors. In a system following the opportunistic strategy, each peer periodically sends its neighbor set to a randomly chosen member of this set itself. Finally, a system following a passive strategy, leaves the structure maintenance to take place as a side effect of the other basic operations such as queries and responses forwarding.

- **Locality and topology awareness**: some overlay networks take into consideration the locality of the underlay, in particular, they try to exploit the knowledge of the geographic position of the physical nodes.

2.3.2 *Examples of structured P2P overlay networks*

In this section we are going to summarize the main features of some of the most popular structured P2P overlay networks. We will not cover all the existing systems, but we will limit ourselves to present a subset of work done so far in this field. Our aim is to give an overview of these systems showing different policies.

In this set we will not cover the Chord DHT algorithm, which will be treated in a proper section of the next Chapter because of its fundamental role in this thesis.

In the first place, we will firstly give a description of the work done by Plaxton Rajaraman and Richa (PRR) in [15], which can be considered the first theoretical definition of a structured overlay network.

PRR

In [15] the problem of accessing shared objects in a distributed network is dealt with. The solution is given through a randomized algorithm that tends to satisfy each access request with a nearby copy. Authors show that under a particular cost model

- the expected cost of an individual access is asymptotically optimal
- if the objects are sufficiently large, with high probability,[2] the memory used for objects store dominates the memory used for the algorithm execution.

modelling cost and network

The cost model used is based on the consideration that obtaining an optimal bound in a given network (in any metric of interest, like throughput or latency) depends on a large set of parameters, like edge delays, edge capacities, buffer space, communication overhead, patterns of user communication, and so on. Then the cost model is simplified considering the combined effect of such parameters as a unique function, specifying the cost of communication of a fixed-length message between any given pair of nodes.

In PRR the network is modeled as a graph G where a set $\mathcal{A}$ of m objects is shared among a set V of n nodes, where $m = \text{poly}(n)$. Each node $u \in V$ and each object $A \in \mathcal{A}$ are identified by, respectively, $(\log n)$-bit and $(\log m)$-bit identifiers. The authors assume that $n = (2^b)^k$ where $b, k \in \mathbb{N}$, b define the number of bits read at time and k is the number of bits used to represent an identifier. Namely, b determines the base used to represent the identifiers and each group of b bits is a *digit*. Furthermore, each node x is assigned a $(\log n)$-bit label $\ell(x)$ uniformly at random.

The cost of communication is modeled by a function $c : V^2 \mapsto \mathbb{R}$. Given $u, v \in V$, $c(u, v)$ is the cost of transmitting a single-word message from u to v.

2 with probability at least $1 - \frac{1}{p(n)}$ where $p(n)$ is a polynomial function on n.

The cost of transmitting a message of l words from u to v is $f(l)(c(u, v))$, where $f : \mathbb{N} \mapsto \mathbb{R}^+$ is a non decreasing function such that $f(1) = 1$.

prefix The random labels are used to build a *neighbor table* at each node. Before describing the neighbor table, we need to define the concept of *prefix* of a digit sequence $\gamma = (\gamma_1 \cdots \gamma_k)$.[3]

$$\text{prefix}_i(\gamma) = \gamma_1 \cdots \gamma_i$$

neighbor table The neighbor table is defined in the following way, it has k rows called *levels*, each level contains 2^b entries corresponding to each possible digit. Let us call $\mathcal{N}$ the neighbor table of node x and $\mathcal{N}(i, j)$ the j-th entry of the i-th level of the table. In $\mathcal{N}(i, j)$ it is stored the node y such that $c(x, y)$ is minimum, $\text{prefix}_{i-1}(\ell(y)) = \text{prefix}_{i-1}(\ell(x))$ and $\ell(y)_i = j$. Node y is said the *primary* (i, j)*-neighbor* of x. In Figure 2.10 we represent a neighbor table with k level and $b = 1$, in this case, then, $k = \log_2(n)$.

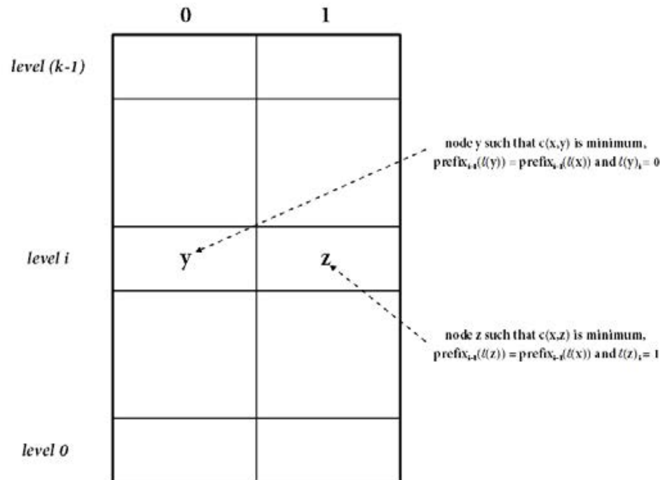


Figure 2.10: PRR neighbor table with $b = 1$

Each entry maintains other two kind of information, the *secondary* (i, j)*-neighbors* set and the *reverse* (i, j)*-neighbor*. Fixed $d \in \mathbb{N}$, the set U of secondary (i, j)-neighbors is defined as follows.

Let us define the set

$$\begin{aligned} W_{i,j} = \; & \{w \in V \setminus \{y\} \mid \\ & \mid \text{prefix}_{i-1}(\ell(w)) = \text{prefix}_{i-1}(\ell(x)) \; \wedge \\ & \wedge \; \ell(w)_i = j \; \wedge \\ & \wedge \; c(x, w) < d \cdot c(x, y)\} \end{aligned}$$

3 In [15] the authors use the *suffix* concept, here we use a later equivalent definition used by Richa in [16] because it is used in real systems like Pastry.

Then $U \subseteq W_{i,j}$ is defined as the set of nodes $u \in W_{i,j}$ such that $c(x,u)$ is minimal.

A node v is a *reverse* (i,j)*-neighbor* of node x if and only if x is a primary (i,j)-neighbor of v.

pointer list

Together with the neighbor table, each node maintains as well a *pointer list* to the object which has been shared into the network. $\mathtt{Ptr}(x)$ is a set of triple (A,y,l) where $A \in \mathcal{A}$, y is a node holding a copy of A and l is an upper bound on the cost $c(x,y)$.

root node

A node $r \in V$ is defined the *root* node of an object $A \in \mathcal{A}$ if $\exists i \in \mathbb{N}$ with $i < k$ such that

i) $\mathtt{prefix}_i(r) = \mathtt{prefix}_i(A)$

ii) if $i < k-1$ $\nexists y \in V$ such that $\mathtt{prefix}_{i+1}(y) = \mathtt{prefix}_{i+1}(A)$

sequence

The notation $\langle\alpha\rangle_l$ indicates the *sequence* of nodes $\alpha_0 \cdots \alpha_l$ (of length $l+1$). A *primary neighbor sequence* for A is a maximal sequence $\langle u\rangle_l$ such that $u_0 \in V$, u_l is the root node for A and $\forall i < l$, u_{i+1} is a primary (i, A_i)-neighbor of u_i.

routing

The routing of a request for object A from node x follows the primary neighbor sequence $\langle x\rangle$ where $x_0 = x$. At each hop i, node x_{i-1}, besides forwarding the request, informs x_i of the cost of the path followed to forward the request to it. At this point the following actions can be performed by x_i:

- If x_i is the root node of A, it sends directly to x the copy of A
- If x_i is not the root node of A, let us call ω the cost of the path followed to forward the request to x_i, it is then

$$\omega = \sum_{j=0}^{i-1} c(x_j, x_{j+1}),$$

  x_i contacts its primary and secondary (i, A_i)-neighbors to check whether they have in their pointer lists a triple (A, z, ω') such that $\omega' \leqslant \omega$. If yes the neighbor with the lower cost is contacted to use the pointer to ask node z to send the object A to x.

- Otherwise x_i forwards the request to x_{i+1}

In Figure 2.11 we represent the search tree, followed by the request for the object $A = 2E3F9C5$. As we can see, at each hop, the node label shares a larger prefix with the object identifier. In the figure the pointer lists are not represented, in order to simplify the search process.

share & unshare

The sharing of a new object follows the same pattern, it is essentially the search of the root node and the insertion of a new pointer in each node visited by the insertion message. The unsharing (delete) of an object generated by a

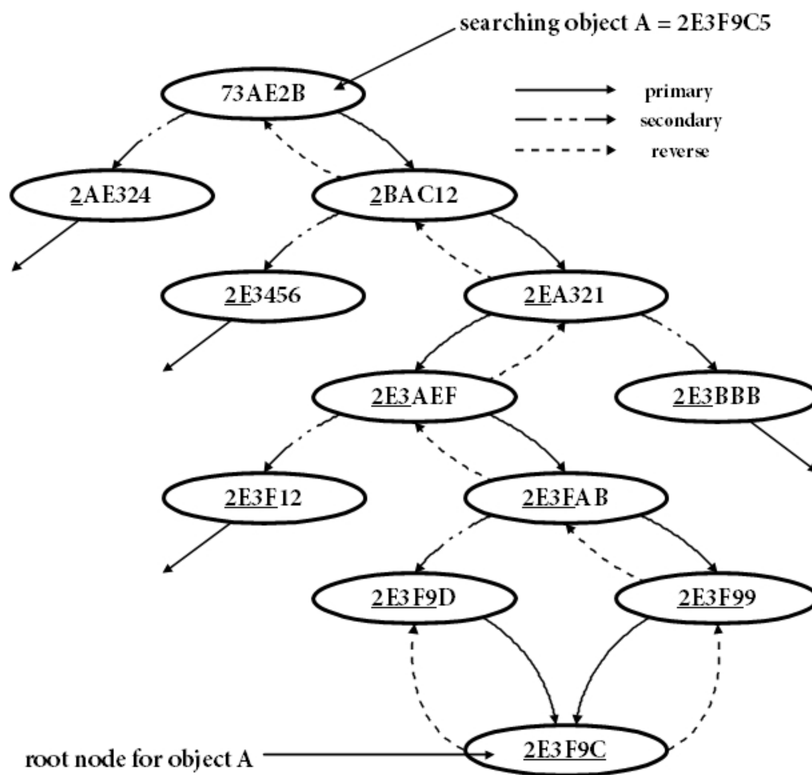


Figure 2.11: PRR search tree with $b = 4$ and $k = 6$

node y consists of the removal of all the pointers $(A, y, \cdot)$ along the sequence $\langle y \rangle$ with $y_0 = y$.

In [15] the authors prove four theorems about bounds on some metrics

- cost of a search operation,
- cost of an insertion,
- size of the auxiliary memory,
- adaptability: number of nodes which auxiliary memory is updated upon an addition or a removal of a node.

It is worth to cite them, but we do not give the proof as it goes beyond the aim of this overview of structured overlay networks.
Given a constant $C = \max\{c(u, v) \mid u, v \in V\}$:

Theorem 2.1 *Let* $x \in V$ *and let* $A \in \mathcal{A}$*. If* y *is the nearest node to* x *that holds a shared copy of A, then the expected cost of a read operation is* $O(f(l(A))c(x, y))$*, where* $l(A)$ *is the number of words composing* A*.*

When a node x tries to read an object A which has currently no shared copy in the network, then the expected cost of the associated operation is $O(C)$.

Theorem 2.2 *The expected cost of an insert operation is* $O(C)$*, and that of a delete operation is* $O(C \log n)$*.*

Theorem 2.3 *Let* q *be the number of objects that can be stored in the main memory of each node. The size of the auxiliary memory at each node is* $O(q \log^2 n)$ *words w.h.p..*

Theorem 2.4 *The adaptability of the access scheme is* $O(\log n)$ *expected and* $O(\log^2 n)$ *w.h.p..*

Following the taxonomy presented in the previous section, PRR is a structured overlay network with logarithmic multi-hop strategy, flat address space, using a prefix matching criteria. Each node is associated with a tree and the resulting graph has a $O(\log n)$ degree. The algorithm takes into consideration the locality through the use of function c. In [15] the authors do not define an overlay maintenance policy.

CAN

In [17] Ratsanamy et all present a DHT algorithm based on a *scalable content-addressable network*. CAN projects nodes and key into a *d-dimensional* Cartesian coordinate space on a *d-torus*. Each node is deterministically assigned a zone of the d-dimensional space and each key is deterministically assigned a point into the space. In a d-dimension CAN two nodes are *neighbors* if their coordinates

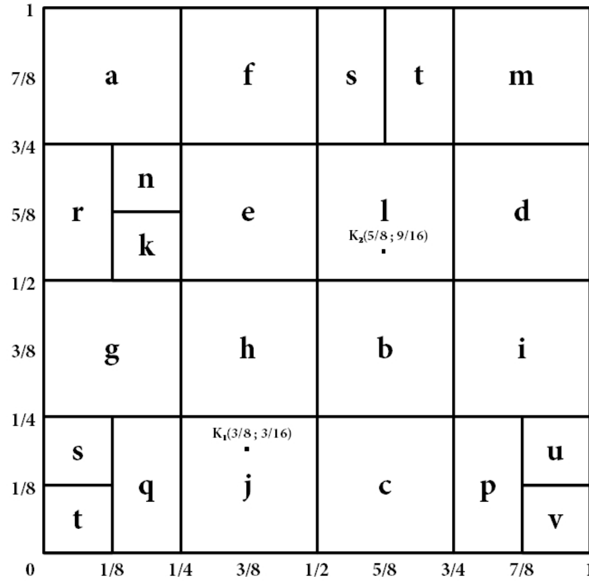


Figure 2.12: Example of CAN in a 2-dimensional space $[0,1] \times [0,1]$

spans overlap along d-1 dimension and they border on one dimension. Given a key K mapped into a point $P(x_1, \ldots, x_d)$ and a node n assigned to the zone $[x_{1_1}, x_{1_2}] \times \ldots \times [x_{d_1}, x_{d_2}]$ if and only if $x_{i_1} \leqslant x_i < x_{i_2}$ with $i = 1, \ldots, d$.

In Figure 2.12 we represent an example of CAN in a 2-dimensional space, node e set of neighbors is $\{n, k, h, l, f\}$.

routing Each node in CAN maintains the list of its neighbors associated with their virtual coordinates, which constitute the routing table of the node itself. In Table 2.1 we report the routing table of node e from Figure 2.12.

Node	Zone coordinates
n	$\left[\frac{1}{8}, \frac{1}{4}\right] \times \left[\frac{5}{8}, \frac{3}{4}\right]$
k	$\left[\frac{1}{8}, \frac{1}{4}\right] \times \left[\frac{1}{2}, \frac{5}{8}\right]$
h	$\left[\frac{1}{4}, \frac{1}{2}\right] \times \left[\frac{1}{4}, \frac{1}{2}\right]$
l	$\left[\frac{1}{2}, \frac{3}{4}\right] \times \left[\frac{1}{2}, \frac{3}{4}\right]$
f	$\left[\frac{1}{4}, \frac{1}{2}\right] \times \left[\frac{3}{4}, 1\right]$

Table 2.1: CAN routing table of node e

Given a key K associated with point P and a node x looking for this key, the routing algorithm followed by the search query greedy proceeds towards the point P. In Figure 2.13 we report a graphical representation of the route towards node j followed by a search query, generated by node d, for a key associated with point K_1.

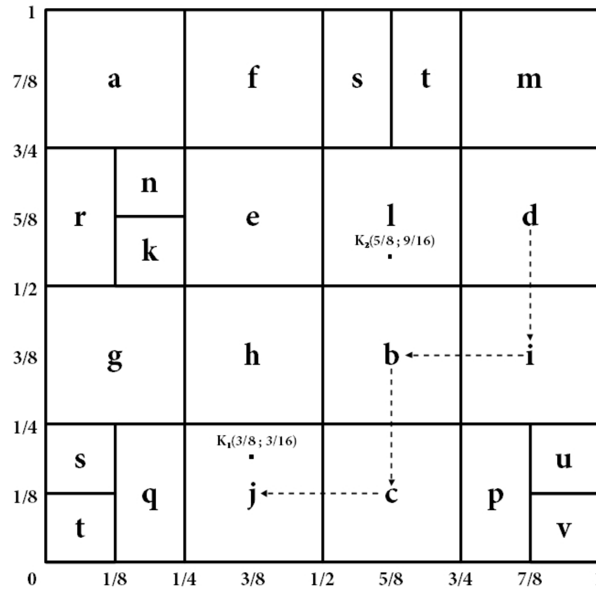


Figure 2.13: CAN routing in a 2-dimensional space

node join A node x has to discover a node which is already in the CAN to join the network. This is done through special bootstrap nodes which maintain a list of part of nodes currently in the system. Node x contact one of those and receive from it a random list of nodes which have already joined the CAN. The first operation to be performed consists in finding a zone for the new node. Node x chooses at random a point P in the d-dimensional space and sends to one of the nodes inside the CAN a join request associated with P. At this point the routing algorithm towards P is executed. Once the zone containing P is reached, it has to be split in half to create a new zone for node x. This process needs a policy ruling the sequence of the dimensions along which the split has to be done. In Figure 2.14 we represent the join of a node x which has been associated with a point P located in the zone of node f, j is the node chosen by x in the list received by the bootstrap node.

The list of neighbors of node f passes from $\{a, e, s\}$ to $\{a, e, x\}$ and the list of neighbors of node x, received from node f, is $\{f, e, s\}$.

node departure When a node y leaves the CAN, two different situation may occur. Firstly, it is possible to merge the y zone with the zone of one of its neighbors, this is possible when the two zones are part of a previously split zone. In Figure 2.15 node s leaves the network, its zone is merged with the t one and all the keys owned by s pass to t.

Secondly, it is not possible to merge the zone left unowned by a node departure to a neighbor zone in order to form a valid zone. The neighbor responsible for the smaller zone handles temporarily both the zones. In Figure 2.16, the departure of node r cannot be managed through the merge of any neighbor

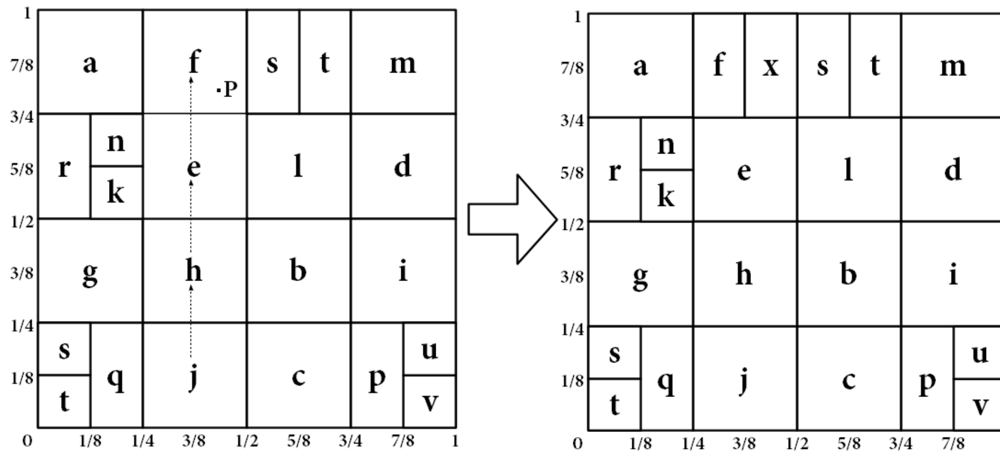


Figure 2.14: Join of a new node x associated with point P, node j is the introducer

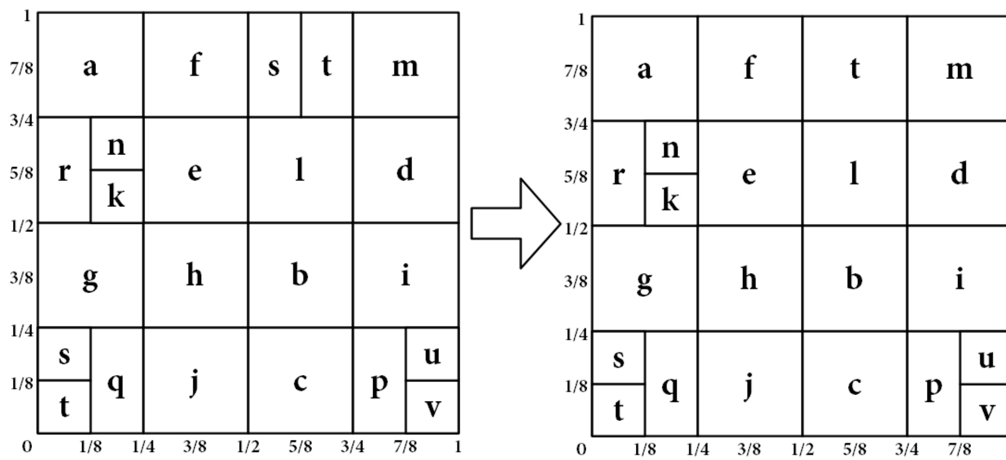


Figure 2.15: Departure of node s, the zone of node t is merged with the unowned zone

zones, then node n handles temporarily the zone until it is possible the creation of a valid zone.

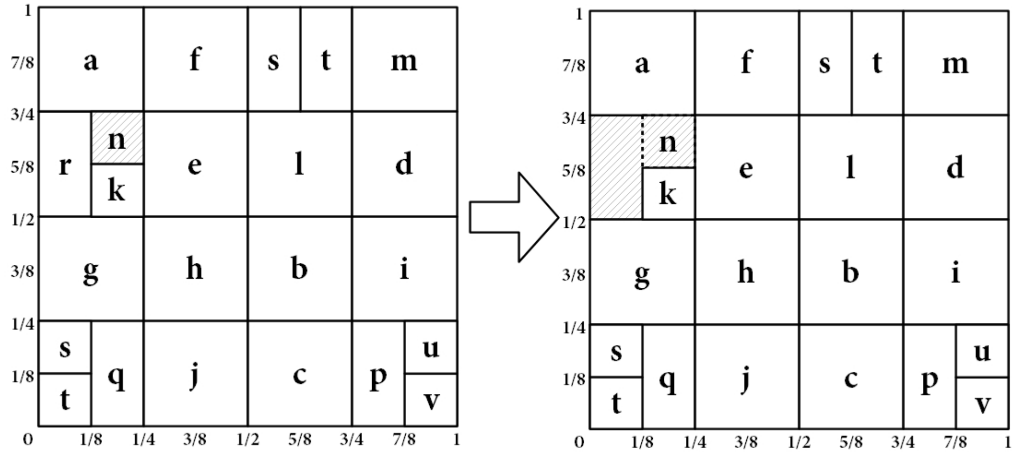


Figure 2.16: Departure of node r, node n temporary takes care of the unowned zone (the merge is not possible)

failures management

Another way for nodes to depart from the CAN is to fail. In CAN each node informs periodically its neighbors about its list of neighbors, when this communication ceases (a threshold is defined to decide when the communication can be considered ceased), the node is considered failed by the neighbors. A node detecting a neighbor's failure starts a takeover algorithm, it initializes a timer proportional to the dimension of its zone, when the timer expires the node sends a TAKEOVER message to all the failed node neighbors, enclosing in the message the size of its own zone. When a node receives a TAKEOVER message, it cancels its timer in case in which its zone is smaller than the one of he sender, otherwise it sends back a TAKEOVER message. At the end of the process a node with a small zone will be responsible for the unowned zone.

As a consequence of a large number of departures it is possible that the CAN could result in a very unbalanced structure, with many nodes taking care of multiple zones. This problem is handled by a mechanism which periodically reassigns the zones to the nodes.

The cost of a search operation in a CAN depends on the dimensions used and on the nodes present in the network. The average length of a path in a d dimensional space divided into n equal zones is in fact $(d/4)(n^{1/d})$. Then, for a fixed dimension d the cost of routing a message is $O(n^{1/d})$.

CAN is a structured overlay network with a multi-hop $O(n^{1/d})$ strategy. The peer address space is flat, the distance metric used is an Euclidean distance in a d-dimensioned space. Its topology is a multi-dimensional Cartesian space. The maintenance mechanism is opportunistic and, finally, in the basic version of the protocol, the locality of the underlay network is not considered.

Pastry

Pastry [18] was designed in the context of project PAST [19, 20] as a middleware for object location and routing scheme in a P2P environment. This DHT protocol takes inspiration directly from PRR, the routing phase is in fact based on a prefix matching criteria.

Nodes are randomly associated to 128-bit identifiers called *nodeId*, then the IDs circular space spans from 0 to $(2^{128}-1)$. In the same way, objects are mapped into keys in the same space. Both nodeId and keys are expressed in base 2^b where $b \in \mathbb{N}$ states the number of bits read at time (exactly like it was for PRR).

node state In Pastry nodes maintain three different structures to represent their partial knowledge of the network, the *routing table*, the *neighborhood set* and the *leaf set*.

The routing table follows the same scheme of PRR neighbor table. A routing table $\mathcal{R}$ of a node A embedded in an N nodes network is composed of $\lceil \log_{2^b} N \rceil$ rows and $2^b - 1$ columns. Hence, each row corresponds to the length of the prefix shared by the nodeIds maintained in it and the local nodeId, each column correspond to the first digit which is different. Formally, adopting the notation used in [18], if we call $\mathcal{R}_l^i$ the entry at row l and column i, it contains the information (IP address and nodeId) about the node having a nodeId X such that:

- nodeId A and nodeId X share the same prefix long l digit and digit at place $l+1$ is equal to i, where i is the i-th digit of the base.
- among nodeIds satisfying the first property, X is the closest node to A according to a proximity metric defined in the underlay network (Pastry uses the IP hops).

The neighborhood set $\mathcal{M}$ contains the information about the $|\mathcal{M}|$ nodes which are the closest to node A according the proximity metric.

The leaf set $\mathcal{L}$ contains the information about the $|\mathcal{L}|/2$ nodes having the numerically closest nodeIds lower than A and the $|\mathcal{L}|/2$ nodes having the closest nodeIds greater than A. Usually $|\mathcal{M}|$ and $|\mathcal{L}|$ are equal to 2^b or 2^{b+1}. In Figure 2.17 we represent the state of a node with nodeId 100231. In the followings with $\mathcal{L}_i$ we refer to the i-th closest node in the leaf set.

routing A message sent in order to search for the key K is routed through a node X following the next mechanism. Firstly, node X checks if $\mathcal{L}_{-|\mathcal{L}|/2} \leqslant K \leqslant \mathcal{L}_{|\mathcal{L}|/2}$. If yes the message is routed to the node in $\mathcal{L}$ with nodeId numerically closest to K. Otherwise, the routing table is used and the message is routed to the node sharing the longest prefix with K (like it was for PRR). It could happen, in some rare cases, that the cell of the routing table is empty or the node is unreachable. Then the message is routed to a node $Y \in \mathcal{L} \cup \mathcal{M} \cup \mathcal{R}$ such that

- the prefix it shares with K is at least long as the one shared between K and X,

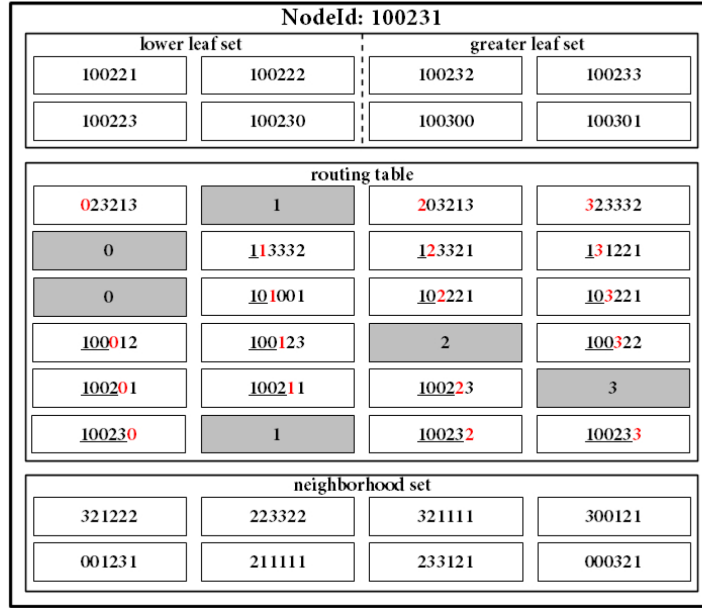


Figure 2.17: A state of a node in Pastry where $b = 2$ and $l = 6$, the red digit is the position relative to the level, the underlined text is the prefix shared with the nodeId

- Y is numerically closer to K than X.

node join

Whenever a node X wants to enter the Pastry network, it has to find a node A already set inside Pastry. This is done through a search for A in the surroundings of the node X, namely, A is searched among nodes, which are near to X with respect to the proximity metric. Once A is found, a message containing the join request is sent to it. The message is routed to a node Z by means of the routing algorithm described earlier. Each node in the path followed by the routing, sends back to X its state tables, then X computes its personal state tables through the information received.

The X's neighborhood set is created through A's neighborhood set (please note that A was a node near to X according to the proximity metric).

The X's leaf set is created through Z's leaf set, as Z is the node numerically closest to X.

The routing table is built through each node Y traversed by the join message. Using the notation from the description of PRR, let us call $\langle Y \rangle_l$ the sequence traveled by the join message with $Y_0 = A$ and $Y_l = Z$ and let us assume that A does not share any prefix with X (it is the most general case). We call $\mathcal{R}(X)$ the routing table of node X, with $\mathcal{R}_i(X)$ we refer to the i-th row of X's routing table. $\mathcal{R}_i(X)$ is created using $\mathcal{R}_i(Y_i)$, its elements in fact share with X i digit, then they are suitable to be used for $\mathcal{R}_i(X)$. This is done for all $i = 1, \ldots, l$. At the

end of the process X has its own state tables and sends it back to all the nodes contained in it ($\mathcal{L}(X) \cup \mathcal{M}(X) \cup \mathcal{R}(X)$).

node departure The procedure used to replace a node X, departed from the Pastry network, is the same in both cases of node leaving with or without warning. In the latter case the departure is discovered when a node does not respond to the messages.

A node Y such that $X \in \mathcal{L}(Y)$, replaces it observing the following strategy: it contacts the numerically farther node Z in the direction of the departed node to get its leaf set (observe that $\mathcal{L}(Y) \cap \mathcal{L}(Z) \neq \phi$), a live node is chosen in $\mathcal{L}(Z)$ to take the place of X.

Let us consider now the case in which node X is such that $X \in \mathcal{R}(Y)$ and let us assume that $X = \mathcal{R}_j^d(Y)$, a new entry for that cell has to be found. We need a node sharing with Y a prefix of j digits, thus nodes appearing in the same row should have in their j-th row a node suitable for $\mathcal{R}_j^d(Y)$. Therefore, nodes in $\mathcal{R}_j^i(Y)$ with $i \neq d$ are contacted until a live node for $\mathcal{R}_j^d(Y)$ is found. If the nodes at row j do not have a live node in their routing table suitable for $\mathcal{R}_j^d(Y)$, the procedure continues with row $j+1$ and so on.

In case $X \in \mathcal{M}(Y)$, each node $Z \in \mathcal{M}(Y)$ is contacted to get its neighborhood set $\mathcal{M}(Z)$. Among the newly discovered nodes, Y chooses the closest ones and updates its neighborhood set accordingly.

Pastry's performances are similar to PRR ones. If we assume accurate routing tables, indeed, the cost of routing a message is $O(\log_{2^b} N)$ hops. In fact, if the message follows the routing table of a node at each step, the search space decreases of a factor of 2^b until we reach the destination. As long as we use a node from the leaf set, the route is one step away to finish. In case of an empty cell in the routing table, as we saw during the routing description, the additive cost is of just one hop. Finally, the messages sent during the join phase are $O(\log_{2^b} N)$.

Like PRR, Pastry uses a logarithmic multi-hop strategy, a flat address space and a prefix matching criteria. The resulting graph has a $O(\log_{2^b} N)$ degree and the locality is considered through the proximity metric of the IP hops distance. In [18], the authors claim that the nodes inside the state tables are maintained relatively close to the local node, this results in a better performance, especially in terms of routing.

With respect to PRR, Pastry adds the information contained in $\mathcal{L}$ and $\mathcal{M}$. Furthermore, there exists a specific overlay maintenance strategy, an active one. Indeed, the communication of the state tables happens only after the detection of a node departure.

Kademlia

Kademlia [21] defines a structured overlay P2P network based on a *bitwise exclusive or* metric (XOR). Each node in Kademlia is identified by a m-bit string, chosen at random when the node joins the network. The objects to be shared are

assigned m-bit keys (in [21] $m = 160$). As usual in DHTs, the keys are assigned to nodes which have an identifier near to them according to a distance function. Kademlia uses the XOR ($\oplus$) operation as a distance function, given two nodes x and y, $d(x, y) = x \oplus y$.

node state A node x in Kademlia stores a routing table which is formed by m sets of nodes called k-buckets. Let us call B_i, with $0 \leqslant i < m$, the i-th k-bucket. B_i contains at most k nodes $\{z_1, \dots z_l\}$ with $l \leqslant k$ such that $d(x, z_j) \in \left[2^i, 2^{i+1}\right)$ with $j = 1, \dots, l$. For each node, Kademlia maintains a triple ⟨IP address, UDP port, Node ID⟩. The k-buckets are sorted chronologically according to the last time a message was received from a node. It is done in order to use a LRU policy for the bucket maintenance. As a consequence, the last added node is always at the tail of the list representing the bucket.

interval	k**-buckets**
$\left[2^0, 2^1\right)$	6
$\left[2^1, 2^2\right)$	5,4
$\left[2^2, 2^3\right)$	3,2,1
$\left[2^3, 2^4\right)$	14,12,10
$\left[2^4, 2^5\right)$	23,21,19

Table 2.2: Kademlia routing table of node $x = 7$, with $m = 32$ and $k = 3$

In Table 2.2 we represent an example of Kademlia routing table for a node $x = 7$, $m = 32$ and $k = 3$ (we represented only the node ids translated into base 10). In Kademlia the node state maintenance procedure does not use any special message. The routing tables in fact are updated each time a message of any kind (request or response) is received by the node. The node identifier of the sender is examined and three possible cases emerge. Let us call x the local node and y the sender node. Let us suppose that B_i is the k-bucket suitable for y.

- If $y \in B_i$, y is moved to the tail of the list, otherwise
- If $|B_i| < k$, y is inserted at the tail in B_i, otherwise
- If $|B_i| = k$, the node a, which was least-recently seen, is contacted. If it does not respond, it is removed and y is inserted at the tail of the list. Otherwise, a is moved to the tail and y is discarded.

primitives It is important to list the four primitives defined by Kademlia. The primitives are `ping`, `store`, `find_node` and `find_value`, all of them are *remote procedure calls* (in short, RPC).

The `ping` RPC contacts a node to probe if it is still alive. The `store` RPC is used to store a new shared object in a node. The `find_node` RPC takes an m-bit

identifier as an input argument and returns the k triples relative to nodes which are closest to the identifier (they can be contained in different k-buckets of the RPC recipient). The `find_value` RPC performs the same operation of `find_node` with the difference that if the recipient has a key stored with the id of the input argument, then the object relative to that key is returned.

node lookup As almost all of the DHT protocols, the basic operation of Kademlia is the search of a node. In Kademlia it is called *node lookup* and it is based on the `find_node` primitive.

Initially, a node x looking for a node y sends, in parallel and asynchronously, a `find_node` RPC to α nodes picked up from the k-bucket closest to y, where α is a system-wide concurrency parameter, usually set to 3. The procedure continues recursively, once x receives the results, it chooses again α nodes among the closest nodes to y, and sends to them the `find_node` RPC. If a `find_node` round returns nodes which are not closer with respect to the closest already seen, x sends the RPC to all the nodes among the k closest ones not yet contacted. The procedure terminates when x has queried and received responses from the k closest nodes seen by it. In Figure 2.18 we represent a node lookup i Kademlia network where $m = 5$, $k = 3$ and $\alpha = 3$. Node 30 sends the request to the three node 12, 14 and 15 which are the closest to 8 in its routing table. The nodes reply to 30 with, respectively, $(8, 9, 10)$, $(8, 10, 11)$ and $(9, 10, 11)$. Node 8 can be found in two of the messages received, hence the lookup finishes.

All the operations in Kademlia are based on the node lookup mechanism. To store an object associated with the key a, the node lookup is performed with argument a and a `store` RPC is sent to the k closest node. To get an object associated with key b the same procedure of node lookup is performed, with the difference that here the `find_value` primitive is used and the procedure is interrupted when the key is found.

Keys in Kademlia are periodically refreshed by re-publication performed by nodes at a fixed rate. If a key is not refreshed, it is considered expired. To maintain consistency in the search process, if a node w where a key a is stored discovers a node v which is closest to a, it replicates the key to v.

node join To join the Kademlia network a node x has to know a node y already set in the system. The first step for x is to insert y in the right k-bucket. Then x starts a node lookup having its identifier as input parameter. In this way it has an initial version of the routing table and at the same time it is inserted in the k-buckets of the nodes traversed by the node lookup. The routing table will be refined during the refreshes which are performed as long as the node receives messages.

node departure There is no a specific mechanism to manage a node departure, in fact the consequences of a node departure affect the network as a side effect of the k-buckets LRU policy. A node which is not anymore participating to Kademlia will not be seen by the others or will not reply to direct requests. Then, gradually, it will be removed from all k-buckets.

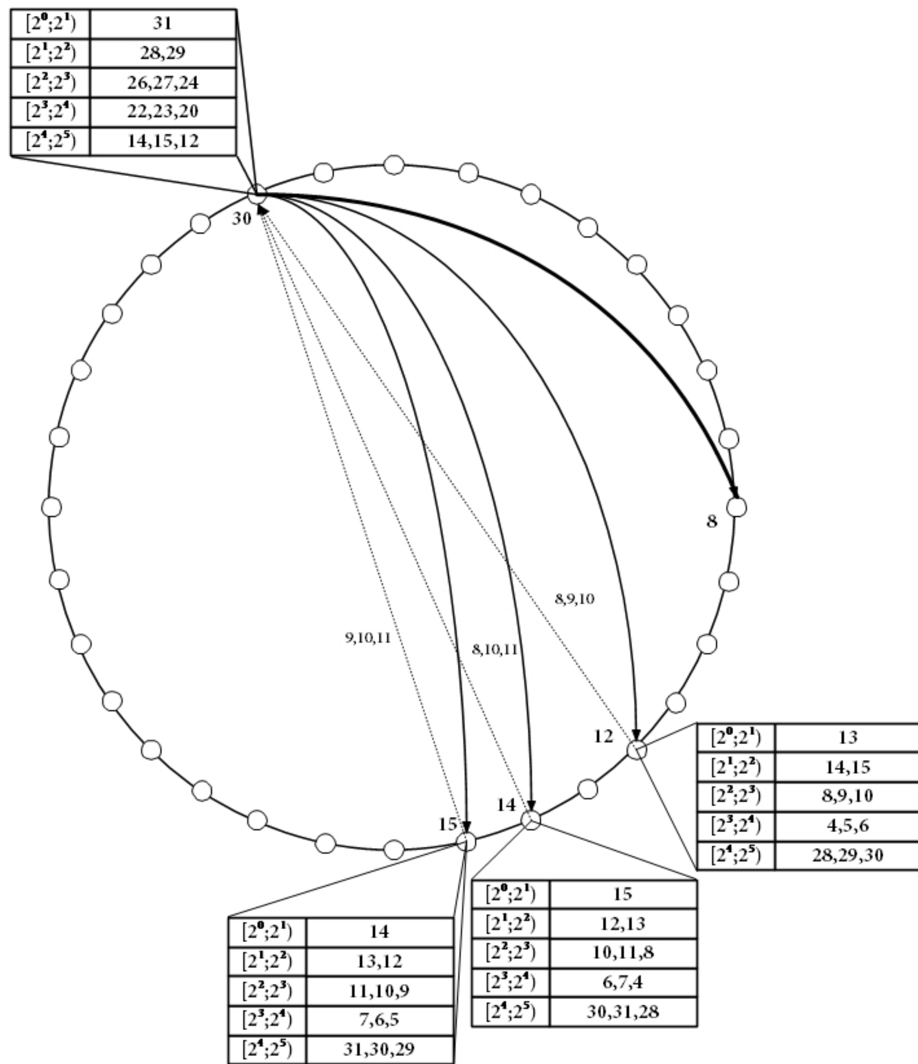


Figure 2.18: Example of node lookup in a Kademlia network where $k = 3$, $m = 5$ and $\alpha = 3$, node 30 looks for node 8

In [21] the authors give a sketch of a proof that the cost of a node lookup is $O(\log n)$ where n is the number of nodes in the network. We do not report the proof here, though it can be intuitively seen from the fact that at each step of the node lookup process, the space of research is at least halved.

With respect to the taxonomy, Kademlia follows a logarithmic multi-hop strategy, the address space is flat and it uses the bitwise XOR as a (non Euclidean) distance metric. The topology can be seen as a ring based on the XOR operation. The overlay network maintenance is performed through a passive strategy, Kademlia is oblivious with respect to the locality of the underlay network.

Kelips

So far, we have described systems with a multi-hop routing strategy, now we talk about Kelips [22], a DHT in which resource lookup times are reduced and stability to failures and churn is increased at the expense of increased memory ($O\sqrt{n}$) and bandwidth usage. In Kelips in fact, a resource lookup is resolved, in normal conditions, with $O(1)$ time. Furthermore, changes on membership in the overlay network are detected and disseminated to the system quickly. Kelips is born as a file sharing system, then, from now we will talk about files instead of general resources.

node state

In Kelips the nodes are distributed into k virtual sets, called *affinity groups*, numbered from 0 to $k-1$. The way the nodes are spread follows the *consistent hashing* policy, introduced in [23]. We will describe consistent hashing in the next Chapter, in which we deal with Chord. Herein we limit ourselves to say that IP address and port number of nodes are used together to obtain the argument of an hash function $\mathcal{H}$ which will map it to the integer interval $[0, k-1]$. A node x such that $\mathcal{H}(IP_x, PORT_x) = i$, with $0 \leqslant i \leqslant k-1$, will be inserted to the i-th affinity group.

The affinity groups' structure is used to build the node state which is composed of three parts

- **Affinity group view**: a set containing the view of the rest of the affinity group in which the node is embedded (the view can be partial)
- **Contacts**: each foreign affinity group is represented in the local node state with a small set (of fixed cardinality c) of nodes contained in it.
- **Filetuples**: is a set enclosing references to the files contained in the nodes of the affinity group including the local node. The reference is composed of the file name and the IP address of the node containing the file, this node is said the *homenode* of the file.

All the entities contained in the node state are associated to a heartbeat count, used to detect if the entry has to be deleted or not. The nodes contained in the

affinity groups and in the sets of contacts are associated to a *round trip time* to establish a preference criterion in the lookup phase.

In Figure 2.19 we show a graphical representation of a single node state and in Table 2.3 we report the relative tables containing the affinity group (Table 2.3a), the contact sets (Table 2.3b) and the filetuples (Table 2.3c). In the last two tables we omitted the heartbeat and the rtt for the contact set and the heartbeat for the filetuples.

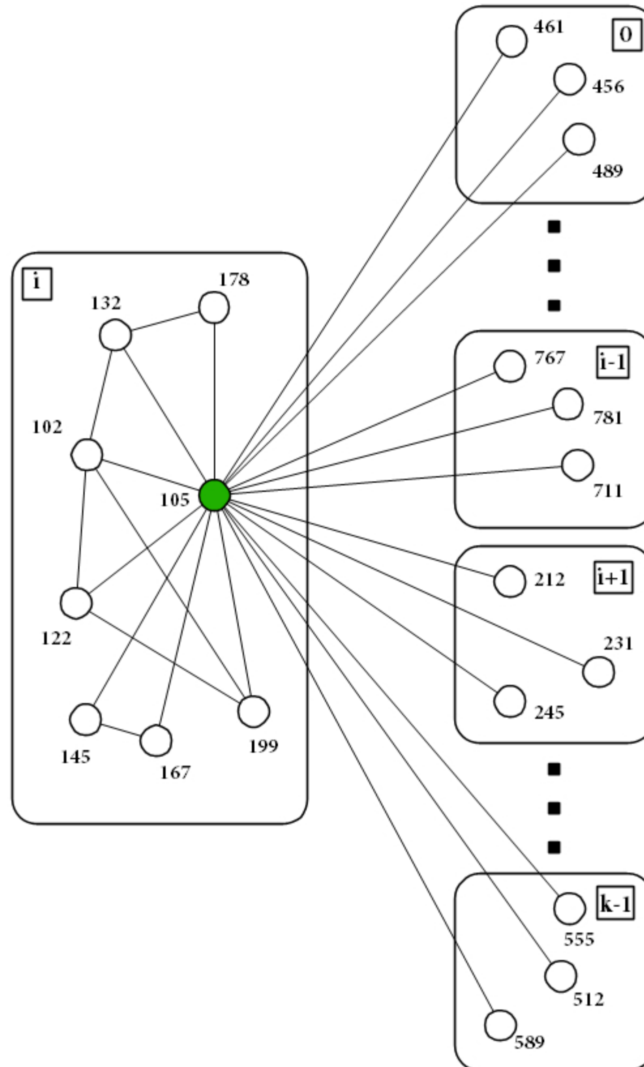


Figure 2.19: Kelips structure from the point of view of node 105 contained in the i-th affinity group, here $c = 3$. On the right we represented the $k-1$ foreign affinity groups. Observe that we reported only the contacts of the node.

The memory usage resulting from this structure is easy to compute. Assuming that $\mathcal{H}$ is an hash function ensuring a uniform distribution of the nodes into the affinity set (such as SHA-1), the cardinality of each affinity group is around $\frac{n}{k}$.

id	hbeat	rtt
178	1232	2ms
132	3211	7ms
102	1211	10ms
122	2044	23ms
145	2134	30ms
167	2155	42ms
199	4566	77ms

(a) Affinity group view

group	contacts
0	461,456,489
⋮	⋮
i-1	767,781,711
i+1	212,231,243
⋮	⋮
k-1	555,512,589

(b) Contact sets

name	homenode
a.txt	102
b.pdf	145
⋮	⋮
z.png	199

(c) Filetuples

Table 2.3: Kelips state of node 105

Space occupied by the contact sets is equal to $c(k-1)$. Space occupied by the Filetuples is $\frac{F}{k}$ where F is the number of files shared in the system, therefore assuming that files are uniformly spread over the nodes. Finally, the total space occupied by a node state is a function in two variables:

$$S(k,n) = \frac{n}{k} + c(k-1) + \frac{F}{k}.$$

Fixing n the function is minimized in $k = \sqrt{\frac{n+F}{c}}$, assuming F proportional to n and observing that c is constant, it can be stated that the optimal k varies as $O(\sqrt{n})$. Consequently $S(k,n)$ varies as $O(\sqrt{n})$.

