Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Geodupa - A Geography-driven Partial Membership Algorithm

Pedro Miguel Siopa da Silva
(28088)

Lisboa

Dezembro 2011

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Geodupa - A Geography-driven Partial Membership Algorithm

Pedro Miguel Siopa da Silva
(28088)

Orientador: Prof. Doutor Sérgio Marco Duarte

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Lisboa

Dezembro 2011

Geodupa - A Geography-driven Partial Membership Algorithm

Agradecimentos

Este documento representa o final de um longo percurso que se demonstrou, a vários níveis, bastante exigente. A sua conclusão não teria sido de todo possível sem o acompanhamento e intervenção de várias pessoas a quem faço questão de dirigir o meu apreço.

Em primeiro lugar, agradeço ao meu orientador, o Professor Doutor Sérgio Marco Duarte, pela dedicação e compreensão demonstradas ao longo deste período. Realço também a sua disponibilidade e suporte que em grande parte contribuíram para a realização desta dissertação.

Quero também agradecer aos colegas e amigos que acompanharam o desenrolar desta dissertação. Companheiros de muitas horas que de alguma forma contribuíram para que o caminho percorrido fosse menos sinuoso.

Lembro ainda todos aqueles que de perto viveram esta caminhada. Pais, irmão, namorada. O apoio e motivação que incansavelmente me prestaram foram essenciais na manutenção do rumo, sobretudo nas alturas mais difíceis. Por isto, e por muito mais, aqui lhes deixo o meu mais profundo e sentido agradecimento, através da dedicatória deste trabalho.

Obrigado!

Abstract

The popularity boost of mobile devices, as well as their technological advances, are fostering the development of new application paradigms. One of which, designated as *Participatory Sensing*, is embedded with a strong community philosophy, in the sense that it relies in users sharing and contributing with data. By gathering, processing and sharing contextual data, new and interesting applications are possible, such as the monitoring of vehicle traffic or road conservation. To enable these applications, it is important to have an underlying communication infrastructure that allows users to exchange information efficiently.

Users of *Participatory Sensing* applications deal, most frequently, with information related to their close physical surroundings. Based on this premise, this dissertation presents a decentralized membership substrate that restrains node visibility to geographical neighborhoods as a way to improve communication performance. To that end, the proposed algorithm divides the user network into two hierarchical levels. The higher-level is managed by an existing *one-hop DHT* and its participants are organized to exploit the partitioning of the physical space. The lower-level is composed by the groups of nodes associated to each region. An experimental evaluation has revealed that it is capable of achieving lower communication costs when compared to a full-membership solution.

Keywords: *Participatory Sensing*, *DHT*, peer-to-peer network, membership algorithm

Resumo

O aumento da popularidade dos dispositivos móveis, assim como os seus avanços tecnológicos, estão a alimentar o desenvolvimento de novos paradigmas aplicacionais. Um dos quais, designado de *Sensoriamento Participado*, está embutido de uma forte filosofia comunitária, no sentido de que se baseia na contribuição e partilha de dados por parte dos utilizadores. Através da recolha, processamento e partilha de dados contextuais, novos e interessantes tipos de aplicações são possíveis, tais como, monitorização do tráfego rodoviário ou conservação das estradas. De forma a possibilitar estas aplicações, é importante que exista uma infra-estrutura de comunicação que permita aos utilizadores trocar informação de forma eficiente.

Os utilizadores de aplicações inspiradas em *Sensoriamento Participado* lidam, com grande frequência, com informação relacionada com os arredores da sua localização física. Tendo em conta esta premissa, esta dissertação apresenta um substrato de filiação descentralizado que restringe a visibilidade dos nós a vizinhanças geográficas com o intuito de melhorar a performances das comunicações. Nesse sentido, o algoritmo proposto divide a rede de utilizadores em dois níveis hierárquicos. O nível superior é gerido por uma *one-hop DHT* existente e os seus participantes estão organizados de forma a explorar o particionamento do espaço físico. O nível inferior é composto pelos grupos de nós associados a cada região. Uma avaliação experimental revelou que este é capaz de atingir custos de comunicação mais baixos quando comparado com uma solução de visibilidade total.

Palavras-chave: *Sensoriamento Participado*, *DHT*, rede *peer-to-peer*, algoritmo de filiação

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation and Context

The recent technological evolution of ubiquitous devices - PDAs, laptops, smartphones or even vehicles - has undoubtedly promoted their growing adoption by the general population. Technical advances, such as longer autonomy, greater processing power or reduced communication costs, together with the ability to gather contextual sensorial information, ranging from image capture to geographical locating (GPS), have contributed to their popularity boost.