It is important to observe that the one-hop property has been reached at an expense of the increase of space used to maintain the routing information, thus we passed from $O(\log n)$ of (almost all) previously described protocols, to $O(\sqrt{n})$. Furthermore, the maintenance of such a structure, performed through an heartbeating mechanism based on a gossip-style protocol [24], requires a higher bandwidth with respect to the previous protocols. Briefly, each node periodically selects a subset of nodes from its state and multicasts them information about its personal state (this is done intra-affinity groups and inter-affinity groups).

lookup When a node x search for a file, it has to map it to the right affinity group through the application of the same hash function $\mathcal{H}$ used to build the affinity groups system. The request is sent to the node which is closest according to the round trip time in the affinity group selected. The node receiving the request checks among its filetuples and sends back to x the homenode address. At this point x sends the request to get the file directly to the homenode.

insertion The insertion of a new file f follows the same scheme of the file lookup, when a node of the right affinity group is contacted, it chooses at random a node h into its affinity group. The node h will be the homenode for f and it is sent to it.

A filetuple is created for f and it is inserted in the gossip stream to be spread among the members of the affinity group.

node join A new node x joins the Kelips network through a node y already located inside the system. Node y sends back to x its state, x initializes its state with the one received and starts to gossip the network to fill its state with the right information. The departure of a node is automatically detected by the gossiping system, its heart beat won't be updated any more and its entry will be gradually deleted from other nodes' states.

Kelips overlay network follows a one-hop strategy for the resource retrieval, the address space is flat, the next-hop decision's criteria is based on the consistent hashing policy and on the round trip time stored in each entry referring to a node. The graph resulting from the Kelips structure has a $O(\sqrt{n})$ degree, the strategy for the node's state's maintenance is active and the locality of the underlay is kept into consideration through the usage of the round trip time.

MADPastry

As a last example we report MADPastry [25], a DHT substrate explicitly designed for the use in a MANET. Applying the DHT concept to a MANET is a challenging problem, in fact there are many issues to take into consideration before designing such a system in a highly dynamic mobile environment.

In [25], the authors list three main problems opposing the possibility to use a DHT in a MANET:

- The connection between the overlay network and the underlay network is not sufficient. A DHT, as we have already seen is almost oblivious of the physical layer. In a MANET context, instead, it is important to have enough knowledge of the physical network, for instance to minimize hop counts, as in a long path over a mobile ad hoc network a message loss is very probable.

- In a MANET, the routing problem is solved usually by the use of broadcast because of the lack of a central infrastructure and the mobility of the nodes. Such a strategy makes the adoption of an overlay completely useless, if not harmful.

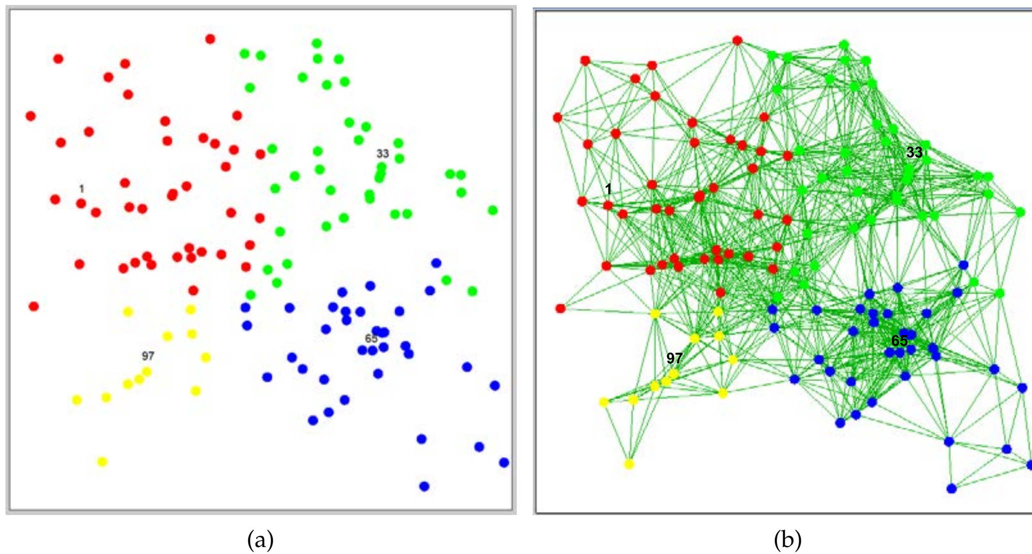
- The cost of maintenance of the routing table in a DHT appears unbearable in a MANET context. In fact a MANET does not have enough bandwidth to manage traffic generated by nodes in order to keep the tables consistent.

The above mentioned reasons show clearly that the application of any of the overlay structured P2P networks to a MANET is unfeasible. The solution is to find a compromise, thus MADPastry is a possible candidate. In fact, it combines Pastry with a well known reactive routing protocol for mobile ad hoc networks,

AODV [26], *ad hoc on demand distance vector*. The objective is to limit, as far as possible, the use of broadcast, thanks to the overlay structure.

Random landmarking

The way MADPastry tries to solve the problem connected to the DHT low knowledge of the physical layer is the usage of the *random landmarking* concept [27]. In this strategy, the set of nodes is divided into clusters through the usage of *landmark keys*. Landmark keys are overlay identifiers, a node responsible for a landmark key becomes a temporary *landmark node*. Nodes inside a cluster share the same prefix, the one of the landmark key. Landmark keys are chosen to divide the address space in equal-sized segments. In Figure 2.20 we report an example of a random landmarking in a mobile ad hoc network composed of 128 nodes and four landmark keys. In 2.20a we represent only the nodes, in 2.20b we added the links. The two figures have been obtained through the implementation of the random landmarking policy in MOMOSE, a mobility models simulator which will be described in Chapter 4.



(a) (b)

Figure 2.20: An example of random landmarking in a MANET with $N = 128$ and $K = 4$, each color is relative to a different cluster, the numbers are written over the current landmark node

The overlay id assumed by a node is of course changing over time, it depends on the places through which the node goes during its movement. Once the node detects that it is closer to a new landmark node (with a different landmark key) with respect to its current one, it changes its id (choosing a random one and concatenating it with the new landmark key prefix) and joins a new cluster. The distance metric used here is the hop count.

A node detects its current affiliation thanks to a beacon which is sent through broadcast by the landmark nodes. To limit the impact of this broadcast, the

beacon message is propagated only to the nodes which are inside the cluster and to the neighbors which are one hop away from them.

node state MADPastry node state is composed from three different routing tables inherited from Pastry and AODV. From the first one it takes the routing table and the leaf set making them "lightweight", while from the latter the standard routing table.

We have already seen the Pastry routing table and the leaf set. MADPastry modifies these two structures to make them usable also in a low-bandwidth context.

The MADPastry routing table passes from the $\lceil \log_{2^b} N \rceil$ rows to $\lceil \log_{2^b} K \rceil$, where K is the number of the landmark keys. For example, if K is chosen to be equal to 2^b, then the Pastry routing table degenerates to a table having only one row and each entry contains a node inside each cluster (in [27] the authors show an example with $b = 4$ and $K = 16$). The reduction of the routing table implies a less expensive maintenance cost, but at the same time, the $O(\log N)$ lookup time is lost.

The leaf set maintains the same structure as in Pastry, the difference here is that the only entries which are kept updated are the closest in both the directions, namely the predecessor and the successor with respect to the ids order. Each node proactively pings both of them either to check if they are alive or if another node has to be selected as new successor(predecessor).

Each node in the cluster broadcasts periodically its id inside the cluster, observing that the leaf set is composed by nodes numerically close to each other, this should be enough to keep a loose consistence of the set.

The AODV routing table is the standard one with no modifications. It is used to perform the physical routing: it maintains the next hop address for each route and a sequence number to keep it updated. The same sequence number is used in order to decide when the route will become obsolete.

routing As we have already said, MADPastry integrates a modified version of Pastry with AODV routing protocol, a node can indeed act in two different ways

1. If the node receives a message because it is a destination of a Pastry forwarding, it decides its next hop following the Pastry protocol, namely it consult its routing table or leaf set to choose the target node.

2. If the node receives a message because it is a physical hop of the AODV protocol, which means that it is a part of one overlay hop, it uses the AODV routing table to decide the next hop, behaving exactly like a classical AODV node.

While a message is routed in the AODV manner, each node inspects the destination overlay id. If a node discovers that its id is closer to the destination one with respect to the next hop one, it intercepts the packet and the next hop is carried

out following the Pastry policy. The reason of that is limiting the network traffic optimizing, at the same time, the length of the path to the destination.

Another issue is how to find a route to a target node when there is no known path for the specific destination. This can happen either when a node selects an entry on its Pastry routing table for which it has not AODV known path, or when an intermediate node of a physical path does not have a valid next hop for a route to the destination.

If the destination is in the same cluster of the node where the problem has arisen, the overlay packet will be broadcast inside the cluster. Otherwise, the standard AODV expanding ring broadcast is performed to discover the right route to the destination.

maintenance In MADPastry the maintenance strategy is hybrid, in fact it is active in the part regarding the leaf set and the landmark cluster membership, while it is passive for the rest of the structures. Indeed, the information about other nodes is known thanks to the overhearing of the packets passing through a node during a routing operation.

That's the reason why each packet propagated over the network contains the following information

- the AODV sequence number of the packet's source,
- the AODV sequence number of the node visited by the packet in the previous hop,
- the Pastry id of the packet's source,
- the Pastry id of the node visited in the previous hop by the packet.

Each node then extracts this information from the packet and, if necessary, updates or creates the routes. The policy states that if the packet contains new information about an existing route, the old data needs to be always overwritten by the new ones.

MADPastry is quite different from a usual DHT because of the particular environment for which it has been designed, however we can still apply the taxonomy, with some exceptions. MADPastry follows a multi-hop strategy, i which the logarithmic cost is lost, the address space is flat, the distance metric is obtained following the prefix matching criterion. The topology of the overlay network is a stretched version of the Pastry one. The maintenance, as we have seen, is active when it goes to keeping the leaf set consistent and passive for the rest of the routing structures. Finally, the locality is taken into consideration as it is not possible to be completely oblivious in a MANET context.

In Table 2.4 we resume the classification of the DHTs that was described in the present chapter. An observation about the geometry of CAN shall be made: the graph resulting is $O(d)$ assuming a d-dimensional space divided in n equal

zones as stated in [17]. Indeed, in such a situation, each node has at most 2d neighbors. We remind that in the case of MADPastry, K is the number of clusters generated by the random landmarking mechanism. Furthermore we observe that the lookup hops cost is given considering the overlay hops.

DHT	# hops	address space	next-hop	geometry(graph degree order)	maintenance	locality
PRR	$O(\log n)$	flat	prefix-match	$O(\log n)$	$\times$	$\surd$
CAN	$O(n^{-k})$	flat	Euclidean distance	$O(d)$	opportunistic	$\times$
Pastry	$O(\log n)$	flat	prefix-match	$O(\log n)$	active	$\surd$
Kademlia	$O(\log n)$	flat	XOR-distance	$O(\log n)$	passive	$\times$
Kelips	$O(1)$	flat	cons.hashing and rrt	$O(\sqrt{n})$	active	$\surd$
MADPastry	$O(\log K)$	flat	prefix-match	$O(\log K)$	hybrid	$\surd$

Table 2.4: DHT comparison according to the taxonomy

3 JXTACH

In this chapter we will describe a way to deal with the dynamism in P2P networks. As we already introduced in Chapter 1, P2P networks can be considered highly dynamic and there exist a lot of mechanisms which deal with this problem.

In Chapter 2 we have introduced the basic notions concerning P2P overlay networks. In this Chapter we are going to describe our results in this area.

The growth of popularity of P2P networks has raised up the need to have the proper instruments to develop services and application over this kind of structure. Sun Microsystem tried to provide users with such an instrument beginning project JXTA in 2001. The aim was to provide a general framework for developing P2P applications. The present work covers the description of this framework as it is the starting point of the result we intend to describe in this chapter.

This result is a pure DHT based version of JXTA framework, we called it JXTACh [28]. We say pure because in JXTA there is a mechanism resembling a DHT, but its characteristics are not respecting the properties of DHTs. JXTA developers explain the use of such a mechanism with the expensive maintenance costs of DHTs. In our opinion, instead, the framework could benefit from such a structure. That is why we decided to replace the old mechanism with Chord, which is one of the several DHT protocols available in the literature.

This work was done as a part of the integrated project IST-015964 AEOLUS: Algorithmic Principles for Building Efficient Overlay Computers [29]. Project AEOLUS has chosen JXTA as a platform over which the overlay computer has been built. We will describe the steps we followed in order to achieve the objective of improving the underlying JXTA framework resource management.

3.1 Introduction

JXTA is an open source project begun by Sun Microsystem in 2001, in order to provide a general framework for developing Peer-To-Peer (in short, P2P) applications. To this aim, JXTA defines and implements a set of protocol specifications which supply the developer with the basic services necessary to build P2P applications.

One of these protocols, that is, the *rendezvous protocol*, determines the way JXTA-based applications announce that a new resource is available and look for existing resources. In particular, this protocol manages a sub-network of the

JXTA overlay network by means of a *loosely-consistent Distributed Hash Table* (in short, DHT) mechanism, whose main component is the *rendezvous peer view*, that is, an ordered table in which each node of the sub-network maintains its partial knowledge of the sub-network itself. The “loosely-consistent” term comes from the fact that JXTA does not guarantee the consistency of all peer views: this may cause much more search misses during a resource lookup process with respect to a “pure” DHT. JXTA tries to solve this problem by using a walker mechanism based on a limited range replication performed during the resource publication process.

The JXTA designers justify the choice of this approach by emphasizing the high maintenance cost that a pure DHT would require in order to manage a resource publication/research process. In particular, in [30] it is said that “the cost of maintaining a consistent distributed index is likely to outweigh the advantages of having one, as we may spend most of our time updating indices”.

Our work started from this point, as we wanted to check if the use of a pure DHT could improve the performances of JXTA. To this aim we have chosen a DHT protocol, Chord, and we replaced the current implementation of the rendezvous service with it.

The process of this work has gone through many phases

1. Understanding JXTA
2. Understanding Chord
3. Reverse engineering JXTA
4. Designing how to implement Chord into JXTA
5. Implementing JXTACh
6. Testing JXTACh against JXTA

A great majority of time was spent on points 1, 3 and 4. It was due to the almost complete lack of the bibliography about JXTA 2.x at the time we started to work on JXTACh (except for the programming guide, which was unfortunately useless for the purpose of understanding the core of JXTA). The only source of documentation available was the source code and some technical papers found on the JAVA webpage. The remainder of this Chapter will continue with a brief overview of the related work followed by the description of the path we represented in the list above.

3.2 RELATED WORK

As far as we know there are very few projects which involve JXTA and “pure” DHTs. One is the JXTA subproject *jxta-meteor* [31, 32] which intends to provide a platform to develop DHTs (as for now the project includes Chord and

CAN [17]) over JXTA protocols (as services). Another project about DHT in JXTA is GISP [33] (*Global Information Sharing Protocol*): this is a proposal for a new DHT protocol, which is intended as a service to be put over JXTA.

The main difference between our JXTA implementation and the above two projects consists in the fact that our project is a new version of JXTA with a new implementation of the rendezvous protocol based on the Chord protocol: this means that our work is placed at a lower level, since our goal is to improve the publication/research process of *every* resource available within the overlay network. It is worth citing [34], which is the first attempt to give a formal description of a DHT-based routing and discovery policy within JXTA.

At the end of this Chapter we will present an experimental evaluation of the performances of JXTACh, by comparing it with the original JXTA version. To this aim we adopted the performance model introduced in [35, 36, 37], where the authors study the JXTA rendezvous protocol performances, by comparing it with the policy of older versions of JXTA, and by using a JXTA subproject benchmark suite [38]. These studies cover average response time, percentage of dropped query and RAM usage: our tests cover the same measures, by significantly increasing the number of peers involved in the simulation.

3.3 understanding jxta and chord

As we have previously said, the first step of our work was to understand the two technologies we were going to merge together, The JXTA framework and the Chord protocol.

3.3.1 *JXTA*

JXTA is an open source project which aims to develop a general framework for peer-to-peer applications. It was designed by Sun Microsystems and later extended thanks to a large number of experts coming from academic and industry institutions. The name JXTA comes from the word *juxtaposed* and intends to emphasize the concept of collaboration with the client-server model rather than its replacement by means of the peer-to-peer model. JXTA provides a common platform which contains the following features, that are recommended by a general peer-to-peer network.

- *Interoperability*: different peer-to-peer systems and communities have to be able to operate together.
- *Platform independence*: the system has to function independently from the programming language, the operating system and the network architectures.
- *Ubiquity*: the system has to operate on any device (not only PCs).

JXTA peers and basic concepts

Peers in JXTA can be subdivided into three categories, each of which reflects the heterogeneity of the devices which can assume the role of a peer (varying from small sensors, cellular phones and PDA, to personal computers, workstations and super computers).

- *Minimal edge peer*: these peers implement only the required core services and use other peers as proxies to access to other services; they can send and receive messages but they don't store advertisements; this role is usually played by devices with limited resources such as sensors or home automation devices.
- *Full-featured edge peer*: these peers implement all the JXTA services; they can send and receive messages and they store advertisements; this is the most common role played in JXTA network by all kind of devices.
- *Super-peer*: these special peers perform some key services for the deployment and functionality of a JXTA network; there are three types of super-peer.
  - **Rendezvous peer**: these peers maintain an index of references to the advertisements published by edge peers; they are also responsible for the propagation of the query and response messages during the advertisement lookup process.
  - **Relay peer**: these peers are used by peers which cannot be reached because of firewalls or NATs.
  - **Proxy peer**: these peers are used by minimal edge peers in order to get access to all JXTA functionalities.

peergroups Peers in JXTA organize themselves in *peer groups*, which have the same set of services and common interests. Each group, in fact, defines a set of services which are available to all the peer members of the group: in JXTA, there exist two core groups, that are the *World Peer Group* and the *Net Peer Group*. The first one is responsible only for the management of physical network connections, physical network (generally broadcast) discovery and physical network topology management. The second core group is the base peer group for applications and services within the JXTA network: most applications and services will instantiate their own peer groups using the Net Peer Group as a base.

services A *service* can be offered either by a single peer (*Peer Service*) or by a set of peers which collaborate to provide it (*Peer Group Service*). Services are described through *modules*, which are generic abstractions of each service or function implemented in JXTA. Services and applications communicate with each other through *JXTA Messages*, which are the basic units of data exchanged between

peers. Messages are sent through *pipes*, virtual communication channels established between two or more peers.

advertisements

In JXTA, each resource (such as peers, groups, pipes and services) is represented by an *advertisement*. By means of the advertisement, which is an XML document of a specified format, each resource communicates to the network that it exists. When a peer needs to find a resource, it has to look for its advertisement in the first place: the advertisement contains the information useful in order to describe the resource it refers to. In Listing 3.1 we report an example of an advertisement, which is relative to a peer.[1]

Listing 3.1: The **A Peer Advertisement**

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE jxta:PA>
3  <jxta:PA xmlns:jxta="http://jxta.org">
4  <PID>
5   urn:jxta:uuid-59616261646162614A78746150325033
6   AB718A7A3BED463D9559CEB1A618085B03
7  </PID>
8  <GID>
9   urn:jxta:jxta-NetGroup
10 </GID>
11 <Name>
12  RdvCarlo02
13 </Name>
14 <Desc>
15  Platform Config Advertisement
16  created by : net.jxta.impl.peergroup.AutomaticConfigurator
17 </Desc>
18 <Svc>
19  <MCID>
20   urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000605
21  </MCID>
22  <Parm>
23   <Rdv>
24    true
25   </Rdv>
26  </Parm>
27 </Svc>
28 <Svc>
29 <MCID>
30  urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000805
31 </MCID>
32 <Parm>
33  <jxta:RA xmlns:jxta="http://jxta.org">
34   <Dst>
35    <jxta:APA xmlns:jxta="http://jxta.org">
36     <EA>
37      tcp://150.217.37.222:3046
```

1 Lines 5-6, 40-41, 44-45 and 48-49 have been split for visualization reasons, they are one unique element.

```
38     </EA>
39     <EA>
40      cbjx://uuid-59616261646162614A78746150325033
41      AB718A7A3BED463D9559CEB1A618085B03
42     </EA>
43     <EA>
44      relay://uuid-59616261646162614A78746150325033
45      AB718A7A3BED463D9559CEB1A618085B03
46     </EA>
47     <EA>
48      jxtatls://uuid-59616261646162614A78746150325033
49      AB718A7A3BED463D9559CEB1A618085B03
50     </EA>
51    </jxta:APA>
52   </Dst>
53  </jxta:RA>
54 </Parm>
55 </Svc>
56 </jxta:PA>
```

JXTA IDs The most important information contained in an advertisement is the *JXTA ID*, which uniquely identifies any JXTA entity. A JXTA ID is expressed by a URN (Uniform Resource Name), a form of URI (Uniform Resource Identifier), which is intended to be a persistent, location-independent, resource identifier.[2] This identifier is a concatenation of different parts: the first one states that the URN is a JXTA URN, while the second one distinguishes between the `jxta` URN format, which is used for specific identifiers, and the `uuid` URN format, which is the most frequently used within JXTA. The first two parts are followed by a hexadecimal sequence: in case of a peer group the sequence contains 128 bits, while in case of a single peer it contains 256 bits (the first 128 bits encode the *World Peer Group*, while the second 128 bits encode the unique identifier of the peer).

messages To communicate with each others, peers use *messages*. In JXTA a message is structured through the XML format and each service defines its own message structure. When needed, messages can be merged or encapsulated in other messages, the XML format facilitates this process. We report in Listing 3.2 the structure of the Discovery Query Message.

JXTA protocols

JXTA defines a common set of open protocols that standardize the manner in which peers discover each other, self-organize in peer groups, advertise and discover network resources, communicate with each other and monitor one another.

These protocols are sub-divided into two groups: the *core specification protocols* which are the protocols that are necessary to a peer in order to be considered

2 See IETF RFC 2141.

Listing 3.2: The **Discovery Query Message**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2  <jxta:DiscoveryQuery>
3   <Type> . . . </Type>
4   <Threshold> . . . </Threshold>
5   <PeerAdv> . . .</PeerAdv>
6   <Attr> . . . </Attr>
7   <Value> . . .</Value>
8  </jxta:DiscoveryQuery>
```

a JXTA peer, and the *standard service protocols*, which are optional but strongly recommended to create a complete JXTA implementation.

- Core specification protocols
  - The *Endpoint Routing Protocol* (ERP) allows a peer to discover a route to another peer, in order to send a message through such a route. In the absence of the direct route between two peers, a peer can find an intermediary which will route the message to the destination peer.
  - The *Peer Resolver Protocol* (PRP) is used for sending a generic resolver query to one or more peers, and for receiving a response (or responses) on the query. The PRP protocol distributes the generic queries to one or more handlers within the group and matches them with the corresponding responses.
- Standard service protocols
  - The *Rendezvous Protocol* (RVP) is the protocol by which peers can subscribe to a propagation service or provide one. Peers can be rendezvous peers or standard peers that are listening to rendezvous peers. The RVP, therefore, allows rendezvous functionality and is used by the PRP in order to propagate messages.
  - The *Peer Discovery Protocol* (PDP) is responsible for publishing and discovering advertisements. PDP uses the PRP for sending and propagating discovery requests.
  - The *Peer Information Protocol* (PIP) allows a peer to obtain the status information on other peers, such as state, uptime, traffic load and capabilities (PIP also uses the PRP).
  - The *Pipe Binding Protocol* (PBP) is employed in order to establish a virtual communication channel or pipe between one or more peers. Using PBP a peer binds pipe ends to a physical endpoint address. PBP uses the PRP for sending and propagating pipe binding requests.

A complete description of all the protocols goes beyond the purposes of this work: for this reason we restrict ourselves to the discussion of the Rendezvous Protocol which is directly responsible for the way JXTA spreads the advertisement references.

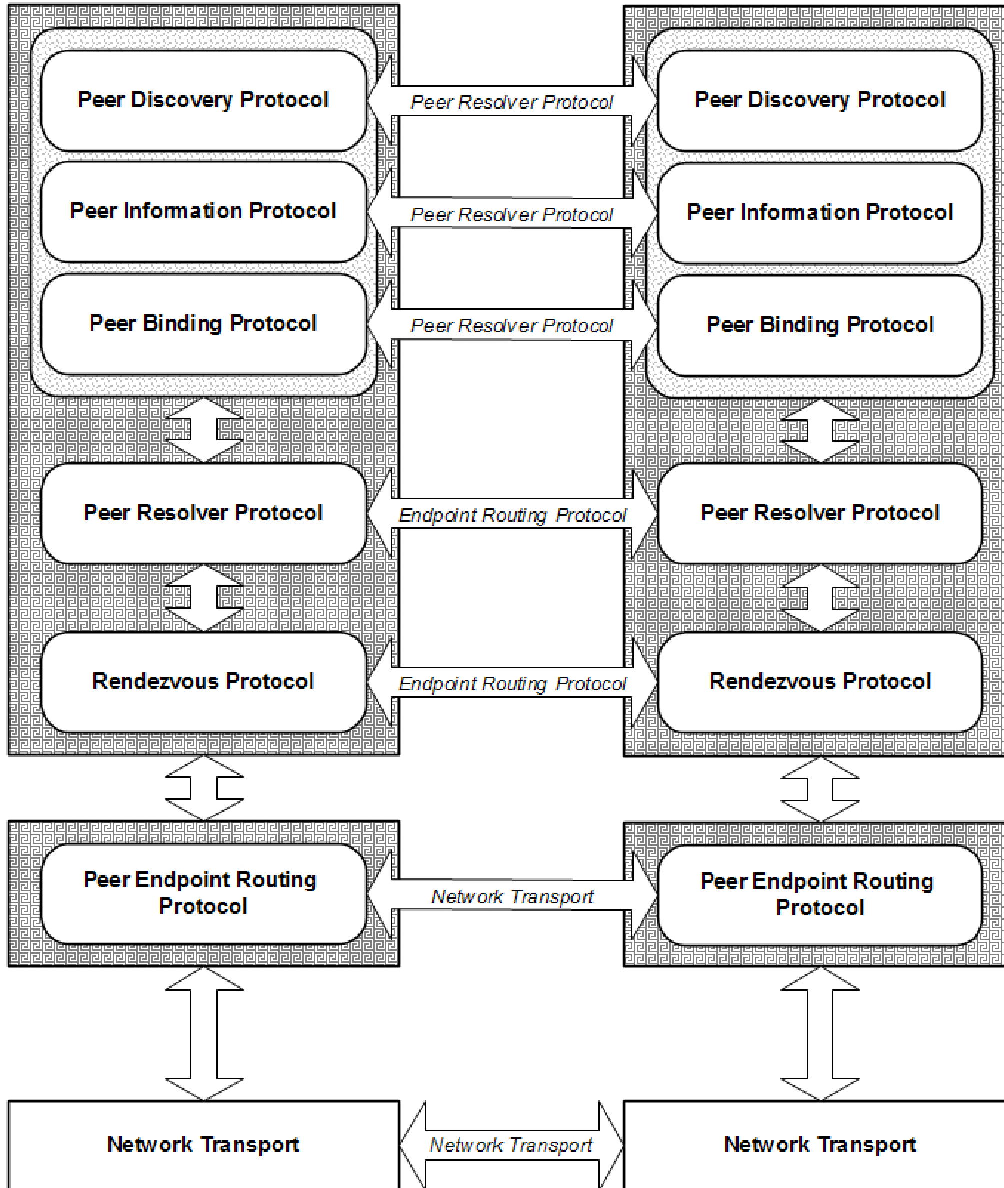


Figure 3.1: The JXTA protocols stack.

In Figure 3.1 the JXTA protocols stack and the protocols used for the communication between two peers are represented.

The JXTA layer architecture

The JXTA architecture is subdivided into three layers as shown in Figure 3.2.

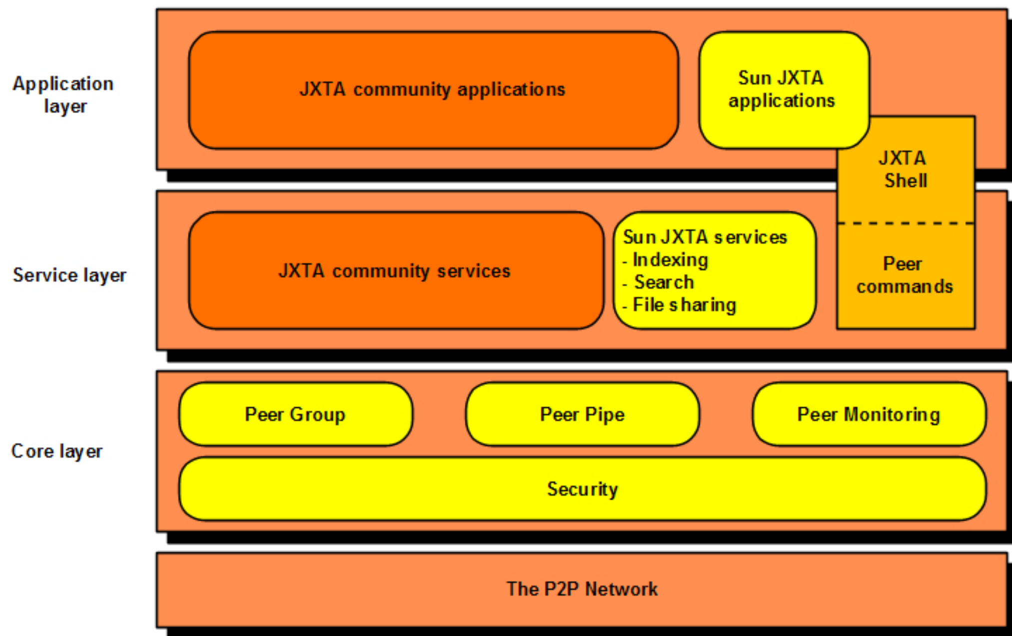


Figure 3.2: The JXTA layer architecture.

- The *core layer* provides the basic features needed by a P2P system, such as:
  - Peers and peer groups creation;
  - Communication system;
  - Security primitives.

  The six protocols we described in Section 3.3.1 are included in this layer, indeed they form the foundations over which all the JXTA services and applications are built.

- The *service layer* provides functionalities which are not necessary for a P2P system to work, but are usually desirable, such as:
  - Resource sharing (this imply a store/search system);
  - Peer authentication;
  - Distributed file systems.

  This layer includes all the services developed by the JXTA community and by the Project JXTA team. They are used as bricks to build any possible P2P application.

- The application layer includes all the classical P2P applications such as:

  - Instant messaging;
  - Video streaming;
  - File sharing.

Every JXTA application is built over the service layer, thus it is a combination of services. Sometimes the concepts of application and service are hard to distinguish. Usually it is the presence of some sort of interface which tells us that we are in presence of an application. However, the JXTA Shell, which has an interface, is implemented as a service. That is the reason why in Figure 3.2 it is represented like an entity over both the layers.

JXTA rendezvous network

The earlier versions of JXTA framework (1.x) were based on the advertisement publication and lookup by means of a flooding mechanism: each peer published and retrieved information through propagation of query and response messages. With the arrival of the version 2.x, JXTA passed to a new approach based on a *loosely-consistent DHT*: to build this mechanism *rendezvous peer* were introduced inside the JXTA structure.

When an edge peer joins the JXTA network, it establishes a lease connection with a rendezvous peer. Each resource the peer wants to offer to the network (including itself) will be published by means of an advertisement: a reference to this advertisement is sent to the rendezvous peer which maintains an index of the advertisement references. This index is called *Shared Resource Distributed Index (SRDI)* and it is one of the three components of the loosely-consistent DHT mechanism. The rendezvous service is, in fact, a set of three components.

- *Rendezvous Peer View (RPV):* an ordered table in which each rendezvous peer stores its partial view of the rendezvous sub-network.[3]
- *Shared Resource Distributed Index:* the index in which each rendezvous stores the advertisement references.
- *The walker:* is the mechanism used by a rendezvous peer in order to manage the research failure.

The RPV manages all the mechanisms to maintain the minimum level of consistency (loosely) among the peer view of the rendezvous peers. This is done through a periodic exchange of peer view entries between rendezvous peers. Each rendezvous selects a set of entries in its peer view, following a certain strategy which by default is set to be random. This set of entries is sent to all

3 Every kind of peers maintains a RPV, in fact there is a possibility for a peer to switch its mode from edge to rendezvous and vice-versa.

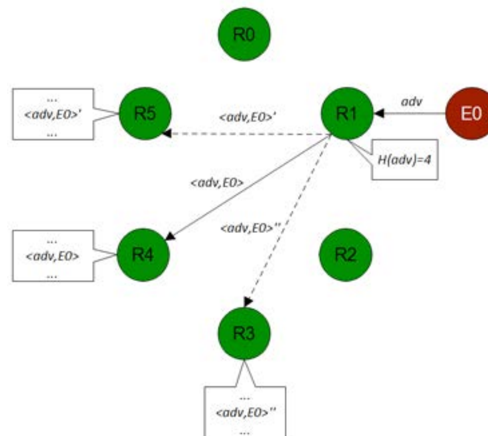


Figure 3.3: JXTA publication process

the rendezvous known by the local peer. If the changes in the network does not occur at a high rate, we can expect that the various RPVs can converge to a consistent common RPV. The assertion of a low rate of changes is anyway too strong in a P2P system, that is one of the reasons why JXTA needs the walker mechanism.

Actually, the word "loosely-consistent" is used because the main difference between JXTA rendezvous service and a pure DHT is that JXTA does not maintain the consistency among peers RPV. It causes much more search misses during the lookup process with respect to a pure DHT. JXTA tries to solve the problem of search misses by using the walker, which is based on the replication performed during the publication process. JXTA, indeed, replicates a reference to an advertisement into a predefined number of neighbors of the rendezvous chosen to keep the reference.

When a peer publishes an advertisement, it contacts the rendezvous it is connected to and sends to it a key to be used in order to build a reference to the advertisement (together with the publisher ID). The rendezvous applies an hash function (SHA-1) to the key and uses the result of the function in order to select a rendezvous within its RPV to which the reference will be sent. As mentioned before, the reference will be replicated also in the neighborhood of the target rendezvous.

In Figure 3.3 E0 publishes an advertisement through its rendezvous R1, the advertisement reference is propagated to R4 and replicated to R3 and R5.

When a peer looks for an advertisement, it contacts the rendezvous it is connected to and sends to it a discovery query containing the advertisement key. The rendezvous applies the same hash function used for the publication process in order to select the rendezvous where the reference should be stored and propagates the query to it. If the target rendezvous contains the reference inside its index, it then propagates the query directly to the peer owning the

advertisement. The latter will send the advertisement directly to the peer looking for it. If the rendezvous does not contain the advertisement in its index (because of the inconsistency of the RPVs), it starts the *limited range walker* process which consists of the propagation of the discovery query to the rendezvous neighborhood until the reference is found or a predefined number of hops has been performed by the message (causing a search miss).

In Figure 3.4 the network did not suffer any change, peer E1 looks for the advertisement published by E0 contacting its rendezvous R2, the discovery query is propagated to R4 owning the reference, R4 contacts E0 which, in turn, sends the advertisement directly to E1.

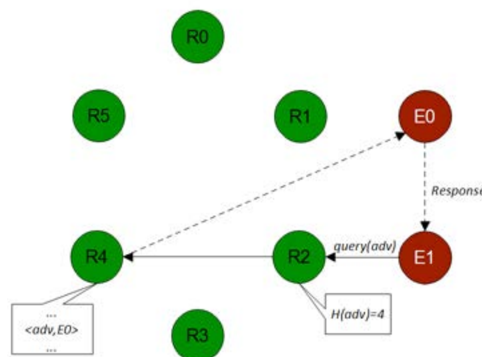


Figure 3.4: JXTA search process

In Figure 3.5 the network suffered a small change, a single peer failure, in this case R4. As we can see, in the new configuration of the network, the old R5 now becomes R4. Then when the discovery query is propagated by R2 to R4, it can be still satisfied thanks to the replication. Determining the number of replicas to spread is a challenging problem, because a too small value could bring as an outcome an excessive use of the walker, a too large value could result in a too heavy load on peers indexes instead.

In Figure 3.6 the network suffered a massive change due to the join of new peers into the system The figure represents the new view of the system. E1 still contacts its rendezvous, now having an inconsistent view of the network and the discovery query is propagated again to the current R4 (one of the new peers) which does not have the reference. This triggers the walker process, the query is propagated to both the direction (up and down in the peer view), until it reaches the current R6 which corresponds to the old R3. The reference is found and the query is forwarded to E1,which in turn sends the advertisement to E0.

According to the JXTA developers, the purpose of such an hybrid approach is to avoid the cost of maintaining the consistency of a pure DHT. Indeed, the RPVs

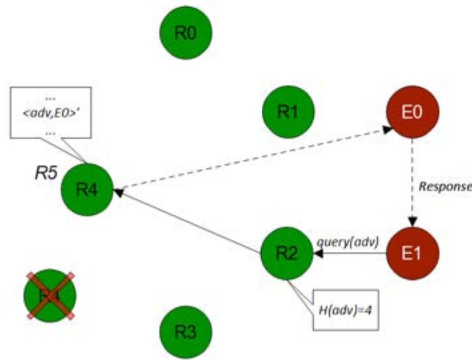


Figure 3.5: JXTA search process having success thanks to replication

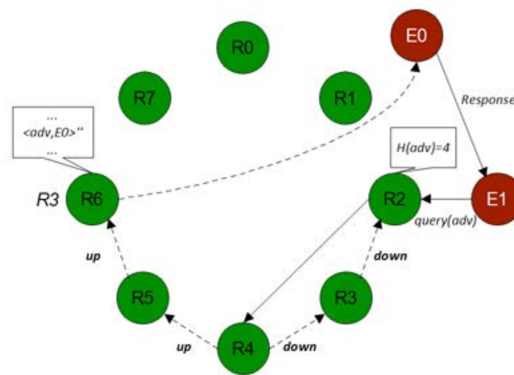


Figure 3.6: JXTA search process using the limited range walker

are maintained loosely consistent through an occasional exchange of information between rendezvous.

This is the question we want to check by the implementation of a pure DHT such as Chord serving as the rendezvous service.

3.3.2 *Chord*

Chord [39, 40] is a scalable protocol for the lookup in a dynamic peer-to-peer system with frequent node arrivals and departures, which uses a variant of *consistent hashing*.

Consistent hashing

Consistent hashing [23] maps nodes and keys into the same set of values through an hash function (i.e. SHA-1): nodes and keys are named through *m-bit* identifiers forming a circular domain modulo 2^m (essentially, $\mathbb{Z}_{2^m}$). An active node n is responsible for a key k if and only if n follows or is equal to k and there is no other active node between k and n. In this case, node n is called the *successor node* of k. In consistent hashing nodes can join and leave the system without harming the key distribution: in fact, when a new node enters the network, it receives from its successor the keys it should be responsible for. Instead, when a node leaves the network, it gives all the keys it is responsible for to its successor. To perform a research in such an environment, it can be enough to let a node maintain a pointer to its successor: a lookup would follow the ring until the node identifier is greater than or equal to the key identifier. This is correct but not efficient.

Scalable key location: the finger table

Chord implements a scalable key location: this is done by means of a particular structure called *finger table*, which is a routing table added to the information about successor and predecessor.

The finger table of node n has m entries, called *fingers*, and three fields for each entry i; with $1 \leqslant i \leqslant m$:

- $finger[i].start$: it contains the identifier $(n + 2^{i-1}) \bmod 2^m$;
- $finger[i].interval$: it is the interval $[finger[i].start, finger[i+1].start)$, by convention we assume that $finger[m+1].start = n$;
- $finger[i].node$: it is the first active node n' such that $n' > finger[i].start$.

In Figure 3.7 a Chord ring with $m = 3$ is represented (each active node is accompanied with its finger table). This kind of structure has two main characteristics: the first one is that each node knows only a small part of the

entire network and its knowledge decreases while going farther in the ring, whereas the second one is that the information contained in a single node is usually not sufficient to retrieve an arbitrary key (for example node p3 in Figure 3.7 does not know the successor of key 1, because node p1 is not present in its finger table).

When a node does not know the successor of a key k, it has to inquire about it the closest node preceding k: in this case, p3 has to contact p5, which knows p1. In general, this process is repeated until the active successor of the key is found: in [39, 40] the next result is proven.

Theorem 3.1 *With high probability, the number of nodes that must be contacted in order to find a successor in a N-node network is* $O(\log N)$.

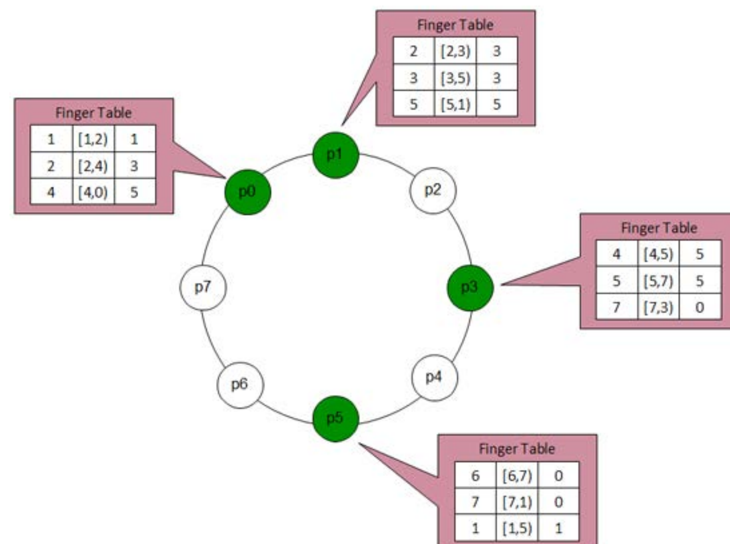


Figure 3.7: Finger tables for a chord ring with $m = 3$, green nodes are active.

Chord main processes

In this section we describe the main operations of the Chord protocol. The procedures implementing such operations are described using a pseudo-code.

THE FIND SUCESSOR PROCESS This procedure is the core of the Chord protocol, as every operation needs to know the successor of a key. It shall become apparent in the rest of our description. Procedure *find_successor* is described in Algorithm 1.

In Algorithm 1, id is the key of which successor we are searching, n represents the local node, in line 2 the check if the interval currently considered contains

Algorithm 1 Successor search: `find_successor`(id)

```
1: n' = n;
2: while (id ∉ (n', n'.successor)) do
3:     n' = n'.closest_preceding_finger(id);
4: end while
5: return n'.successor;
```

the key is made. If yes we return the right successor, otherwise in line 3 we call the *closest_preceding_finger* procedure which is described in Algorithm 2.

Algorithm 2 Preceding finger search: `closest_preceding_finger`(id)

```
1: for i = m downto 1 do
2:     if (finger[i].successor ∈ (n, id)) then
3:         return finger[i].successor;
4:     end if
5: end for
6: return n;
```

The *closest_preceding_finger* procedure visits in decreasing order the finger table to find the node which is the closest to the key we are looking for. Once such a node is found, it is returned as a result of the procedure.

It is clear that the procedure to find the successor of a node is a multi-hop process, at each iteration (line 2 of Algorithm 1) a new node is contacted.

THE STORE PROCESS When a node stores a new data, its hash value has to be computed. Then, the peer responsible for the key generated by the hash function has to be found. This is done through the *find_successor* procedure, described above. In Figure 3.8 we report the store process which proceeds in the following way: peer $p5$ is going to store key k; supposing that $\mathcal{H}(k) = 1$ (where $\mathcal{H}$ is the chosen hash function), k is sent to peer $p1$. Indeed, peer $p5$ knows directly the successor of 1 from its finger table (the information is in the third row).

THE SEARCH PROCESS The search process follows the same steps of the store one. The only difference is the last step in which the peer owning the data sends it to the peer requesting it. To be more precise, a lookup in a Chord ring for a certain key k consists essentially of searching the node n such that $n = successor(k)$. In Figure 3.9 we report the search process in which peer $p3$ looks for key k This time $p3$ does not know directly the successor of k, in fact, after consulting its finger table, $p3$ can at most send the query to peer $p0$ which is the closest preceding node to the key. Upon receiving the query, $p0$ discovers that the successor of k coincides with its successor, that is $p1$. Thus, the query is forwarded to $p1$ which, in turn, replies to $p3$ with the researched key.

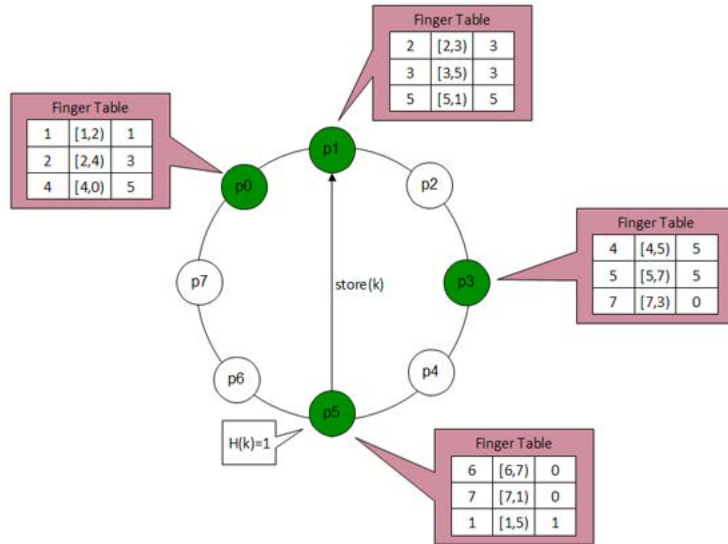


Figure 3.8: Chord store process

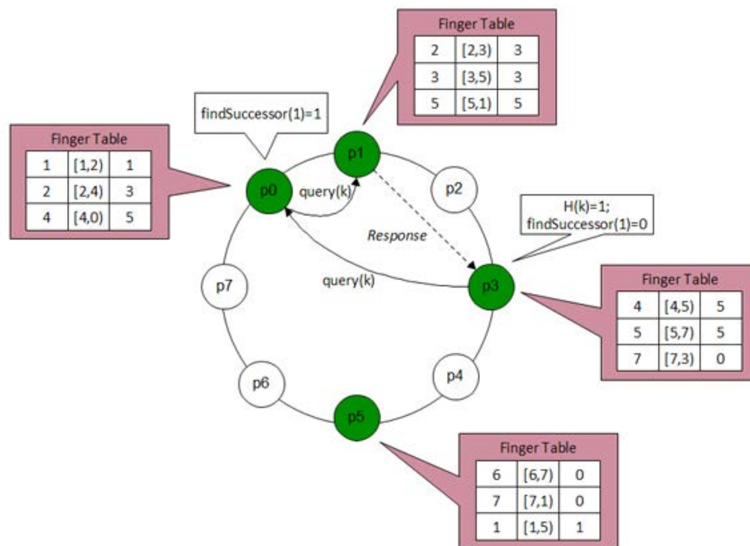


Figure 3.9: Chord search process

NODES JOIN AND LEAVE When a node n wants to join the network, it has to ask node n', already inside the ring (if it exists), to find the successor of n and to build a finger table for it: this is done by the *join* procedure. The join procedure is described in Algorithm 3.

Algorithm 3 Initialization: `join(n')`

```
1: if (n') then
2:    init_finger_table(n');
3:    update_others();
4: else
5:    for i = 1 to m do
6:       finger[i].node = n;
7:    end for
8:    successor = n;
9:    predecessor = n;
10: end if
```

The first check (line 1) is performed in order to control whether there is a node n' available to build the finger table for the local node n. If not, n initializes the ring. Then all the fields of the finger table and both the successor and the predecessor, are initialized with node n itself (lines 5, 8 and 9). Otherwise, the local node has to contact the node already located in the ring in order to "introduce" it (line 2). This means that the remote node n' has to compute both the finger table and the successor for node n. This has to be done on the basis of the information that n' has about the network (through its personal finger table). The `init_finger_table` procedure is described in Algorithm 4. The next step to be performed is updating the finger tables of the nodes preceding n in the ring, this is done through the procedure `update_others` which is described in Algorithm 5.

Algorithm 4 Finger Table creation: `init_finger_table(n')`

```
1: finger[1].node = n'.find_successor(finger[1].start);
2: successor = finger[1].node;
3: predecessor = successor.predecessor;
4: successor.predecessor = n;
5: for i = 1 to m − 1 do
6:    if finger[i + 1].start ∈ [n, finger[i].node) then
7:       finger[i + 1].node = finger[i].node;
8:    else
9:       finger[i + 1].node = n'.find_successor(finger[i + 1].start);
10:   end if
11: end for
```

The `init_finger_table` procedure initializes all the successor fields in the finger table entries. The first *node* field is found separately (line 1) as it will be used to set the *successor* and the *predecessor* of the local node (lines 2 and 3). Furthermore the successor is informed that its predecessor has been changed into the local node (line 4). The `for` statement (line 5) visits the rest of the finger table checking if the *node* field of the i-th entry is valid also for the $(i+1)$-th one (line 6). In that case the value is maintained (line 7). Otherwise we use the `find_successor` procedure to find the right *node* field for the entry (line 9).

Algorithm 5 Updating other nodes: `update_others`

1: **for** $i = 1$ to m **do**
2: $p =$`find_predecessor`$(n - 2^{i-1})$;
3: p.update_finger_table(n, i);
4: **end for**

The `update_others` procedure updates the finger tables of nodes that could have n as $node$ field in some of their entries. It is essentially a `for` loop which visit backward the ring calling a `find_predecessor` procedure.[4] In Algorithm 6 we describe the `update_finger_table` procedure.

Algorithm 6 Updating finger table entries: `update_finger_table`(s, i)

1: **if** $s \in [n, finger[i].node)$ **then**
2: $finger[i].node = s$;
3: $p = predecessor$;
4: p.`update_finger_table`(s, i);
5: **end if**

The last event of a node join is the transfer of the keys for which it will be responsible. This operation has to be done by the successor of the entering node. Let n be the id of the joining node, s the id of the successor determined for n and p the predecessor determined for n (which was the old predecessor of s). All the keys k such that $p < k \leqslant n$ has to be transferred to n.

The leave of a node corresponds to the change of the information in its predecessor and in its successor (essentially the first becomes the predecessor of the second one). The update of the finger tables of the nodes that could contain a leaving node is managed by the stabilization protocol that we are going to present in the following section.

STABILIZING THE RING In a P2P system peers can join and leave the network whenever they want. Such "freedom" causes unpredictable network environment

4 The `find_predecessor` just search for the predecessor of a key, it is similar to the `find_successor` procedure, we do not show its pseudo-code.

which leads to the most complex design challenge of a P2P protocol: how to make P2P service available under high churn? The Chord protocol is optimized in a way which manages this high churn and grants the best degree of coherence among finger tables.

The way Chord manages the high dynamism of the ring is obtained by two main procedures, `stabilize` and `fix_fingers`. The first one is described in Algorithm 7 and the second in Algorithm 9.

Algorithm 7 Checking the successor: `stabilize()`

```
1: x = successor.predecessor;
2: if (x ∈ (n, successor)) then
3:     successor = x;
4:     successor.notify(n);
5: end if
```

The `stabilize` procedure is executed periodically at a fixed rate. It is simply a check if our successor is still valid or if there is a need to upgrade it because of the entrance in the ring of a new node which has become our successor. Once the successor is changed we have to inform the new one that the local node has become its new predecessor. It is done by the `notify` procedure, described in Algorithm 8.

Algorithm 8 Notifying: `notify(n′)`

```
1: if ((predecessor = null) or (n′ ∈ (predecessor, n))) then
2:     predecessor = n′;
3: end if
```

The `fix_fingers` procedure is also executed periodically at a fixed rate in each node. It chooses randomly an entry of the finger table and calls the `find_successor` procedure with its *start* field as a parameter. The purpose of such a mechanism is to keep the finger table of the local node as consistent as possible with respect to the finger tables of remote nodes.

Algorithm 9 Upgrading entries: `fix_fingers()`

```
1: i = random(1, m);
2: finger[i].node = find_successor(finger[i].start);
```

In Chord it is very important for the successor pointer to be always updated. That is the aim of the `stabilize` procedure. However, the `stabilize` can be used also to detect nodes failures. In fact, if the successor does not reply to the local node after a given threshold, the local node can infer that the node has failed and it can inform the rest of the ring of its failure.

To be more efficient we could apply as well a procedure which checks if the predecessor is still up. For that reason in Chord a `check_Predecessor` procedure has been defined (Algorithm 10). This procedure performs the same operation, described above for the successor, this time with respect to the predecessor.[5]

Algorithm 10 Checking the predecessor: `check_Predecessor`

1: **if** (predecessor has failed) **then**
2: predecessor = null;
3: **end if**

According to the taxonomy we described in Section 2.3.1, Chord is an overlay structured P2P network following a logarithmic multi-hop strategy: its address space is flat and the distance metric used for the next-hop decision criteria is the linear distance in the ring. The topology, as we have seen, has a form of a ring, the graph obtained from the structure of the finger table has a logarithmic degree. The overlay network maintenance described is opportunistic and there is no cognition of the underlay network locality.

3.4 JXTA RENDEZVOUS SERVICE REVERSE ENGINEERING

The issues described in section 3.3.1 are a result of a selection of information both from books [41, 42, 43, 44] pertaining to the first versions of JXTA (1.x) and from papers and reports [30, 45] discussing its latest versions (2.x) written by the JXTA developers. What was completely missing was a guide which could enable us to move easily inside the JXTA code. We had to examine in details the code in order to understand in which way the things we learned from the above mentioned sources were implemented. We were able to understand thanks to the Javadoc and the comments disposed by the developers.

In the present chapter we will describe the elements we discovered during the reverse engineering process on which we spent a considerable amount of time. We will limit our description to the parts which were modified by us. The analysis concerned almost all the framework, nonetheless, for the purpose of this work it is enough to describe the three parts of the rendezvous service already shown in section 3.3.1.

3.4.1 *The Rendezvous Peer View*

The Rendezvous Peer View (RPV) is implemented in the package listed below.

- Package `net.jxta.impl.rendezvous.rpv`

5 In the condition at line 2 we hide the procedure based on a threshold we explained for the `stabilize`.

1. PeerView
2. PeerViewDestination
3. PeerViewElement
4. PeerViewEvent
5. PeerViewListener
6. PeerViewRandomStrategy
7. PeerViewRandomWithReplaceStrategy
8. PeerViewSequentialStrategy
9. PeerViewStrategy

All the classes inside this package aim at making the peer view work. The classes from 1 to 3 are responsible for the structure of the peer view, while classes 4 and 5 manage the events which can be generated by the view. Finally the last four classes manage the strategy thanks to which the peer view is visited.

The PeerView class is the central class of this package, all the other classes are auxiliary. In Figure 3.10 we represent a pseudo-UML class diagram of the package rpv,[6] we decided to give a different pattern to each group of classes and to represent the PeerView class with a different style to underline its importance.

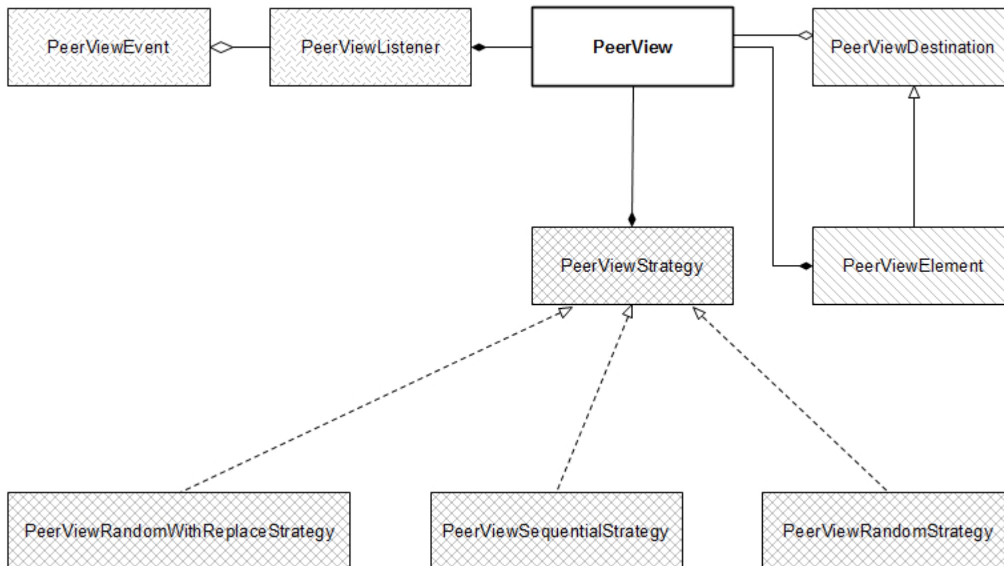


Figure 3.10: Pseudo-UML representation of the rpv package

The PeerViewElement class and the PeerViewDestination class implement the elements composing the peer view. As we can see from the class diagram,

6 This class diagram is a portion of the whole class diagram. The PeerView class, for instance, has other relations which are omitted.

the first one is the extension of the second one. This is justified by the fact that the JXTA peer view is an ordered table and `PeerViewDestination` implements the comparable part of an element of the peer view, which is the ID of the peer (see Figure 3.11).

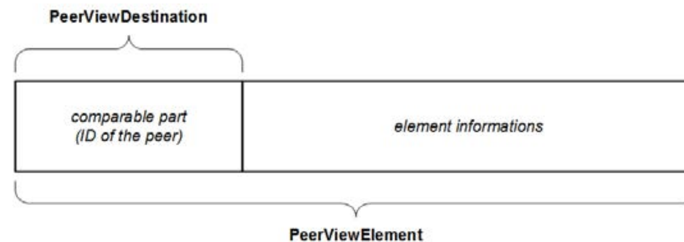


Figure 3.11: PeerViewElement schema

The attributes added by the `PeerViewElement` class consist of the following parameters:

- the time of creation of the element,
- the last update of the element,
- the Endpoint Service used by the element,
- the rendezvous advertisement of the peer represented by this element,
- the known state of the peer (alive or not),
- a messenger to send message to this peer.

The `PeerViewStrategy` class is an interface which defines the general strategy for iterating over the values in a peer view, in other words, it delineates the way the structure is visited. As we can see from the class diagram, JXTA defines three different types of iteration over the peer view. The `PeerViewSequentialStrategy` class implements a sequential iteration over the peer view elements,

- the `PeerViewRandomStrategy` class,
- the `PeerViewRandomWithReplaceStrategy` class,

both implement a random strategy, the first one by using a support data structure, the second one by using directly the peer view.

The `PeerViewEvent` class and the `PeerViewListener` class implement the event system in order to manage the dynamics of the peer view. Elements of the event system are the add, the remove and the failure of a rendezvous inside the view.

As we have mentioned above, the `PeerView` class is the main class of the package and it implements all the features described in Section 3.3.1. The description of each method can be found in the Javadoc furnished with the

JXTA framework. Thus, we will bring the description of some of it only when necessary.

The periodic exchange of information is implemented by a `TimerTask` which uses a `PeerViewStrategy` to implement the policy used for choosing the elements which shall be sent to the rendezvous peers known by the local peer. To be more precise, this operation is performed through a method named `kick()` which is periodically called by the `KickerTask`, an inner class extending `TimerTask`, in its `run()` method.

The `KickerTask` is not the only `TimerTask` used in `PeerView`. All the operations in need of a periodic execution are indeed implemented through the extension of this class. The `TimedSendTask` performs the periodic send of an advertisement to a specific destination peer. This operation has a limited duration in time and it is performed in the course of the initialization, if seed peers are used.

The `WhatchdogTask` checks periodically the availability of the up peer and of the down peer. The `AdvertisingGroupQueryTask` sends periodically a query request regarding the advertising group, a group in which the peer view advertises and broadcasts its existence. The `OpenPipesTask` makes sure that the peer view changes its behavior properly when switching from edge mode to rendezvous mode, and vice-versa.

3.4.2 *The Shared Resource Distributed Index*

The Rendezvous Shared Resource Distributed Index (SRDI) is implemented in the package listed below

- Package `net.jxta.impl.cm`
  1. `Cm`
  2. `Indexer`
  3. `Srdi`
  4. `SrdiCache`
  5. `SrdiIndex`

This package contains the implementation of the whole cache system (cm stands for cache manager). The interesting class here is Srdi, which contains the implementation of the mechanisms both to distribute and replicate the advertisements published by peers and, at the same time, to search for them. It is at this point that the system computes the hash function which decides where to send the replicas of the advertisements published or propagates a research query to the proper rendezvous.

In order to better describe the `Srdi` class, it is necessary to make a further step and to show the structure of the messages used by the SRDI.[7] An SRDI

7 SRDI is indeed a service, this means that it defines its personal message format.

message has the structure represented in Listing 3.3(it is implemented in the `SrdiMessage` class which is a part of the package `net.jxta.protocol`). We give

Listing 3.3: The **Srdi Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:GenSRDI xmlns:jxta="http://jxta.org">
    <PeerID> . . . </PeerID>
    <TTL> . . . </TTL>
    <PrimaryKey> . . . </PrimaryKey>
    <Entries key="" value="" expiration=""> . . . </Entries>
    .
    .
    .
    <Entries key="" value="" expiration=""> . . . </Entries>
</jxta:GenSRDI>
```

a brief description of the fields of the message,

- `PeerID`: contains the ID of the peer which is the source of the message.
- `TTL`: the time to live of this message, this is a limit to the propagation jumps of the message.
- `PrimaryKey`: the primary key of an Srdi message is the type of advertisement to be published or researched.
- `Entries`: this field is a list, as an advertisement can have multiple references: it is caused by the fact that we can search for an advertisement using different keys such as its name or its ID (possible keys depend on the kind of the advertisement). The attributes of this field are:
  - `key`: the attribute used for this publication (name, ID,...)
  - `value`: the value assumed by the attribute
  - `expiration`: the expiration time after which this reference will cease to be valid.

Each element of the list formed by the `Entries` fields is used to form the index which will be used as an argument for the hash function in the following way

$$index = (PrimaryKey + key + value) \tag{3.1}$$

During the publication process, the reference $< index, peerID >$ is firstly published in the rendezvous directly connected to the publishing peer, where the *peerID* element is the ID of the peer which published the reference. Then the SRDI has to select the right one to which it will send the reference, among

the rendezvous peers contained in the peer view. This is done by selecting the position of the element in the peer view through the following formula:

$$pos = \frac{h \cdot |PW|}{|HS|} \tag{3.2}$$

where

- $h = \mathcal{H}(\mathtt{index})$, where $\mathcal{H}$ is the hash function used (JXTA uses SHA-1)
- $|PW|$ is the cardinality of the peer view
- $|HS| = 2^b$ is the cardinality of the codomain of the hash function, b is the number of bits used to represent $\mathtt{index}$.

This computation is performed by a single method in the `Srdi` class, the method `getReplicaPeer`, which is used in both the processes, publication and search. Due to its importance, we report the whole code of the method in Listing 3.4.

As we can see from the listing, the method is assigned as an input parameter a `String` called `expression`: this is the implementation of *index* (epression 3.1). At lines 7-10 we can find the computation of $\mathcal{H}(\mathtt{index})$, at lines 15-17 expression 3.2 is computed, finally at line 18 the method selects the element at position `pos`.

Each entry of the SRDI message is processed by `getReplicaPeer`, this is executed by the method `replicateEntries`. This method prepares a set of bins each of which is associated to a single destination peer. Once the bins are ready, `replicateEntries` uses the method `pushSrdi` to propagate each reference to its proper peer.

We will describe the search process in the following section, as it involves the walker mechanism.

3.4.3 *The Walker*

The Walker is implemented in the packages listed below:

- Package `net.jxta.impl.rendezvous`
  1. `PeerConnection`
  2. `RdvGreeter`
  3. `RdvWalk`
  4. `RdvWalker`
  5. `RendezVousPropagateMessage`
  6. `RendezvousServiceImpl`

Listing 3.4: The `getReplicaPeer` method

```
public PeerID getReplicaPeer(String expression) {
 PeerID pid = null;
 Vector rpv = getGlobalPeerView();

 if (rpv.size() >= RPV_REPLICATION_THRESHOLD) {
  BigInteger digest = null;
  synchronized(jxtaHash) {
   jxtaHash.update(expression);
   digest = jxtaHash.getDigestInteger().abs();
  }
  BigInteger sizeOfSpace =
   java.math.BigInteger.valueOf(rpv.size());
  BigInteger sizeOfHashSpace =
   BigInteger.ONE.shiftLeft(8 * digest.toByteArray().length);
  int pos =
  (digest.multiply(sizeOfSpace))
                    .divide(sizeOfHashSpace).intValue();
  pid = (PeerID) rpv.elementAt(pos);
  if (LOG.isEnabledFor(Level.DEBUG)) {
   LOG.debug("[" + group.getPeerGroupName() +
                        + " / " + handlername +
                        + "] Found a direct peer " + pid);
  }
  return pid;
 } else {
  return null;
 }
}
```

7. `RendezvousServiceInterface`
8. `RendezvousServiceProvider`
9. `StdRendezvousService`

- Package `net.jxta.impl.rendezvous.limited`
    1. `LimitedRangeGreeter`
    2. `LimitedRangeWalk`
    3. `LimitedRangeWalker`

The first package is a general package implementing the rendezvous service. In this section we are focusing on the walker, thus we will describe just the classes implementing it. The classes implementing the walker mechanism in the `rendezvous` package are `RdvGreeter`, `RdvWalker` and `RdvWalk`. The `limited` package is entirely dedicated to the walker. Specifically, it implements the *limited range walker* described in Section 3.3.1.

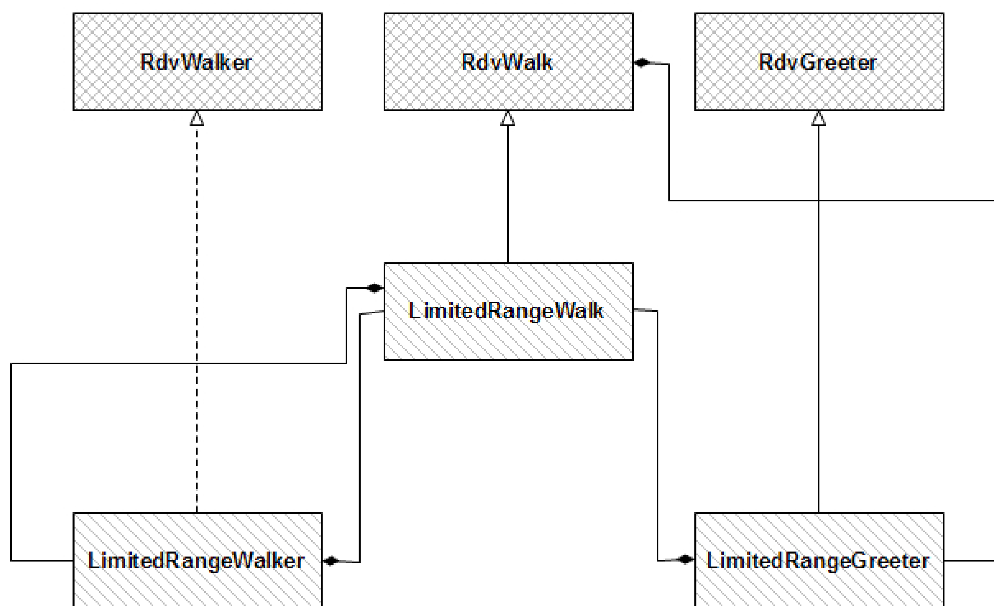


Figure 3.12: Pseudo-UML representation of the walker implementation.

In Figure 3.12 we represent a pseudo-UML class diagram of the walker implementation: in the upper part the classes which are a part of the `rendezvous` package can be found. They represent the general classes to be extended (or implemented in the case of `RdvWalker`) in order to obtain any policy we want to define for the walker mechanism. We will show in the next chapter how the chord policy was implemented as an extension of these classes (we created, in fact, a `ChordWalk`, a `ChordWalker` and a `ChordGreeter`). In the original JXTA, it was the limited range walker to be implemented.

The classes are structured in such a way because the walker mechanism is divided into two parts, the sending one and the receiving one: the sending one is `RdvWalker` (`LimitedRangeWalker`) while the receiving one is `RdvGreeter` (`LimitedRangeGreeter`). The `RdvWalk` (`LimitedRangeWalk`) is used as a container for those two parts. In Figure 3.13 we represent a high level schema of the walk structure.

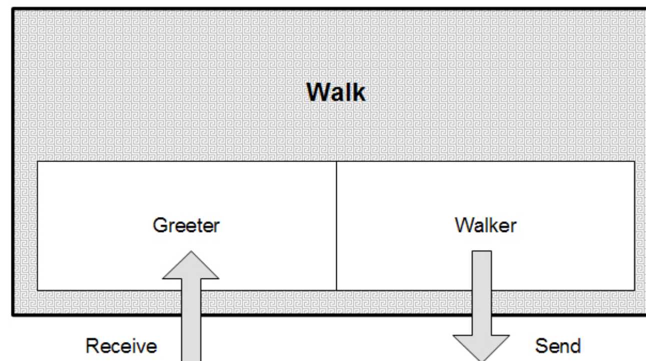


Figure 3.13: A walk schema.

As we said right above, the walker is responsible for the send operation of the propagated query, in Figure 3.14 we represent the interaction between the SRDI and the walker in the query resolution mechanism. All the following operations are invoked by the Discovery Service, such invocations are implemented in class `DiscoveryServiceImpl`.

1. A peer generates a query for an advertisement and sends it to the rendezvous it is connected to.

2. The rendezvous peer receiving the query checks if the advertisement is contained inside its cache: if yes: it replies with the advertisement directly to the sending peer, otherwise, a passage to the next step is needed, in which the work of the SRDI starts.

3. Firstly, we check the local instance of the SRDI in order to control whether it contains some references $< adv, peer >$ to the advertisement: if yes the rendezvous peer propagates the query to the $peer$ which, in turn, sends the advertisement to the peer that generated that query, otherwise, the next step follows.

4. This step consists of the computing the expression 3.2, if the result is different from the local peer we go to step 5, otherwise step 6 follows.

5. The query is forwarded to the peer contained at the position equal to the result of expression 3.2 in the peer view (the receiving rendezvous start from step 2).

6. The walker mechanism is invoked and we start to linearly propagate the message along the peer view.

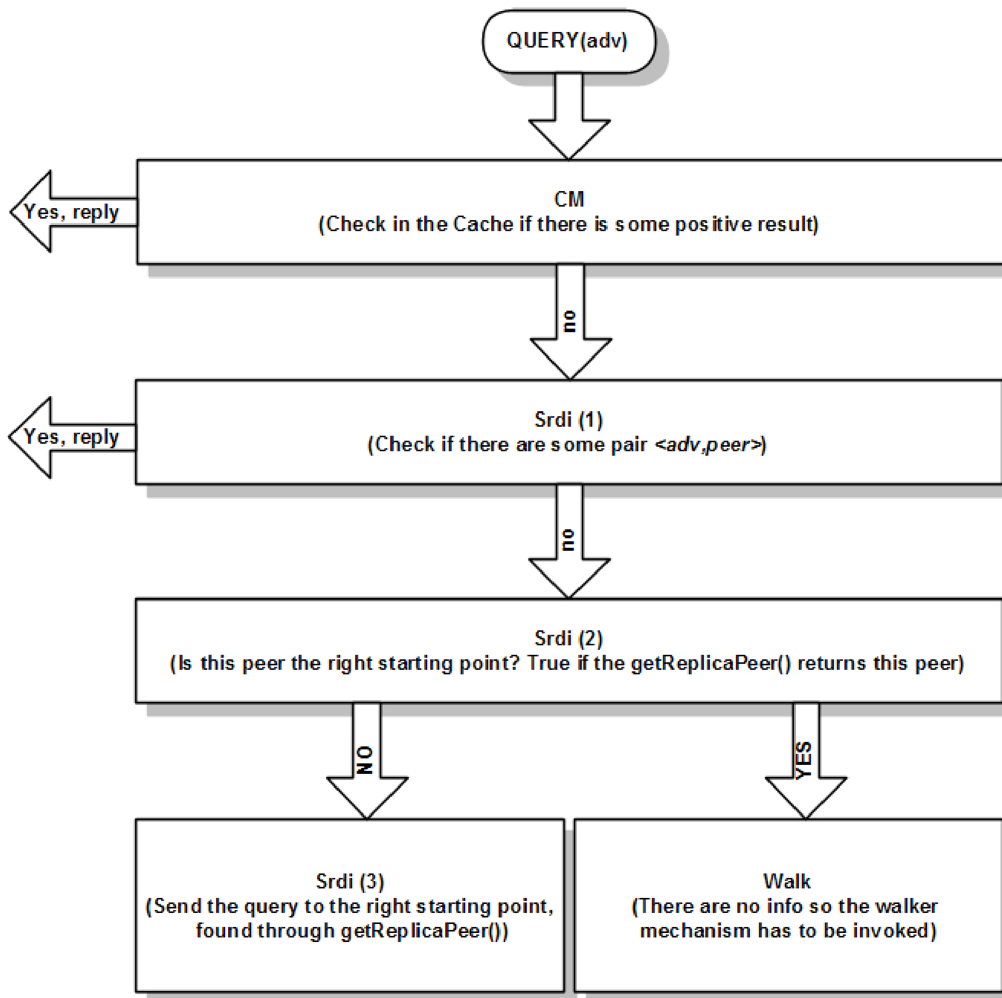


Figure 3.14: Query schema.

When a rendezvous peer contacts the walker for the first time, in order to propagate a query, the first operation performed by the walker consists in attaching a `LimitedRangeRdvMessage` to the message containing the query (a `DiscoveryQueryMessage`). This message is realized through two classes, the abstract class `LimitedRangeRdvMessage` in the package `net.jxta.protocol` and its extension `LimitedRangeRdvMsg` inside the package `net.jxta.impl.protocol`. It adds information which will be used during the walking phase. For the purpose of the description we report the structure of the `LimitedRangeRdvMessage` in Listing 3.5.

The message is composed by the fields below:

Listing 3.5: The **Limited Range Rdv Message**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <jxta:LimitedRangeRdvMessage xmlns:jxta="http://jxta.org">
3     <TTL> . . . </TTL>
4     <DIR> . . . </DIR>
5     <SrcPeerID> . . .</SrcPeerID>
6     <SrcSvcName> . . .</SrcSvcName>
7     <SrcSvcParams> . . .</SrcSvcParams>
8     <SrcRouteAdv> . . .</SrcRouteAdv>
9 </jxta:LimitedRangeRdvMessage>
```

- `TTL`: this element contains the time to live of the message. The value allows JXTA to limit the number of propagation jumps of the query during the walker phase. At each jump of the message, in fact, the walker decreases the value. The operation is performed by the `checkMessage` method in class `LimitedRangeGreeter`.
- `DIR`: this element contains a numeric value which indicates the direction the message shall follow along the peer view propagation. There exist three possibilities:
  - *UP*: the message has to be propagated to the up peer with respect to the peer view order (the numeric value is 1)
  - *DOWN*: the message has to be propagated to the down peer with respect to the peer view order (the numeric value is 2)
  - *BOTH*: the message has to be propagated to both the peers the up and the down one, with respect to the peer view order (the numeric value is 3). The field assume this value at the beginning of the walker phase as we saw in section 3.3.1. After the first step this field is modified into *UP* or *DOWN* according to the position of the receiving peer with respect to the sending one.

  These fields are used by the method `walkMessage` of the `LimitedRangeWalker` class, which is directly responsible for the propagation of the messages.
- `SrcPeerID`: this element contains the peer ID of the peer from which the query was originated.
- `SrcSvcName` and `SrcSvcParams` are the name and the parameters of the service which has to analyze the message when it is delivered.
- `SrcRouteAdv`: this optional element contains the route advertisement of the peer requesting the advertisement and is used in case in which target node do not know a route to the requesting peer.

This message will be analyzed by the `LimitedRangeGreeter`. This class, in addition to extending the `RdvGreeter` class, implements the `EndpointListener` interface (like all the classes in JXTA intended to be a "message receiver"). The message analysis is performed by the `processIncomingMessage` method which checks the `LimitedRangeRdvMessage` and then delivers the message to the proper service (through a chain of listener calls).

3.5 JXTACH DESIGN AND IMPLEMENTATION

After the reverse engineering process, we started the design process of a new version of the framework. In this work we limited ourselves to describe the rendezvous service. The reason of such a choice is that our change touched almost only this service[8], the rest of the framework remained unchanged.

In this section we give the reasons on the basis of which we designed the injection of the Chord DHT protocol inside the JXTA framework and, at the same time, we describe how we implemented it, on the basis of the information we collected during the previous phases.

In order to describe the design and the implementation of the rendezvous service in JXTACh, we will follow the chapter 3.4 schema. Each section will contain the design description and the implementation description.

The last section is dedicated to the description of an utility class implementing the check operations regarding the inclusion of a value in a certain interval embedded in a circular domain.

3.5.1 *The distributed hash table*

Design

As we have seen, the table in JXTA is implemented through the rendezvous peer view, an ordered table containing information about the rendezvous peers known by the local peer.

Our work begun from here. Indeed, the first step to take was to think of how to operate on the package `rpv` (see Section 3.4.1). The package maintained the same name but we decreased the number of classes, indeed some of the old classes were useless to the new implementation. We list below the new package:

1. `PeerView`
2. `PeerViewElement`
3. `PeerViewListener`

8 We had to modify small portions of Discovery service implementation as well. Is has not been reported here due to its little importance.

The new `PeerView` class is the implementation of the Chord Finger Table. We maintained the same name just for the reason of transparency with respect to the other parts of the JXTA framework. We have decided then to drop six of the existing classes, each of which supported by its valid reason. The removed classes are:

- `PeerViewDestination`: has been removed because because a comparable part in the elements of the table is no longer needed. As we saw in Section 3.3.2, the Chord finger table follows a different strategy, it is not just an ordered table according to the IDs order, it has a fixed infrastructure (the start fields structure seen in Section 3.3.2).
- `PeerViewEvent`: was used by the original JXTA to handle the *add*, *remove* and *failure* in the peer view. In Chord the table dimension is fixed according to the number of bits used to represent the IDs, then in JXTACh we do not have a concept of *add* and *remove* of elements. We have the concept of the *change* of an entry, which can be seen in three cases:
  - In case in which a new peer entering the Chord ring has to become its new *node* field (see Section 3.3.2).
  - In case in which a peer contained in the finger table leaves the ring.
  - In case in which a peer contained in the finger table fails.

  We have chosen not to use a special listener to deal with these events, but to handle directly with the `PeerView` class which is an extension of `EndpointListener` and `RendezvousListener`.[9]
- `PeerViewRandomStrategy`,
  `PeerViewRandomWithReplaceStrategy`,
  `PeerViewSequentialStrategy`,
  `PeerViewStrategy`: are used essentially to choose elements which are sent to other rendezvous peers in order to try to guarantee the consistence among the peer views (see Section 3.4.1). As we use the Chord protocol in JXTACh such a strategy is not needed. We apply instead the processes analyzed in Section 3.3.2.

Implementation

Before describing the classes of package `rpv` we have to describe a small change performed on the class which creates a new instance of the finger table/peer view. This class is `RendezVousServiceImpl`. We have had to postpone the creation of the finger table with respect to the procedure followed in the original JXTA. It appeared necessary, because during the testing phase, we have realized that in a high dynamic environment, with many peers entering together, we have

9 We left `PeerViewListener` for transparency reasons, the drop of this class is under evaluation.

encountered some communication problem. From our first experiments results, that was due to the lack of the routing information, particularly the Route Advertisement in some cases could not be created. The solution adopted was to wait for the termination of the route advertisement creation.

The `PeerViewElement` class implements an entry of the finger table. To give a complete description of the fields we refer to the Javadoc which is available together with the source code. Here we add a description to the added fields in order to obtain a finger table entry.

- `rdvJxtaID`: contains the *start* field of a finger table entry.[10]
- `radv`: contains the advertisement of the peer contained in the *node* field.
- `successor`: contains the ID of the peer contained in the *node* field.[11]
- `destAddress`: consists of the `EdpointAddress` contained in the *node* field.

In Figure 3.15 we give a graphic representation of the theoretical design of a finger table entry (top) with the relative practical realization (bottom). In the figure we also point out that the implementation of the *interval* field is useless because this information can be obtained without an explicit field (ith i^{th} interval is equal to $[\mathtt{fingerTable.start}(i), \mathtt{fingerTable.start}(i+1))$).

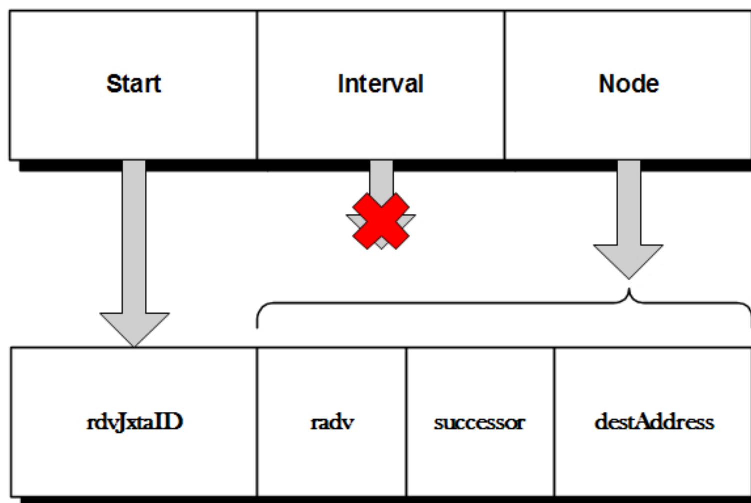


Figure 3.15: Finger table entry implementation.

The `PeerView` class is a class implementing the Chord finger table. The implementation starts directly from its JXTA counter part. That gives a possibility to maintain an higher level of compatibility with the rest of the framework.

10 The name is temporary.
11 It has this name because in some papers the *node* field is also referred to as *successor*.

processRdvRequest	processRdvResponse	waitingFingerTable
processFtRequest	processEdgeFtRequest	receiveAck
processFtResponse	processFsRequest	processFsResponse
processStabilizeRequest	processStabilizeResponse	processNotify
processFixFingersRequest	processFixFingersResponse	processPredecessorNotify
processKeysMessage	processRouteUpdate	

Table 3.1: Message processing methods.

We have left some ideas from the original `PeerView` class, some of the code is unchanged, but the greatest part of the original code has been replaced with a new code.

Here we have implemented all the processes described in section 3.3.2, Below, follows the classification of the methods implemented by us:

- Initialization methods
- Maintenance methods
- Termination methods
- General or utility methods

In the remaining part of this section we will describe the `PeerView` class through these categories. Before arriving to that process, it is important to give a description of the communication system used in this class. This is an `EndpointListener` interface implementation, thus the messages used for the initialization, the maintenance and the termination of the finger table are managed through the characteristics inherited from it.

The `EndpointListener` interface defines a method which manages the incoming messages, called `processIncomingMessage`. In our implementation, this method is used as a dispatcher: it checks the fields of the message received and, on the base of the one present, it chooses the right method to call in order to process the message. In Table 3.1 we report all the methods, each of which is responsible for a specific message (the name of the method is quite self-explanatory). While describing the `PeerView` class we will point out the actual use of these messages.

initialization (join of a new node into the network) When a new peer enters the system, it has to be initialized in order to participate in the protocol. In Figure 3.16 we represent the join procedure from the message exchange point of view. For the remainder of this section, we will refer to the entering node as a *local peer* and to the introducing peer as a *remote peer*. In our implementation, the first operation performed by a new peer in order to join the network is to check for existing rendezvous peers. These rendezvous peers

are responsible for giving to the new peer all the necessary information so that it follows the protocol (they are the potential introducing peers as shown in Section 3.3.2).

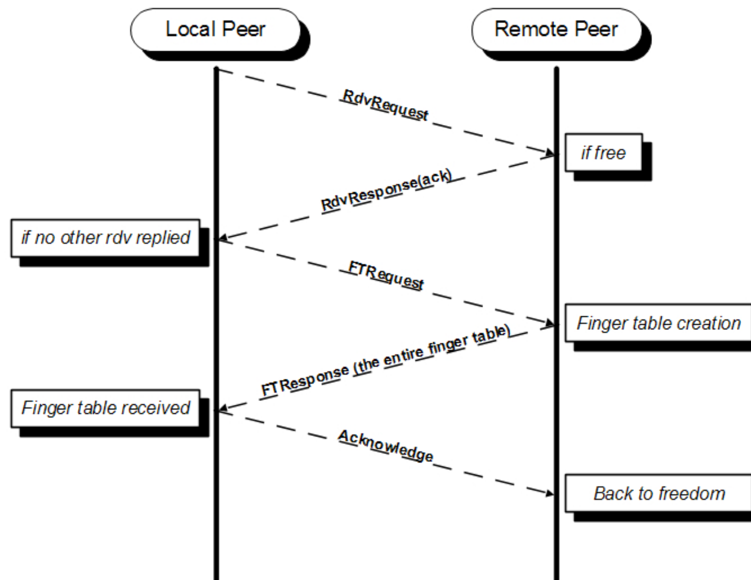


Figure 3.16: The join process, in the boxes there are the operations performed when the message is received.

The check performed by the new peer follows a specific procedure, At first, the new peer searches for *seed peers* (if they are defined), which are special peers that should be placed into "always-on" machines. Their name derives from their function being the roots over which the protocol grows up. Together with the seed peers check, the local peer tries to find rendezvous peers in its local network broadcasting a message.

The join communication protocol

This process is implemented through an inner class `RdvMessageTask`, an extension of the `TimerTask` class. This task sends periodically (the period is set through a static attribute of the class) a message (RdvRequest) to the seed peers and broadcasts the same message to the local network. The message contains a request to join the network, namely the request to find a remote peer available for introducing the local peer into the network.[12] When it goes to receiving, the message is managed by the `PeerView` class itself (the `processRdvRequest` method), if the remote peer is free (it has not already accepted another peer's request), it accepts to introduce the local peer into the network through sending to it an acknowledge message (RdvResponse). When the local peer receives the acknowledge (the `processRdvResponse` method), it sends the real request for a finger table (FTRequest) to the remote peer if and only if there was not any prior reply from another rendezvous peer (we remind that the request is sent

12 This is done by both the main type of peers, rendezvous and edge.

to all the seed peers and to the local network as well). Now the remote peer can start the finger table creation procedure (the `processFTRequest` method). Once finished the finger table is sent back to the local peer, which, in turn (the `processFTResponse` method), sends back an acknowledge message (Acknowledge) to the remote peer. Finally, the remote peer can go back to a free state. At the end of this process the local peer becomes a part of the Chord ring.

The finger table construction

We have just described the communication protocol which allows to add a new node into the ring. We have left undefined the issue of building the finger table for the local peer by the remote peer. This mechanism surely deserves a proper description.

The method `createFingerTable` is responsible for the finger table creation. At first, the method checks if the finger table of the remote peer should be updated as a consequence of the join of the new node. It is done through visiting the finger table and updating the *node* information if needed. Let ID_l be the ID of the local peer, we call $\mathcal{FT}_i$ the i-th entry of the finger table of the remote peer. We use the point notation to refer to the fields of the entry. Below we have given its formal description:

$$\mathcal{FT}_i \text{ is updated} \Leftrightarrow ID_l \in [\mathcal{FT}_i.start, \mathcal{FT}_i.node) \quad (3.3)$$

where with *start* and *node* we refer respectively to the ID contained in the field `rdvJxtaID` and to the ID contained in the field `successor` (see Section 3.5.1).

Once the remote peer has updated its finger table, it can start to create the finger table of the local peer. The remote peer builds a new finger table (through `findStartSuccessor` method) on the basis of its personal information, namely its finger table, and, if it is necessary, contacts the other rendezvous peers with the *find_Successor* method for the finger table creation purpose.[13]

There are in fact two cases for each of the considered *start* field. Let us call $start_i$ the i-th start field considered during the finger table construction, where m is the dimension of the finger table. The two cases are

1. $start_i \in (\mathcal{FT}_0.start, \mathcal{FT}_{m-1}.start]$
2. $start_i \notin (\mathcal{FT}_0.start, \mathcal{FT}_{m-1}.start]$

In Figure 3.17, we graphically represent the visibility of the remote peer which generates the two cases listed above. The dashed line represents the *visibility interval* and the start fields falling in this range correspond to case 1, while the dot-dashed line represents the *invisible interval* and the start fields falling in this interval correspond to case 2.

In the first case the $node_i$ field can be determined solely with the information contained in the remote peer finger table (this is done with use of the

13 There exist multiple versions of *find_successor* process in our implementation. It is due to the fact that there are multiple moments in which this process is needed, each of which has some special requirements.

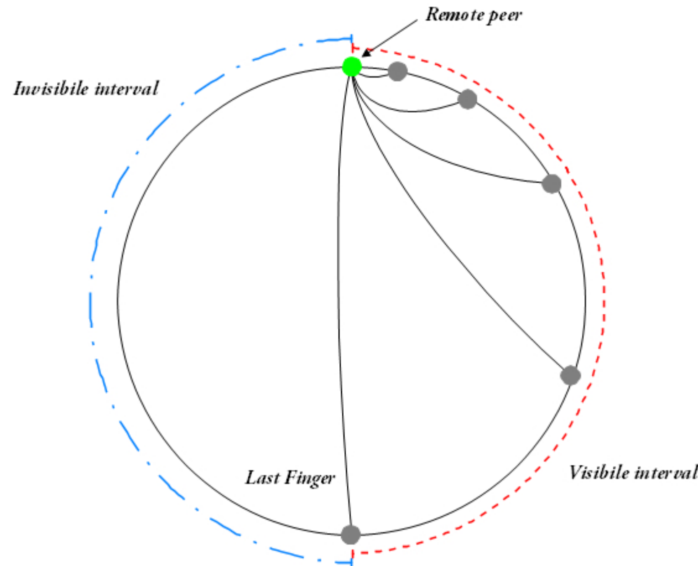


Figure 3.17: Finger table visibility interval.

`checkFingerTableEntry` method). At first, we have given a temporary value to the $node_i$ field, chosen among the ID of the remote peer, the ID of the local peer and the ID of the predecessor of the remote peer. Then the finger table is visited and when the condition

$$\mathcal{FT}_j.node \in [start_i, node_i)$$

is verified, the $\mathcal{FT}_j.node$ becomes the new $node_i$.

In the other case, the finger table of the remote peer does not have the necessary information to determine the $node_i$ field. One solution to this problem could be to set the field with the remote node id, we have decided to follow a different approach. We observed that the peer contained in the last $node$ field has surely more information about the current start field taken into consideration. Hence, we contact the last $node$ and we use its finger table in order to determine the right $node_j$, with use of the same process described above (the search of the right $node_j$ is done by `searchSuccessorIntoFingerTable` method). Any inconsistency can be corrected by the fix fingers process which will be described later on.

The communication process is managed through a `TimerTask` extension, `FindSuccessorTimerTask`. The last $node$ is contacted and a message is sent periodically to minimize the probability of the request to be lost.

There remains the last operation to be done after a peer join: the keys transfer. As we have seen in Section 3.3.2 the keys (in JXTACh case the references to the advertisements) for which the local peer should be responsible of, have to be transferred to it from the successor assigned to the local peer.

This operation is performed by the `sendKeysInterval` method. This method selects from the Srdi the references that have to be maintained by the new peer. This is done visiting the Srdi and for each reference r which is of the form $< peer_i, adv_j >$ we apply the following:

$$k = \mathcal{H}(r) \tag{3.4}$$

and then we have

$$r \rightarrow \begin{cases} \text{sent} & \text{if } p < k \leqslant n, \\ \text{not sent} & \text{otherwise.} \end{cases}$$

where n is the new peer and p is the successor of n.

MAINTENANCE A DHT needs to be maintained consistent in order to achieve its best performances. We have seen in Section 3.3.2 that there are some procedures that guarantee the consistency of distributed hash table. Here we are going to describe how these processes are implemented in JXTACh. All these three procedures are periodic and the best way to implement it has been, in our opinion, by using a `TimerTask` extension for each procedure. Below we have listed the name of each process with its respective timer task:

- *stabilize* → `StabilizeMessageTask` (see Algorithm 7)
- *fix_fingers* → `FixFingersMessageTask` (see Algorithm 9)
- *check_predecessor* → `CheckPredecessorMessageTask` (see Algorithm 10)

Stabilize process

The stabilize timer task sends periodically a message to the local peer successor to check if any new node entered the network. It means that the local peer has to update its successor with any new entered node. The message could be expressed in a question "who's your predecessor?": if the reply contains the local peer ID, nothing changes, otherwise the successor has to be modified by adding the peer ID contained in the reply message (if it's correct) and the local node has to notify the new successor that it has become his predecessor (Algorithm 8). Formally, let us call id_n the ID of the local node n, id_s the ID of the successor s, id_p the ID of the current predecessor p of s and finally, s' is the new successor after a stabilize run, we have

$$s' = \begin{cases} p & \text{if } id_n < id_p < id_s, \text{ and } n \text{ has to notify } p \\ s & \text{if } id_n = id_p. \end{cases}$$

At the same time the stabilize process is used to check if the successor is still alive. The local peer maintains a counter to remember how many times the stabilize message did not receive an answer. This is done to decide when a peer considers its successor failed. Of course, it is not possible for a single peer

in a distributed environment to distinguish between a failed peer and a very slow peer. For this reason we decided to give a time threshold over which we consider the peer failed. This limit states how many times the peer can send the same stabilize message to its successor without receiving a reply. For example, the current value for the number of stabilize messages which does not receive a reply is set to 20. The message is sent each 15 seconds. So the total period of time we give to a peer for giving a sign of life before retaining it failed, is 300 seconds (i.e. 5 minutes).[14]

When a peer is considered failed, the local peer sends a notification to the other peers to let them know that the peer is failed then it chooses the first `node` field in its finger table, different from the failed one, as its new successor. This node probably will not be the right one, but the stabilize mechanism finds the right successor in few steps.

All these operations are performed in the `run` method of the timer task implementing the stabilize process. The messages are managed by two of the methods shown in Table 3.1, `processStabilizeRequest` and `processStabilizeResponse`. In case of a successor update instead, a notify message is sent and it is processed by the `processNotify` method.