These ongoing improvements have encouraged an emerging and promising research area - *Participatory Sensing* [2, 3, 22]. Its leading concept is the collection, processing and sharing of contextual sensorial data through the deployment of wide-area sensor networks composed by personal mobile devices. The broad and continuous mobility of the users foresees a model that provides real-time monitoring of the various aspects of the physical world, particularly in densely populated areas.

Several interesting examples of applications that rely on this model have already been proposed and are often designed to monitor sensorial data in cities for community benefit [17, 19]. The monitoring of air quality [18] or of traffic and road conditions [10, 16, 5] are just a few specific aspects of its use. An alternative application domain has a more social connotation. For instance, a system to monitor and share the experience of cyclists through the deployment of sensors in bicycles [4] or even one to record, in detail, the daily routine of the user [14].

Intuitively, the deployment of such wide-area networks is dependent on an underlying communication infrastructure that spans its participants and enables data sharing. A potential solution is the adoption of a centralized infrastructure where the sensor nodes store and retrieve information by directly interacting with one or more "base stations". While benefiting on some aspects, such as simple software implementation or easy data indexation, this approach suffers from the usual drawbacks of centralized approaches: single point of failure and possible communication bottlenecks. Moreover, the costs of deploying and maintaining such an infrastructure make it nearly unfeasible in the absence of a notorious funding entity.

A purely decentralized peer-to-peer model thus comes to mind. It avoids any kind of centralized component and relegates the entire system's actions to the mobile sensors. This approach

avoids the usual centralized drawbacks and the inherent costs of deploying dense sensing infrastructures. However, even though it's technically possible to put this model into practice, there is a crucial problem: the amount of work required from each sensor. Despite the ongoing technical advances of ubiquitous devices, the current processing and energy capabilities are not yet satisfactory to render this solution realistic.

A different kind of approach might still be considered, one that tries to combine the advantages of centralized and purely decentralized architectures. Instead of the actual mobile devices, it's possible to consider fixed hardware as the infrastructure building blocks. The peer-to-peer network is then composed by applications running on the users' Personal Computers, which share information and act as servers to the mobile devices, thus shifting most of the work from the mobile to the fixed devices. On one hand, this approach solves the purely decentralized problem related to the mobile devices' limited resources. On the other hand, since the network hardware components are supplied by the users themselves, it preserves the *Participatory Sensing*'s community driven philosophy.

Having established a reasonable infrastructure model, the problem of inter-node visibility is yet to be addressed. This has to consider the strong geographical correlation of the *Participatory Sensing* application domain. For instance, a user in Lisbon is more likely to be interested in sensorial data from its close vicinity than, say, from Porto or Madrid. Thus, participating nodes need to have a more detailed knowledge of the peers located nearby. As distance grows, the importance is likely to decrease in proportion and the accuracy of node visibility grows to be progressively less relevant. This has to be taken into account when choosing the peer-to-peer architecture.

In the matter of peer-to-peer architectures there are two main approaches to consider: unstructured and structured overlays. An unstructured peer-to-peer overlay has an arbitrary network topology where nodes are randomly connected to each other. Although these networks are capable of accommodating large numbers of nodes, the lack of a well established organization leads to inefficient communications as nodes have no clear knowledge of the network arrangement. The process of discovering the address of other nodes is not deterministic and is usually performed by flooding the network, possibly causing its congestion. Obviously, the unstructured architecture scheme is not an appropriate candidate as it performs poorly regarding the need to have a detailed awareness of nodes in a geographical neighborhood.

Structured overlays [23, 21, 9, 1, 7, 8] are an alternative to unstructured overlays. Commonly known as Distributed Hash Tables (DHT) , these are mainly used as repositories of data and provide similar, while distributed, lookup operations as usually found in hash table data structures. For the purpose of inter-node coordination and overlay maintenance, each participant features an identifier, derived from some unique attribute (e.g., ip address), and keeps information regarding its peers. Most DHTs guarantee deterministic message exchange and routing table sizes that grow logarithmically with the network size, thus favoring the potential to accommodate large numbers of participants without compromising the correct behavior and efficiency of the system (scalability).

The monitoring of large physical areas is reliant on large numbers of participating nodes. Therefore applications based on the *Participatory Sensing* model need to carefully deal with the scalability aspect. Moreover, they also require some degree of inter-node organization in order to enable the participants to readily communicate with the relevant peers. For this reason, the adoption of a DHT as the foundation for the underlying node architecture seems a reasonable approach.

Although some DHTs handle the scalability problem quite well, there is an issue in adopting a DHT. It is related with the nodes' identifiers dispersal and their consequent lack of geographical correlation. There is no relationship between the identifier of a node and its coordinates in space. In other words, two nodes with close identifier values might not be physically close to each other. This poses a problem as every node should be able to differentiate others based on their distance to itself. The generation of identifiers should obey a proximity metric that mapped the coordinates of nodes in a way that physically close nodes are assigned close identifier values. Unfortunately, the more mainstream DHTs do not readily provide such mapping.