The fix fingers process

The *fix fingers* process checks periodically if the fingers of the finger table are correct or need to be updated. This is done by choosing randomly a finger in the table and starting the find successor procedure for the *start* value. In our implementation there is an on-purpose method which implements the find successor mechanism for the fix fingers process, it is the `findSuccessorFix` method. The message receiving, for requests and for responses, is managed respectively by `processFixFingersRequest` and `processFixFingersResponse` methods. At the end of the chain of calls for the find successor process, when the response is processed, the peer which generated the call can check if the randomly chosen finger has to be updated or not.

The check predecessor process

The *check predecessor* process checks periodically if the predecessor is alive. As in the stabilize process, we have set a threshold to define when the predecessor peer can be considered failed, the default threshold is set to 20 messages without a reply and the interval of message send is set at the same level of the stabilize one (15 seconds). To simplify this process management we have decided to leave the responsibility for that directly to the `processIncomingMessage` method. Let us observe that the check does not announce its predecessor's failure as it is performed by the stabilize of the predecessor of the failed node.

TERMINATION When a peer voluntarily exits from the network, there are some operations which have to be performed in order to let it go. First of all, the peer has to warn its successor and its predecessor about its leaving. This is

14 This time is totally customizable, due to the fact that we have released all the source code, maintaining the JXTA philosophy.

getRadv	getSelfID	getPredecessor
setPredecessor	getSuccessor	setSuccessor
getLocalFinger	publishRadv	publishRouteAdv
publishRouteFromRdv	translateAddr	translateID
translateIDfromString	updateSelfFTEntries	updateFTEntry
addSeed	getFingerTableElement	getView
getRelayPeers	checkValue	

Table 3.2: General or utility methods.

done through a simple message exchange in which the exiting peer sends to its successor a message "here is your new predecessor" referring to its predecessor and symmetrically, communicates to its predecessor "here is your new successor" referring to its successor. In our implementation this is implemented through the `leave` method.

The `leave` method is responsible as well for the second operation which has to be performed before the peer is able to leave. The keys it was responsible for have to be transferred to its successor. This is done by the method `sendKeys` which is called by `leave`. The method has a direct access to the SRDI[15] from which it collects the stored keys. Upon completion, the keys are sent to the successor which will process the message through the `processKeysMessage` method.

After these two operations the peer has to stop all the tasks active at the moment, the `leave` method is part of the method which does all such things which it is method `stop`.

GENERAL OR UTILITY The last set of methods implement getters and setters for some of the attributes of the `PeerView` class and a set of operations which are useful but not necessary to the implementation of the protocol. For a complete description of such methods, we address the reader to the Javadoc of the framework. A list of them can be found in Table 3.2.

3.5.2 *The distributed index*

Design

The class structure of the package `cm` has not changed. In fact, it was perfectly feasible to modify directly the source code of the class in order to obtain the Chord behavior. We had to modify some methods and to add some new ones. They will be described in the next section.

15 to permit this access we added some methods in the `cm` package, for the description see the next section

Implementation

In Section 3.4.2 we have seen how the SRDI is implemented. We have stated as well that the most important class was the `Srdi` class, in which distribution and replication of keys take place. For that reason, the main modifications performed by us have been related to this class. The following methods have been modified:

- `replicateEntries`
- all the three `forwardQuery`
- `getReplicaPeer`
- `getGlobalPeerView`

and the following methods have been added

- `closestPrecedingFingers`
- `deleteKey`

As it was for the original `Srdi` class, the central method used in the new version is `getReplicaPeer` This time, of course, the method has to deal with a peer view which is implemented as a finger table, hence, it behaves quite differently. We underline the fact that the method interface has not changed, it still receives a `String` (see expression 3.1 in Section 3.4.2) and returns a `PeerID`. We report the entire code in Listing 3.6.

As we have seen, the old `getReplicaPeer` is limited to computing the position of the rendezvous peer responsible for the key processed currently. In the new version of the method, after the hash function has been calculated, we have to check if the local peer is the right successor for the key processed (lines 20-24). If this is the case the method returns the local peer ID, otherwise we have to find the closest preceding finger for the key (line 30, method `closestPrecedingFinger`). If the local peer is the closest preceding finger, we directly return the successor ID. We have to remember that this process is embedded in both the main operations involving the Srdi, the publication and the search.

The `replicateEntries` method was modified as well. In fact in the old version of the Srdi, a publication could not jump through the rendezvous network. Here we have the Chord ring and, as we have already seen, the publication process follows the same behavior of the search one. Then in the new version of the method we added the ttl decrement to limit jumps of the Srdi message (it is set to $m = 128$).

As listed above we have changed four other methods: the three versions of `forwardQuery` and the `getGlobalPeerView` method. The first three were slightly changed, we have just modified the time to live, we have set it to $m = 128$, the latter one returns the elements of the `PeerView` and it has been modified to deal with the new structure of the class.

Listing 3.6: The new `getReplicaPeer` method

```
public PeerID getReplicaPeer(String expression) {
 BigInteger bigDig;
 PeerID pid = null;
 Vector rpv = getGlobalPeerView();
 RendezVousService rvs = group.getRendezVousService();
 PeerView fingerTable = rvs.getFingerTable();
 synchronized(jxtaHash) {//hash computation
  jxtaHash.update(expression);
  BigInteger bigDig = jxtaHash.getDigestInteger().abs();
  bigDig = bigDig.mod(modulo);
 }
 BigInteger a =
  new BigInteger(fingerTable.
      getSelfID().toString().substring(46, 78),16);
 BigInteger b = null;
 if (fingerTable.getPredecessor()!=null)
 b =
  new BigInteger(fingerTable.getPredecessor().
                  toString().substring(46, 78),16);
 if (b!=null &&
    && CircularDomain.inCloseDx(bigDig.mod(modulo),
    b.mod(modulo),
    a.mod(modulo),
    modulo))
 {
  return fingerTable.getRadv().getPeerID();
 }
 else {
  // let's search the closest preceding finger
  PeerID pidRes = closestPrecedingFinger(bigDig).getPeerID();
  if (pidRes.toString().
      equals(fingerTable.getSelfID().toString()))
  {
   pidRes = ((PeerViewElement)fingerTable.
              getLocalFinger().get(0)).getRadv().getPeerID();
   return pidRes;
  }
  return pidRes;
 }
}
```

3.5.3 *The walker*

Design

We decided to maintain the walker structure of the original JXTA. The structure of the new walker mechanism is the same as the one of the limited range walker. There is a new package, `net.jxta.impl.rendezvous.chord`, which replaces the `limited` package described in Section 3.4.3. The new package contains the following classes:

- `ChordWalk`
- `ChordWalker`
- `ChordGreeter`
- `ChordMsg`
- `ChordMessage`

As we have already anticipated in Section 3.4.3, the class structure of the old walking policy was maintained. We added here the implementation of the message used for the Chord policy as, in our opinion, it is more logical to have included all the classes in the same package. Again, it has been easy to reuse the pre-existing structure in order to design the message propagating policy of Chord.

Implementation

We followed the JXTA style implementation, indeed the `ChordWalk` class is an extension of the `RdvWalk` interface and it has the same role of the `LimitedRangeWalk` in the old version of JXTA. It is a container for the two classes which takes care of the input and output of the messages.

The Chord Rdv Message

Before describing the two main classes we have to describe the new kind of message we implemented for JXTACh, the `ChordMessage`. In Listing 3.7 we report the new message structure. If we compare it with the old version (the `limitedRangeMessage`), we can see that there are two new fields: the `Jump` field and the `Hash` field, where the latter in some sense replaces the `DIR` field of the old message (see Listing 3.5).

The `Jump` field signals to the processing class that this message has arrived to the destination and that it has to be processed by the local peer. To be precise, the field can assume two values, true or false. In the first case, the result is the one described above, in the second case the message has to jump, following the Chord policy.

The `Hash` field is used for storing the hash value which is used for the propagation during the find successor process. We maintained the possibility to use

Listing 3.7: The **Chord Rdv Message**

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jxta:ChordMessage xmlns:jxta="http://jxta.org">
3      <TTL> . . . </TTL>
4      <Hash> . . . </Hash>
5      <Jump> . . . </Jump>
6      <SrcPeerID> . . .</SrcPeerID>
7      <SrcSvcName> . . .</SrcSvcName>
8      <SrcSvcParams> . . .</SrcSvcParams>
9      <SrcRouteAdv> . . .</SrcRouteAdv>
10 </jxta:ChordMessage>
```

the old direction, defining new values for the UP, DOWN and BOTH directions. the values are, respectively, 2^m, $2^m + 1$ and $2^m + 2$. It was necessary in order to maintain the compatibility with the old version and that is why we have stated above that this field replaces the old `DIR` field.

The rest of the message remains unchanged, we point out that the `TTL` is set when the message is created, to $m = 128$.

The Chord Walker

The `ChordWalker` class is structured like the `LimitedRangeWalker` class, which is its counterpart in JXTA. There are new methods added and some changed, to manage the Chord protocol. The first method we describe is the `sendMessage` method, this was modified with respect to the one in `LimitedRangeWalker` class. At first the method checks if the `ChordMessage` has been already defined (during a propagation process) or no (beginning of the propagation process). In the latter case we have to create a new `ChordMessage` which sets all the fields. The most important one is the `Hash` field which will guide the message through the propagation along the Chord ring.

To compute the hash value we have implemented the `getHashFromMsgType` method. It was necessary to implement such an on-purpose method because there are different types of messages passing through the walker: the Resolver query message (which contains a Discovery query message) and the Resolver response message (which contains a Discovery response message) both with a different field to extract and to use in order to compute the hash value.

The following three methods complete the Chord walker implementation, thus we give a brief description of each method:

- `walkChordMessage`: checks if the message has to be treated like in the old protocol or if it has to follow the find successor mechanism,
- `findSuccessor`: implements the find successor process described in Section 3.3.2 in Algorithm 1,
- `closestPrecedingFinger`: implements the closest preceding finger described in Section 3.3.2 in Algorithm 2.

Finally, we have implemented a new `send` method, which consists in sending the message directly through the Endpoint service.

The Chord Greeter The `ChordGreeter` class is essentially responsible for the following tasks:

1. receiving the message containing the `ChordMessage`,
2. extracting the `ChordMessage` from the message,
3. checking if the `ChordMessage` has the right structure,
4. extracting the information contained in the `ChordMessage`, with particular interest in the `Jump` field,
5. according to the value of `Jump`
   a) propagating if the value is true.
   b) delivering the message to the stack of protocols if the value is false (the successor peer has been reached).

The main method performing this tasks is the `processIncomingMessage` method. It is in fact responsible of tasks 1, 4 and 5. Tasks 2 and 3 are performed by the `getChordMessage` and the `checkMessage` methods respectively through a call in the main method.

3.5.4 *A simple utility class*

We have added the `CircularDomain` class to the framework. This class implements inclusion check in an interval embedded in a circular domain (essentially $\mathbb{Z}_{2^m}$). It was, in our opinion, the best solution in order to perform the comparisons requested by the Chord Protocol (see Section 3.3.2). The class is a collection of four static methods. Each method is relative to any possible type of interval, open, close, left close and right close. The `CircularDomain` class is a part of the `net.jxta.impl.rendezvous.finger` package.

3.6 EXPERIMENTATION PHASE

In this section we will describe the results we obtained during the experimentation phase of JXTACh.

The environment for our tests was a LAN composed by 39 PCs of the same kind: they were all AMD ATHLON X2 Dual Core 5600+ 2,9 GHz with 4GB of RAM, running Debian Linux. In order to obtain a larger rendezvous network we decided to run 5 rendezvous peer in each machine, while in the machines dedicated to the edge peers we decided to run a single peer.

We have chosen to use the following metrics in order to perform our analysis (these metrics are also proposed in [37] and are, in our opinion, the most relevant ones for a performance comparison of this kind of protocols):

- *Lookup time:* the time required to get the result of an advertisement research operation.
- *Memory load:* the percentage of memory used by a single rendezvous peer.
- *CPU load:* the percentage of CPU time used by a single rendezvous peer.
- *Dropped query percentage:* the percentage of query lost.

To trace and to measure the memory and the CPU load, we used the linux command `top`.

Moreover, we have distinguished between a static and a dynamic environment: in the first case, rendezvous peers are stable and do not disconnect from the overlay network, while in the second case rendezvous peers can disconnect making the system unstable and forcing the protocols to react to the situation. In particular, we have analyzed two different ways to introduce dynamism into the network: the first one is to let the peers disconnect in a "gentle" way (that is, by announcing their departure from the network), while the second one is to make the peers fail, so that the protocol cannot predict when a peer will abandon the network (we call that an "abrupt" disconnection).

Finally, we have chosen to use the following set of parameters to be varied in order to study the network behavior under different conditions:

- *Query rate*: this tells us how much query load the protocol can stand.
- *Presence of negative query*: this tells us if the presence of negative queries imply increasing or decreasing performance.
- *Gentle or abrupt disconnections of rendezvous peers*: this tells us how good is the protocol to react to a search miss and how good is the system to solve an "unexpected" situation.

In the remaining part of this chapter, we will use the following notation:

- R : set of rendezvous peers.
- E_p : set of publishing peers.
- E_r : set of searching peers.
- k_p : number of advertisements to be published.
- k_r : cardinality of an advertisement block to be searched.
- t_q : query rate for the searching peers.
- lt_{min} : minimum life time of a peer.
- lt_{max} : maximum life time of a peer.

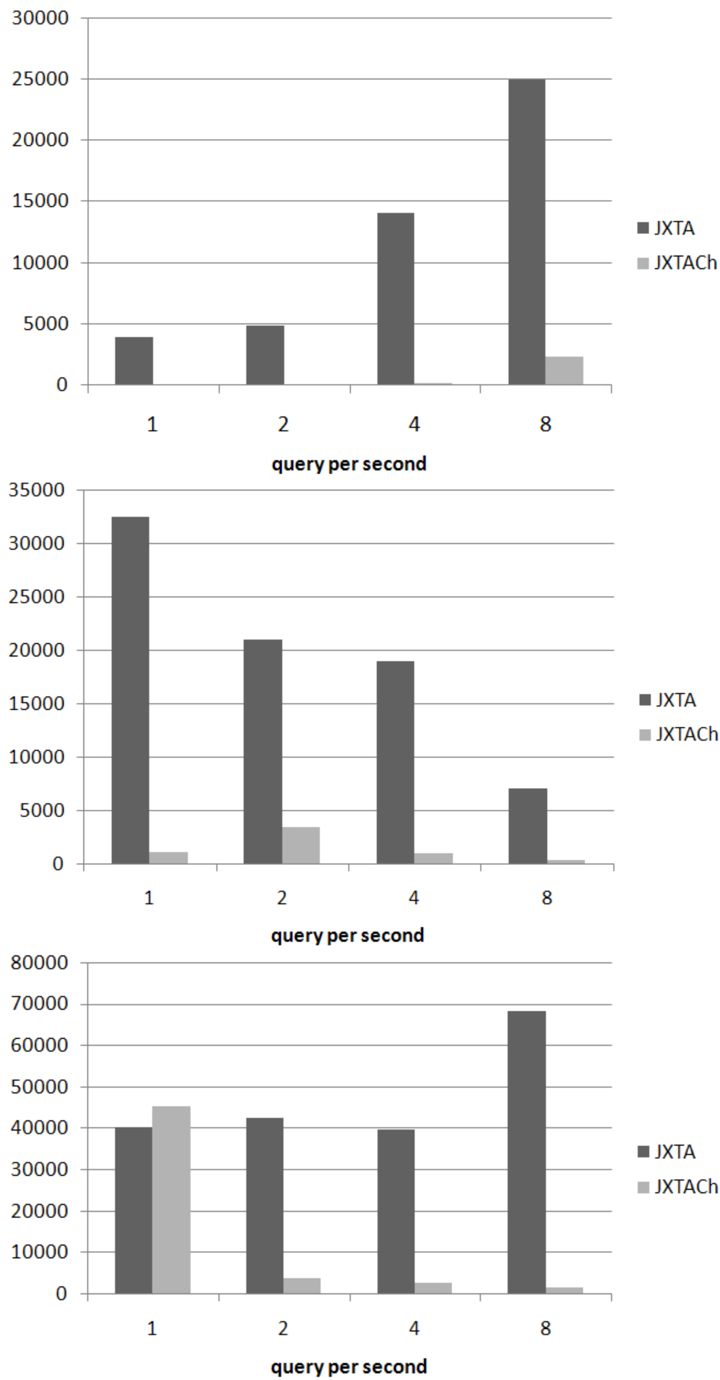


Figure 3.18: Average lookup time (milliseconds) in static, "gentle" dynamic and "abrupt" dynamic environment.

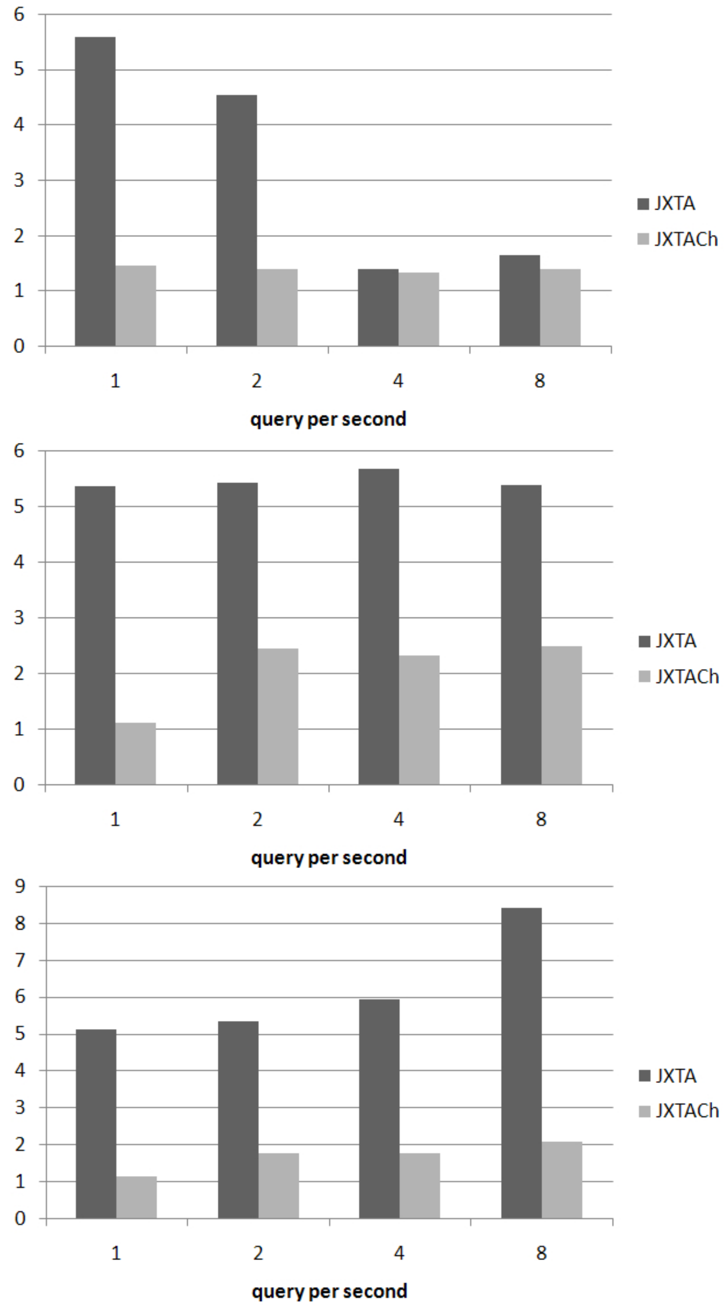


Figure 3.19: Average RAM usage percentage comparison in static, "gentle" dynamic and "abrupt" dynamic environment.

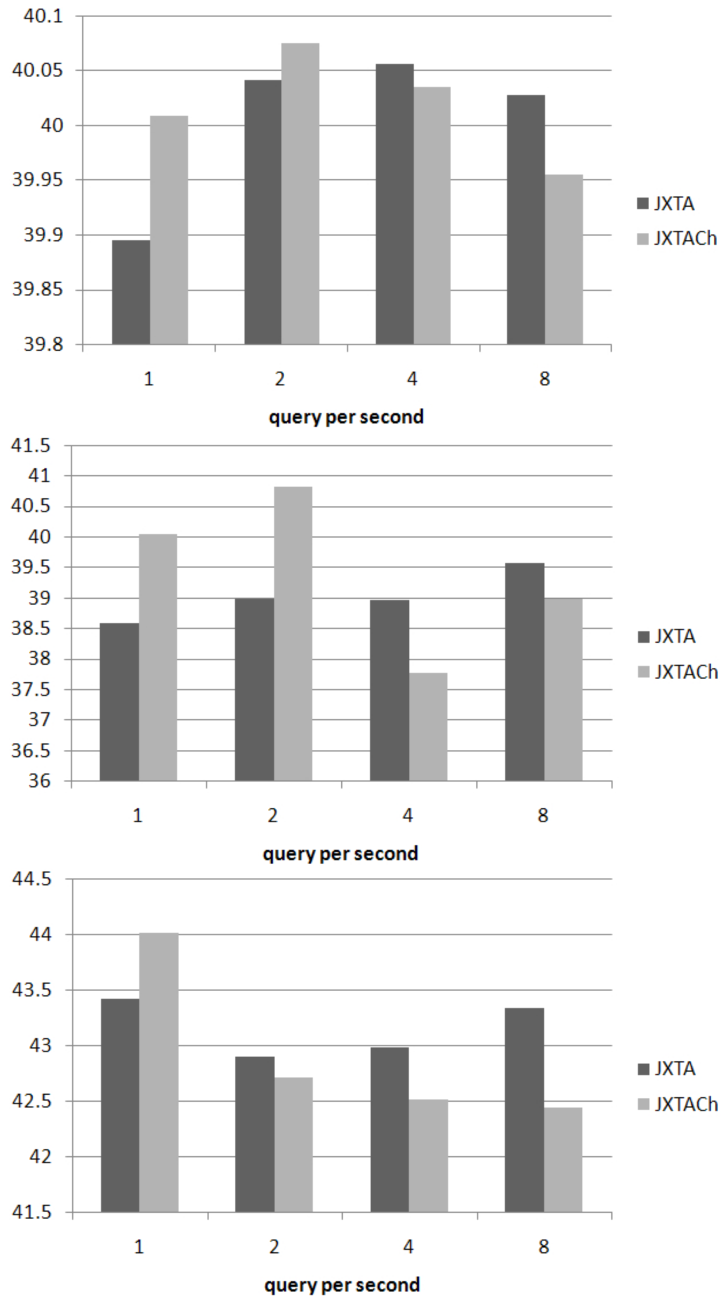


Figure 3.20: Average CPU time usage percentage in static, "gentle" dynamic and "abrupt" dynamic environment.

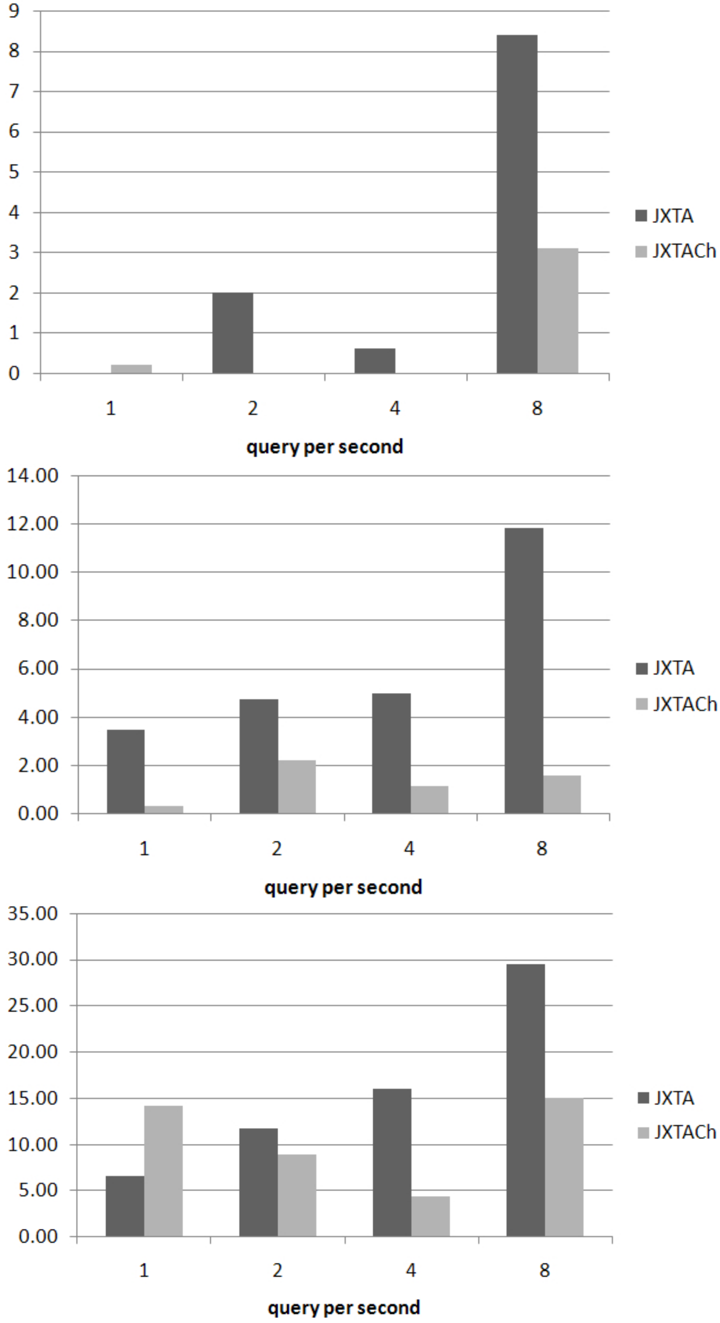


Figure 3.21: Average dropped query percentage comparison in static, "gentle" dynamic and "abrupt" dynamic environment.

- rt : maximum reconnection time.

Finally, the values of all parameters which are not explicitly mentioned are set equal to the JXTA default values.

3.6.1 *Static case*

In this case, we set up an overlay network with $|R| = 160$, $|E_p| = 3$ and $|E_r| = 3$. Each edge peer $e \in E_p$ publishes $k_p = 1000$ advertisements, while each edge peer $e \in E_r$ looks for the k_p advertisements in blocks of $k_r = 100$: for each block, e turns itself off and waits a small amount of time rt to reconnect and to restart with the next block. We experimented $t_q = 1, 2, 4, 8$.

Response time

In the left part of Figure 3.18 we can observe the comparison between the average lookup times: it is clear that, in a static situation, JXTACh is much faster than JXTA. The main reason is likely to be the heaviest traffic load that JXTA generates: indeed, the original protocol makes a massive use of broadcast operations.

Memory load

The left part of Figure 3.19 reports the average percentage of RAM used by the two protocols. Even in this case, it is clear that JXTACh uses less memory than JXTA. The first reason is likely to be the use of the finger tables, which limits the dimension of the peer view to 128 fields, while the rendezvous peer view, in the LAN case, might contain all the peers involved in the test (i.e. $|R|$). Another reason could be the fact that a bigger amount of messages is sent by JXTA, thus overfilling the buffers.

CPU load

In the left part of Figure 3.20, it can be seen that JXTA has a slightly better performance with respect to JXTACh in the case of 1 and 2 query per second: this small difference (less than 1% in each case) might be induced by the cost of the Chord finger table maintainance. In the 4 and 8 query per second case, instead, we can see that JXTACh has a better behaviour: this is likely to be due to the cost of the messages sent by JXTA, that starts to influence the percentage of CPU time used. Observe that it is anyway difficult to draw any deeper conclusion from such small differences.

Dropped query

In a static environment there should not be a big loss of queries, since the rendezvous peers are always on line. The results confirm this fact, as we can see in the left part of Figure 3.21: we always have less than 9% of queries lost (the larger value is in the case of 8 query per second). We also notice that in JXTA case the percentage is much higher than in JXTACh: again, this is likely to be due to the fact that JXTA uses much more messages that overfill the buffers causing the loss of queries.

3.6.2 *Dynamic case with gentle disconnections*

In this case, we have chosen the following parameter values: $|R| = 160$, $|E_p| = 3$, $|E_r| = 3$, $k_p = 3000$, $k_r = 200$, $rt = 40$ minutes. We experimented $t_q = 1, 2, 4, 8$. Each rendezvous peer $r \in R$ connects to the overlay network, remains connected for at least $lt_{min} = 2$ hours and for at most $lt_{max} = 6$ hours.

Response time

In the middle part of Figure 3.18 we can see that, even in this case, JXTACh has a better behavior than JXTA. For each query rate, indeed, JXTACh is one order of magnitude faster than JXTA. We can also notice that, in the case of JXTA, the average response time decreases as we increase the query rate: this might be surprising, but it is likely to be due to the fact that the percentage of dropped queries increases as the query rate increases. This means that the protocol performs better at expense of reliability.

Memory load

In a dynamic environment the percentage of memory used by the protocols increases with respect to the static case, but, as it can be seen by comparing the left and the middle part of Figure 3.19, the increase in the case of JXTA is higher than the one in the case of JXTACh. This is especially true as we increase the query rate: for example, in the case of 4 query per second, in JXTA we go from 1,5% to 5,5%, while in JXTACh we go from 1,5% to 2,2%. It's interesting also to compare the behavior of the two protocols with respect to a single rendezvous peer. In Figure 3.22, we show the values of memory load sampled each 5 seconds during the whole test in a single rendezvous peer: as we can see, JXTACh is clearly less memory-consuming than JXTA.

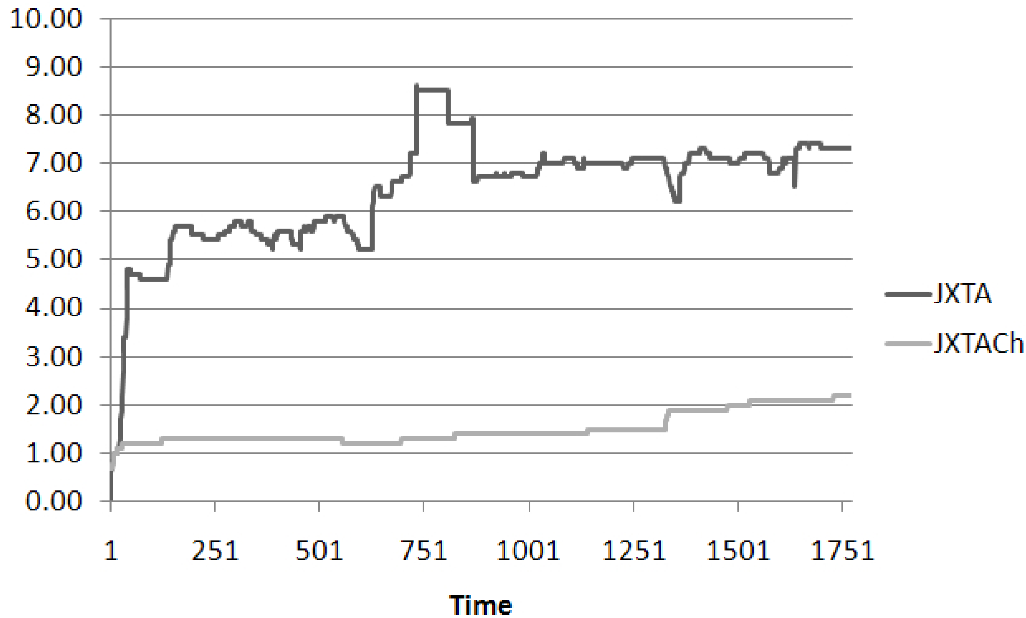


Figure 3.22: RAM usage percentage in a rendezvous peer (sampled each 5 seconds) in a "gentle" dynamic environment.

CPU load

In this case, the CPU load does not differ too much from the static case, as it can be seen by comparing the left part of Figure 3.20 and the middle part of Figure 3.20.

Dropped query

As expected, in a dynamic environment the loss of query starts to be significant, and, again, JXTACh has a better behavior. As we can see from the middle part of Figure 3.21, the percentage of dropped queries increases with the increase of the query rate in the case of JXTA. In the case of JXTACh, instead, the percentage remains stable: at most, we have a 2% of queries lost.

3.6.3 *Dynamic case with abrupt disconnections*

In this case, the test setting is exactly the same as the one in the previous section. This time we observe how the two protocols react to unexpected disconnections.

Response time

In this environment, the performances of the two protocols are comparable only in the case of 1 query per second, while for the other query rates JXTACh continues to perform better (see the right part of Figure 3.18). As we expected

there is a decay of performances of both protocols with respect to the "gentle" case due to the unexpected disconnection of rendezvous peers, but there is still a difference of one order of magnitude between the two protocols.

Memory load

The memory usage increases only in JXTA: indeed, JXTACh still uses the same amount of RAM, while JXTA's memory usage raises up till 8,4% in the case of 8 query per second (compare the middle and the right part of Figure 3.19). This means that, not only in the presence of unexpected failures, JXTACh performs better than JXTA, but also that, in the case of the first protocol, the kind of disconnections does not affect the memory load.

CPU load

In the presence of failures the average processor time usage increases for both the protocols (compare the middle and the right part of Figure 3.20): the difference between the two protocols is still small (order of 1 point of percentage).

Dropped query

Like in the case of the lookup time, if we compare the middle and the right part of Figure 3.21, we notice an increment on the percentage of queries lost (as expected the maximum query loss is in the case of 8 query per second). With respect to this metric, we also notice that in the case of 1 query per second there is a worst behavior in JXTACh with respect to JXTA: it is possible that the network experienced an interval of high instability and the protocol could not recover properly (this is witnessed also from the worse behavior in the average lookup time).

3.6.4 *Incidence of negative queries*

To check the impact of negative queries on the lookup time, we set up a different environment. We decided to have a single block of queries with $k_r = k_p = 3000$, $|R| = 160$, $|E_p| = |E_r| = 3$, and $t_q = 0.2q/s$. For each rendezvous, we set $lt_{min} = 2$ hours, $lt_{max} = 6$ hours, and $rt = 40$ minutes. We performed the tests in both dynamic cases. Once again JXTACh performs better than JXTA, as it can be seen from the following table (values are expressed in milliseconds):

	JXTA	JXTACh
gentle	16679.09	2699.92
abrupt	1972.82	344.96

It is interesting to compare the lookup time of this case with the case of all positive query. We notice from the previous results that the presence of negative queries results in a lower response time: this is likely to be due to the fact that the traffic generated by a query loss is not as costly as the response traffic.

The values shown in the table might seem surprising because the average lookup time in the "abrupt" case is lower than the one in the "gentle" case. This is likely to be due to the fact that in this test we have chosen a lower query rate which allows the protocols to react to the failures in the case of "abrupt" disconnection: this reaction does not imply any special action, so the message load of the whole system does not change. In the "gentle" case instead, both the protocol generate special messages either to warn the system they are logging off (JXTA) or to pass the keys they are responsible for to their successors (JXTACh). These messages generate higher traffic that might have an high impact on the performances of both protocols in a LAN environment. The important result here is that given the same environment, JXTACh performs better than JXTA even in presence of negative queries.

3.7 CONCLUSIONS

The results obtained from our experiments on JXTACh lead us to the conclusion that it is possible to use a pure DHT for the publication/research of advertisement in JXTA: this means that the rendezvous protocol can be implemented with a pure DHT. In fact the main result of our work is the integration of Chord as the JXTA rendezvous protocol: this integration required a deep study of the JXTA project and a hard work of reverse engineering. JXTACh source code is available at its official web site [46].

In a local area network, our tests witness an improvement on every measure we considered and in each kind of environment, static or dynamic, with "gentle" or "abrupt" disconnections. This means that not only the injection of a pure DHT can be performed in JXTA, but also that this would result in a better behavior from the point of view of the response time, of the memory and CPU usage, and of the reliability.

In the very next future, we intend to perform experiments with different parameter combinations and multiple runs, and then to proceed in an incremental way, moving to more and more complex networks. We are ready to test JXTACh on several subnetworks of class C of the same network of class B and, just after, within the global Internet environment: to this aim, we will take advantage of the fact that this project is partially funded by a European project which has more than 20 partners. Our will is to use the project testbed to perform the same test we did for the local environment. An alternative viable way to increase the test scale could be also the use of distributed environments like PlanetLab. The last step will be to test the new framework with a real application, in order to

verify the transparency with respect to the original JXTA framework. Finally, we believe it could be interesting to perform a comparison in terms of data traffic.

4 MOBILE AD HOC NETWORKS SIMULATION: MOMOSE

In the previous chapters we have described algorithms which were designed (almost all) for wired networks, in this chapter we move to the context of wireless networks, more specifically into the context of *ad hoc networks* (also known as multi-hop radio networks).

An ad hoc network is a wireless network where there is no a wired backbone to relay on. Nodes are connected to each others only through the wireless medium: two nodes can communicate only if they both are within each other transmission range. The name ad hoc is because this kind of network is usually deployed for a specific purpose.

When nodes are able to move, an ad hoc network becomes a *mobile ad hoc network* (in short, MANET). In such a network the dynamism increase because of mobility and the resulting system become more and more complex. It is then important to have instruments to model the behavior of MANETs and good metrics to measure the most important parameters influencing the system.

This chapter describes MOMOSE [47, 48], a highly flexible and easily extensible environment for the simulation of mobility models. MOMOSE not only allows a programmer to easily integrate a new mobility model into the set of models already included in its distribution, but it also allows the user to let the nodes of the MANET move in different ways by associating any mobility model to any subset of the nodes themselves. The environment allows the user to define and to subsequently use configuration windows, whose content depends on the mobility model, and its GUI includes a simulation window that allows the user to manage and control the execution of a simulation. Moreover, MOMOSE allows the user to simulate the movement of the nodes within a "realistic" environment, where obstacles (such as buildings and barriers) are present and limit the movement of the nodes. Finally, MOMOSE can be easily adapted in order to record, during the simulation time, all the data necessary for the evaluation of the performance of any communication protocol or of any MANET-based application. As far as we know, MOMOSE is the only mobility model simulation environment that has all these features and which is currently available as a free software project.

4.1 INTRODUCTION

A *mobile wireless ad hoc network* (in short, *MANET*) is a computer network in which no pre-existing communication infrastructure exists. Communication links

are wireless, and nodes are free to move and organize themselves in an arbitrary fashion. These networks are expected to have several applications because of the minimal configuration and the quick deployment they require: natural or human-induced disasters, inter-vehicular communication, law enforcement, military conflicts, and emergency medical situations are just a few examples of application areas in which MANETs are expected to play an important role.

Since the nodes of a MANET are mobile, the network topology may change rapidly and unpredictably over time. It is then important, in order to evaluate the performance of any communication protocol or of any MANET-based application, to be able to accurately simulate the mobility traces of the nodes that will eventually utilize the protocol or the application. To this aim, we need a *mobility model* that describes the movement pattern of mobile users, and how their location, velocity and acceleration change over time.

We can distinguish two basic approaches in order to obtain a mobility model. The first approach consists of constructing the mobility model on the ground of accurate information about the mobility traces of users: however, obtaining real mobility traces is usually a great challenge. For this reason, various researchers proposed different kinds of mobility models that are not trace-driven and that are called *synthetic mobility models*. A great variety of such mobility models have been proposed in the literature, which differ according to at least one of the following criteria [49]: the *geographic constraints* that a mobile node has to deal with, the *scale* the model is designed to work for, and *the individuality* which is determined by the node aggregation level of the model. Some examples of mobility models that have been proposed in the past are[1] the random walk model [50, 51, 52, 53], the random waypoint model and its many variations [54], the random direction model and its many variations [55, 56], the boundless simulation area model [57], the Gauss-Markov model [57], the city section model [57], the exponential correlated random model [58], the column model [59], the nomadic community model [59], the pursue model [59], the reference point group model [58], and, more recently, the real-world environment model [60], the virtual track group model [61], the ripple model [62], the clustered model [63], the social model [64], and the TCP-based worm spread model [65].

Following the scheme we used in this thesis, we will give an overview of the background of the argument of this chapter. We will focus on the synthetic mobility models we listed above giving a general description of some of them. Afterwards, we will describe the main topic of this chapter, the MOMOSE tool, which is a highly flexible and easily extensible environment for the simulation of mobility models. Indeed, MOMOSE not only allows a programmer to easily integrate a new mobility model into the set of models already included in its distribution, but it also allows the user to let the nodes of the MANET move

1 This list is certainly not exhaustive: our goal, however, is just to give a flavor of the huge quantity of mobility models that have been proposed in the literature and of how important the mobility simulation topic is within the MANET research area.

in different ways by associating any mobility model to any subset of the nodes themselves. Moreover, MOMOSE can be easily adapted in order to record, during the simulation time, all the data necessary for the evaluation of the performance of any communication protocol or of any MANET-based application.

The chapter is structured as follows. In section 4.2 we describe some of the most adopted mobility models. In section 4.3 we introduce our simulation tool and the technical motivations behind our primary design choices. In Section 4.4, we describe the software architecture of MOMOSE and we briefly explain how a programmer can use MOMOSE in order to develop a new mobility model and/or a new data recorder. In Section 4.5 we present the experiments that we have executed in order to evaluate the performance differences between the two simulation engines included in MOMOSE. In Section 4.6 we summarize some of the tools that, according to our opinion, are the most related to our application and we present some experimental performance comparisons. In Section 4.7 we briefly describe two case studies, while we conclude in Section 4.8 by presenting some research directions.

4.2 MOBILITY MODELS OVERVIEW

As we stated above, mobility models are an instrument to describe the mobility pattern of mobile entities. In this section we will introduce a subset of the existing mobility models, some of them are already implemented in the simulator, while the others, as we will see, can be easily added to MOMOSE.

There exist many taxonomies according to which the mobility models are classified. Here we follow the one reported in [66] where mobility models are grouped into:

- Random based mobility models
- Mobility models with temporal dependency
- Mobility models with spatial dependency
- Mobility models with geographical restrictions

In all the descriptions we assume the mobile nodes represented by geometrical points spread for simplicity in a plane.

4.2.1 *Random based mobility models*

In random based mobility models, the mobile nodes are free to move in an empty simulation area. Their movement is totally random and unconstrained. Each node chooses uniformly at random its velocity and its direction and this choice is totally uncorrelated with the choices of the other nodes.

Random walk Model

The *random walk* mobility model (also known as Brownian motion) was first mathematically described by Einstein in 1926, the basic idea of this model come from the fact that movements of many entities in nature are difficult to predict, then they appear to be completely random. In this model a node n moves at a randomly chosen velocity v such that $v \in [v_{min}, v_{max}]$, where v_{min} and v_{max} are predefined parameters, and with a randomly chosen direction α such that $\alpha \in [0, 2\pi]$. Velocity and direction are maintained by n for a predetermined constant time t or a predetermined constant distance d. When t expires or d has been covered a new velocity and a new direction are chosen. The simulation area is bounded and a node hitting the simulation border bounces with an angle which depends on the incoming direction. The node continues along the new direction determined after the bounce.

Random waypoint Model

The *random waypoint* mobility model [67] is based on the same idea of the random walk but differs from it in some aspects. A node following the random waypoint model choose at random a point P in the simulation area and starts to move towards it with a velocity v chosen uniformly at random in $[v_{min}, v_{max}]$. Once P is reached n stays there for a predefined pause time t_p. When t_p expires, the node choose another point and another velocity and starts again to move.

The random waypoint model have a problem known as *density waves* in the average number of neighbors [56, 57]. A density wave is the clustering of nodes around certain points in the simulation area. The random waypoint model has a density wave around the center of the simulation area. This means that the probability for a node to pass through a point which is close to the center of the area, during his movements, is high.

Random direction model

The *random direction* mobility model has been designed to overcome the density wave phenomenon. Each node moving according to this model choose a direction α uniformly at random in $[0, 2\pi]$ and a velocity v uniformly at random in $[v_{min}, v_{max}]$ and starts to move. Once the node reaches a border of the simulation area, it stops there for a predefined pause time t_p. When t_p expires, the node choose a new direction α' uniformly at random in $[0, \pi]$ and a new velocity v' in the same way it was done at the first step.

There exists also a modified version of the random direction model, in which the nodes are not forced to reach the border of the simulation area. A node following the modified model choose the direction like in the original one, but this time it chooses also a destination point P anywhere along the direction. Once P is reached the node wait for a time t_p and then restart the procedure.

Let us observe that in this version the direction ranges always in $[0, 2\pi]$, as there is no contact with the border. This model can be simulated with the random walk model adding to it pause times.

4.2.2 *Mobility models with temporal dependency*

All the models considered so far have something in common, they are memoryless: velocity and direction selection are not influenced by the previous choices. Mobility models with temporal dependencies release this assumption. There is a relationship between two consecutive steps of the simulation.

Boundless simulation area model

In the *boundless simulation area* mobility model [68, 57] (shortly BSA) the simulation area does not have physical bounds, a node reaching the border appears on the opposite side of the simulation area. In this way it is like the nodes were traveling over the surface of a torus (Figure 4.1).

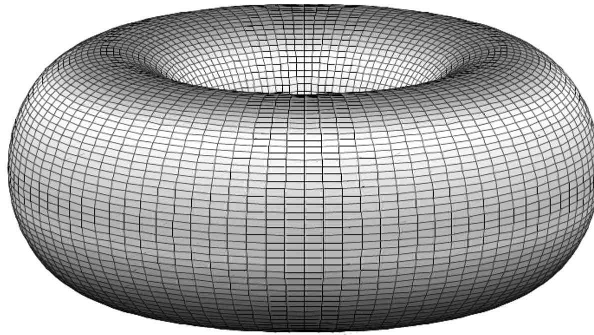


Figure 4.1: A torus

The BSA uses a vector $\bar{v} = (v, \theta)$ to describe the movement of a node, where v is the velocity and θ is the direction. The position of the node is represented as the coordinates pair (x, y). In this model the direction and the velocity are updated at each δt according to the following formulas

$$\begin{aligned}
v(t+\Delta t) &= \min\{\max[v(t)+\Delta v, 0], V_{max}\}, \\
\theta(t+\Delta t) &= \theta(t)+\Delta\theta, \\
x(t+\Delta t) &= x(t)+v(t)\cdot\cos\theta(t), \\
y(t+\Delta t) &= y(t)+v(t)\cdot\sin\theta(t).
\end{aligned}$$

where,

- V_{max} is a predefined parameter defining the maximum velocity reachable by the nodes,

- Δv is chosen uniformly at random in $[-A_{max} \cdot \Delta t, A_{max} \cdot \Delta t]$ where A_{max} is the maximum acceleration reachable by a given node,
- $\Delta\theta$ is the change of direction uniformly distributed in $[-\alpha \cdot \Delta t, \alpha \cdot \Delta t]$ where α is the maximum angular change of direction a node can perform.

Gauss-Markov model

The *Gauss-Markov* (in short GM) mobility model was not designed to simulate ad hoc networks. It was first proposed in [69] to simulate a wireless personal communication service (PCS) network. The application to a mobile ad hoc network has been proposed later in [70].

In the Gauss-Markov model a velocity v and a direction θ are initially assigned to each node. The direction and the velocity of each node at the i-th time interval are calculated on the base of their values at $(i-1)$-st time interval. The update is governed by the following equations

$$\begin{aligned} v_i &= \alpha v_{i-1} + (1-\alpha)\bar{v} + \sqrt{(1-\alpha^2) v_{x_{i-1}}} \\ \theta_i &= \alpha \theta_{i-1} + (1-\alpha)\bar{\theta} + \sqrt{(1-\alpha^2) \theta_{x_{i-1}}} \end{aligned}$$

The values v_i and θ_i are respectively the velocity and the direction of a node at the i-th time interval; $\bar{v}$ and $\bar{\theta}$ are two constants representing the mean value of v and θ as $i \to \infty$; finally, $v_{x_{i-1}}$ and $\theta_{x_{i-1}}$ are random variables from a Gaussian distribution. However, the most important parameter in these two equations is α. It is a tuning parameter such that $0 \leqslant \alpha \leqslant 1$ and it defines the degree of randomness of the model. Indeed, the model ranges from totally random values of v_i and θ_i when $\alpha = 0$, to constant values when $\alpha = 1$. In the former case the model describe a Brownian motion, in the latter a uniform linear motion is modeled.

The position (x_i, y_i) of the node at the i-th time interval is also calculated on the base of the position, velocity and direction at the $(i-1)$-st time interval. This is done through the following equations

$$\begin{aligned} x_i &= x_{i-1} + v_{i-1} \cos \theta_{i-1} \\ y_i &= y_{i-1} + v_{i-1} \sin \theta_{i-1} \end{aligned}$$

In the GM model nodes do not remain too long near to the border of the simulation area. Indeed, when a node arrives close to the border, the value of $\bar{\theta}$ is changed to force the node to get away from the border. For example, if a node arrives in the surroundings of the bottom border of the simulation area, $\bar{\theta}$ is set to $\frac{\pi}{2}$ (Figure 4.2).

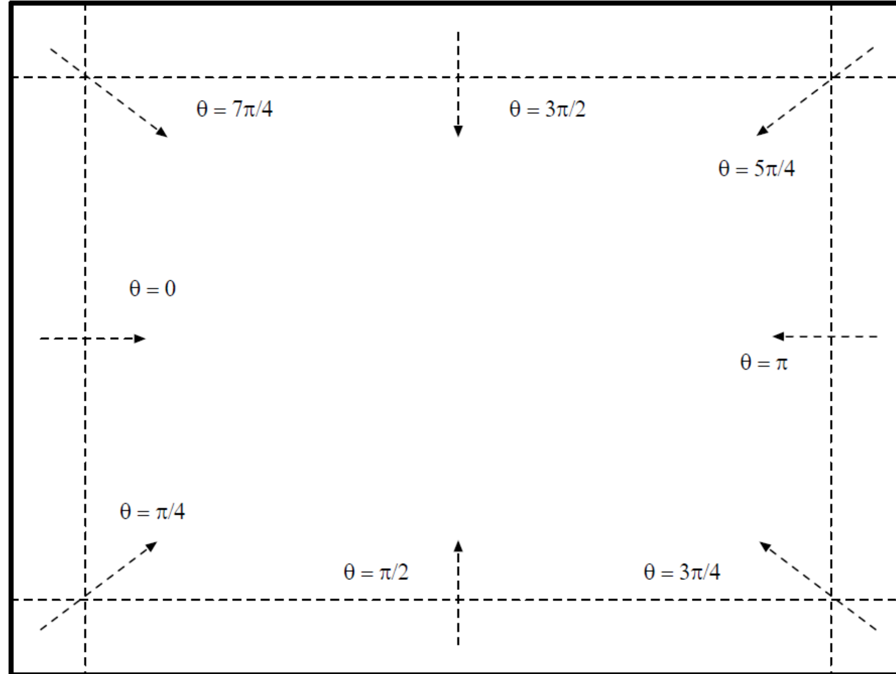


Figure 4.2: The values of the mean value of θ assumed when a node approaches to the border of the simulation area

4.2.3 *Mobility models with spatial dependency*

Mobility models with spatial dependency are models where the location, direction and velocity of a single node are correlated to the ones of the other nodes. This kind of models is necessary when we want to simulate, for example, situations in which groups of nodes collaborate to perform some task. Such models are also knows as *group mobility models* [57].

Exponential correlated random model

The *exponential correlated random* mobility model [58, 57] is one of the first examples of *group mobility model*. Indeed, it can simulate single nodes or group of nodes movements. The movements of each group are created by a motion function. The position at time t is referred as $\vec{b}(t)$. The position $\vec{b}(t+1)$ is defined through the motion function and depends from $\vec{b}(t)$ deviated by a random $\vec{r}$:

$$b(t+1) = b(t)e^{-\frac{1}{\tau}} + \left(\sigma\sqrt{1-\left(e^{-\frac{1}{\tau}}\right)^2}\right) r$$

where τ adjust the degree of change between two subsequent instants, a small τ corresponds to a large change, and r is a random Gaussian variable with variance σ.

The main problem of this model is that it is difficult to define a complete set of (τ, σ) for each group to obtain a given motion pattern.

Column model

The *column* mobility model [59, 57] simulate the movement of a group of nodes positioned over along a line (or a column). An example in real world might be the movement of a line of robots during an anti-personal mines deactivation operation. The initial phase of the model is to build a reference grid, which can be a line or a column. A set of reference point are positioned over the grid. Each node is assigned a reference point. The column of reference points starts to move and the nodes move around their reference point according to an entity mobility model (e.g. the random walk). Each reference point is moved according to an advance vector which is a predefined offset calculated via a random distance d and a random angle $\alpha \in [0, \pi]$ (the limited angle is due to the fact that movement is in a forward direction only). The offset is applied to all the reference point, in this way the column structure is maintained. When the reference points move, the nodes move together with them and subsequently start again to move according to their personal mobility model. We report a graphical representation of the nodes movements in Figure 4.3.

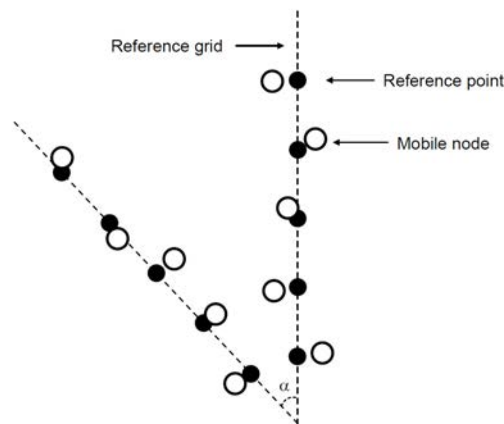


Figure 4.3: Movements of five nodes following the column model

Nomadic community model

The *nomadic community* mobility model [59, 57] simulate the movement of a group of nodes around a single reference point. The model can describe a group of tourists following the guide in a museum. In this model there is a single reference point around which all the nodes move according to an entity mobility model (e.g. the random walk). Each time the reference point moves, all the nodes follow it and then restart to move according to their personal model. The entity

mobility model parameters define how far each node can go from the reference point. The movement of a single node can be expressed through the following equation:

$$P_i(t+1) = Q(t+1) - P_i(t) + rv_i(t+1)$$

where $P_i(t)$ is the position of the i-th node at time t, $Q(t)$ is the position of the reference point at time t and $rv_i(t)$ is a random offset for each node obtained through the parameter of the entity mobility model followed by the single nodes.

This model is similar to the column model, however, in this case nodes have more freedom of movement, they do not have to keep the line order like in the column model (see Figure 4.4).

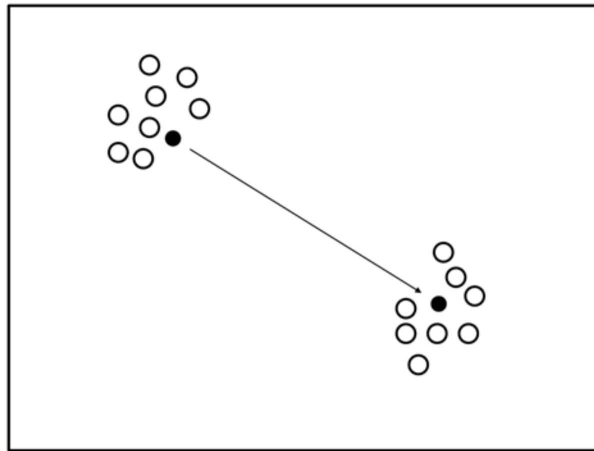


Figure 4.4: Movements of nodes following the nomadic model

Pursue model

The *pursue* mobility model [59, 57] simulate the movement of a group of nodes pursuing another node moving in the simulation area. An example from real world might be a group of police men chasing an escaping criminal. In this model the escaping node is used as a reference point, the movement of the chaser nodes is governed by a single equation:

$$P_i(t+1) = P_i(t) + a(Q(t+1) - Q(t)) + rv_i(t)$$

where $P_i(t)$ is the position of the i-th node at time t, a is the acceleration of the target node, $Q(t)$ is the position of the target node at time t and $rv_i(t)$ is a random offset for each node obtained through the parameter of an entity mobility model (e.g. the random walk). The random offset has to be limited in order to maintain the nodes chasing the target (see Figure 4.5).

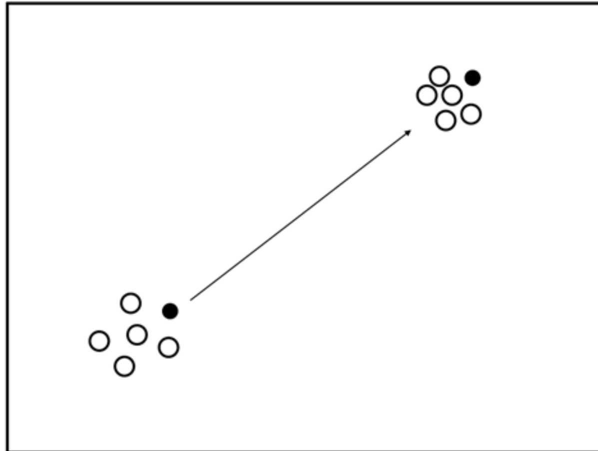


Figure 4.5: Movements of nodes following the pursue model

Reference point group model

The *reference point group* mobility model [58, 57] (in short, RPGM) can be considered the generalization of all the group mobility models we have seen so far. In fact the other models can be easily obtained modifying the RPGM parameters. The RPGM models both the movement of a group of nodes and the movements of a single node within the group. The group moves along the path covered by a logical center. The logical center change of position is defined by a group motion vector $\vec{GM}$. Each node have a reference point moving according to the group motion vector. The nodes randomly move around their reference points. The movements of each node at each time step are obtained combining the movement of the group center (hence their reference points) with a random motion vector $\vec{RM}$. The vector $\vec{RM}$ represents the random motion of a node around its reference point, its length is uniformly distributed within a certain radius and its direction is uniformly distributed in $[0, 2\pi]$.

A way to implement the RPGM might be using the random waypoint model for both the logical center and the nodes with the difference that the nodes does not have pause times, they will stop when the logical center stops. An example from real world might be an emergency situation like an avalanche rescue involving men and dogs [57]. The reference points model the human rescuer while the nodes model the dogs moving around their guides constrained by the leashes.

4.2.4 *Mobility models with geographical restrictions*

In real scenarios, it is rare that moving entities can move without any physical obstacle. For instance, people movements inside a building are constrained by the rooms shape, cars move according to the streets surrounding the buildings,

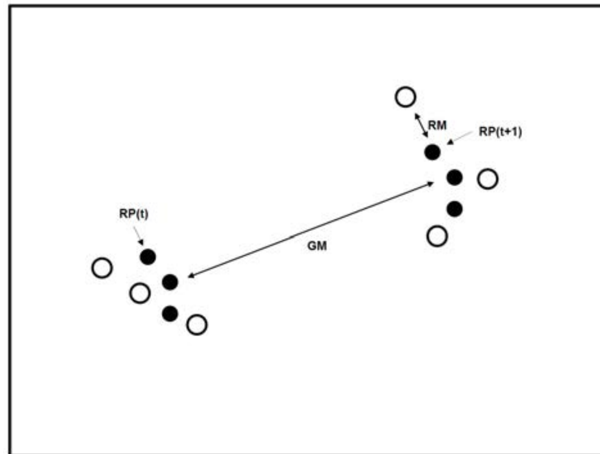


Figure 4.6: Movements of nodes following the RPGM model

students roaming in a campus are constrained by buildings and lawns. Furthermore the movements are not completely random, but they usually follow a pattern which is common to many moving entities. The mobility models with geographical restrictions are models trying to capture all these aspects.

City section model

The *city section* mobility model represents nodes moving over a grid representing the street network of a city (or part of it). Each street have its proper characteristics in term of speed limits. Each node start the simulation at some point in some street, chooses uniformly at random a destination point and moves towards it. To reach the destination, the node chooses the shortest path in terms of time, taking into consideration the speed limits. In addition each node has to respect a minimum distance from other nodes present in the same street. When the node reaches the destination it waits for a specified pause time, another destination is selected and the process restarts.

This model represent a better simulation of the real world, in fact people moving in a city are usually constrained by obstacles, other nodes, streets and speed limits.

Manhattan model

The Manhattan mobility model [71] is similar to the city section model, indeed, it models the behavior of mobile nodes embedded in a street network. The street network is represented with a map composed of vertical and horizontal streets. A node can move in both the directions in a street. When a node arrives to a crossroad, it has to choose one of the three possible direction according to a probability distribution: in [71] the probability to turn left or right is set to 0.25,

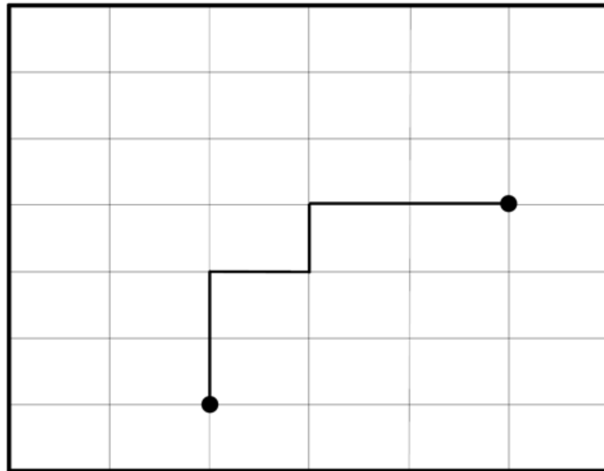


Figure 4.7: Movements of nodes following the city model

while the probability to continue in the same street is set to 0.5. The velocity of each node at time slot is dependent on its velocity at the previous time slot and it is constrained by the node preceding it on the same street.

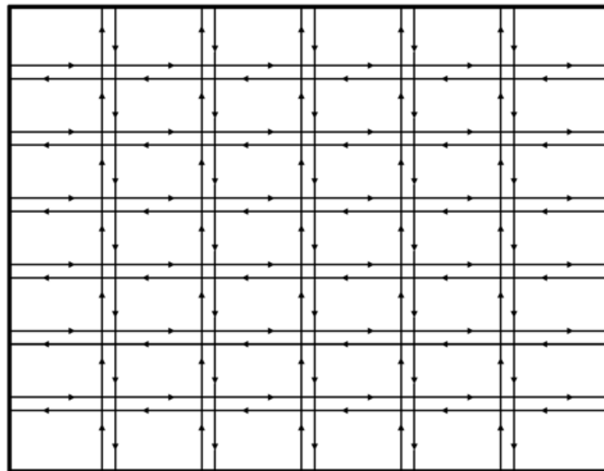


Figure 4.8: A graphical representation of the simulation area of the Manhattan model

Real-world environment model

The real world environment mobility model (also known as Obstacle mobility model) was designed to increase the realism of the simulation. In this model the nodes move in a simulation area in which there are some obstacles. The presence of obstacle implies a different transmission model as well. Indeed, the radio signal of a wireless device is subject to phenomena as diffraction,

reflection, scattering, multi-path propagation and attenuation due to the presence of physical objects.

Four component are considered in the model:

1. *Obstacle construction*: the obstacle are modeled as arbitrarily complex polygonal shapes (i.e. a list of vertices). Each object has doors which permit a mobile node to enter inside it (this is done to model the possibility for a mobile entity to enter inside a building).

2. *Pathways construction*: the pathways are constructed as a Voronoi diagram of the obstacle corners. The nodes are forced to follow the edges of the Voronoi diagram, eliminating in this way the randomness of the paths and generalizing the intuitive notion that the pathways typically run in the middle of two adjacent buildings. The corners of the Voronoi cells are the vertices of the pathways, the intersection of an obstacle boundary with a Voronoi diagram edge is considered a door of the obstacle.

3. *Node movement*: the starting point of each node is chosen uniformly at random, at each step a node choose a destination point and a velocity and starts to move. The route selection from a starting point to a destination point is computed through a shortest path algorithm (e.g. Dijkstra algorithm) on the pathways graph.

4. *Signal propagation model*: the obstacle model computes the approximate signal attenuation experienced by the radio wave.

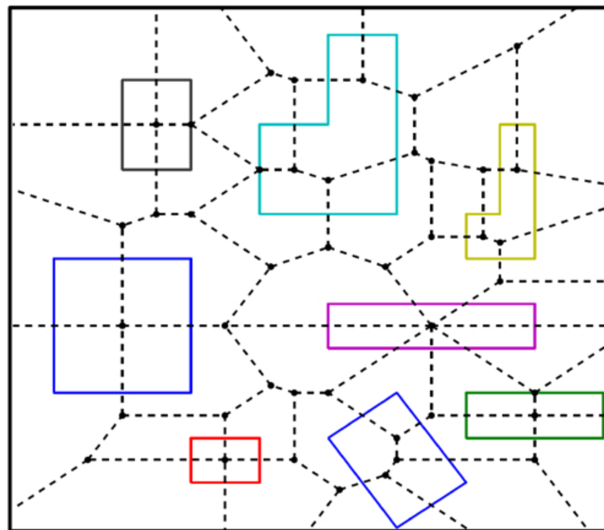


Figure 4.9: An example of the simulation area of the obstacle model

Virtual track group model

The virtual track mobility model is a group mobility model simulating the movement of groups of nodes and single nodes in the same simulation area. This model starts from the idea that in some mobile environment there might be both groups of node moving following a common path, and single nodes moving independently according to a personal model. Furthermore the model admits the possibility for a group to split in smaller groups or to merge with other groups forming a bigger group.

At the beginning of a simulation performed through the virtual track model, a certain number of *switch stations* are randomly positioned on the simulation area. The switch stations are then interconnected through a network of *virtual tracks*. A virtual track exist between two switch stations if their euclidean distance is smaller than a predefined maximum value. Each virtual track has also a width which can be either predefined either randomly chosen. Once this preliminary operation is terminated, the nodes have to be spread over the simulation area. The groups of nodes are distributed along the virtual tracks, while the single nodes are distributed randomly without taking into account the virtual tracks.

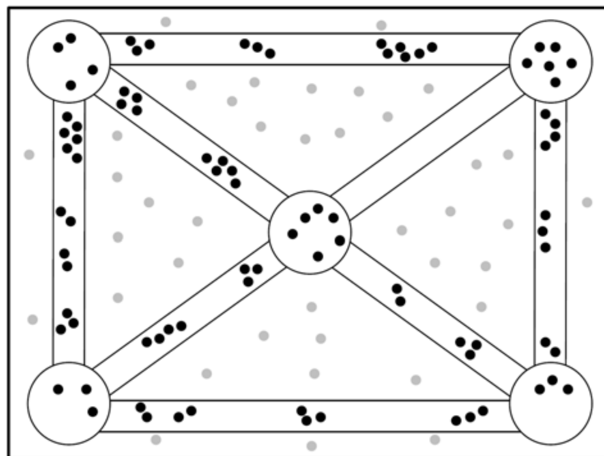


Figure 4.10: The virtual track model simulation area, the big circles are the switch stations connected through the virtual tracks, the black spots are nodes in groups, the grey spots are the single nodes.

The group mobility is constrained by the structure formed by the switch stations and the virtual tracks in the following way: each group select as destination one of the two switch stations connected by the virtual track containing it. The movement towards the switch station is modeled with the random waypoint model with two specific constraints: the first one is that at each step the destination point chosen has to be closer to the switch station than the source one, the second one is that the destination point must be inside the same virtual track. This movement is applied to all the members of the group. In addition, each

node within the group can have a small internal mobility under the constraints of the group and tracks. When the group reaches the switch station, it chooses a new destination and moves towards it.

The split and merge of groups happen at switch stations. The split is modeled using a group stability threshold. When a group arrive to a switch station, each node check whether its stability value is below the threshold, if yes it chooses a different direction with respect to the original group. The merge of the groups happen if two groups choose the same track when going out from a station.

Finally, the movements of individual nodes are modeled through the random waypoint model.

4.3 a description of momose features

While developing MOMOSE we have taken into account the following basic principles.