Most of the DHTs known so far are able to perform node lookups in no more than O($log\ N$) hops (with N the number of nodes in the network) by storing routing information of, at most, O($log\ N$) other nodes in the network. To perform a lookup, several nodes in sequence need to be contacted which might result in high latency. For this reason, a different approach might be considered, one that offers the ability to lookup nodes in a single hop. Commonly designated as "one-hop", these DHTs [7, 13] require the maintenance of full-membership information at each node. While granting less effort on performing lookups, this approach is also conditioned by a few shortcomings. Namely, limited scalability and high bandwidth consumption caused by the maintenance of full membership information at each node.

The goal of this dissertation is then design of a membership substrate able to act as the underlying peer-to-peer architecture for a network composed by the users' Personal Computers. To preserve the Participatory Sensing premise that a node needs a detailed view of its close geographical neighborhood, each node will feature a set of physical coordinates and full-membership will be applied to geographically delimited areas. The physical space is to be partitioned and a node that belongs to a specific partition will have complete knowledge of its partition colleagues, allowing low latency message exchange, in a single hop. Since nodes do not need to have the same level of knowledge when it comes to distant nodes, the full membership notion will be relaxed to farther partitions, thus reducing membership maintenance costs and increasing scalability.

1.2 Objective

The main endeavor is the design of a peer-to-peer network architecture to support applications based on the Participatory Sensing model, where each node has complete knowledge of others in a close geographical neighborhood and sparse awareness of nodes positioned farther away.

This dissertation thus presents a two-level, explicitly hierarchical network. The higher level is to be composed by an independent full-membership network of super-nodes. It spans the entire physical space and each super-node is responsible to manage a fraction of it. The management of the higher level is the responsibility of a pre-existing one-hop DHT: Catadupa [13].

Catadupa is a full-membership substrate designed to support peer-to-peer content-based routing. The choice of this one-hop DHT is motivated by some of its interesting properties, e.g. it constantly tries to balance the load throughout the nodes, taking bandwidth capacity into account. It also comprises fault-tolerance mechanisms and deterministically assures accurate membership information at each node.

The lower level of the proposed substrate is composed by sub-nodes who need to be aware of all the peers located in the same geographical neighborhood. These must also be aware of, at least one, super-node in their vicinity. This is motivated by the fact that, since super-nodes are dispersed throughout the physical space and are all aware of each other, having at least one in range allows sub-nodes, among others, to be able to communicate with farther peers.

1.3 Main Contributions

The main contributions expected from the elaboration of this thesis are:

- The design of a two-level geography-oriented membership substrate that provides its participants with full knowledge of their closely surrounding peers.
- An evaluation study, taking several relevant metrics into account, in which the impact of the configuration parameters is observed.

1.4 Document Structure

This document is composed by six chapters:

• **Chapter 1:** The current chapter. It establishes a common background and provides the motivation for this thesis.

• **Chapter 2:** Provides a general description of peer-to-peer architectures complemented with previous work on the area.

• **Chapter 3:** Presents the problem at hands and the challenges expected in the design of the proposed algorithm, also iterating through the strategies adopted to solve them.

• **Chapter 4:** Describes the process of implementing the solutions proposed in chapter 3, in a simulation environment.

• **Chapter 5:** Some metrics are defined and used to assess the algorithm's behavior under different configurations.

• **Chapter 6:** This chapter presents the final considerations and possible future improvements to the algorithm.

2. Related Work

This chapter presents a general overview of this dissertation's relevant state-of-the-art by presenting some of its previously accomplished studies.

2.1 Overlay Networks

A peer-to-peer overlay network can be perceived as a logical organization of nodes built on top of an existing network. Participating nodes exchange information through virtual links whose path might consist of several hops in the underlying network. These are commonly implemented to provide an abstraction to the developers of applications and to enlarge the set of services provided by the underlying network, namely, the ability to efficiently discover information.

There are two main approaches to consider in the matter of peer-to-peer overlay networks: unstructured and structured overlays. The former follows an approach where the network membership information at each node is minimized, thus requiring additional effort and resource consumption to perform queries. Conversely, structured overlays attempt to maintain proper membership information at each node in order to increase communication performance.

2.1.1 Unstructured overlays

An unstructured peer-to-peer overlay, like Gnutella or Kazaa, is a network that follows an ad-hoc philosophy regarding its organizational rules. As illustrated in figure 2.1, the network topology is usually arbitrary. This happens due to the random and constraint-free relationships established between nodes. The absence of a centralized entity transfers the efforts of inter-node discovery to the nodes themselves. Therefore, every time a node wants to send a message to another, it has to discover its network address without the possibility of exploring a predetermined path, relying only in the assistance of its peers.