extensibility. Within the MOMOSE framework, a programmer can easily implement new mobility models: we believe this is an interesting feature of the framework, since research on mobility models is still very active and new models are continuously introduced in the literature (see, for example, [72]). Observe that there are basically no limits on the complexity of the mobility model the user wants to implement: for example, it is not difficult to implement models with more complex trajectories (the authors have already implemented a variation of the random waypoint model in which the trajectory followed by the nodes is a Bezier curve) or in which the nodes follow external mobility traces obtained by any trace repository. The user can also define and easily implement appropriate *data recorders*, that is, sets of data structures and methods, in order to collect, during the simulation, the data necessary for the evaluation of a mobility model or of a specific protocol. Observe that this feature does not really make MOMOSE a network simulator (such as the ones cited in Section 4.6), since many aspects of the simulation of a computer network mainly related to the lower levels of the network architecture are not taken into account by the MOMOSE framework.

adaptability. Two kinds of adaptability features have been mainly considered. On one hand, we believed it was important to allow different mobility patterns to be used within the same simulation and that it was not plausible to assume all nodes moving according to the same mobility model. For this reason, by using MOMOSE and without writing any line of code, the user can easily simulate the movement of a set of nodes by using *any* combination of the mobility models included in the MOMOSE distribution and of the newly implemented mobility models. On the other hand, we

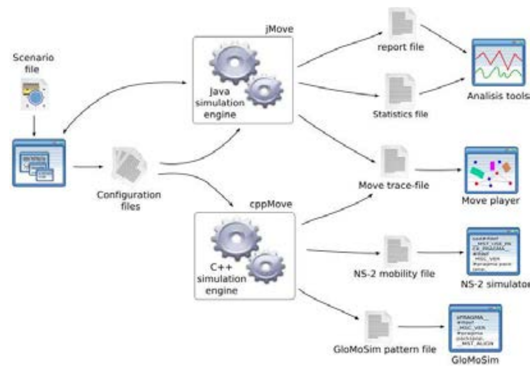


Figure 4.11: The MOMOSE flow diagram

believed that the same simulation should have been used in order to analyze different aspects and consequences of the node mobility. For this reason, during a simulation, one or more data recorders can be used, which allow the user to collect the interesting data (see Figure 4.11). For example, a data recorder could compute the distribution of the nodes during the simulation time in a given area, while another data recorder could compute the degree of each node (that is, the number of active connections for each node): both data recorders could store this information into the same output file, which could be subsequently used in order to produce statistics or reports. The current distribution of MOMOSE includes a data recorder that produces trace files compatible with the *ns-2* network simulation environment [73], which is one of the most popular network simulators within the research community. As far as we know, the mobility modules produced for this simulator include only very simple mobility models, such as the random waypoint model [74] and the random walk model [75]: MOMOSE can hence be used in order to produce more realistic mobility patterns, which can be subsequently used by *ns-2* while simulating and evaluating any network protocol.

efficiency. This is quite an obvious requisite of any simulation tool. Even in this case, two different kinds of efficiency have been identified. On one hand, while evaluating the characteristics of a mobility model, we have considered very important to visually *and* efficiently analyze the behavior of a set of nodes moving according to the model itself. The graphical user interface (in short, GUI) of MOMOSE is based on the `Java` *Swing* classes and on the *OpenGL* standard [76]: as we will state in a later section, using the OpenGL libraries makes our framework extremely performant for what concerns the visualization of a simulation. On the other hand, an efficient simulation engine turns out to be useful whenever massive simulations

have to be performed in order to collect sufficient data for successive elaborations. The MOMOSE simulation engine has been developed both in `Java` and in `C++` (see Figure 4.11): the former one is deeply integrated with the GUI and allows the user to interactively control the simulation and its visualization, while the latter is optimized from a performance point of view and is accessible only by means of command lines. The reason why we decided to include two different simulation engines is strictly connected to the main characteristics of the two programming languages. `Java` is highly portable and the Swing classes behave essentially in the same way, independently from the used processor/operating system platform: on the contrary, several different graphical libraries have been developed in `C++`, and it does not seem that any of them can be considered as the standard one. Hence, the `Java` version of the simulation engine is particularly appropriate during the development phase of a mobility model and/or of a data recorder: in this case, indeed, the GUI allows the user to easily configure the simulation parameters and to observe in real-time the node movement and the evolution of the simulation itself. On the other hand, the `C++` engine, which has to be compiled for any possible platform/operating system platform, has the advantage of being significantly faster than the `Java` engine: hence, this version of the engine is more appropriate for the execution of simulations with a long simulation time and/or with a huge number of nodes (as we will see in a later section, its efficiency outperforms or at least compares with the one of similar mobility model simulation tools).

mobility model and data recorder configurability. During each simulation, the set of the nodes of the network is partitioned into an arbitrary number of subsets, each one corresponding to the specific mobility model governing the movement of the nodes in the subset: however, since each node has an own logic unit which is independent from the other nodes, different nodes in the same subset are allowed to play different roles within the same mobility model (for example, in the `PursueModel` [58] it is necessary to allow one node to act as the *leader* of the subset). The behavior of a mobility model is typically determined by the value of some model-specific parameters: for example, in the case of the *Gauss-Markov mobility model* [77] the parameter α which determines the randomness degree of the model has to be specified, while in several other models typical parameters are the acceleration value or the angular velocity value [78]. In general, a unique set of parameters which can be used for any mobility model does not exist: for this reason, MOMOSE allows the user to define and to subsequently use *configuration windows* whose content depends on the mobility model (see the foreground window of Figure 4.12). In this way, it is very easy to tune all the parameters of a specific model, both

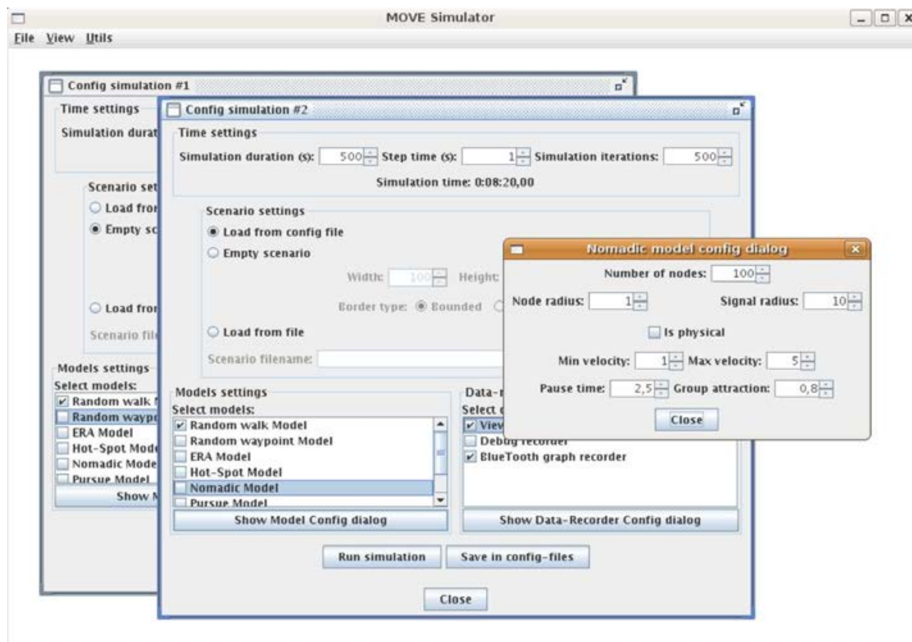


Figure 4.12: The simulation and mobility model configuration windows

the parameters common to all mobility models (such as the number of nodes moving according to the model or the maximum node transmission range) and the parameters which are meaningful for that specific model only (such as the attraction degree in the case of the *nomadic model* [77]): moreover, this parameter tuning can be saved in appropriate files, which can be successively reloaded. A similar approach can be also followed in the case of data recorders whose behavior depends on the value of specific parameters: even in this case the user can define and subsequently use specific configuration windows in order to set up parameters such as the name of the file into which the collected data will be written.

simulation configurability. By means of the MOMOSE GUI, the user can control any aspect of a simulation both before its beginning, by interacting with its configuration window, and during its execution, by interacting with the simulation window. The *configuration window* (see the background windows of Figure 4.12) allows the user to set up the simulation time and the simulation scenario (which can range from an empty area to any environment specified in an appropriate file). It also allows the user to select and configure the mobility models and the data recorders to be used during the simulation. Starting from the simulation configuration window, the user can directly start the simulation itself or can save the current configuration into an appropriate file, which can be subsequently

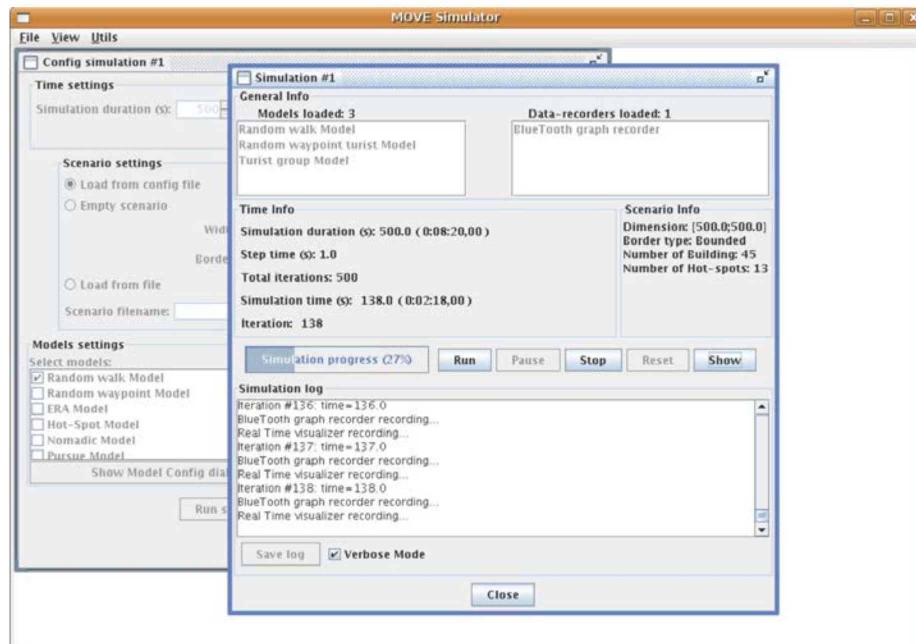


Figure 4.13: The simulation window

used by one of the two engines previously described in order to execute the simulation and collect the required data (clearly, a configuration file can be reloaded and modified at any subsequent moment).

user friendly interface. The *simulation window* (see Figure 4.13) allows the user to manage and control the execution of a simulation: in particular, at any moment the user can pause and restart the execution, can stop it, can see log messages produced by the simulation engine, and can activate a graphical window which shows, in real-time, the movement of all the nodes within the specified scenario (along with other information related to the simulation). Moreover, MOMOSE includes an OpenGL *player* which allows the user to visualize and graphically analyze the evolution of a simulation, which was previously saved into appropriate files by a default data recorder included in the MOMOSE distribution. Within the main drawing area of the player, the simulation scenario and the movement of the nodes are shown (see Figure 4.14): on the left of this area, some basic information about the scenario and the simulation time is given together with some tools for controlling the simulation itself (as with any other player, the user is also allowed to pause and restart the simulation and to fast advance it both forward and backward). During the simulation, some additional information can be visualized within the drawing area, such as the node IDs, the node transmission ranges, and the communication graph.

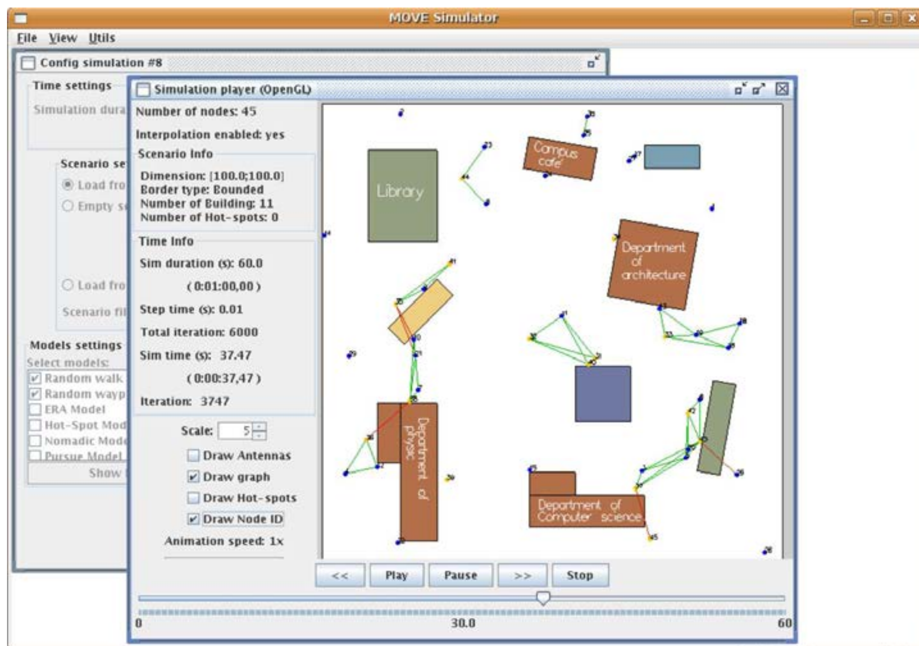


Figure 4.14: The OpenGL player

Since long simulations with a large number of nodes might produce huge trace files, MOMOSE allows the user to save these files in a compressed form (in particular, by using the `gzip` standard): these compressed files can be directly loaded and visualized by the player.

adherence to reality. MOMOSE allows the user to simulate the movement of the nodes within a "realistic" environment, where obstacles (such as buildings and barriers) are present and limit the movement of the nodes.[2] The definition of a scenario is flexible enough to allow the user to define significantly different situations, ranging from people moving within a building or a campus to robots moving within a disaster recovery environment or to vehicles moving within an urban environment. Indeed, a *scenario* is defined by means of an XML file which contains the list of obstacles that are present in the simulation area. Each obstacle is formed by one or more polygons (in particular, squares, rectangles and/or circles): for each polygon, the XML code specifies its position, its rotation angle, its color, its name and its attenuation factor (which is a number between 0 and 1). A scenario can also contain a set of *hot-spots*, that is, specific points of the simulation area which are of particular interest for the nodes: by

2 Even though this is not strictly related to the simulation of mobility models, MOMOSE also allows the user to let the obstacles attenuate the transmission signals sent by the communication units: this features can be useful for developing specific data recorders.

means of this feature, it is possible to define within the scenario a graph, which can be used by a mobility model while deciding the movement of a node (such as in the case of the *pathway model* [79]). The XML standard allows the user to easily define new scenarios: however, MOMOSE includes the possibility of generating a scenario starting from a *Scalable Vector Graphics* (in short, *SVG*) file. Hence, the user can create the scenario by using any drawing program, in order to subsequently export it in the SVG format and, hence, to translate it into the XML code required by the MOMOSE simulation engine.

portability. As we already noticed, choosing `Java` and `C++` as development languages, makes MOMOSE easily portable both in terms of user interface and in terms of simulation engines. It is also worth noting that, since the simulation configuration files, the scenario definition files and the player trace files are written by using the XML technology and they are all independent from each other, they can be immediately ported on different processor/operating system platforms, provided that an instance of MOMOSE has been installed.

4.4 THE SOFTWARE ARCHITECTURE AND THE SIMULATION EXECUTION FLOW

Apart from the GUI, the main software components of MOMOSE are the simulation engine,[3] the models, the nodes and the data recorders. Each of the latter three components is represented by means of an abstract `Java` class. The MOMOSE distribution includes several *template classes* that can be extended by the programmer in order to develop personalized mobility models and data recorders.

The simulation engine contains several components managing the following different aspects of a simulation.

- The *mobility model manager* is in charge of the nodes and of the mobility models during the simulation.
- The *physical engine manager* computes the collisions between the nodes and the obstacles which are present into the scenario and, hence, moves the nodes within the simulation area.
- The *scenario manager* is in charge of the logical representation of the simulated environment and of all the objects which are contained in the environment itself.

3 From a functional point of view, the `Java` and the `C++` simulation engines are equivalent: all the classes that represent the different simulator components have the same interface and do the same task. In this way, the programmer can switch from one engine to the other without having to modify a single line of the code.

- The *data recorder manager* allows the data recorders to store the data collected during the simulation.
- The *time manager* is in charge of the advance of the simulation clock.

A simulation execution is divided into three phases: the *simulation setup* phase, the *simulation cycle* phase, and the *simulation end* phase. During the setup phase all the data structures necessary for the simulation execution are initialized: the behavior of this phase is determined by the information read from the simulation configuration file, produced by means of the GUI or directly written by the user. The different components of the simulation engine are initialized one after the other starting from the time manager data structures and, subsequently, the scenario and all the objects that are contained within it. Successively, the mobility models and the node generation are initialized: during this step, each model can setup its own data structures and create the nodes that will move into the simulation area. The nodes have some common properties (such as the ID, the transmission range, and the velocity vector); moreover, each node can be defined as a physical object, so that it can collide with other nodes. During their generation, the nodes are also assigned an initial position: to this aim, the mobility model can analyze the scenario, if necessary (for example, in order to position a node onto an hot-spot or to avoid to position a node onto a wall). At the end of this step, all the information concerning the models and the nodes are passed to the initialization step of the mobility model manager and of the physical engine manager. The last step of the setup phase consists of the initialization of the data recorders and of their manager: similarly to the mobility model initialization, each data recorder setup its own data structures (such as the output file or counter variables).

Once the setup phase is done, the simulation cycle starts. At each cycle, the time manager updates the internal clock of the simulator and checks for the ending conditions. If the simulation is not ended up, the mobility model manager makes each node and model choose the next operation to be performed. Both models and nodes may perform any required operation: for instance, a model may generate a new target for its nodes, may change the role of some or of all its nodes and may interact with other models, while a node may switch on or off its own transmission device, may change its own role, may change its speed and direction, and may modify its transmission range (notice that, by modifying the transmission range, energy saving arguments can be also taken into account). While such operations are performed, the simulator keeps models and nodes informed about the simulation time, the scenario and the states of the other nodes in order to let them take the correct decisions. For instance, a node may need information about the scenario in order to determine whether collisions may occur while moving at a given speed and direction, or it may need information about the hot-spot list in order to choose its next target destination. After the mobility model manager step, the physical engine

manager gets the simulation control and computes the new node positions, taking into consideration the collisions between nodes and scenario objects and among nodes. In order to speed up these operations, the physical engine manager represents the simulation area by a *BSP tree*:[4] such a tree is built during the setup phase and it allows the physical engine manager to save computational time, since only the collisions between a node and its surrounding physical objects are considered. At the end of any simulation cycle, each data recorder collects the required data and records system information at current time. Data recorders may access all simulation data (such as time, scenario, and system state) and may access all the information about models and nodes involved in the simulation: for instance, one data recorder might compute the communication graph and draw its diameter, the node degrees, and the number of connected components, while another data recorder might write the logs of node positions and states at the current simulation cycle.

The last phase of a simulation is the simulation end, during which a procedure is invoked for any data recorder that allows it to execute some final tasks. For example, it is possible to close the open files, to evaluate some performance values by using the collected data, or to create reports and the similar. During this phase, the simulation engine erases all temporary data structures. Finally, the simulation ends and the output files created by the data recorders can be used for the analysis or can be exploited by other tools.

4.4.1 *Extending MOMOSE*

As previously stated, MOMOSE allows the programmer to create new mobility models and new data recorders starting from a set of template classes. In order to develop a new mobility model, the programmer has to implement three classes, which manage the creation of the model, represent the model itself, and represent a node moving according to the model, respectively. Optionally, the programmer can implement two additional classes, which represent the model configuration window and whose task is reading all the necessary information starting from a configuration file, respectively: these two classes should allow the user to easily manage the model parameters. The structure of the classes that implement a data recorder is quite similar to the one of the classes that implement a mobility model. In particular, the programmer has to define two classes, which manage the creation of the data recorder, and actually implement the data recorder itself, respectively. Optionally, the programmer can implement a configuration window class and a class which collects the data recorder set up information starting from a configuration file. More details concerning the extension of the MOMOSE framework can be found in [47].

---

4 The Binary Space Partitioning [80] technique is a recursive partitioning of the space into convex sub-spaces, which is described by means of a binary tree (called BSP tree).

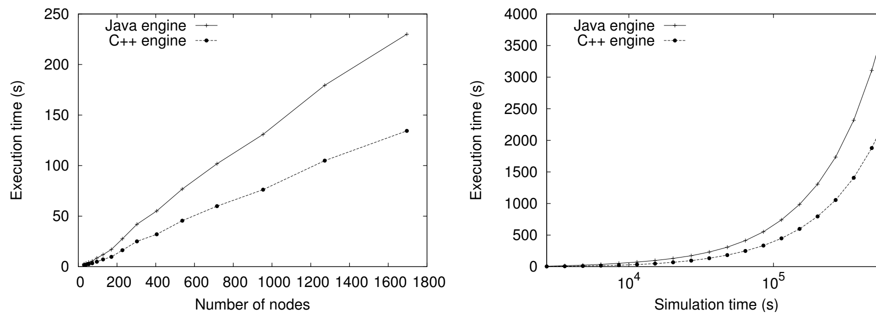


Figure 4.15: A comparison of the average execution time of the Java and the C++ engines with respect to the number of nodes (on the left) and with respect to the simulation time (on the right, in a logarithmic scale).

4.5 java and c++ performance comparison

In this section a performance comparison is presented between the Java and the C++ simulation engines. The analysis is performed by comparing the running times of the two engines when executing on the same simulation framework (that is, the same simulation time and the same number of nodes). In order to evaluate the real performances, all additional I/O components have been removed. The simulations have been executed on a Intel Pentium 4 2.4 GHz processor, with 512 MB of RAM running a Linux (kernel version 2.6.17-10) operating system.

The first comparison focuses on the number n of nodes. Ten different values of n have been considered: for each of them, twenty simulations have been executed and the average execution time has been computed. In particular, for each execution the simulation time has been set equal to 10800 seconds, while the n nodes have been partitioned into three equally sized sets moving according to the random waypoint, the random walk, and the nomadic mobility model, respectively: in the left part of Figure 4.15, the average execution time is shown. It is evident that the C++ engine is significantly more performing than the Java engine: it also seems that this better performance does not depend on the number of nodes.

However, one might think that the execution time is too short and that the initial overhead, that a Java program has usually to pay for,[5] cannot be compensated in such short execution times. For this reason, we have performed a second kind of comparison which focuses on the simulation time t. Ten different values of t have been considered: for each of them, twenty simulations have been executed and the average execution time has been computed. In particular, for each execution 900 nodes have been simulated, partitioned in a

5 For instance, the Java interpreter usually performs a code validity check, which is done only the first time a method is invoked.

way similar to the previously described one: in the right part of Figure 4.15, the average execution time is shown (in a logarithmic scale). Even in this case, the better performance of the C++ engine is quite evident. Even though the performance relative difference slightly decreases while the simulation time increases, it seems that asymptotically this difference tends to a value close to 40%.

Both the Java and the C++ engine can deal with a very large number of nodes: indeed, we have experimented up to 100000 nodes. Clearly, the simulation execution time depends on the complexity of the used mobility models and of the used data recorders. Moreover, this time also depends on the number of obstacles which are present in a scenario: however, we have experimentally verified that the ratio between the execution time with no obstacles and the execution time with obstacles does not depend on the number of nodes (for instance, in the case of the scenario represented in Figure 5.10, this ratio is approximately equal to 0.1).

4.6 related work and performance comparison

Several network simulators are available on the web and most of them include tools for the generation of node mobility patterns. Four popular examples of such simulators are *ns-2* [81], the *GloMoSim* environment [82], *GTNetS* [83], and *OMNeT++* along with its mobility framework [84, 85]. Within these frameworks the node mobility support is only one of their several features and not even the most important one: indeed, these tools are mostly devoted to the simulation of network protocols and they take into account many aspects of a protocol execution, starting from the physical layer up to the application one. Clearly, this makes these tools much "heavier" than a simulator like MOMOSE, which is mainly focused on the development and the analysis of mobility models and whose goal is allowing a user to quickly determine the characteristics of a specific mobility pattern: this is true even in terms of the size of the software needed to be installed (basically, MOMOSE is two orders of magnitude lighter than the previously mentioned tools). Moreover, the complexity of these simulators usually makes harder the task of developing and testing new mobility models, both from a programming point of view and from a final user point of view.

For all these reasons, we believe that the natural main competitors of MOMOSE are *CanuMoboSim*, a framework for user mobility modeling [86], and *Sinalgo*, one of the most recent network simulator developed in Java [87].

The *CanuMoboSim* framework includes a number of mobility models and parsers of geographic data in various formats, and it is based on the notion of *extension module*. In particular, a group of nodes can be extended by specifying the mobility model according to which all nodes in the group move during the simulation. Moreover, a simulation can be extended by specifying its spatial en-

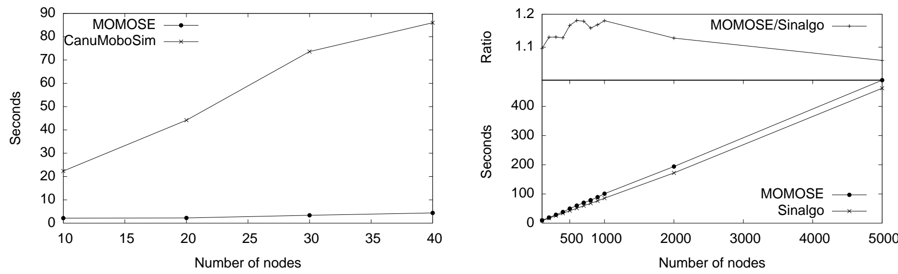


Figure 4.16: A comparison of *CanuMoboSim*, *Sinalgo*, and MOMOSE with respect to the simulation execution time (7200 simulated seconds and increasing number of nodes)

vironment (which can also be loaded from a GDF file [88] and from which points of interests can be extracted) or the graph representation of the movement area (which can be used in the case of a graph-based mobility model). Several other extension modules are available in the CanuMoboSim framework: these modules, for instance, allow the user to produce different kinds of output (similarly to the data recorders of MOMOSE) and to visualize the simulation (similarly to the player of MOMOSE). The parameters determining the simulation behavior and the characteristics of a mobility model are specified within a XML file and no configuration window is available to the final user. Moreover, the simulation window seems to be quite basic and no control buttons are available within it. Finally, as far as we can see, the node have no communication range associated and the engine does not manage the collision between the nodes and the obstacles which are present in the simulation area (even though, clearly, these features can be added to the framework by defining suitable extension modules). Most importantly, MOMOSE significantly outperforms *CanuMoboSim* in terms of execution time, as we will show at the end of this section.

A more recent simulation framework for testing and validating network algorithms is *Sinalgo*, which, in order to guarantee easy extensibility, offers a set of extension points, called *models*: among the models included in its distribution, the framework contains the *mobility model*, that describes how the nodes change their position over time. As in the case of MOMOSE, more than one mobility model can be used during each simulation. Moreover, the programmer can create new mobility models by implementing a subclass of the class `MobilityModel`, which must define the method `getNextPos`: the movement of the nodes during the simulation is shown within a simulation window, which allows the visualization of the simulation area that is either a two-dimensional or a three-dimensional rectangle. The framework also allows the programmer to refer to a simulation scenario containing obstacles by means of images representing maps: however, it is left to the mobility model to decide how these obstacles interfere with the movement of the nodes. In other words, there is nothing analogous to the physical engine manager of MOMOSE (which

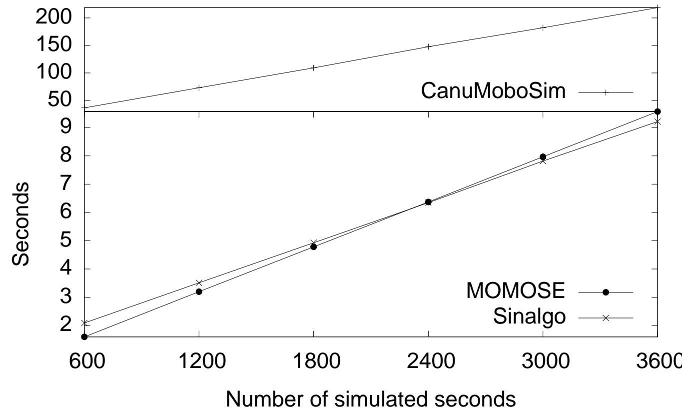


Figure 4.17: A comparison of *CanuMoboSim*, *Sinalgo*, and MOMOSE with respect to the simulation execution time (increasing simulated time and 200 nodes)

computes the collisions between the nodes and the obstacles which are present into the scenario) and, instead, this management is delegated to the specific mobility model. Moreover, the parameters determining the characteristics of a mobility model are specified within a XML file and no configuration window is available to the final user. Finally, *Sinalgo* does not allow the user to use multiple data recorders during one simulation, unless new code is written implementing all the desired recorders.

We conclude this section by presenting some experimental results concerning the performances of MOMOSE (with the C++ engine), *CanuMoboSim*, and *Sinalgo*: these results are, in our opinion, particularly valuable whenever massive simulation data have to be collected. In particular, we have realized the following three experiments.

1. For a number of nodes ranging between 100 and 5000 (in the case of MOMOSE versus *Sinalgo*) and between 10 and 40 (in the case of MOMOSE versus *CanuMoboSim*), we let all the nodes move according to the random waypoint mobility model for 7200 seconds: the positions of the nodes have been recorded into a file according to the *ns-2* syntax. The results of this first experiment are shown in Figure 4.16 (the values represent the average over 10 experiment executions):[6] it is evident that both MOMOSE and *Sinalgo* outperform *CanuMoboSim* by at least one order of magnitude (observe that we were not able to deal with a much larger number of nodes with *CanuMoboSim*). On the other hand, MOMOSE and *Sinalgo* seem to have similar performances as it can be seen in the upper plot of the right part of the figure: indeed, the ratio between the performances of MOMOSE

6 As one might expect, the variance of all the experiments turns out to be very low and it is not shown.

and of *Sinalgo* tends to be a value lower than 1.1 (by computing the linear regression for both value series, it turns out that this ratio is equal to 1.05).

2. For a simulation time ranging between 600 and 3600 seconds, we let 200 nodes move according to the random waypoint mobility model: the results of this second experiment are shown in Figure 4.17 (once again, the values represent the average over 10 experiment executions). Even in this case, both MOMOSE and *Sinalgo* clearly outperform *CanuMoboSim*: moreover, it seems that for short simulation periods (that is, shorter than 40 minutes), MOMOSE performs better than *Sinalgo*.

3. For a number of nodes ranging between 100 and 200, we let all nodes move according to the random waypoint mobility model and we used MOMOSE and *Sinalgo* to visualize the entire simulation. In this third experiment, MOMOSE clearly outperforms *Sinalgo*: indeed, during the same execution time MOMOSE is able to visualize a simulation time which is at least two orders of magnitude greater than the one visualized by *Sinalgo*. Hence, MOMOSE is able to visualize the simulation at a refresh rate much lower than real-time even in the case of a large number of nodes. We believe that this result is mainly due to the fact that MOMOSE uses the OpenGL libraries in order to implement the visualization.

4.7 TWO CASE STUDIES

In the first case study we have replicated the experiments described in [89] concerning a localization algorithm based on the estimate of the received signal power: the localization problem is one of the most important research topic within the field of sensor networks, and it is strictly related to routing protocols and energy consumption. In order to replicate these experiments, we used the random waypoint model (which was already included in MOMOSE) and we designed a new parametric data recorder, which computes, during the simulation, the real position of a sensor, the position computed by the algorithm proposed in [89], and the error between these two values. Our experimental results strongly agree with the ones presented in [89], thus validating the correctness of our simulation tool and proving the easiness of using it in order to design and realize a new set of experiments.

The second case study has concerned the development of a new mobility model. Considering that different nodes may move according to different mobility models and that the mobility behavior of a node may vary during time because of changes of its environment, we let the nodes of a network move according to mobility models that are determined by the roles played by the nodes themselves: these roles, in turn, can be determined by computing colorings of the graph induced by the communication network [90]. Observe that

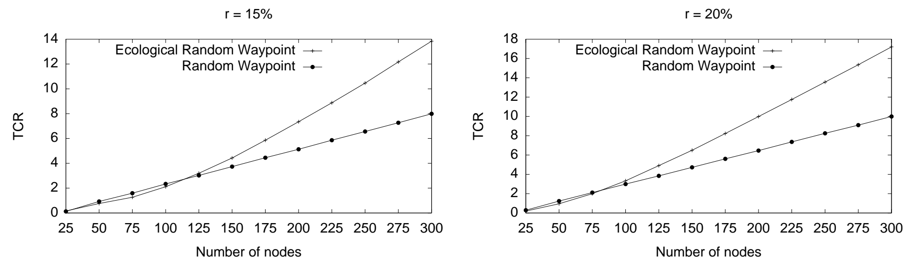


Figure 4.18: Experimental results on the topology change rate of two different mobility models

prior applications of social network analysis to the development of MANET mobility models assume that the structure of the social network is known *a priori* and that this structure does not change over time [64, 91]: in the role assignment based approach, instead, the social network structure is determined by the topology of the MANET, which in turn changes over time due to the movement of the nodes. Our experiments show that the combination of a role assignment algorithm (called ecological [92])[7] with a simple mobility model (that is, the random waypoint model) produce movement patterns with significantly higher mobility "values" with respect to some mobility metric [93] than the original mobility model itself. For example, let us consider the *Topology Change Rate* (in short, TCR) measure [94], according to which the number of link changes during a simulation is taken into account: a mobility model is considered as more mobile as such a number is larger. Moreover, let us assume that different nodes may move according to different instances of the random waypoint mobility model and the instance associated to each node may vary during time, where an instance of the mobility model consists only of the region within which the target point is chosen (in other words, we assume that all nodes have the same speed characteristics). In this case, the experiments show that, for several values r of the communication range, there exists a threshold value n_r such that the pure random waypoint model model is "more mobile" when the number of nodes is at most equal to n_r, while its combination with the ecological role assignment is "more mobile" in the other case. Figure 4.18 summarizes these results in the case in which the communication range is equal to 15% and 25% of the side of a square simulation area (in the experiments, the role assignment algorithm use four roles and a quadrant of the simulation area is associated to each role). Note that these experiments allowed us to confirm the easiness of designing and developing new mobility models within the MOMOSE framework.

7 Informally, an ecological role assignment of a graph is a coloring of the nodes of the graph such that the colors present in a particular node's neighborhood determine the color of that node.

4.8 CONCLUSIONS

In this chapter we have described MOMOSE, a new environment for the development and simulation of mobility models for mobile wireless ad-hoc networks, whose main characteristics are flexibility and extensibility. MOMOSE has already been applied in three interesting and non trivial case studies. Two were described here, the last one will be described in Chapter 5 as it involves the Bluetooth technology. In this way we have showed how MOMOSE can be used while analyzing different aspects of MANETs. These case studies seem to confirm the flexibility and extensibility of MOMOSE: for this reason, we believe that our framework can become a very useful tool for evaluating the effects of mobility on the performance of a protocol for MANETs and we hope that a wide use of the tool itself will allow us to further improve it.

The current distribution of MOMOSE [95] includes an implementation of the random walk model, of the random waypoint model, of the nomadic community model, of the pursue model, and of all the models necessary to analyze the case studies described in the thesis: a first natural further step will be to integrate the distribution with the implementation of all the mobility models referred to the introduction and described in Section 4.2. The distribution also includes a debug data-recorder, a *ns-2* data-recorder, a real-time visualizer data-recorder, a player data-recorder, and all the data recorders necessary to analyze the case studies described in the previous section: a second further step will be to develop a *GloMoSim* data-recorder and a data-recorder that can be used for simulations implemented within the *Sinalgo* framework. Moreover, we intend to improve the SVG parser (which is currently limited to a subset of possible figures) and to develop one or more parsers which would allow the user to import geographic data files (such as the GDF files). Finally, as a result of the performance comparison between MOMOSE and *Sinalgo*, it seems that there is space for some improvement of our framework in terms of simulation execution time.

5 BLUETOOTH AD HOC NETWORKS

In the last few years wireless networks have become increasingly popular. As a result, many different technologies have been developed to build ad hoc networks. Among the most important, Bluetooth, IrDA (Infrared Data Association), HomeRF, IEEE 802.11. All of them can be used to build an ad hoc network.

In this chapter we will concentrate on Bluetooth, in particular, we study the connectivity properties of a family of random graphs which closely model the Bluetooth's device discovery process, where each device tries to connect to other devices within its visibility (transmission) range in order to establish reliable communication channels yielding a connected topology.

Specifically, we will provide both analytical and experimental evidence that when the visibility range of each node (i.e., device) is limited to a vanishing function of n, the total number of nodes in the system, full connectivity can still be achieved with high probability by letting each node connect only to a "small" number of visible neighbors. Our results have extended previous studies, where connectivity properties were analyzed only for the case of a constant visibility range, and provide evidence that Bluetooth can indeed be used for establishing large ad hoc networks [96, 97].

Before showing our results, we will introduce the Bluetooth technology, focusing on the network formation problem, to let the reader be more confident with the issues we have dealt with.

5.1 BLUETOOTH OVERVIEW

A critical problem in setting up ad hoc networks is guaranteeing connectivity while minimizing power consumption and, in some cases, the number of active connections per node. Among others, *Bluetooth* [98] is a popular enabling technology for ad hoc networks, which was originally introduced in 1999 by a Special Interest Group (Bluetooth SIG [99]) formed by more than 1800 manufacturers for the deployment of Personal Area Networks (PANs), typically consisting of cellular phones, laptops, wireless peripherals and PDAs. Several arguments have been raised to foster the use of Bluetooth for the establishment of large ad hoc networks, due to its low cost, availability, suitability for small devices, and low power consumption (see, for example, [100]). However, a number of challenges arise in this context, particularly for what concerns network formation [101, 102].

5.1.1 *Bluetooth architecture*

In Figure 5.1 we report the Bluetooth architecture as it is represented in [100], we will not give a deep description of the layers of the stack as it goes beyond the purpose of this chapter. We will limit ourselves to overview the main concepts, summarizing the contents of [100] proceeding in a bottom-up fashion.

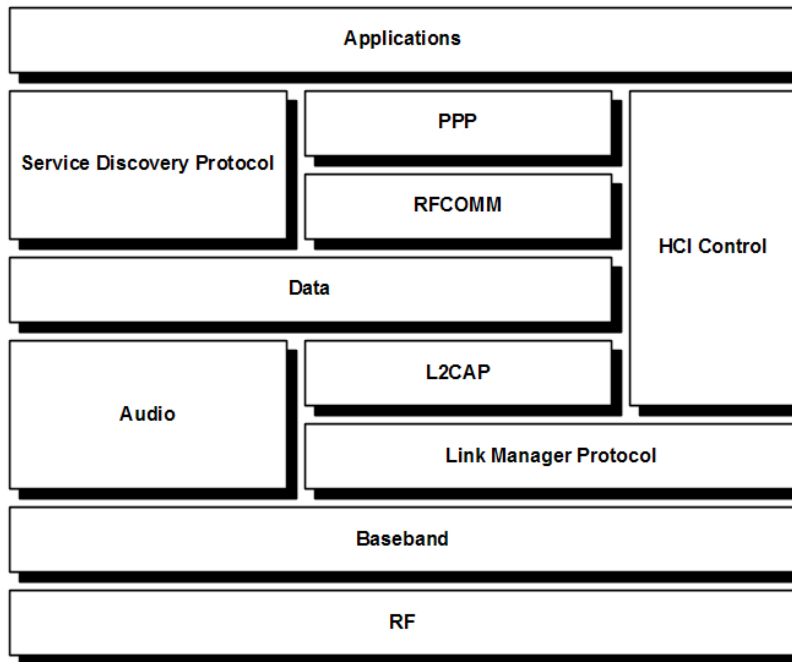


Figure 5.1: The Bluetooth architecture.

Radio Frequency (RF)

This layer takes care of the physical communication of Bluetooth nodes. Bluetooth devices operate in the unlicensed 2.4 GHz ISM (Industrial Scientific Medical) band. In order to transmit radio signals, Bluetooth implements the *Frequency-hopping spread spectrum* (FHSS, [103]) method using 79 frequencies (23 in countries with limitation in the ISM band). The hopping rate is set to 1600 hops per seconds, this means that the device remains in a frequency for 625μs. The frequencies sequence is pseudo-randomly decided.

Bluetooth devices are classified into three power classes based on their maximum output power. As a result each class has different transmission range. We report the three classes in Table 5.1 with the relative transmission range.

Power Class	Maximum permitted power	Range
Class 1	100 mW	~ 100 m
Class 2	2.5 mW	~ 10 m
Class 3	1 mW	~ 1 m

Table 5.1: Bluetooth classes

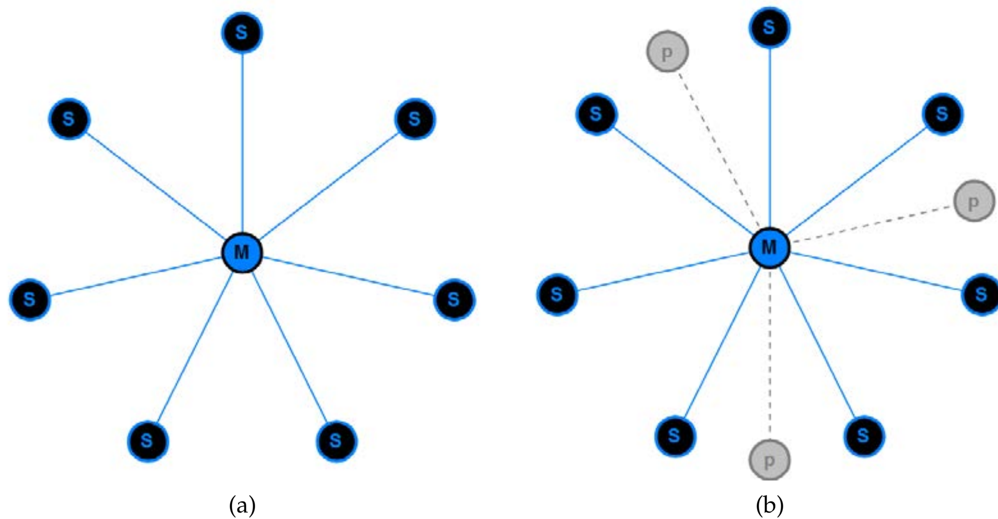


Figure 5.2: Bluetooth piconets with parked nodes (b) and without (a)

Baseband

This layer specifies how the radio layer should be used to manage the communication between Bluetooth devices. In Bluetooth nodes are firstly organized in small subnetworks, called *piconets*, in which two roles are possible, *master* and *slave*. Precisely, there is only one master for each piconet connected in star topology to multiple slaves (see Figure 5.2a). Bluetooth specification establishes that the number of slaves in a piconet must be at most seven, other slaves can be connected to the same master but they have to be in a non operative state called *park* (see Figure 5.2b).

The master operates as a router for the slaves, in fact each intra-piconet communication passes through the master. Namely, the master can communicate with all the slaves, but the slaves have to pass through the master to communicate with each others.

The baseband layer defines two phases to form a piconet

- *Inquiry*: it is the phase in which the nodes discover their neighbors

- *Paging*: it is the phase in which the links between nodes are established.

Both the phases follow the same pattern, nodes alternate two states, *inquiry* (*page*) and *inquiry scan* (*page scan*). The communication can happen only between two nodes which are in complementary states. The basic difference between inquiry and page is that during the inquiry phase each node operates in broadcast mode: a node in inquiry state broadcasts the network searching for nodes in inquiry scan state. During the page phase, instead, each device operates in a point-to-point manner: a node in page state communicates with nodes in page scan state which were previously discovered through the inquiry procedure.

In addition, the baseband layer defines the *medium access control*, thus it determines how the nodes communicates. The communication between master and slave is totally controlled by the first one. A master node divide the time in *time slots* each of 625µs (it is the same interval of time we have seen for the radio signal, this means that each slot coincides with a different frequency, unless we are transmitting a multi-slot packet). In this way, it creates a time division duplex system, dividing slots in even and odd. The master transmits in the even slots while the slaves are enabled to transmit in the odd ones. To avoid multiple conflicting slave to master transmissions, the master assign each slot to a specific slave. Each slave is able to transmit only in the time slots which have been assigned to it. The sequence of frequencies adopted is the one used by the master, the slaves in fact synchronize their frequency pattern to the one of the master at the moment in which the piconet is created.

Power saving issues are taken into account as well. For this reason there exist different states in which the power consumption is limited:

- *idle* mode: a node in such a mode performs the scan operation for just 1% of the time,

- *park* mode: a node in such a mode does not listen to the master (in fact it is not even addressed),

- *sniff* mode: a node in such a mode wakes up periodically to communicate with the master (at predefined time slots),

- *hold* mode: a node in such a mode "sleeps" until the master sends to it a wake-up signal.

The last purpose of this layer is to define the type of logical links which can be created between the nodes and to define the packets exchanged through these channels, there exist two kinds of links

- ACL, *asynchronous connectionless link*, is a master-slave channel and works in a point-to-multi-point manner. A master can communicate through an ACL link with multiple slaves.

- SCO, *synchronous connection-oriented*, is a point-to-point connection, in each piconet there can be up to three SCO links. It is a duplex channel which can be used for example for voice transmission (using both even and odd slots).

The structure of Bluetooth packets is reported in Figure 5.3. It consists of an access code (72 bits), an header (54 bits) and a payload ($0 - 2745$ bits).

The *access code* is used to identify the piconet where the packet has been generated. A message can be received by a node only if the access code matches with the one of the piconet master.

The *header* is subdivided into six fields

- *Address*: it is the address of the receiver of the packet.
- *Type*: the type of packet is determined by the type of link used to carry it (ACL or SCO), by the number of slots the packet will cover (1, 3 or 5), and by the type of error correcting code used.
- *Flow*: signals if the node sending the packet wants to stop the flow of messages (e.g. because it has the buffer full).
- *Acknowledgment*: it is used to acknowledge the receive of a packet
- *Sequence*: it is a sequence number for the packet, the protocol uses a stop-and-wait policy, thus a single bit is enough.
- *Checksum*: it is the error correcting code.

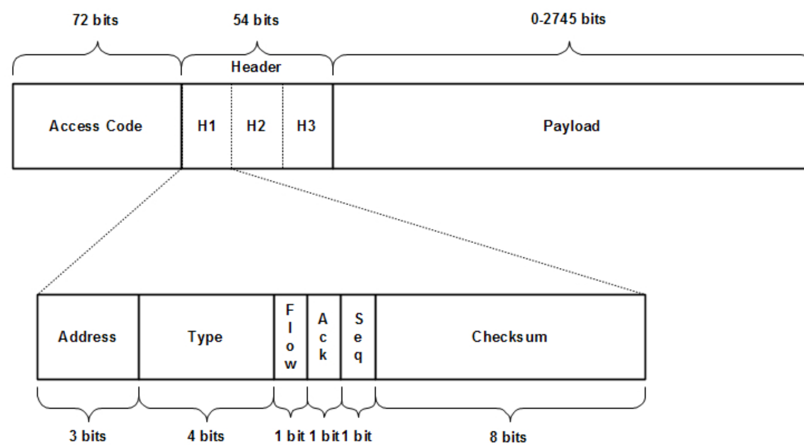


Figure 5.3: Bluetooth packet structure.

The header is encoded with a *forward error correcting* code with rate 1/3 to obtain high transmission reliability. Essentially, each bit of the header is repeated

three times, hence the header field is three times the dimension of 18 bits we described above. Referring to Figure 5.3, H1, H2 and H3 contain exactly the same data.

Let us observe that in case of multi-slot packet transmission, the frequency remain the same as long as we are transmitting the package. If the packet is k-slot, with $k = 1$, 3 or 5 and the frequency used for the transmission is f_i, with $i = 0, \dots, 78$, when the transmission terminates, the frequency adopted for the next slot is f_{i+k}. The purpose of that is to keep the transmission frequency synchronized with respect to the other slaves.

Finally, the *payload* contains the data transmitted with the packet. Two types of payload are defined, according to the type of link the packet is sent through.

Link Manager Protocol and Host Controller Interface

This layer controls the baseband layer. The links are created, secured and controlled in this layer. The paging phase is controlled here, the role assigned to the node and the switch of it from master to slave and vice versa is a task of this layer as well. Furthermore, the link manager supervises and handles the exchanging of multi-slots packets.

The *host controller interface* (HCI) is present in the devices which are not integrated with the hosts where the applications using the Bluetooth transport protocol run. For example, devices which can be attached via an USB port. Thus, the host controller responsibility is to interpret the information received from the host and direct it to the right component of the Bluetooth stack.

L2CAP layer

The logical link control and adaptation protocol (L2CAP) is the frontier between the lower levels and the upper levels of the Bluetooth architecture. The first group is implemented via hardware, while the second group is implemented through the software. L2CAP takes the responsibility of the segmentation of packets received from the upper layers and of the reassembly of the packets coming from the lower layers. In the latter case it also dispatches the packet to the right protocol.

Service Discovery Protocol

The *Service discovery protocol* is used to determine which Bluetooth services are available on a particular device. A device can act both as a service client, using services provided by other devices, and as a service server providing services to other devices. Each node can have only one server process active, while it can maintain multiple client remote connections.

Applications/Profiles

The *application* layer is also known as *profiles* layer. Indeed, the applications are defined by a set of profiles which are established by the Bluetooth specifications. In order to use Bluetooth wireless technology, indeed, a device must be able to interpret these profiles. Bluetooth profiles are general behaviors through which Bluetooth enabled devices communicate with each other.

Bluetooth technology defines a wide range of profiles describing many different types of use cases. By following a guidance provided in Bluetooth specifications, developers can create applications to work with other devices conforming to the Bluetooth specification as well.

Each profile specification contains information on the following topics at least:

- Dependencies on other profiles.
- Suggested user interface formats.
- Specific parts of the Bluetooth protocol stack used by the profile. To perform its task, each profile uses particular options and parameters at each layer of the stack. This may include an outline of the required service record, if appropriate.

A complete list of Bluetooth profiles can be found in [99].

5.1.2 *From piconets to scatternets, the Bluetooth topology*

We have seen that Bluetooth is organized hierarchically. Nodes are grouped into *piconets*, with each piconet containing one master and multiple slaves The number of active slaves can be at most seven (Figure 5.2).

To form larger networks, piconets are interconnected through special nodes named *bridge* in order to form a *scatternet*. Bluetooth specification does not give any common guideline on how a scatternet may be formed, tough it puts some constraint on bridge role. A node acting as a bridge can not be a master in more than one piconet. (Figure 5.4).

Bluetooth scatternet formation (BSF) can be decomposed into three main steps

- *device discovery*,
- *piconet formation*,
- *piconet interconnection*.

Each of these steps poses interesting algorithmic challenges for which several solutions have been proposed [98]. In particular, during the first step each device attempts at discovering other devices contained within its visibility range and at establishing reliable communication channels with them, in order to form a

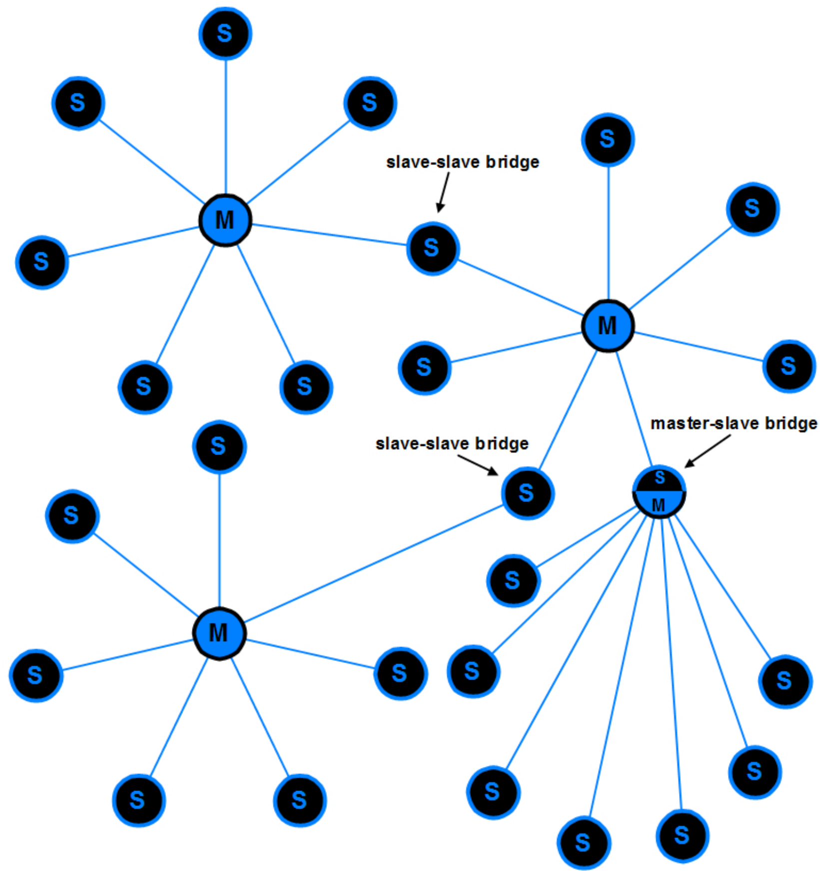


Figure 5.4: A Bluetooth scatternet.

connected topology, called the *Bluetooth topology*, which underlies the subsequent piconet formation and piconet interconnection steps.

As we have seen, the Bluetooth stack does not have any special layer where the scatternet formation policy is defined. The only constraints given by the Bluetooth specification are that piconets formed by one master and up to seven slaves should be connected only through master-slave or slave-slave bridges, and the resulting scatternet should be connected. Then this problem is left totally open to the researchers and to the developers. As a result, there exist a lot of BSF protocols. We will give a brief overview of the protocols which, in our opinion, are the most representative.

BTCP: Bluetooth Topology Construction Protocol

BTCP [104] is the first trial to define a Bluetooth scatternet formation protocol, it works under some specific requirements:

- All the devices participating to the network formation fall into the transmission range of each other.
- The number of devices participating to the network formation is at most 36.

Moreover, the authors add four properties that the network must satisfy after the protocol termination:

- A bridge may be used to connect only two piconets
- Given the number of nodes N, the resulting scatternet should be composed by the minimum number of piconets
- The resulting scatternet should be fully connected, each master should be connected to each other master through a bridge
- Two piconets can share only one bridge

The protocol proceeds through three phases (we report a graphical representation in Figure 5.5):

- I - **Coordinator election**. The first phase is dedicated to the election of a leader which will manage the creation of piconets and then of the final scatternet. A distributed asynchronous leader election algorithm is used. Each node maintains a counter of votes which is initialized to 1 at the beginning of the protocol execution. During the discovery phase, each time two nodes discover each others (during the inquiry phase), they compare their votes counters. The node having the highest value prevail and survive to the other which gives to it all its votes and information about nodes

which previously lost with it. If the votes counters have the same value, the node with the higher Bluetooth identifier wins. The loser enters in the page scan mode (in this way it eliminates itself from the leader election phase, not being able to hear the inquiry messages). The node surviving at the end of this process is the leader and the next phase can start (Figure 5.5b).

- II - **Role determination**. Now the network has a leader which knows all the information about the nodes participating to the protocol. If the number of nodes N is less than eight, the scatternet will be composed by a single piconet with the leader as the unique master. If $N > 8$ then it is needed to select among the other nodes the right masters to minimize the number of piconets composing the scatternet (keeping the four properties stated above). The protocol's authors prove in [105] that this number P is known for networks in which $1 < N \leqslant 36$. That is the reason of the limitation to 36 nodes. Once P has been computed, the leader chooses itself and other $P-1$ nodes to be the masters (Figure 5.5c) and $P(P-1)/2$ nodes to be the bridges of the scatternet, the rest of the nodes are assigned the role of slave. As the last action of this phase the leader send to each master the list of the bridges and slaves it assigned to them (Figure 5.5d).

- III - **Connection establishment**. The last phase is the effective formation of the scatternet with the activation of the links of each master according to the lists received from the leader. Once the formation of a piconet is completed (Figure 5.5e) the local master sends a notification to the leader, when all the notifications arrive to the leader the protocol terminates (Figure 5.5f).

It is clear that the main phase of the protocol is the first one. It is important to notice that in order to finish the election phase, the leader has to know in some way that he won the election. In BTCP this is achieved with the use of a timeout. Each node is assigned a timeout at the beginning of the protocol, the timeout is reset each time a node wins a comparison. When the timeout expires the node considers itself the leader. Finding the right timeout value is the most challenging issue. A solution to this problem is given in [105].

This protocol has two main problems, which coincide with the constraints we reported at the beginning of this description. Indeed, it is quite limiting to restrict the applicability of a protocol to a configuration in which all the nodes can see each other and, in the same way, it is limiting to restrict the number of nodes to 36. A reason for such restrictions could be that this is the first attempt to give a solution to the BSF problem. The fact that the Bluetooth technology was intended just as a cable replacement in a personal area network could be a reason as well.

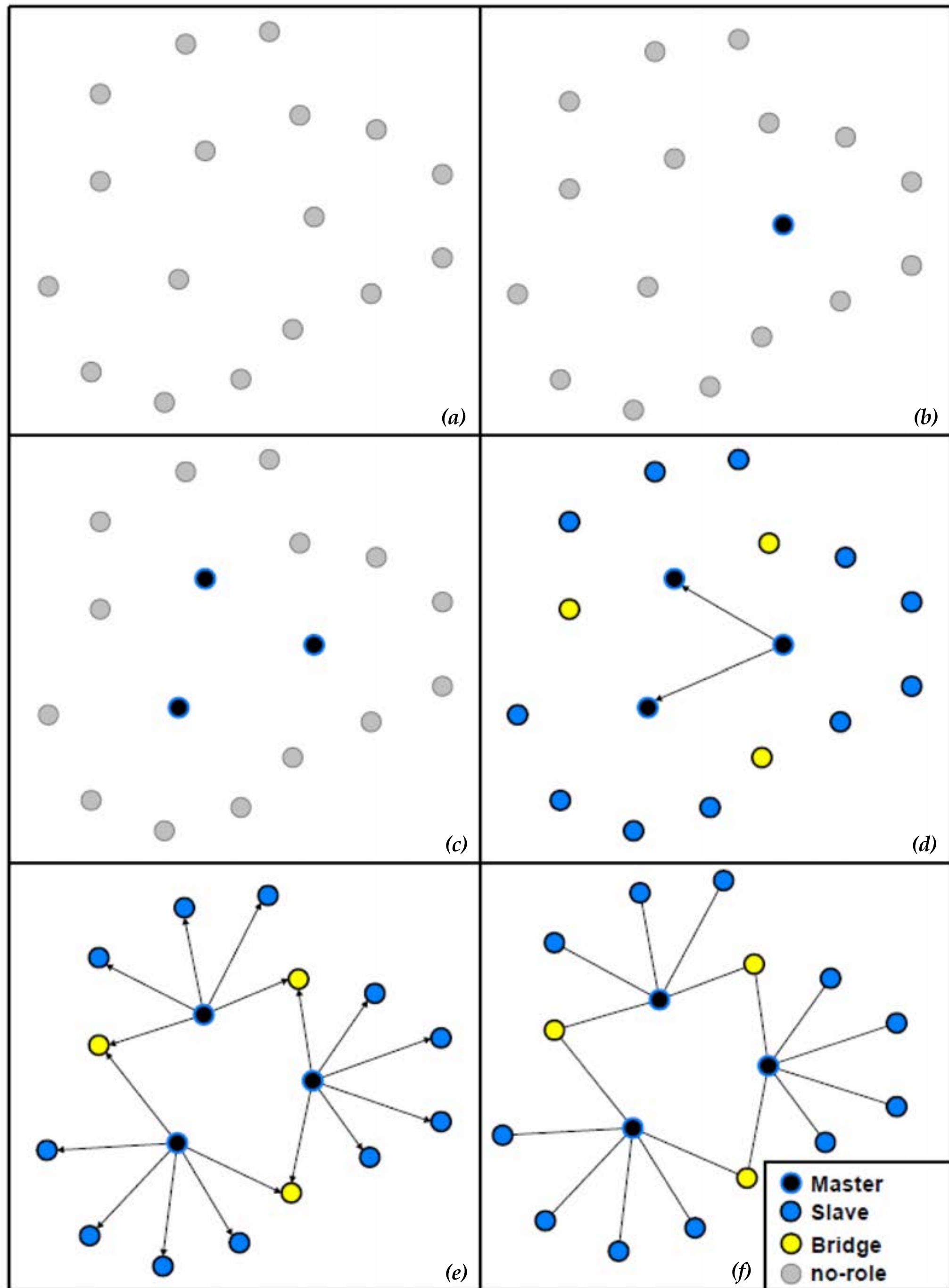


Figure 5.5: The BTCP protocol execution sequence.

Bluetrees

Bluetrees [106] is a BSF protocol which forms a scatternet having a tree topology. There exists two version of the algorithm, the first one is based on a single tree, while the second one is based on a forest successively connected to form a connected topology. Both the versions of the protocol start from the graph formed by the visibility range of the nodes, we call it *visibility graph*. For both the versions, it is required that the visibility graph is connected. In Figure 5.6 and Figure 5.7 we report the steps of both the versions of the protocol. In both the figures the dashed lines represent the links of the visibility graph obtained after the device discovery phase (Figure 5.6b and Figure 5.7b), while a continuous line represents a link of a piconet or a link of the final scatternet.

The single tree approach is based on the computation of a spanning tree over the visibility graph, the result is called *bluetree*. A node is selected as the root of the spanning tree and it is called *blueroot* (Figure 5.6c, the black node). The bluetree is built in the following way.

1. The role of master is assigned to the blueroot and the role of slave is assigned to all its children. This is the first piconet created by the protocol (Figure 5.6d).

2. Recursively, if the current node is not a leaf, it is assigned the additional role of master (it becomes a master-slave bridge) and all its children become its slaves in a new piconet. If the current node is a leaf, it remains a slave and the recursive call is closed (Figure 5.6e). The procedure ends when all the nodes have received their role.

We observe that at the end of the protocol (Figure 5.6f) the nodes will have a role among master (only the blueroot), master-slave (the internal nodes) and slave (the leaves).

The protocol is based on the assumption that each node knows whether it is the blueroot or not and each node knows all the identifiers of its neighbors and if they already are part of a piconet. The tree formation phase is performed through the page mode of the nodes, each step of the recursive procedure described above correspond to a page phase of the current node considered. If a node is neighbor of two masters (with respect to the visibility graph) it joins the piconet of the master which paged it first.

Following this protocol, it is possible that a master could have more than seven slaves, the authors solve this problem through the following geometrical observation: in an open, interference-free and obstacle-free environment, if a node n has more than five neighbors, then there are at least two nodes among these neighbors that are neighbors themselves. Let us observe that this is true if all the nodes have the same visibility range, which is the case of a Bluetooth network built with devices of the same power class. Hence, the protocol is equipped with the following procedure: when a master have more than five

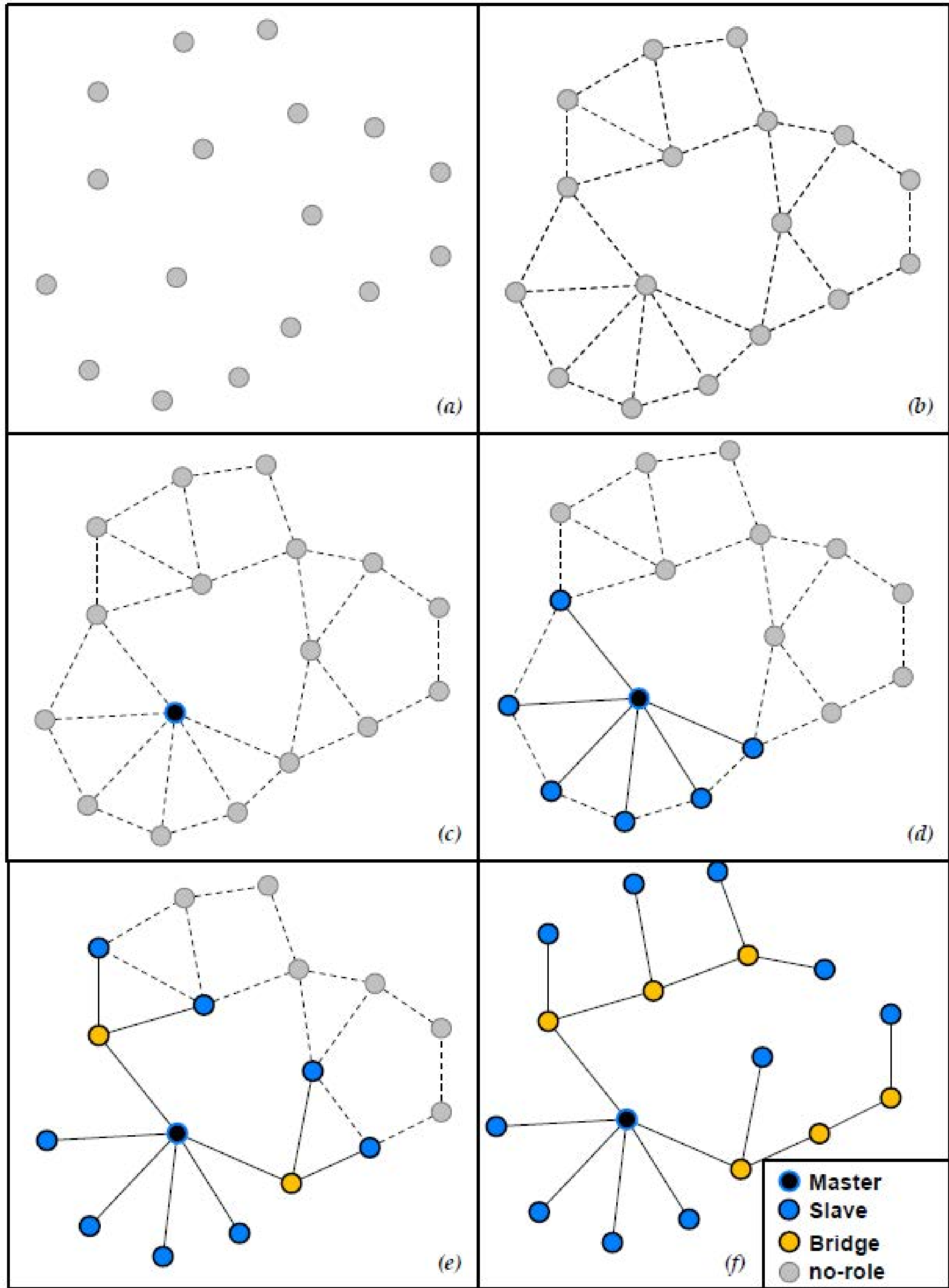


Figure 5.6: The Bluetrees protocol execution sequence.

slaves, it contacts its slaves in order to know which their neighborhoods. In this way, the master can select two slaves which are neighbors each other. The node with fewest neighbors is chosen and it becomes the master of a new piconet with the other node as a slave, which, in turn, disconnect from its original master.

The forest approach is a two phase protocol. In the first phase some *init* nodes are chosen (those having the greatest identifier among their neighbors, see Figure 5.7c, the black nodes) and they start the formation of bluetrees following the classical protocol with two essential modification:

- each time a piconet is formed the slaves node will be informed about which node is the blueroot,
- when two nodes from two different bluetrees interact, they have to exchange information about their respective roots.

At the end of this phase there will be a forest of bluetrees (Figure 5.7d).

In the second phase the disjoint bluetrees have to be connected to form the final scatternet. This is done considering each bluetree as a single virtual node and applying to the virtual graph obtained the single tree approach. This have to be done keeping the structure and the roles constraints (Figure 5.7e). The final scatternet is still a tree (Figure 5.7f).

This protocol has a problem which is strictly correlated to its hierarchical approach. The choice of a tree topology exposes the final scatternet to problems of connectivity. Indeed, if just one internal node fails, the scatternet becomes disconnected. Furthermore, the routing in a structure like this is expensive, especially in a large system. Finally the internal nodes become easily bottlenecks.

BlueStars

BlueStars [107] has been designed to solve the problems of the previously described protocols. The authors claim that their algorithm works for general multihop Bluetooth networks. The resulting topology is a mesh with multiple paths between any pair of nodes. The selection of masters node follow a best-fit policy thanks to the fact that each node is associated with a weight (a real value) proportional to the grade of suitability of the node to be a master. The generated scatternet is connected if the visibility graph is connected. Finally, the nodes does not need any further hardware (e.g. GPS units).

Like almost all the already described protocols, BlueStars is organized into three phases which can be matched with the three steps of the scatternet formation we listed at the beginning of this section. At first, the nodes discover their neighborhoods. At second, the piconets are formed (they are called *BlueStars* because of the topology of the piconets). Finally, the BlueStars interconnect with each others forming the final *BlueConstellation* which corresponds to the scatternet.

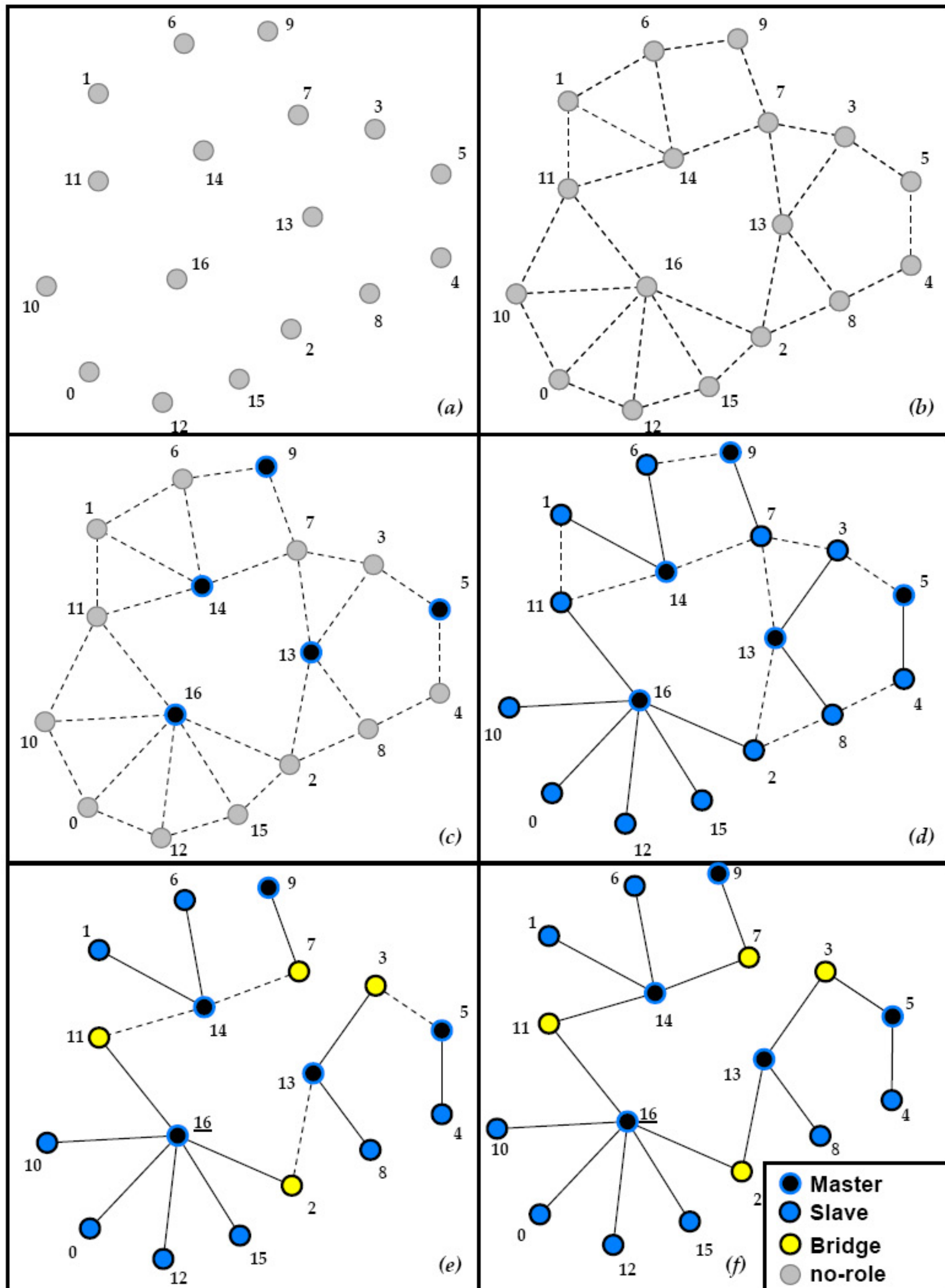


Figure 5.7: The distributed Bluetrees protocol execution sequence.

In Figure 5.8 we report a graphical representation of the steps performed during the scatternet formation. In this figure as well, the dashed lines represent the connections after the device discovery phase, while the continuous lines represent the links in a piconet or the links in the final scatternet.

- I - **Topology discovery**: in this phase nodes start to look for their neighbors through the alternation of inquiry and inquiry scan modes. The aim of this phase is to let the nodes have a complete picture of their neighborhood. To do that, each time two nodes interact, they create a temporary piconet only to exchange the information about themselves (identifier and weight). Once the information has been exchanged, the piconet is disrupted (Figure 5.8a).

- II - **BlueStars formation**: in this phase the formation of the piconets is performed. First of all, some of the nodes are elected as init nodes (Figure 5.8b). This is done through a mechanism which is similar to the one we saw in the distributed version of Bluetree, the nodes having a weight which is the highest with respect to the weights of their neighbors become init nodes. There is the possibility that two nodes have the same weight, in this case the node with the higher identifier is considered the greatest (in Figure 5.8 we reported only the weight which are all different just for simplicity). The init nodes will be all masters. The init nodes enter in page mode and start to page their neighbors (Figure 5.8c). The rest of the nodes enters in page scan mode.

  In general, a node x which is not an init node follows the next scheme. It waits the decision of its *bigger neighbors*, the first bigger neighbor paging it will become its master, if all the bigger neighbors decide to be slaves in other piconets then x decide to become a master itself. When a node x decides its role, it starts to page its *smaller neighbors* to let them take their own decision. At each exchange of information, all the nodes inform each others about their neighborhood, this information will be used during the third phase of the protocol. This process is also known as *pecking protocol*.

  At the end of this phase each node have chosen its own role and the network is composed of many disjoint BlueStars, ready to be interconnected in the following phase (Figure 5.8d).

- III - **BlueConstellation formation**: the last phase is based on some preliminary definition. Two masters having a two-hop or three-hop path connecting them (through one or two slaves) are said *neighboring masters* (in short, *mNeighbors*). A master is called an *init master* (in short, *iMaster*) if it has the highest weight among its mNeighbors.

  The nodes which are along the paths connecting two mNeighbors will be used as gateways. If there are two paths or more connecting two masters, the choice among them depends on their type. In case of two-hop paths,

the one including the node having the biggest weight is chosen (e.g. in Figure 5.8e, the path including node 6 is preferred to the one containing node 5). In case of three-hop paths the sum of weights of the nodes included in the paths is considered and it is chosen the one having the greatest sum (e.g. in Figure 5.8e, the path including nodes 10 and 8 is preferred to the one containing nodes 2 and 8).

All the information used to build such a preliminary structure have been exchanged during the second phase (Figure 5.8e, where the iMasters are indicated through an i following the weight and the paths between the mNeighbors are represented through a red line).

We can now describe the BlueConstellation formation. The iMasters and the non-iMasters will perform different operations. Each iMaster contacts its own gateways and instructs them to go to page mode to establish a link to each of its mNeighbors. In case of a two-hop path, the gateway pages directly the mNeighbor, it becomes its temporary master and switch the roles just after, becoming a slave-slave bridge (in Figure 5.8f this happens to node 16 and 18). In case of a three-hop path, the gateway connected to the iMaster pages the gateway connected to the mNeighbor in order to form a new piconet which will be the link between the two original masters. The master of the new piconet will become a master-slave bridge while the slave will become a slave-slave bridge (In Figure 5.8f this happens to node 6 and 3).

Each non-iMaster will instruct its gateways connected to bigger mNeighbors to go to page scan, then, if there are some gateway slaves of bigger mNeighbors to whom it has to interconnect, it goes to page mode. Once these potential links are established, the node starts to behave like an iMaster toward the gateways linking it with smaller mNeighbors.

At the end of these three phases the resulting scatternet is connected (Figure 5.8f). In [107] the authors formally prove it through the correctness of the three phases.

LSBS: Li-Stojmenovic BlueStars

LSBS [108] is a BSF protocol having as principal aim the construction of a connected scatternet in which the masters have up to seven slaves, without any parked node. The protocol combines the Yao construction proposed in [109] with the BlueStars protocol. Each node is associated with a weight like in the BlueStars protocol. Furthermore, the protocol needs devices equipped with specific hardware in order to sense their geographical position (like GPS).

The protocol starts with the same device discovery protocol we have seen in BlueStars. In this phase it is possible that some nodes in the visibility range might not be discovered, while the protocol needs to operate in a unit disk

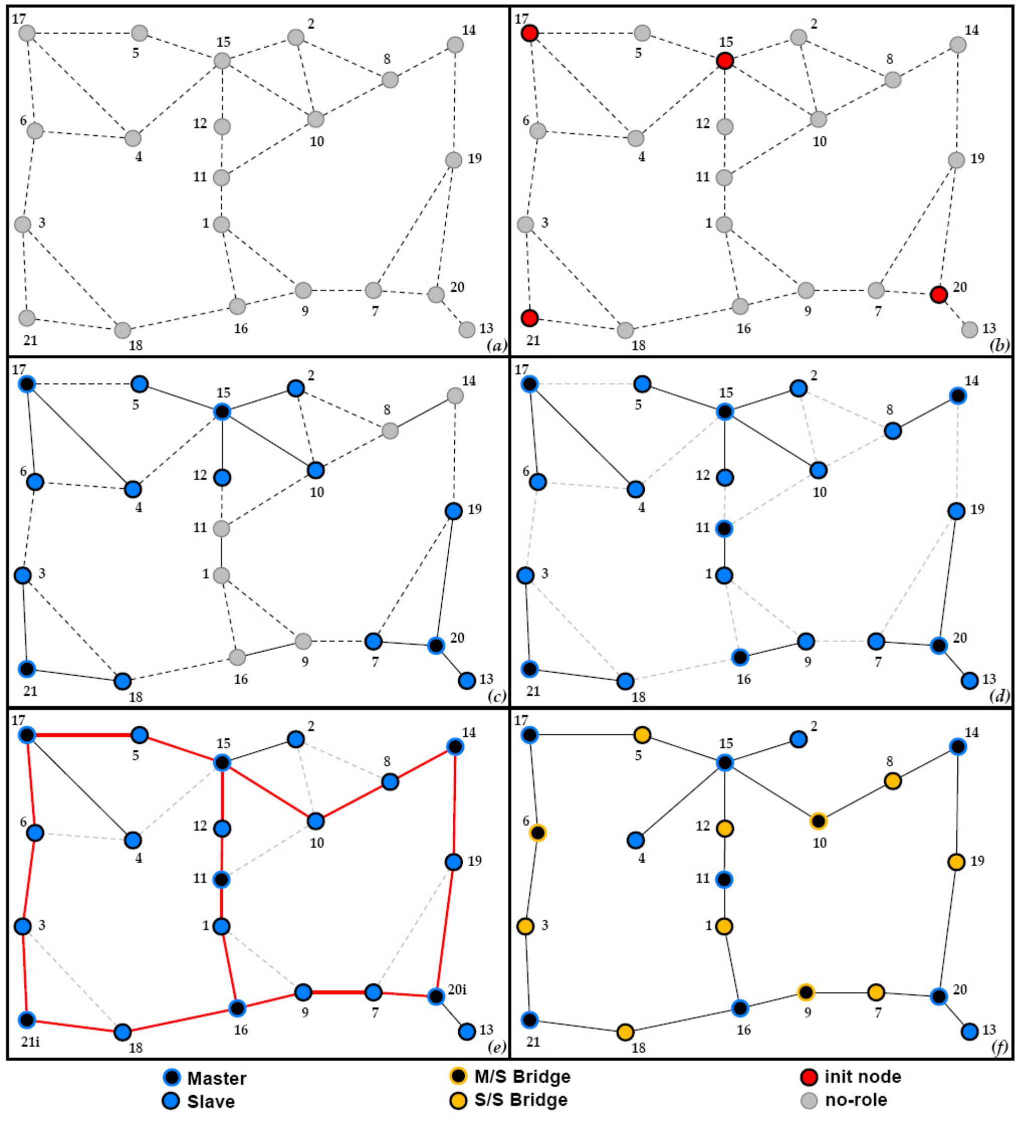


Figure 5.8: The BlueStars protocol execution sequence.

graph. Because of that, the LSBS adds a *replenish phase* aimed at adding the undiscovered nodes. At the end of the discovery phase, the nodes exchange each other the list of their neighbors through a process which follows the scheme of the pecking protocol. Thanks to the location data, each node v can build a set of nodes A_v which are within v's transmission range but were not discovered during the inquiry phase. Upon A_v construction completion, v starts to page A_v's elements in a round robin fashion. Each node $u \in A_v$ contacted by v exchanges with it the list of neighbors, in this way A_v and A_u might acquire new nodes. At the end of this process the resulting topology should coincide with the unit disk graph.

The Yao construction is applied to the resulting topology to reduce the degree of each node to a value $k \leqslant 7$. The process is the following: each node v divides its visibility disk into k equal sectors, in each sector v selects the closest node u according to the euclidean distance. The link with u is then a candidate to be a link in the final structure. Indeed, it is maintained only if also u selects v as its closer neighbor in one of its sectors. The rest of the links in the sector are then removed (see Figure 5.9). To perform this construction, each node must communicate with its neighborhood the information about the nodes it selected. This is done again through the pecking protocol.

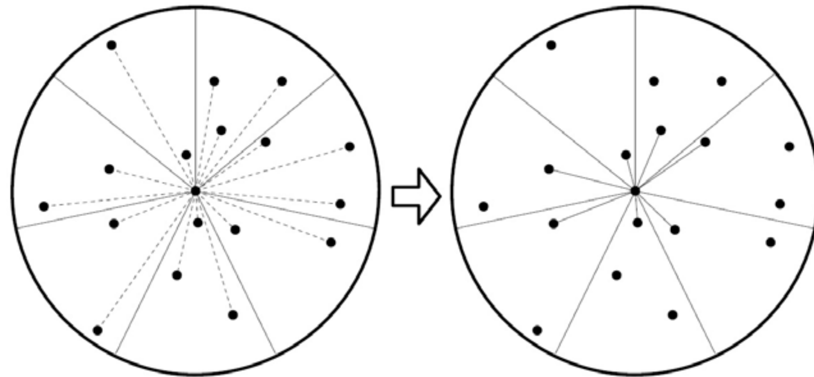


Figure 5.9: The Yao construction.

Provided that the Yao construction operates on a connected topology, the connectivity of the resulting topology is guaranteed.

Once the connected topology is built, the protocol continues as the BlueStars scatternet formation protocol.

The problem in this protocol is the replenish phase, which is time consuming and leads to considerable overhead due to the exchange of neighbors lists between neighbors. Furthermore, LSBS needs extra-hardware to operate, all the devices has to be equipped with GPS-like components.

Blue Pleiades

Blue Pleiades [110] is a protocol which combines the BlueStars approach with LSBS approach aiming at taking the best features from them and eliminating their drawbacks.

As it was in LSBS, Blue Pleiades aims at building a final scatternet of limited degree. The authors propose a new device discovery protocol modeling the system through a geometric random graph defined in the following way:

Definition 5.1 *Fixed* $r > 0$ *and* $\rho \geqslant 1$ *and* $1 \leqslant c \leqslant c^*$ *where* $r \in \mathbb{R}$ *and* $\rho, c, c^* \in \mathbb{N}$, $G^n_{r,c,c^*,\rho}$ *is the geometric random graph built in the following way:*

- *The vertex set consists of* n *points picked independently according the uniform distribution in* $[0, 1]^2$.
- *The edges are connections formed through the following asynchronous process: each vertex attempts to connect to* c *nodes chosen uniformly at random among those within distance* r. *A vertex which is requested connection declines it if its degree is already* c^*. *A vertex attempting to connect can make* ρ *tries for each of its* c *connections.*

This graph is clearly a sub-graph of the visibility graph, the protocol correctness is based on the fact that given a connected visibility graph, with high probability it is possible to build a connected $G^n_{r,c,c^*,\rho}$ fixing $r \geqslant 0$ choosing $c \geqslant 3$ and $c^* = \frac{737280}{r^2}c$ and $\rho = \gamma \lceil \log n \rceil$, with $\gamma > 0$. The proof of that is given in [110].

During the Blue Pleiades device discovery phase, then, each node connects to c nodes falling within its visibility range. The device discovery has a time limit to avoid undefined waiting to the nodes having less than c neighbors. The authors optimistically set $c = c^*$ and use a time-out instead of ρ, but give experimental evidence that for increasing values of n, the topology is connected.

Once the device discovery is terminated, the protocol continues with the BlueStars piconet and scatternet formation applied on the topology obtained.

A Blue Pleiades simulation

We have simulated Blue Pleiades as a case study for MOMOSE. The purpose of the case study has been to evaluate the connectivity performance of part of the protocol in order to more efficiently perform the device discovery phase.

We have then implemented the protocol and we have evaluated its connectivity performance when used for a MANET formed by thousands of devices moving in an environment with obstacles simulating the historic center of Florence, Italy (see Figure 5.10).

We have performed our experiments varying the number nodes n, the number of neighbor c chosen to form the BT topology, and the transmission range r. For

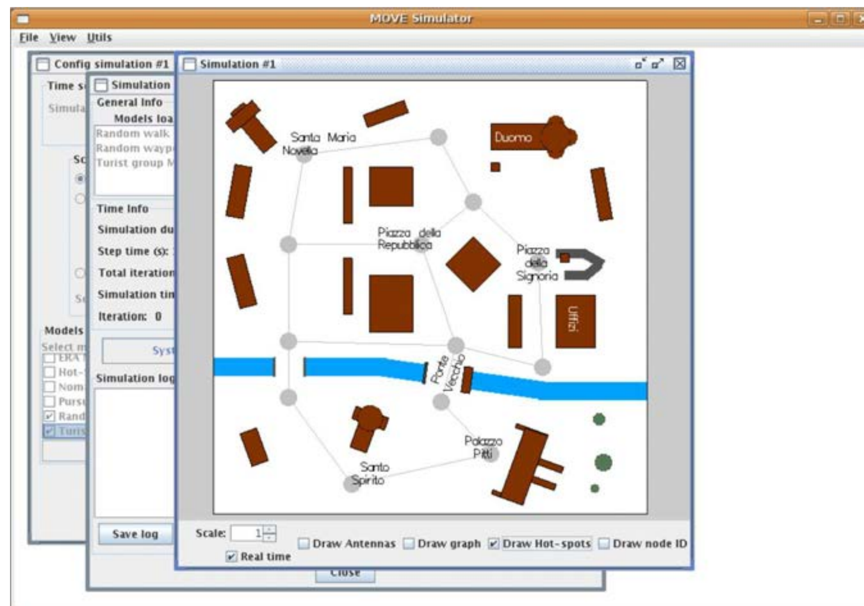


Figure 5.10: The Florence's historic center scenario used while evaluating the Blue Pleiades protocol

all the values of n and c we have performed 50 simulations where the nodes were equally divided to follow three different mobility models: two of such models were designed and implemented ad hoc for simulating the tourist movements across the Florence center monuments. The third one was the random walk mobility model we have seen in Section 4.2. We give a brief description of the first two models:

- *Random Waypoint Tourist mobility*: it is a variation of the random waypoint model we have described in Section 4.2. This version has been designed to simulate the movement of individual tourists. The model starts defining a list of hot spots positioned in the simulation area. Each hot spot corresponds to a point of interest of the center of Florence. The mobile nodes are distributed uniformly at random in the simulation area. During the simulation, each node chooses the hot spot which is the closest to it as a destination point. If there are not obstacles between the node and the hot spot, the node starts to move towards it. Otherwise, the node chooses a destination point according to the random waypoint model. Once the destination is reached, the procedure is repeated. Once the node has reached a monument, it can either decide to choose another monument, or decide to choose a random point in the simulation area.
- *Tourist Group*: it is a variation of the Pursue model we have described in Section 4.2. This version has been designed to simulate groups of tourists each one following its guide. In this model the hot spots we have described

in the random waypoint tourist model are connected through a graph (see Figure 5.10). Such a graph is used to build a path followed by a node representing a tourist guide. Each edge has a weight corresponding to the degree of interest for the monument which is decided by the guide. Each group is positioned along an edge connecting two hot spots. During the simulation, each guide node decides the next hot spot to be visited and starts to move towards it followed by its group. Once the hot spot is reached, the nodes stop for a certain amount of time. When the wait time expires the guide chooses another hot spot, which has not been still visited, and the group moves towards it. Once the list of hot spots to visit has been completed, the guide defines a new set of weights for the edges and the "trip" can restart.

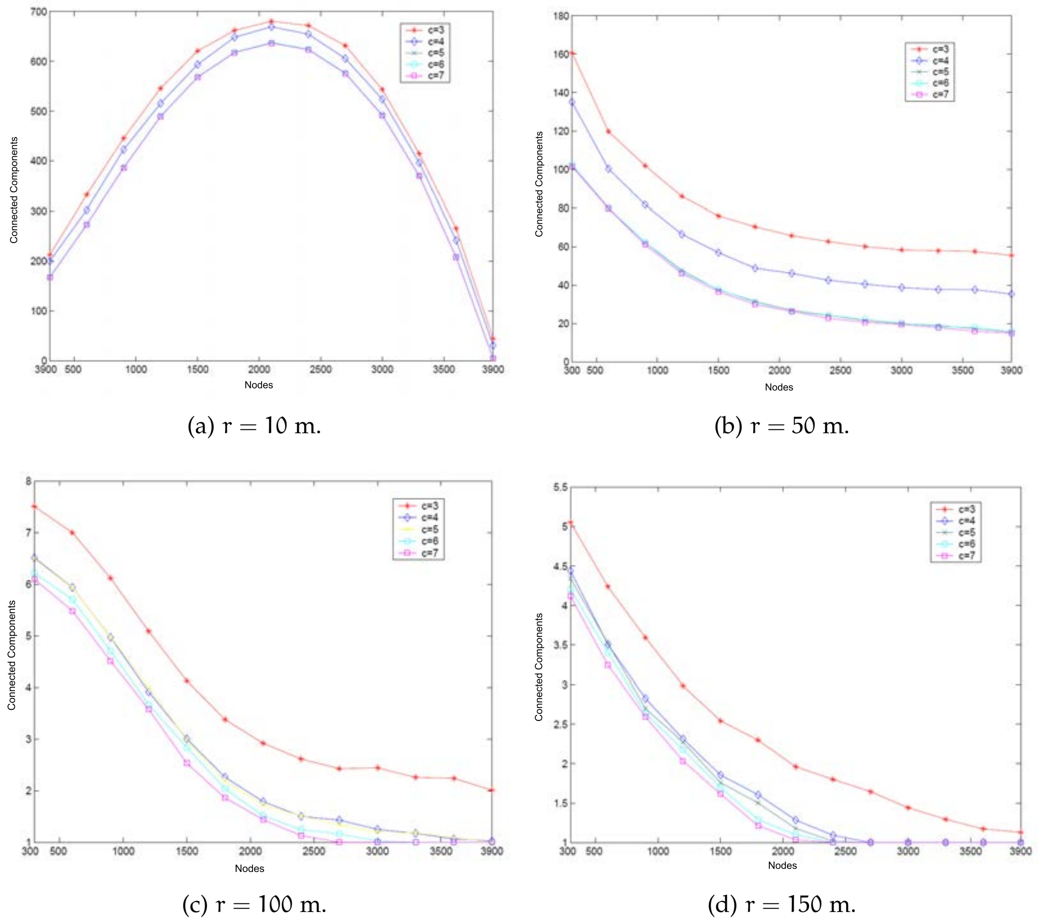
The simulation time has been set to 540 minutes (9 hours). The data extracted from the simulations were the number of connected component and the number of node included in the biggest connected component.

The results of the experiments are visible in Figure 5.11 and Figure 5.12. In all the figures the different functions refer to different values of c.

As it is visible from the charts of Figure 5.11, if the number of selected neighbors is at least 5, the number of connected components produced, in both the scenarios, by the Blue Pleiades protocol does not increase too much with respect to the case in which 6 or 7 neighbors are selected. The values for $c = 3$, 4 are instead quite higher especially in the cases of $r = 10, 50$, while when we increase the transmission range, that is $r = 100, 150$, the values of the two functions are closer. This is a predictable fact: indeed, increasing the transmission range, it is more likely that more nodes fall into each others transmission range.

The results for the number of connected components are reflected also into the charts visible in Figure 5.12. In the case of the percentage of nodes included into the biggest connected component as well, the case of $c = 3, 4$ shows a worse behavior with respect to the cases in which $c = 5, 6, 7$.

We have introduced the main concepts of Bluetooth technology and we gave an overview of some of the most important scatternet formation protocols. We have reported the protocols in a chronological order, this has been done to remark the fact that over the years the protocols design have moved toward two main directions. On one hand, there was an effort to limit the number of neighbors of a node. On the other hand, there was an effort to increase the number of nodes forming the Bluetooth scatternet. Limiting the number of neighbors results into scatternets where the number of parked nodes is minimized (in some cases it is equal to 0) and given the fact that the unparking phase its time-consuming, the overall performance of the system increases. Furthermore, if we think about a mobile environment, the less time a device discovery protocol takes, the best such a protocol can be used in a highly dynamic environment as a MANET.



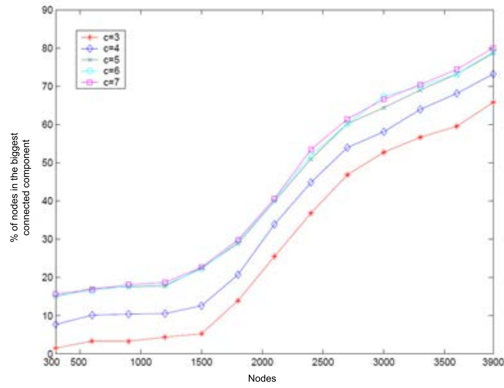
(a) $r = 10$ m.

(b) $r = 50$ m.

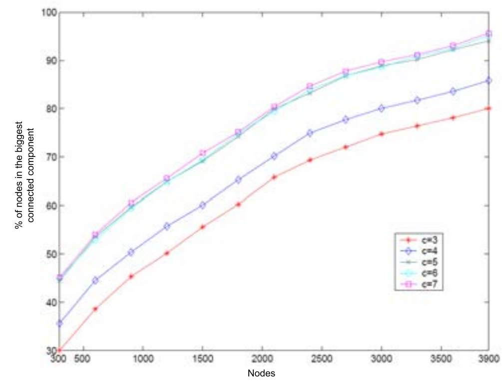
(c) $r = 100$ m.

(d) $r = 150$ m.

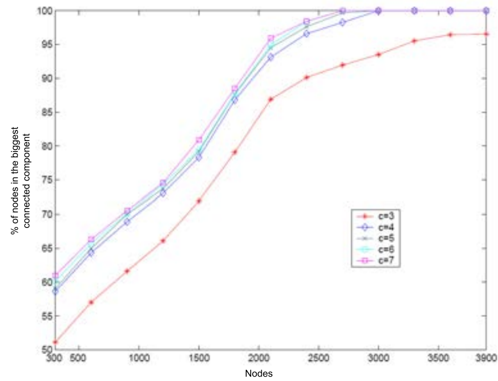
Figure 5.11: Number of connected components of the BT topology for some values of c, as the number of nodes increases for four different values of transmission range.



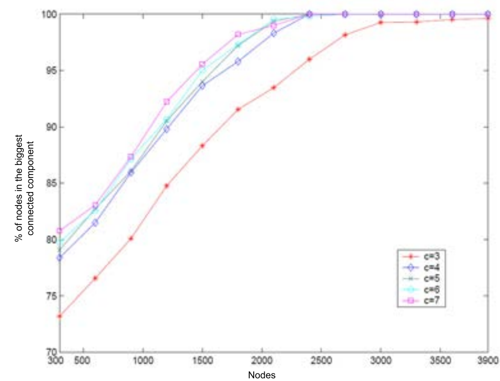
(a) r = 10 m.



(b) r = 50 m.



(c) r = 100 m.



(d) r = 150 m.

Figure 5.12: Percentage of nodes included into the biggest connected component for some values of c, as the number of nodes increases for four different values of transmission range.

The reason of increasing the number of nodes forming the scatternet is instead aimed at proving that Bluetooth could be used to build large ad hoc networks.

Finally, in all the protocols we have described, the transmission range of each node was assumed constant. We moved one step forward considering the visibility range as a variable parameter.

5.1.3 *From constant* r *to* $r(n)$

Since requiring each device to discover *all* of its neighbors is too time consuming [101], a crucial problem consists of deciding how many neighbors have to be selected in order to guarantee that the resulting Bluetooth topology is connected. Indeed, obtaining connectivity under degree limitations has been indicated in [100] as a major challenge for the adoption of the Bluetooth technology for large ad hoc networks.

In [111] the device discovery step has been effectively modeled as follows. The devices are regarded as a set of n nodes randomly and uniformly distributed in a square of unit side. Each node has a visibility range of $r(n)$, i.e., it can "see" all other nodes within Euclidean distance $r(n)$. Given a function $c(n)$, each node selects as neighbors $c(n)$ visible nodes at random, picking all visible nodes if their number is less than $c(n)$. Observe that the process is unidirectional in nature: however, each link established in this way becomes bidirectional. As a consequence, the final degree of each node may be much higher than $c(n)$, in the case that the node was selected as a neighbor by many other nodes. We refer to $BT(r(n), c(n))$ as the resulting (undirected) graph (observe that $BT(r(n), c(n))$ is a generalization of the well-studied random geometric graph [112, 113, 114] which can be obtained by setting $c(n) \geqslant n - 1$).

Previous studies on the connectivity properties of $BT(r(n), c(n))$ have considered only the case where each node is able to see a constant fraction of all other nodes, that is, the visibility range $r(n)$ is a constant. For this particular case, the experimental analysis conducted in [111] has shown that setting $c(n)$ to a small constant is sufficient to yield connectivity for $BT(r(n), c(n))$ almost always. The experimental evidence has been later substantiated by the analysis in [115], which shows that, for constant $r(n)$, $c(n) = 2$ is sufficient to achieve connectivity with high probability. Also, in [116] it was proved that constant $c(n)$ (though much larger, in the order of the millions) is also sufficient to guarantee linear expansion of $BT(r(n), c(n))$. These results suggest that device discovery can be performed efficiently whenever the network is sufficiently small (even though not necessarily a PAN). However, the assumption of constant $r(n)$ becomes quickly unfeasible as the number of devices to be connected increases, which would be the case when adopting Bluetooth for building large ad hoc networks. A motivation rooted in the technology of the devices for studying vanishing visibility range is the presence of the interference effect. Indeed, as the number

of devices increases, the communication between two nodes which would be in each other visibility range becomes more difficult because of the increasing density of transmitting sources.

We extend the above studies by providing both analytical and experimental evidence that, when the visibility range is a vanishing function of n, the device discovery step in Bluetooth can still be performed efficiently while guaranteeing connectivity, by letting each device discover only a"small", although non constant, number of neighbors. In particular, we prove that if $r(n) = \Omega(\sqrt{\ln n/n})$, then $BT(r(n), c(n))$ is connected with high probability as long as $c(n) = \Omega(\ln(1/r(n)))$. We remark that the lower bound on $r(n)$ cannot be improved since it is known that when $r(n) \leqslant \delta\sqrt{\ln n/n}$, for some constant $0 < \delta < 1$, the *visibility graph* where each node is connected to *all* nodes in its visibility range is disconnected with high probability [112, 117]. A challenging open question is whether the lower bound on the value of $c(n)$ required for connectivity is tight. We give a partial analytical answer to this question by showing that in fact $c(n) = 3$ is sufficient to attain connectivity with high probability, as long as $r(n) \geqslant n^{-\epsilon}$, for some constant $0 < \epsilon < 1/2$, but each node must choose two of the three neighbors sufficiently close to it.

We also report on a massive set of experiments conducted in order to assess the real performance of the two previously described protocols. Quite surprisingly, the experiments indicate that, even when the visibility range function is close to the aforementioned lower bound, the number of neighbors needed for connectivity exhibits an extremely weak dependence on $r(n)$: in fact, for values of n up to the hundreds of thousands $c(n) = 3$ suffices almost always, *independently of how the neighbors are chosen*. Moreover, the experiments show that the expected maximum total degree featured by the topologies obtained by choosing three neighbors for each node is much smaller than the one featured by the visibility graph, while the diameter is only slightly larger.

Even though our results are mainly motivated by the question of whether Bluetooth is suitable as a large-scale ad hoc network technology, we believe that they may be of interest for other wireless network scenarios [118].

The rest of the chapter is organized as follows. Section 5.2 analyzes the connectivity of $BT(r(n), c(n))$ when $c(n)$ is $\Theta(\log(1/r(n)))$. Section 5.3 analyzes the case of $c(n) = 3$ under further constraints on neighbor selection. Section 5.4 reports the results of our experiments, while in Section 5.5 we conclude with some final considerations and proposals for further research.

5.2 CONNECTIVITY OF BT$(r(n), c(n))$

Consider a set V of n nodes randomly and uniformly distributed in a unit-side square. Each node $v \in V$ has a visibility range of $r(n)$, i.e., v can "see" all nodes u at Euclidean distance $d(v, u) \leqslant r(n) \leqslant 1$.[1]

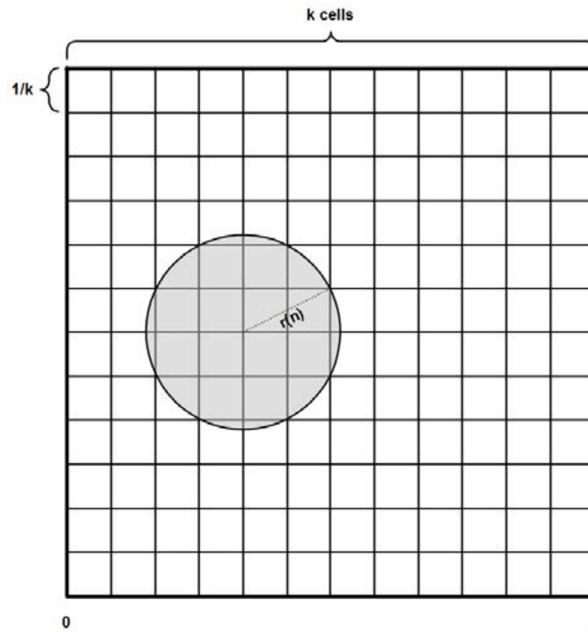


Figure 5.13: The tessellated unit square with $k = \left\lceil \frac{\sqrt{5}}{r(n)} \right\rceil$.

Let the unit square be tessellated into k^2 square *cells* of side $1/k$, where $k = \lceil \sqrt{5}/r(n) \rceil$ (see Figure 5.13). Consequently, any two nodes residing in the same or in adjacent cells (i.e., cells sharing a side) are at distance at most $r(n)$: hence, they are visible from one another. Most of our results hold *with high probability* (*w.h.p.* for short) by which we mean that the probability of the stated event is at least $1 - 1/\text{poly}(n)$, where $\text{poly}(n)$ denotes some polynomial function of n. We need the following technical fact.

Proposition 5.1 *Let* $\alpha = 9/10$ *and* $\beta = 11/10$*. There exists a constant* $\gamma_1 > 0$ *such that for every* $r(n) \geqslant \gamma_1 \sqrt{\ln n / n}$ *the following events occur together w.h.p.*

1. *Every cell contains at least* $\alpha n / k^2$ *and at most* $\beta n / k^2$ *nodes.*
2. *Every node has at least* $(\alpha/4)\pi n r^2(n)$ *and at most* $\beta \pi n r^2(n)$ *other nodes in its visibility range.*

1 In fact, the maximum significant range in the case of the unit square is $\sqrt{2}$. Placing an upper bound of 1 allows us to simplify some of the proofs, however, all of the results in the chapter would still hold up to the maximum range.

Proof. It is sufficient to show that any given cell contains at least $\alpha n/k^2$ and at most $\beta n/k^2$ nodes with probability greater than or equal to $1 - 1/(2n^2)$, and that any given node has at least $(\alpha/4)\pi n r^2(n)$ and at most $\beta \pi n r^2(n)$ other nodes in its visibility range, with probability greater than or equal to $1 - 1/(2n^2)$. Since there are less than n cells and exactly n nodes, the proposition will follow by a simple application of the union bound. Fix a cell Q. By using Chernoff's bound [119] and the fact $k \leqslant 4/r(n)$, we obtain that the probability that Q contains more that $\beta n/k^2$ nodes is less than

$$\left(\frac{e^{\beta-1}}{\beta^\beta}\right)^{\gamma_1{}^2 \ln n/16}.$$

A symmetrical argument shows that the probability that Q contains less than $\alpha n/k^2$ nodes is less than

$$\left(\frac{e^{1-\alpha}}{(2-\alpha)^{2-\alpha}}\right)^{\gamma_1{}^2 \ln n/16}.$$

By choosing a suitable constant γ_1, both upper bounds can be made smaller than $1/(4n^2)$. Therefore, the probability that the number of nodes in Q is between $\alpha n/k^2$ and $\beta n/k^2$ is at least $1 - 1/(2n^2)$.

The probability bound regarding the number of nodes in the visibility range of any fixed node can be proved in a similar fashion by making the further observation that for $r(n) \leqslant 1$, the visibility range of any node covers an area of the unit square which is at least $(\pi/4)r^2(n)$ and at most $\pi r^2(n)$. □

The rest of the section is devoted to the proof of the following theorem.

Theorem 5.1 *There exist two positive real constants* γ_1, γ_2 *such that, if* $r(n) \geqslant \gamma_1 \sqrt{\ln n/n}$ *and* $c(n) = \gamma_2 \ln(1/r(n))$ *then* $\mathrm{BT}(r(n), c(n))$ *is connected w.h.p.*

Let $\epsilon = 1/8$. In the proof of the theorem we distinguish between the case $r(n) \leqslant n^{-\epsilon}$ and the case $r(n) > n^{-\epsilon}$, which are dealt with separately in the following subsections. Moreover, in both cases we condition on the events expressed by Proposition 5.1, which occur with high probability.

5.2.1 *Case* $\gamma_1 \sqrt{\ln n/n} \leqslant r(n) \leqslant n^{-\epsilon}$

We fix the lower bound for $r(n)$ to be the same under which Proposition 5.1 holds. In the range of $r(n)$ considered in this case, we have that $c(n) = \gamma_2 \ln(1/r(n)) = \Theta(\ln n)$. Let Q be an arbitrary cell and let G_Q denote the subgraph of $\mathrm{BT}(r(n), c(n))$ formed by nodes and edges internal to Q. We first show that every G_Q is connected and then prove that for every pair of adjacent cells there exists an edge in $\mathrm{BT}(r(n), c(n))$ whose endpoints are in the two cells.

Lemma 5.1 *With high probability, every* G_Q *is connected.*

Proof. Fix an arbitrary cell Q and let A_Q be the event that, for every partition of the nodes in Q into two nonempty subsets, there is at least an edge with endpoints in distinct subsets. Observe that the subgraph $G_Q \subseteq BT(r(n), c(n))$ is connected if and only if A_Q occurs. Then:

$$\begin{aligned} 1 - \Pr(A_Q) &\leqslant \sum_{s=1}^{\beta n/(2k^2)} \binom{\beta n/k^2}{s} \left(1 - \frac{\alpha n/k^2 - s}{\beta \pi n r^2(n)}\right)^{sc(n)} \cdot \\ &\quad \cdot \left(1 - \frac{s}{\beta \pi n r^2(n)}\right)^{(\alpha n/k^2 - s)c(n)} \\ &\leqslant \sum_{s=1}^{\beta n/(2k^2)} \exp\left(s \ln \frac{e\beta n}{sk^2} - \frac{2sc(n)}{\beta \pi n r^2(n)} \left(\frac{\alpha n}{k^2} - s\right)\right). \end{aligned}$$

Note that for the values of s in the summation range, we have that

$$\ln(e\beta n/(sk^2)) = O(\ln n)$$

and

$$((\alpha n/k^2) - s)/(\beta \pi n r^2(n)) = \Theta(1),$$

therefore by choosing the constant γ_2 in the expression for $c(n)$ large enough, the summation is dominated by its first term and can be made as small as $1/n^2$. The lemma follows by applying the union bound over all k^2 cells. □

Lemma 5.2 *With high probability, for every pair of adjacent cells* Q_1 *and* Q_2 *there is an edge* $(u, v) \in BT(r(n), c(n))$ *such that* u *resides in* Q_1 *and* v *resides in* Q_2.

Proof. Consider an arbitrary pair of adjacent cells Q_1 and Q_2 and let B_{Q_1,Q_2} denote the event that there is at least one edge in $BT(r(n), c(n))$ between the two cells. Since we are conditioning on the events described in Proposition 5.1, we have that

$$\begin{aligned} 1 - \Pr(B_{Q_1,Q_2}) &\leqslant \left(1 - \frac{\alpha n/k^2}{\beta \pi n r^2(n)}\right)^{2c(n)\alpha n/k^2} \\ &\leqslant \exp\left(-\frac{\alpha n/k^2}{\beta \pi n r^2(n)}(2c(n)\alpha n/k^2)\right) \\ &\leqslant \exp(-\zeta \ln^2 n), \end{aligned}$$

where ζ is a positive constant. The lemma follows by applying the union bound over all $O(n)$ pairs of adjacent cells. □

For the case $\gamma_1 \sqrt{\ln n/n} \leqslant r(n) \leqslant \delta n^{-\epsilon}$, Theorem 5.1 follows by combining the results of the above two lemmas.

5.2.2 Case $n^{-\epsilon} < r(n) \leqslant 1$

We generalize and simplify the argument which was used in [115] for the case $r(n) = \Theta(1)$. Specifically, we first show that $BT(r(n), c(n))$ contains a large connected component C, and then we show that for every node v there is a path from v to C. Again, we condition on the events stated in Proposition 5.1, which occur with high probability.

Lemma 5.3 *For* $n^{-\epsilon} < r(n) \leqslant 1$ *and* $c(n) \geqslant 2$, $BT(r(n), c(n))$ *contains a connected component of size* $n/(8k^2)$, *w.h.p.*

Proof. The argument is a simple adaptation of the one used in the proof of Proposition 3 in [115], which we highlight in the following for the sake of completeness. Starting from an arbitrary node u, consider a sequential discovery procedure where each node chooses two random neighbors out of those nodes in its visibility range. A simple application of the Chernoff bound [119] shows that each node has at least $n/(2k^2)$ other nodes in its visibility range with probability at least $1 - k^2 e^{-n/(8k^2)}$. This implies that the probability of reaching a previously discovered node in the first $2\log_2 n - 2$ neighbor selections is at most $8k^2 \log_2^2 n/n = O\left(\log^2 n/n^{1-2\epsilon}\right) = O\left(\log^2 n/n^{3/4}\right)$. Hence, by stopping the sequential discovery after $2\log_2 n - 2$ neighbor selections have been executed, we get, with probability $1 - O\left(\log^2 n/n^{3/4}\right)$, a full binary tree rooted at u with $\log_2 n$ leaves. Now, from these leaves we run $\log_2 n$ independent sequential discoveries until a total of $n/(8k^2)$ nodes are discovered. By reasoning as in [115] we argue that this process stochastically dominates a branching process [120] beginning with $\log_2 n$ individuals and binomial offspring distribution with parameters 2 and 3/4. We conclude that the probability of failing to reach $n/(8k^2)$ nodes is bounded from above by the probability that the branching process dies out, which is at most $(1/9)^{\log_2 n}$.

Putting it all together, the probability that the component grown from u has size at least $n/(8k^2)$ is at least

$$1 - O\left(\frac{\log^2 n}{n^{3/4}} + \frac{1}{n^{\log_2 9}}\right),$$

which proves the lemma. □

Let C be the connected component of size at least $n/(8k^2)$ which, by the above lemma, exists w.h.p. By the pigeonhole principle there must exist a cell Q containing at least $n/(8k^4)$ nodes of C. Let $V(Q, C)$ the set of nodes residing in Q and belonging to C. We have:

Lemma 5.4 *With high probability, for each node* u *there exists a path in* $BT(r(n), c(n))$ *from* u *to some node in* $V(Q, C)$.

Proof. Consider a directed version of $\mathrm{BT}(r(n), c(n))$ where an edge (u, v) is directed from u to v if u selected v during the neighbor selection process. Since we are conditioning on the event stated in the second point of Proposition 5.1, our choice of $c(n)$ implies that the outdegree of each node is *exactly* $c(n)$ w.h.p. Pick an arbitrary node u and run a sequential breadth-first exploration from u in such a directed version of $\mathrm{BT}(r(n), c(n))$. With respect to this exploration, we say that a failure occurs whenever an edge (v_1, v_2) is considered during the exploration of v_1, but node v_2 has been already discovered. Let m be a suitable value, to be fixed later by the analysis. We stop the exploration as soon as one of the following events happen: (a) the $c(n)$-th failure occurs; or (b) m nodes are discovered but not yet explored. We now prove an upper bound on the probability that the exploration stops due to event (a). In this case, it is easy to show that any time before the $c(n)$-th failure occurs, the tree formed by the nodes discovered so far has at most one internal node of degree one, hence the tree contains less than m leaves (i.e., the unexplored nodes) and less than m internal nodes, for a total of less than $2m$ nodes altogether.

From the second point of Proposition 5.1 it follows that the probability that event (a) happens prior to event (b) is at most

$$\binom{m \cdot c(n)}{c(n)} \left(\frac{2m}{(\alpha/4)\pi n r^2(n) - c(n)} \right)^{c(n)} \leqslant \left(\frac{2em^2}{(\alpha/4)\pi n r^2(n) - c(n)} \right)^{c(n)}, \quad (5.1)$$

where the binomial coefficient bounds from above the number of ways of fixing $c(n)$ failures in the node explorations, which are less than m, while the subsequent factor bounds from above the probability of a fixed configuration of $c(n)$ failures when less than $2m$ nodes have been discovered.

We will choose m so to make the upper bound given in Equation 5.1 vanishingly small in n. Therefore, we may condition on the event that m unexplored nodes, say $w_1, w_2, \dots, w_m$, are reached via breadth-first exploration from u before $c(n)$ failures occur. We now estimate the probability that $\mathrm{BT}(r(n), c(n))$ contains a path from w_i to a node in $V(Q, C)$. Observe that from the cell containing w_i there is a sequence of at most $2k$ pairwise adjacent cells ending at Q. Specifically, we estimate the probability that $\mathrm{BT}(r(n), c(n))$ contains a path from w_i to $V(Q, C)$ following such a sequence of cells, with the constraint that the path contains one node per cell and these nodes do not belong to the set of at most $2m$ nodes initially discovered from u or to the $m - 1$ paths constructed for any other w_j, with $j \neq i$. This probability is at least $p^{2k}q$, where p is the probability of extending the path one cell further, and q is the probability of ending,

in the last step, in a node of $V(Q, C)$. By using the bounds in Proposition 5.1 we have that

$$\begin{aligned} p &\geqslant \left(1-\left(1-\frac{\alpha n/k^2 - 3m}{\beta\pi n r^2(n)}\right)^{c(n)}\right) \\ q &\geqslant \frac{n/(8k^4)}{\beta\pi n r^2(n)} = \frac{1}{8\beta\pi k^4 r^2(n)}. \end{aligned}$$

Recall that $c(n) = \gamma_2 \ln(1/r(n)) = \Theta(\ln k)$. If we take $m = o(n/k^2)$ and γ_2 large enough, we have that

$$p^{2k} \geqslant \tau$$

for some constant $0 < \tau < 1$. It follows that the probability that all of the w_is fail to reach $V(Q, C)$ is at most

$$(1-\tau q)^m \leqslant \left(1-\frac{\tau}{8\beta\pi k^4 r^2(n)}\right)^m = \left(1-\frac{\tau}{\sigma k^2}\right)^m, \tag{5.2}$$

for some positive constant σ.

By combining Equations 5.1 and 5.2, we get that the probability that u is not connected to $V(Q, C)$ is at most

$$\left(\frac{2em^2}{(\alpha/4)\pi n r^2(n) - c(n)}\right)^{c(n)} + \left(1-\frac{\tau}{\sigma k^2}\right)^m.$$

Now, since $r(n) > n^{-1/8}$, we have that $k = O(n^{1/8})$. If we choose $m = \Theta(n^{1/3})$ we have that $m = o(n/k^2)$, as required above, and $m = \omega(k^2 \ln n)$. This, combined with the choice of $c(n)$, ensures that the above probability is smaller than $1/n^2$. The lemma follows by applying the union bound over all nodes u. □

For the case $r(n) > n^{-\epsilon}$, Theorem 5.1 follows by combining the results of the above two lemmas.

5.3 ACHIEVING $c(n) = 3$ USING A DOUBLE CHOICE PROTOCOL

In the previous section we showed that selecting $c(n) = O(\ln(1/r(n)))$ visible neighbors at random is sufficient to enforce global connectivity for all values of $r(n)$ which guarantee connectivity of the visibility graph. Whether these many neighbors are necessary remains a challenging open question. As a step towards this objective, we show that, at least for large enough (yet non constant) radii, $c(n) = 3$ always suffices under a slightly different neighbor selection protocol where each node is required to direct the selection of some neighbors within a certain geographical region. Such a phenomenon provides evidence that the $O(\ln(1/r(n)))$ bound on $c(n)$ is not likely to be tight.

More formally, consider again the tessellation of the unit square into k^2 square cells of side $1/k$, with $k = \lceil \sqrt{5}/r(n) \rceil$. Define $BT(r(n), 2, 1)$ to be the undirected graph resulting by letting each node select two neighbors at random among the nodes residing in its cell, and another neighbor at random among all visible nodes. Observe that if applied in a practical scenario, this *double-choice* protocol would require each node to infer geographical information about its location and the location of the nodes in its visibility range. For example, this information could be provided by a GPS device.[2]

Theorem 5.2 *There exists a constant* ϵ*,* $0 < \epsilon < 1/2$ *such that if* $r(n) = \Omega(n^{-\epsilon})$*, then* $BT(r(n), 2, 1)$ *is connected w.h.p.*

Proof. We employ the same approach used in Subsection 5.2.1. Specifically, we first argue that, with high probability, for all cells Q the graph G_Q induced by the nodes in Q is connected, and that for every pair of adjacent cells there is an edge with endpoints in the two cells. Since by the first point of Proposition 5.1, each cell Q contains $\Omega(n^{1-2\epsilon})$ nodes w.h.p., the main result of [115] implies that two neighbors selected by each node in Q suffice to guarantee connectivity of G_Q with probability at least $1 - 1/n^{\delta(1-2\epsilon)}$, for a suitable positive constant $\delta < 1$. Then, choosing ϵ smaller than $\delta/(2(1+\delta))$ and applying the union bound, all cells will be internally connected with high probability. In order to prove connectivity between adjacent cells, we proceed as in the proof of Lemma 5.2. In particular, consider an arbitrary pair of adjacent cells Q_1 and Q_2, and let B_{Q_1,Q_2} denote the event that there is at least one edge in $BT(r(n), 2, 1)$ between the two cells. By conditioning on the events described in Proposition 5.1, we have that

$$\begin{aligned} 1 - \Pr(B_{Q_1,Q_2}) &\leqslant \left(1 - \frac{\alpha n/k^2}{\beta \pi n r^2(n)}\right)^{2\alpha n/k^2} \\ &\leqslant \exp\left(-\frac{\alpha n/k^2}{\beta \pi n r^2(n)}(2\alpha n/k^2)\right) \\ &\leqslant \exp(-\zeta n^{1-2\epsilon}), \end{aligned}$$

where ζ is a positive constant. The theorem follows by applying the union bound over all $O(n)$ pairs of adjacent cells. □

5.4 EXPERIMENTS

We have designed an extensive suite of experiments aimed at comparing the connectivity and other topological properties of the graphs analyzed in the pre-

2 A full discussion on the feasibility of this approach is outside the scope of this thesis, since the analysis of the double-choice protocol is mostly meant to provide evidence that the selection of very few neighbors may suffice in order to build a connected topology.

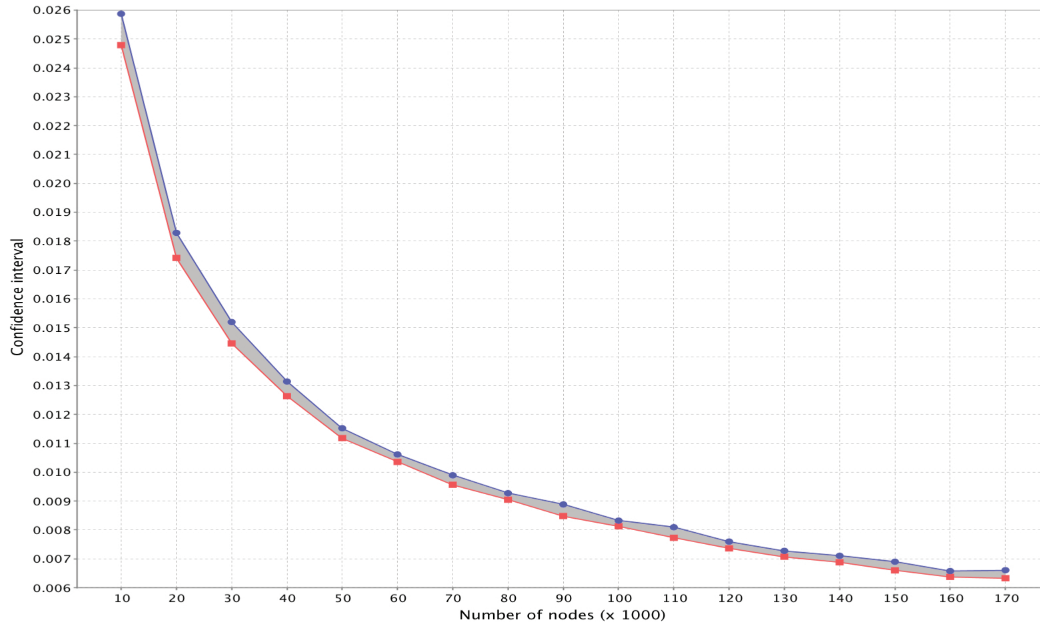


Figure 5.14: The 95% confidence intervals of the minimum range of the visibility graph

vious sections.[3] In a first set of experiments, for values of n ranging from 10000 to 170000 with step 10000, we have performed 20 times the following binary search of the minimum range that guarantees connectivity of the visibility graph associated with the placement (i.e., the graph where each node connects to all its visible neighbors).

1. Set the search interval to $\left[r_{\text{left}} = 0.1\sqrt{\ln/n}, r_{\text{right}} = 0.99\sqrt{\ln/n}\right]$.
   Comment: r_{left} (resp., r_{right}) is a value which guarantees, in practice, that the corresponding visibility graph is always disconnected (resp., connected).

2. If the length of the search interval is less than $0.005r_{\text{left}}$, then return r_{right}.

3. Generate 50 placements of n nodes in the unit square and verify whether for all of them the range $r = (r_{\text{left}} + r_{\text{right}})/2$ guarantees connectivity of the visibility graph associated with the placement. If this the case, then set $r_{\text{right}} = r$ otherwise set $r_{\text{left}} = r$. Go to step 2.

Figure 5.14 plots, for each value of n, the lower (r_{lb}) and upper (r_{ub}) endpoints of the 95% confidence intervals yielded by the 20 estimates of the minimum range provided by the above experiments.

We repeated the above procedure to estimate the minimum ranges r_{sc} and r_{dc} which guarantee connectivity of $BT(r_{\text{sc}}, 3)$ and $BT(r_{\text{dc}}, 2, 1)$, respectively, so to

3 The implemented code makes use of the Boost Graph Libraries [121] for computing the number of connected components and for performing a breadth first search of a graph.

n	$[r_{lb}; r_{ub}]$	r_{sc}	r_{dc}
10000	[0.0247868;0.0258708]	0.0251758	0.0253024
20000	[0.0174184;0.0182865]	0.0178253	0.0177519
30000	[0.0144605;0.0151994]	0.0146205	0.0149556
40000	[0.0126296;0.0131409]	0.0129900	0.0126957
50000	[0.0111809;0.0115222]	0.0115894*	0.0112902
60000	[0.0103549;0.0106147]	0.0105789	0.0108026*
70000	[0.0095643;0.0098931]	0.0097572	0.0098735
80000	[0.0090486;0.0092731]	0.0092001	0.0091857
90000	[0.0084843;0.0088834]	0.0087328	0.0087249
100000	[0.0081272;0.0083245]	0.0082948	0.0082109
110000	[0.0077308;0.0080951]	0.0078290	0.0080540
120000	[0.0073631;0.0075985]	0.0076215*	0.0075632
130000	[0.0070734;0.0072776]	0.0072350	0.0072433
140000	[0.0068810;0.0071065]	0.0069794	0.0070066
150000	[0.0066020;0.0069010]	0.0068197	0.0067747
160000	[0.0063759;0.0065772]	0.0066330*	0.0066525*
170000	[0.0063293;0.0066083]	0.0063751	0.0063459

Table 5.2: Comparison between the ranges r_{sc} and r_{dc} which guarantee connectivity for $BT(r_{sc}, 3)$ and $BT(r_{dc}, 2, 1)$, and the 95% confidence intervals for the minimum range that guarantees connectivity for the visibility graph. Starred values highlight outliers.

appreciate whether there is a significant discrepancy between these minimum ranges and the one obtained for the visibility graph. As before, the binary search procedure has been repeated 20 times. At each execution of Step 3, we check whether connectivity is achieved for all of 50 graphs, each obtained by a random placement of n nodes and the neighbor selection protocol, and restrict the search interval accordingly.

Table 5.2 reports, for each value of n, the confidence interval $[r_{lb}; r_{ub}]$ for the minimum range of the visibility graph, and the values of r_{sc} and r_{dc}, averaged over the 20 estimates. According to these experiments, r_{sc} is very close to r_{lb} (always within 4% for all values of n) and it is almost always within the confidence interval itself (apart from the three starred cases). Also, r_{dc} features a very similar similar behavior. In fact, interestingly, connectivity of $BT(r(n), 2, 1)$ does not seem to require that $r(n) \in \Omega(\sqrt{1/n^{\epsilon}})$ as implied by the analysis, since it is attained for values of $r(n)$ close to r_{lb}.

In a second set of experiments we measured the maximum degree of the graphs $BT(r(n), 3)$ and $BT(r(n), 2, 1)$, and of the visibility graph with visibility

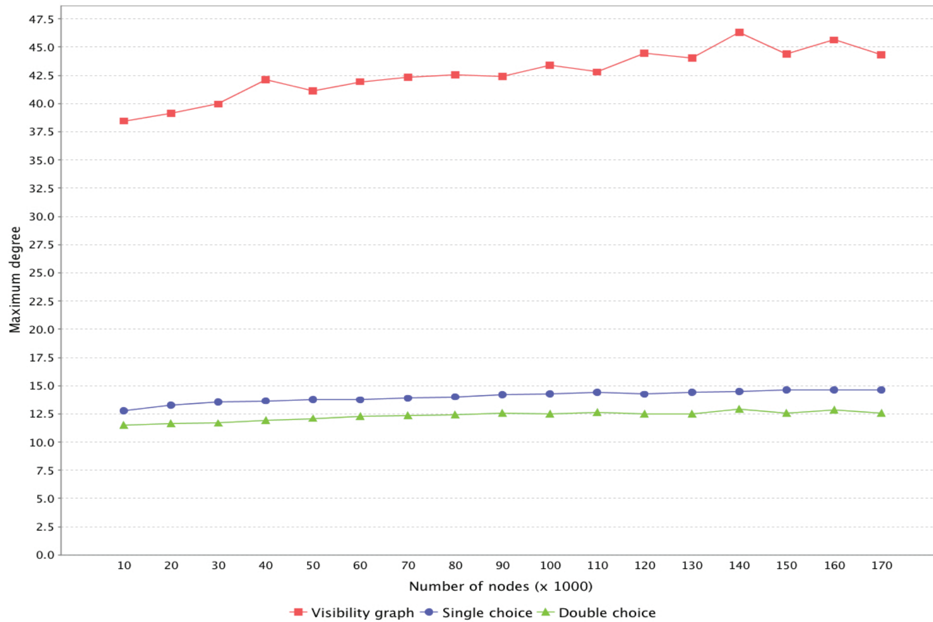


Figure 5.15: Comparison of the average maximum degree of $BT(r(n), 3)$, $BT(r(n), 2, 1)$, and of the visibility graph with range $r(n)$

range $r(n)$, where $r(n)$ is chosen to be an approximation of the smallest value which guarantees connectivity in all three cases. In particular, the average maximum degrees have been computed over 50 random placements for which the values $(r_{lb} + r_{ub})/2$, r_{sc}, and r_{dc}, respectively, guaranteed connectivity of the corresponding graphs. The results of these experiments are depicted in Figure 5.15 for each value of n. It can be seen that $BT(r(n), 2, 1)$ exhibits a slightly smaller maximum degree than $BT(r(n), 3)$, and, clearly, both graphs have a much smaller maximum degree than the visibility graph whose expected maximum degree can be analytically shown to be $\Theta(\ln n)$ when $r(n) = \Theta(\sqrt{(\ln n)/n})$.

One last set of experiments concerned the estimation of the average diameter of $BT(r(n), 3)$ and $BT(r(n), 2, 1)$, and of the visibility graph with visibility range $r(n)$, where $r(n)$ is chosen as in the previous experiments. In order to avoid expensive all-pairs shortest paths computations, we have chosen to approximate the diameter as twice the maximum height of 30 breadth-first search trees rooted at randomly chosen nodes. The results of these experiments are depicted in Figure 5.16, once again reporting, for each n the averages over 50 estimates. It can be seen that $BT(r(n), 3)$ has a diameter which is smaller than the one of $BT(r(n), 2, 1)$, and not considerably larger than the one of the visibility graph.

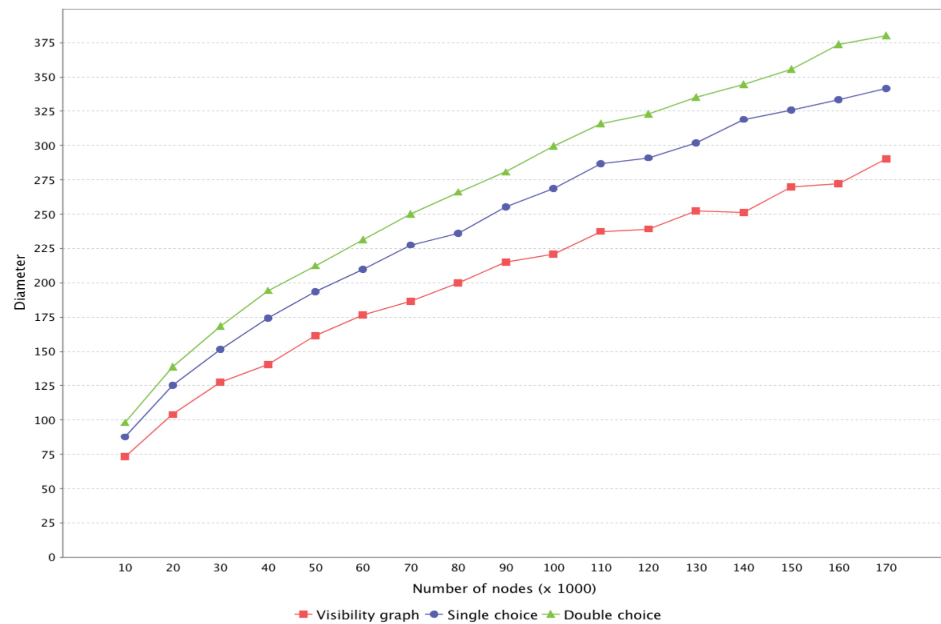


Figure 5.16: Comparison of the average diameter of $BT(r(n), 3)$, $BT(r(n), 2, 1)$, and the visibility graph with range $r(n)$

5.5 CONCLUSIONS

In this chapter we have analyzed th Bluetooth technology which is a perfect example of ad hoc network. We have described the BSF protocols to the motivations which have guided us to the main result contained in this chapter.

The main theoretical contribution of this chapter is a proof of connectivity for the Bluetooth graph when the visibility range $r(n)$ is a vanishing function of the number n of nodes and each node selects only a logarithmic number of neighbors with respect to $1/r(n)$. Also, we introduce and analyze a novel neighbor selection protocol based on a double choice mechanism, which ensures connectivity when a total of only three neighbors are selected by each node. In this chapter we also reported the results of extensive experiments which validate the theoretical findings. In fact, from the simulation results, we can conclude that, within the range of 10000 to 170000 nodes, the three protocols seem to be statistically indistinguishable for what concerns connectivity; on the other hand, if the degree of a node is the main concern, then the protocols based on the choice of fewer neighbors offer a substantial advantage, while if the diameter is the main concern, then the visibility graph is slightly superior (recall however that requiring each node to discover all of its visible neighbors may be unfeasible in the practical Bluetooth scenario). From a theoretical point of view, the experimental results substantiate that the best avenue for future research is to tighten the bound on $c(n)$ that guarantees connectivity of $BT(r(n), c(n))$,

since it appears that the double choice protocol (which is in fact of limited practical applicability) does not provide any significant advantage in practice.

6 CONCLUSIONS

In this thesis we have dealt with algorithmic, simulative and experimental aspects of dynamic networks. We have considered as dynamic all the networks in which the dynamism is caused directly by the intrinsic nature of such networks. Specifically, we have worked on two specific types of dynamic networks: *peer-to-peer networks* and *mobile ad hoc networks*.

In P2P networks, the dynamism is generated by the fact that such networks are designed to let a large number of nodes (peers) cooperate in order to perform some kind of task. Such a task is usually giving benefits to the whole network. However, the involvement of a peer in a P2P network is not mandatory. A peer can freely decide to leave the network whenever it wants to. This makes the network highly dynamic, since it may be subjected to a high churn of nodes joining and leaving the system.

In such a scenario, it is thus important to face this dynamism with the right instruments. Distributed hash tables (in short DHT) are one of these instruments. DHTs are becoming more and more popular in the P2P area. In this thesis we have given a tangible demonstration of the fact that the usage of a DHT in a P2P system, can noticeably improve its performance. We have seen that providing a preexisting P2P framework like JXTA with a pure DHT protocol like Chord, can increase the overall performance in terms of lookup for shared resources up to one order of magnitude.

In MANETs, the dynamism is generated by the capability of movement of the devices participating in the formation of the ad hoc network. The topology of a MANET is continuously changing, therefore studying its features is extremely challenging. It is then important to have both practical and theoretical instruments to model the behavior of MANETs and good metrics to measure the most important parameters influencing such systems. This thesis contains results from both the areas of the study of MANETs, practical and theoretical.

The most popular instruments which simulate the movement of wireless devices composing a MANET are the mobility models. Because of the difficulty to obtain concrete traces from the real world, there has been a lot of interest in synthetic mobility models, which are not trace-driven. We have given an overview of such mobility models and we have shown a practical tool, MOMOSE, which allows to simulate the behavior of such a complex system. The importance of having such an extensible instrument to cover the large number of existing mobility models is twofold: at first, it gives the possibility to simulate and study a wide range of phenomena connected to the MANETs; at second, it gives the

possibility to explore the characteristics of the existing mobility models in order to improve them and in order to design new models.

We have finally covered one of the most popular technologies used to build MANETs, the Bluetooth technology. Specifically, we have focused on the device discovery phase of the construction of the Bluetooth topology. In this area the main contribution has been theoretical and experimental. Indeed, we have provided both analytical and experimental evidence that when the visibility range of each node is limited to a vanishing function of the total number of nodes in the system, full connectivity can still be achieved with high probability by letting each node connect only to a "small" number of its visible neighbors. The connection to the fewer number of neighbors possible means a fastest reconfiguration of the system. This is clearly a fundamental property in a mobile system like a MANET.

LIST OF FIGURES

Figure 1.1 Network classification according size. 3
Figure 1.2 A small example of overlay network. 5
Figure 1.3 An infrastructured WLAN, continuous lines represent wired links, dashed ones represent wireless links. 7
Figure 1.4 A schematic representation of an ad hoc network. 9
Figure 2.1 Abstraction of the P2P overlay network architecture. 12
Figure 2.2 Unstructured topology of an overlay network. 13
Figure 2.3 Flooding over unstructured topology. 14
Figure 2.4 Random walk over unstructured topology. 14
Figure 2.5 Gnutella ultra-peers structure. 16
Figure 2.6 Routing of a search query in Freenet. 17
Figure 2.7 Phases of connection of BitTorrent. 19
Figure 2.8 Regions where one-hop and multi-hop approaches should be used ([12]). 21
Figure 2.9 Hierachical overlay P2P network. The g_i are the bottom-level groups 21
Figure 2.10 PRR neighbor table with $b = 1$ 24
Figure 2.11 PRR search tree with $b = 4$ and $k = 6$ 26
Figure 2.12 Example of CAN in a 2-dimensional space $[0, 1] \times [0, 1]$ 28
Figure 2.13 CAN routing in a 2-dimensional space 29
Figure 2.14 Join of a new node x associated with point P, node j is the introducer 30
Figure 2.15 Departure of node s, the zone of node t is merged with the unowned zone 30
Figure 2.16 Departure of node r, node n temporary takes care of the unowned zone (the merge is not possible) 31
Figure 2.17 A state of a node in Pastry where $b = 2$ and $l = 6$, the red digit is the position relative to the level, the underlined text is the prefix shared with the nodeId 33
Figure 2.18 Example of node lookup in a Kademlia network where $k = 3$, $m = 5$ and $\alpha = 3$, node 30 looks for node 8 37
Figure 2.19 Kelips structure from the point of view of node 105 contained in the i-th affinity group, here $c = 3$. On the right we represented the $k - 1$ foreign affinity groups. Observe that we reported only the contacts of the node. 39

Figure 2.20 An example of random landmarking in a MANET with $N = 128$ and $K = 4$, each color is relative to a different cluster, the numbers are written over the current landmark node 42
Figure 3.1 The JXTA protocols stack. 54
Figure 3.2 The JXTA layer architecture. 55
Figure 3.3 JXTA publication process 57
Figure 3.4 JXTA search process 58
Figure 3.5 JXTA search process having success thanks to replication 59
Figure 3.6 JXTA search process using the limited range walker 59
Figure 3.7 Finger tables for a chord ring with $m = 3$, green nodes are active. 61
Figure 3.8 Chord store process 63
Figure 3.9 Chord search process 63
Figure 3.10 Pseudo-UML representation of the rpv package 68
Figure 3.11 PeerViewElement schema 69
Figure 3.12 Pseudo-UML representation of the walker implementation. 74
Figure 3.13 A walk schema. 75
Figure 3.14 Query schema. 76
Figure 3.15 Finger table entry implementation. 80
Figure 3.16 The join process, in the boxes there are the operations performed when the message is received. 82
Figure 3.17 Finger table visibility interval. 84
Figure 3.18 Average lookup time (milliseconds) in static, "gentle" dynamic and "abrupt" dynamic environment. 94
Figure 3.19 Average RAM usage percentage comparison in static, "gentle" dynamic and "abrupt" dynamic environment. 95
Figure 3.20 Average CPU time usage percentage in static, "gentle" dynamic and "abrupt" dynamic environment. 96
Figure 3.21 Average dropped query percentage comparison in static, "gentle" dynamic and "abrupt" dynamic environment. 97
Figure 3.22 RAM usage percentage in a rendezvous peer (sampled each 5 seconds) in a "gentle" dynamic environment. 100
Figure 4.1 A torus 109
Figure 4.2 The values of the mean value of θ assumed when a node approaches to the border of the simulation area 111
Figure 4.3 Movements of five nodes following the column model 112
Figure 4.4 Movements of nodes following the nomadic model 113
Figure 4.5 Movements of nodes following the pursue model 114
Figure 4.6 Movements of nodes following the RPGM model 115
Figure 4.7 Movements of nodes following the city model 116

Figure 4.8 A graphical representation of the simulation area of the Manhattan model 116
Figure 4.9 An example of the simulation area of the obstacle model 117
Figure 4.10 The virtual track model simulation area, the big circles are the switch stations connected through the virtual tracks, the black spots are nodes in groups, the grey spots are the single nodes. 118
Figure 4.11 The MOMOSE flow diagram 120
Figure 4.12 The simulation and mobility model configuration windows 122
Figure 4.13 The simulation window 123
Figure 4.14 The OpenGL player 124
Figure 4.15 A comparison of the average execution time of the Java and the C++ engines with respect to the number of nodes (on the left) and with respect to the simulation time (on the right, in a logarithmic scale). 128
Figure 4.16 A comparison of *CanuMoboSim*, *Sinalgo*, and MOMOSE with respect to the simulation execution time (7200 simulated seconds and increasing number of nodes) 130
Figure 4.17 A comparison of *CanuMoboSim*, *Sinalgo*, and MOMOSE with respect to the simulation execution time (increasing simulated time and 200 nodes) 131
Figure 4.18 Experimental results on the topology change rate of two different mobility models 133
Figure 5.1 The Bluetooth architecture. 136
Figure 5.2 Bluetooth piconets with parked nodes (b) and without (a) 137
Figure 5.3 Bluetooth packet structure. 139
Figure 5.4 A Bluetooth scatternet. 142
Figure 5.5 The BTCP protocol execution sequence. 145
Figure 5.6 The Bluetrees protocol execution sequence. 147
Figure 5.7 The distributed Bluetrees protocol execution sequence. 149
Figure 5.8 The BlueStars protocol execution sequence. 152
Figure 5.9 The Yao construction. 153
Figure 5.10 The Florence's historic center scenario used while evaluating the Blue Pleiades protocol 155
Figure 5.11 Number of connected components of the BT topology for some values of c, as the number of nodes increases for four different values of transmission range. 157
Figure 5.12 Percentage of nodes included into the biggest connected component for some values of c, as the number of nodes increases for four different values of transmission range. 158

Figure 5.13 The tessellated unit square with $k = \left\lceil \frac{\sqrt{5}}{r(n)} \right\rceil$. 161
Figure 5.14 The 95% confidence intervals of the minimum range of the visibility graph 168
Figure 5.15 Comparison of the average maximum degree of $BT(r(n), 3)$, $BT(r(n), 2, 1)$, and of the visibility graph with range $r(n)$ 170
Figure 5.16 Comparison of the average diameter of $BT(r(n), 3)$, $BT(r(n), 2, 1)$, and the visibility graph with range $r(n)$ 171

BIBLIOGRAPHY

[1] V. Paxson. End-to-end routing behavior in the internet. *IEEE/ACM Trans. Netw.*, vol. 5(5):pp. 601–615, 1997. (Cited on page 3.)

[2] R. Govindan and A. Reddy. An analysis of internet inter-domain topology and route stability. In *INFOCOM '97: Proceedings of the INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, p. 850. IEEE Computer Society, Washington, DC, USA, 1997. (Cited on page 3.)

[3] R. V. Oliveira, B. Zhang, and L. Zhang. Observing the evolution of internet as topology. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 313–324. ACM, New York, NY, USA, 2007. (Cited on page 3.)

[4] Y. Ke, L. Deng, W. Ng, and D.-L. Lee. Web dynamics and their ramifications for the development of web search engines. *Computer Networks*, vol. 50(10):pp. 1430 – 1447, 2006. I. Web Dynamics. (Cited on page 4.)

[5] S. B. Handurukande, A.-M. Kermarrec, F. L. Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. In *EuroSys*, pp. 359–371. 2006. (Cited on page 4.)

[6] K. Zhang, N. Antonopoulos, and Z. Mahmood. A review of incentive mechanisms in peer-to-peer systems. In *Proceedings of The First International Conference on Advances in P2P Systems, AP2PS2009*, pp. 45–50. 2009. (Cited on page 4.)

[7] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, vol. 7:pp. 72–93, 2005. (Cited on page 11.)

[8] A. laszlo Barabasi, R. Albert, and H. Jeong. Scale-free characteristics of random networks: The topology of the world-wide web, 2000. (Cited on page 15.)

[9] L. Li, D. Alderson, R. Tanaka, J. C. Doyle, and W. Willinger. Towards a theory of scale-free graphs: Definition, properties, and implications (extended version), Oct 2005. URL `http://arxiv.org/abs/cond-mat/0501169`. (Cited on page 15.)

[10] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, vol. 2009:pp. 46–??, 2001. (Cited on page 17.)

[11] J. F. Buford, H. Yu, and E. K. Lua. *P2P Networking and Applications*. Morgan Kaufmann, 2008. (Cited on page 20.)

[12] R. Rodrigues and C. Blake. When multi-hop peer-to-peer lookup matters. In *IPTPS, Peer-to-Peer Systems III, Third International Workshop*, pp. 112–122. 2004. (Cited on pages 20, 21, and 175.)

[13] L. Garces-Erice, E. Biersack, P. A. Felber, K. W. Ross, and G. Urvoy-Keller. Hierarchical peer-to-peer systems. In *in: Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par*, pp. 643–657. 2003. (Cited on page 21.)

[14] E. Meshkova, J. Riihijärvi, M. Petrova, and P. Mähönen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Comput. Netw.*, vol. 52(11):pp. 2097–2128, 2008. (Cited on page 22.)

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA*, pp. 311–320. 1997. (Cited on pages 23, 24, and 27.)

[16] A. W. Richa and C. Scheideler. Overlay networks for peer-to-peer networks. In T. Gonzales, editor, *Handbook of Approximation Algorithms and Metaheuristics*. 2005. (Cited on page 24.)

[17] S.Ratnasamy, P.Francis, M.Handley, R.Karp, and S.Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 161–172. 2001. (Cited on pages 27, 45, and 49.)

[18] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pp. 329–350. 2001. (Cited on pages 32 and 34.)

[19] P. Druschel and A. I. T. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS*, pp. 75–80. 2001. (Cited on page 32.)

[20] A. I. T. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, pp. 188–201. 2001. (Cited on page 32.)

[21] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, pp. 53–65. 2002. (Cited on pages 34, 35, and 38.)

[22] I. Gupta, K. P. Birman, P. Linga, A. J. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *IPTPS*, pp. 160–169. 2003. (Cited on page 38.)

[23] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the 29th annual ACM symposium on Theory of computing*, pp. 654–663. 1997. (Cited on pages 38 and 60.)

[24] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Tech. rep., Cornell University, Ithaca, NY, USA, 1998. (Cited on page 40.)

[25] T. Zahn and J. Schiller. MADPastry: A DHT Substrate for Practicably Sized MANETs. In *Proc. of 5th Workshop on Applications and Services in Wireless Networks (ASWN2005)*. Paris, France, June 2005. (Cited on page 41.)

[26] E. M. Belding-Royer and C. E. Perkins. Multicast operation of the ad-hoc on-demand distance vector routing protocol. In *MOBICOM*, pp. 207–218. 1999. (Cited on page 42.)

[27] R. Winter, T. Zahn, and J. H. Schiller. Random landmarking in mobile, topology-aware peer-to-peer networks. In *FTDCS*, pp. 319–324. 2004. (Cited on pages 42 and 43.)

[28] C. Nocentini, P. Crescenzi, and L. Lanzi. Performance evaluation of a chord-based jxta implementation. In *Proceedings of the 1st International Conference on Advances in P2P Systems AP2PS2009*, pp. 7–12. 2009. (Cited on page 47.)

[29] Aeolus: Algorithmic principles for building efficient overlay computers official web site. http://aeolus.ceid.upatras.gr/. (Cited on page 47.)

[30] B. Traversat, M. Abdelaziz, and E. Pouyoul. Project jxta: A loosely-consistent dht rendezvous walker. http://research.sun.com/spotlight/misc/jxta-dht.pdf, 2003. (Cited on pages 48 and 67.)

[31] N. Jiang, C. Schmidt, V. Matossian, and M. Parashar. Enabling applications in sensor-based pervasive environments. In *Proceedings of the 1st Workshop on Broadband Advanced Sensor Networks (BaseNets 2004)*. 2004. (Cited on page 48.)

[32] Meteor. jxta-meteor official web site. https://jxta-meteor.dev.java.net/. (Cited on page 48.)

[33] D. Kato. Gisp: Global information sharing protocol "a distributed index for peer-to-peer systems". In *Proceedings of the 2nd International Conference on Peer-to-Peer Computing (P2P'02)*, p. 65. 2002. (Cited on page 49.)

[34] N. Theodoloz. *DHT-based Routing and Discovery in JXTA*. Master's thesis, School of Computer and Communication Sciences Swedish Institute of Computer Science, 2004. (Cited on page 49.)

[35] E. Halepovic. *Performance Evaluation and Benchmarking of the JXTA Peer-To-Peer Platform*. Master's thesis, University of Saskatchewan, 2004. (Cited on page 49.)

[36] E. Halepovic and R. Deters. The costs of using jxta. In *Third International Conference on Peer-to-Peer Computing (P2P'03)*, p. 160. 2003. (Cited on page 49.)

[37] E. Halepovic, R. Deters, and B. Traversat. Performance evaluation of jxta rendezvous. In *LNCS, vol. 3291, On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pp. 1125–1142. 2004. (Cited on pages 49 and 92.)

[38] JxtaBenchmarking. jxta-benchmarking official web site. https://jxta-benchmarking.dev.java.net/. (Cited on page 49.)

[39] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 149–160. 2001. (Cited on pages 60 and 61.)

[40] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE/ACM Transaction on networking*, vol. 11(1):pp. 17–32, 2003. (Cited on pages 60 and 61.)

[41] D. Brookshier, D. Govoni, N. Krishnan, and J. C. Soto. *JXTA: Java P2P Programming*. Sams Publishing, 2002. (Cited on page 67.)

[42] J. D. Gradecki. *Mastering JXTA: Building Java Peer-to-Peer Applications*. Wiley, 2002. (Cited on page 67.)

[43] S. Oaks, B. Traversat, and L. Gong. *JXTA in a nutshell*. O'Reilly, 2002. (Cited on page 67.)

[44] B. Wilson. *JXTA*. New Riders Publishing, 1 ed., 2002. (Cited on page 67.)

[45] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.-C. Hugly, E. Pouyoul, and B. Yeager. Project jxta 2.0 super-peer virtual network. http://research.sun.com/spotlight/misc/jxta.pdf, 2003. (Cited on page 67.)

[46] C. Nocentini. Jxtach official web site. http://www.dsi.unifi.it/ nocentin/jxtach/. (Cited on page 102.)

[47] S. Boschi, M. Di Ianni, P. Crescenzi, G. Rossi, and P. Vocca. Momose: a mobility model simulation environment for mobile wireless ad-hoc networks. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pp. 1–10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2008. (Cited on pages 105 and 127.)

[48] S. Boschi, P. Crescenzi, M. D. Ianni, C. Nocentini, G. Rossi, and P. Vocca. Momose: A mobility model simulation environment for mobile wireless ad-hoc networks. unpublished. (Cited on page 105.)

[49] J. Capka and R. Boutaba. A mobility management tool-the realistic mobility model. In *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005), IEEE International Conference on*, vol. 2, pp. 242–246 Vol. 2. Aug. 2005. (Cited on page 106.)

[50] W. Feller. *An Introduction to Probability Theory and its Applications, Volume 1*. Wiley, 1968. (Cited on page 106.)

[51] B. D. Hughes. *Random walks and random environments*. Oxford University Press, 1995. (Cited on page 106.)

[52] S. Pólya. Über eine aufgabe der wahrscheinlichkeitstheorie betreffend die irrfahrt im strassennetz. *Mathematische Annalen*, vol. 84(1):pp. 149–160, 1921. (Cited on page 106.)

[53] P. Révész. *Random walk in random and non-random environments*. World Scientific, 1990. (Cited on page 106.)

[54] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 1996. (Cited on page 106.)

[55] C. Bettstetter. Mobility modeling in wireless networks: categorization, smooth movement, and border effects. *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5(3):pp. 55–66, 2001. (Cited on page 106.)

[56] E. Royer, P. Melliar-Smith, and L. Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Communications, 2001. ICC*

2001. IEEE International Conference on, pp. 857–861 vol.3. 2001. (Cited on pages 106 and 108.)

[57] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications & Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, vol. 2:pp. 483–502, 2002. (Cited on pages 106, 108, 109, 111, 112, 113, and 114.)

[58] X. Hong, M. Gerla, G. Pei, and C. chuan Chiang. A group mobility model for ad hoc wireless networks. In *Proc. ACM Intern. Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems*, pp. 53–60. 1999. (Cited on pages 106, 111, 114, and 121.)

[59] M. Sanchez and P. Manzoni. A java-based ad hoc networks simulator. In *SCS Western Multiconference Web-based Simulation Track*. San Francisco, January 1999. (Cited on pages 106, 112, and 113.)

[60] A. P. Jardosh, E. M. Belding-Royer, K. C. Almeroth, and S. Suri. Real-world environment models for mobile network evaluation. *IEEE Journal on Selected Areas in Communications*, vol. 23(3):pp. 622–632, 2005. (Cited on page 106.)

[61] B. Zhou, K. Xu, and M. Gerla. Group and swarm mobility models for ad hoc network scenarios using virtual tracks. In *Military Communications Conference, 2004. MILCOM 2004. IEEE*, vol. 1, pp. 289–294 Vol. 1. Oct.-3 Nov. 2004. (Cited on page 106.)

[62] C.-H. Chen, H.-T. Wu, and K.-W. Ke. Flexible mobility models towards uniform nodal spatial distribution and adjustable average speed. In *Vehicular Technology Conference, 2005. VTC-2005-Fall. 2005 IEEE 62nd*, vol. 4, pp. 2292–2296. Sept., 2005. (Cited on page 106.)

[63] S. Lim, C. Yu, and C. Das. Clustered mobility model for scale-free wireless networks. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pp. 231–238. Nov. 2006. (Cited on page 106.)

[64] M. Musolesi, S. Hailes, and C. Mascolo. An ad hoc mobility model founded on social network theory. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pp. 20–24. ACM, New York, NY, USA, 2004. (Cited on pages 106 and 133.)

[65] M. Abdelhafez, G. F. Riley, R. G. Cole, and N. Phamdo. Modeling and simulations of tcp manet worms. In *Proc. 21st International Workshop on Principles of Advanced and Distributed Simulation*, pp. 123–130. 2007. (Cited on page 106.)

[66] F. Bai and A. Helmy. *A survey of mobility models*, chap. 1. Springer, 2006. (Cited on page 107.)

[67] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. G. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *MOBICOM*, pp. 85–97. 1998. (Cited on page 108.)

[68] Z. Haas. A new routing protocol for the reconfigurable wireless networks. In *Universal Personal Communications Record, 1997. Conference Record., 1997 IEEE 6th International Conference on*, vol. 2, pp. 562–566 vol.2. Oct 1997. (Cited on page 109.)

[69] B. Liang and Z. J. Haas. Predictive distance-based mobility management for pcs networks. In *INFOCOM*, pp. 1377–1384. 1999. (Cited on page 110.)

[70] V. Tolety and V. Tolety. Load reduction in ad hoc networks using mobile servers. Master's thesis at Colorado School of Mines, 1999. (Cited on page 110.)

[71] F. Bai, N. Sadagopan, and A. Helmy. Importan> a framework to systematically analyze theimpact of mobility on performance of routing protocols for adhoc networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 2. March-3 April 2003. (Cited on page 115.)

[72] C.-H. Chen, H.-T. Wu, and K.-W. Ke. General ripple mobility model: A novel mobility model of uniform spatial distribution and diverse average speed. *IEICE Transactions on Communications*, vol. 91-B(7):pp. 2224–2233, 2008. (Cited on page 119.)

[73] K. Fall and K. Varadhan. *The ns Manual*, 2007. (Cited on page 120.)

[74] E. Hyytiä, H. Koskinen, P. Lassila, A. Penttinen, J. Roszik, and J. Virtamo. Random waypoint model in wireless networks. Networks and Algorithms: complexity in Physics and Computer Science, June 2005. (Cited on page 120.)

[75] G. Lin, N. G, and R. Rajaraman. Mobility models for ad hoc network simulation. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 1, pp. 454–463. March 2004. (Cited on page 120.)

[76] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The official guide to learning OpenGL*. Addison Wesley, 2003. (Cited on page 120.)

[77] D. Shukla. Mobility models in ad-hoc networks. KReSIT-IIT, November 2001. (Cited on pages 121 and 122.)

[78] C. Bettstetter. Smooth is better than sharp: a random mobility model for simulation of wireless networks. In *MSWIM '01: Proceedings of the 4th ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pp. 19–27. ACM, New York, NY, USA, 2001. (Cited on page 121.)

[79] J. Tian, J. Hahner, C. Becker, I. Stepanov, and K. Rothermel. Graph-based mobility model for mobile ad hoc network simulation. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pp. 337–344. April 2002. (Cited on page 125.)

[80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pp. 124–133. ACM, New York, NY, USA, 1980. (Cited on page 127.)

[81] The network simulator ns-2. http://www.isi.edu/nsnam/ns/. (Cited on page 129.)

[82] GloMoSim. Global mobile information systems simulation library. http://pcl.cs.ucla.edu/projects/glomosim/. (Cited on page 129.)

[83] GTNetS. http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/. (Cited on page 129.)

[84] Omnet++. http://www.omnetpp.org/index.php. (Cited on page 129.)

[85] Mobility framework for omnet++. http://mobility-fw.sourceforge.net/. (Cited on page 129.)

[86] I. Stepanov, J. Hahner, C. Becker, J. Tian, and K. Rothermel. A meta-model and framework for user mobility in mobile networks. In *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pp. 231–238. Sept.-1 Oct. 2003. (Cited on page 129.)

[87] Sinalgo - simulator for network algorithms. http://dcg.ethz.ch/projects/sinalgo/. (Cited on page 129.)

[88] I. O. for Standardization. Intelligent transport systems - geographic data files (gdf) - overall data specification. ISO 14825:2004, 2004. (Cited on page 130.)

[89] P. Bergamo and G. Mazzini. Localization in sensor networks with fading and mobility. In *Proc. 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, pp. 750–754. 2002. (Cited on page 132.)

[90] U. Brandes and T. Erlebach. *Network Analysis: Methodological Foundations (Lecture Notes in Computer Science)*, *Lecture Notes in Computer Science*, vol. 3418. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (Cited on page 132.)

[91] M. Musolesi and C. Mascolo. Designing mobility models based on social network theory. *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 11(3):pp. 59–70, 2007. (Cited on page 133.)

[92] S. P. Borgatti and M. G. Everett. Ecological and perfect colorings. *Social Networks*, vol. 16(1):pp. 43 – 55, 1994. (Cited on page 133.)

[93] S. Xu, K. L. Blackmore, and H. M. Jones. An analysis framework for mobility metrics in mobile ad hoc networks. *EURASIP J. Wirel. Commun. Netw.*, vol. 2007(1):pp. 26–26, 2007. (Cited on page 133.)

[94] X. Pérez-Costa, C. Bettstetter, and H. Hartenstein. Toward a mobility metric for comparable & reproducible results in ad hoc networks research. *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 7(4):pp. 58–60, 2003. (Cited on page 133.)

[95] Momose. http://sourceforge.net/projects/momose/. (Cited on page 134.)

[96] P. Crescenzi, C. Nocentini, A. Pietracaprina, G. Pucci, and C. Sandri. On the connectivity of bluetooth-based ad hoc networks. In *Euro-Par*, pp. 960–969. 2007. (Cited on page 135.)

[97] P. Crescenzi, C. Nocentini, A. Pietracaprina, and G. Pucci. On the connectivity of bluetooth-based ad hoc networks. *Concurrency and Computation: Practice and Experience*, vol. 21(7):pp. 875–887, 2009. (Cited on page 135.)

[98] R. Whitaker, L. Hodge, and I. Chlamtac. Bluetooth scatternet formation: a survey. *Ad Hoc Networks*, vol. 3:pp. 403–450, 2005. (Cited on pages 135 and 141.)

[99] B. S. I. Group. http://www.bluetooth.com. (Cited on pages 135 and 141.)

[100] I. Stojmenovic and N. Zaguia. Bluetooth scatternet formation in ad hoc wireless networks. In J. Misic and V. Misic, editors, *Performance Modeling and Analysis of Bluetooth Networks*, pp. 147–171. Auerbach Publications, 2006. (Cited on pages 135, 136, and 159.)

[101] S. Basagni, R. Bruno, G. Mambrini, and C. Petrioli. Comparative performance evaluation of scatternet formation protocols for networks of Bluetooth devices. *Wireless Networks*, vol. 10(2):pp. 197–213, 2004. (Cited on pages 135 and 159.)

[102] R. Kettimuthu and S. Muthukrishnan. Is Bluetooth suitable for large-scale sensor networks? In *Proc. of the 2005 Intl. Conf. on Wireless Networks*, pp. 448–454. 2005. (Cited on page 135.)

[103] P. Chandra, D. M. Dobkin, A. Bensky, R. Olexa, D. Lide, and F. Dowla. *Wireless Networking: Know It All*. Newnes, 2007. (Cited on page 136.)

[104] T. Salonidis, P. Bhagwat, L. Tassiulas, and R. O. LaMaire. Distributed topology construction of bluetooth personal area networks. In *INFOCOM*, pp. 1577–1586. 2001. (Cited on page 143.)

[105] T. Salonidis, P. Bhagwat, L. Tassiulas, and R. O. LaMaire. Proximity awareness and ad hoc network establishment in bluetooth. Tech. rep., University of Maryland, 2001. (Cited on page 144.)

[106] G. Zaruba, S. Basagni, and I. Chlamtac. Bluetrees-scatternet formation to enable bluetooth-based ad hoc networks. In *Communications, 2001. ICC 2001. IEEE International Conference on*, vol. 1, pp. 273–277 vol.1. Jun 2001. (Cited on page 146.)

[107] C. Petrioli, S. Basagni, and I. Chlamtac. Configuring bluestars: Multi-hop scatternet formation for bluetooth networks. *IEEE Trans. Computers*, vol. 52(6):pp. 779–790, 2003. (Cited on pages 148 and 151.)

[108] X.-Y. Li, I. Stojmenovic, and Y. Wang. Partial delaunay triangulation and degree limited localized bluetooth scatternet formation. *IEEE Trans. Parallel Distrib. Syst.*, vol. 15(4):pp. 350–361, 2004. (Cited on page 151.)

[109] A. C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. Tech. rep., Stanford University, Stanford, CA, USA, 1977. (Cited on page 151.)

[110] D. P. Dubhashi, O. Häggström, G. Mambrini, A. Panconesi, and C. Petrioli. Blue pleiades, a new solution for device discovery and scatternet formation in multi-hop bluetooth networks. *Wireless Networks*, vol. 13(1):pp. 107–125, 2007. (Cited on page 154.)

[111] F. Ferraguto, G. Mambrini, A. Panconesi, and C. Petrioli. A new approach to device discovery and scatternet formation in Bluetooth networks. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, pp. 221–228. 2004. (Cited on page 159.)

[112] M. J. P. Appel and R. P. Russo. The connectivity of a graph on uniform points on $[0,1]^d$. *Statistics & Probability Letters*, vol. 60(4):pp. 351–357, 2002. (Cited on pages 159 and 160.)

[113] P. Gupta and P. R. Kumar. *Stochastic Analysis, Control, Optimization and Applications: A Volume in Honor of W. H. Fleming*, chap. Critical power for asymptotic connectivity in wireless networks, pp. 547–566. Birkhäuser, 1999. (Cited on page 159.)

[114] M. D. Penrose. *Random Geometric Graphs*. Oxford University Press, New York, USA, 2003. (Cited on page 159.)

[115] D. Dubhashi, C. Johansson, O. Häggstrom, A. Panconesi, and M. Sozio. Irrigating ad hoc networks in constant time. In *Proc. of the 17th ACM Symp. on Parallel Algorithms and Architectures*, pp. 106–115. Jul. 2005. (Cited on pages 159, 164, and 167.)

[116] A. Panconesi and J. Radhakrishnan. Expansion properties of (secure) wireless networks. In *Proc. of the 16th ACM Symp. on Parallel Algorithms and Architectures*, pp. 281–285. 2004. (Cited on page 159.)

[117] R. Ellis, X. Jia, and C. Yan. On random points in the unit disk. *Random Structures and Algorithms*, vol. 29(1):pp. 14–25, 2005. (Cited on page 160.)

[118] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, vol. 38:pp. 393–422, 2002. (Cited on page 160.)

[119] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, vol. 33(6):pp. 305–308, Feb. 1990. (Cited on pages 162 and 164.)

[120] T. Harris. *The Theory of Branching Processes*. Springer, Berlin, Germany, 1963. (Cited on page 164.)

[121] J. G. Siek, L. Lee, and A. Lumsdaine. *Boost Graph Library, The: User Guide and Reference Manual*. Addison Wesley Professional, Reading MA, Dec. 2001. (Cited on page 168.)

[122] A. Boukerche. *Algorithms and Protocols for Wireless sensor networks*. Wiley, 2009.

[123] A. Boukerche. *Algorithms and Protocols for Wireless and Mobile Ad Hoc Networks*. Wiley, 2009.

[124] S. Basagni, M.Conti, S. Giordano, and I. Stojmenovic. *Mobile Ad Hoc Networking*. Wiley, 2004.

[125] JXTA. Jxta project official web site. http://jxta.dev.java.net.

[126] Chord. Chord official web site. http://pdos.csail.mit.edu/chord/.

[127] R. Bagrodia, M. Takai, Y. an Chen, X. Zeng, and J. Martin. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, vol. 31:pp. 77–85, 1998.

[128] F. Bai, N. Sadagopan, and A. Helmy. User manual for important mobility tool generators in ns-2 simulator. University of Southern California, February 2004.

[129] D. Dubhashi, O. Häggström, G. Mambrini, A. Panconesi, and C. Petrioli. Blue pleiades, a new solution for device discovery and scatternet formation in multi-hop bluetooth networks. *Wireless Networks*, vol. 13(1):pp. 107–125, 2007.

[130] P. Johansson, M. Kazantzidis, R. Kapoor, and M. Gerla. Bluetooth: an enabler for personal area networking. *Network, IEEE*, vol. 15(5):pp. 28–37, Sep/Oct 2001.

[131] J. Yoon, M. Liu, and B. Noble. Random waypoint considered harmful. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 2, pp. 1312–1321 vol.2. March-3 April 2003.

[132] Mobigen. http://www.soe.ucsc.edu/˜mmosko/mobigen/.

[133] The rice university monarch project. http://www.monarch.cs.rice.edu/.

[134] K. Murota and K. Hirade. Gmsk modulation for digital mobile radio telephony. *Communications, IEEE Transactions on*, vol. 29(7):pp. 1044–1050, Jul 1981.

[135] Z. Wang, R. J. Thomas, and Z. J. Haas. Bluenet - a new scatternet formation scheme. In *HICSS*, p. 61. 2002.