Since inter-node links are arbitrary, the discovery process is not trivial and is usually performed by probing, whether by flooding the network, possibly causing its congestion, or by "Random Walks" where a random node is selected at each step of the path. The lack of hints

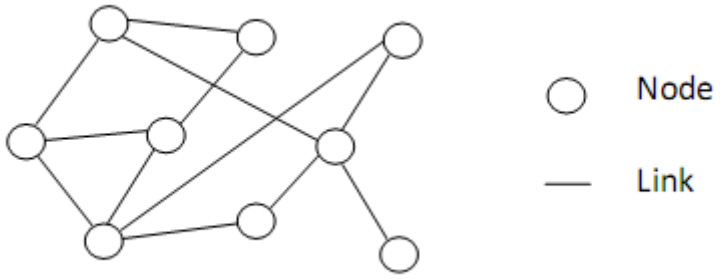


Figure 2.1: Arbitrary topology of unstructured overlays.

that would lead the probing in the right direction causes the blind querying of a large fraction of non useful nodes and render this solution nondeterministic in its ability to produce results.

However limiting the mentioned drawbacks might seem, unstructured networks also have some strong points. Namely, the absence of a rigid topology, leading to low assembly and maintenance efforts. Also, the arbitrary nature of the links between nodes causes the load to usually be distributed throughout the nodes and introduces path redundancy, endowing unstructured overlays with high degrees of resilience against node failures.

2.1.2 Structured overlays

Structured overlays are a sub-class of the overlay network concept and have been intensely targeted by researchers in the recent past. This interest was motivated by the growing need to share information and by the lack of capacity revealed by unstructured overlays (section 2.1.1) in fulfilling this necessity. This has led to the design of several structured overlays, each one with its own distinctive features regarding some key aspects. Namely, the **topological rules** defining the organization of nodes in the network and which peers they communicate with; **scalability**, the potential to accommodate large numbers of nodes without compromising the correct behavior and efficiency of the system; **fault-tolerance**, the ability to keep operating correctly in the presence of node failures; **performance**, conceptually perceived as the trade-off between the amount of work performed and the resources consumed in the process; **security**, which comprises the procedures carried out to defend the network and its resources against illicit access.

Structured overlays are often used as data repositories and provide similar, while distributed, lookup operations as found in hash table data structures. This led structured overlays to be intuitively designated as *Distributed Hash Tables* (DHTs). In this kind of overlay, node organization follows a set of common rules. Usually, each node is assigned an identifier obtained by applying a uniform hash function to some unique attribute (e.g., ip-address). To determine the node responsible for some data, the same hash function is applied to the data identifier. From this process results an identifier which can then be compared by some proximity metric to the identifiers of the nodes in order to find the appropriate one to store/get the data. This ability to map data into nodes allows queries to be deterministically routed to their correct destination.

As an example of a DHT network architecture, figure 2.2 illustrates the topology of Chord [23], in which nodes are organized in an identifier ring. The figure shows a Chord ring composed by three nodes - 0, 1 and 3 - and three pieces of data with the values 1, 2 and 6 as their respective keys. The nodes responsible for some data are called its *successors* and are the ones that actually store it. In this case, node 1 is the successor of key 1, node 3 is the successor of key 2 and node 0 is the successor of key 6.

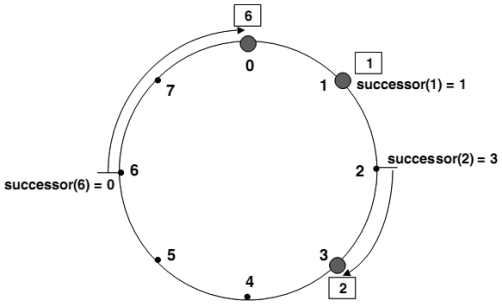


Figure 2.2: Topology of a structured overlay - Chord[23].

Unlike unstructured overlays, these invest most of their work in maintaining correct membership information at each node as a way to increase query performance. This scheme can follow several directions regarding optimization. For instance, on the subject of scalability, DHTs can be conceptually assembled into three groups. The **first** group [23, 21] directs its efforts into reducing the routing information kept at each node. By maintaining tables with logarithmical dimension (considering the total size of the network) they manage to achieve logarithmical lookups. Following the tendency to minimize the information kept at each node, the

second group [12, 11] attempts to keep constant routing table sizes and still allow communications to be performed with logarithmical complexity. Finally, the **third** group [7, 8, 9, 13] takes its optimization attempts in the opposite direction. That is, routing tables have their dimensions increased in the attempt to minimize query complexity and latency. However, the large amount of information maintained at each node reveals scalability issues as it is bound to increase with the size and dynamism of the network.

2.2 Significant Projects

In this section several projects are described as a means to provide an insight into the work previously done in the area of network overlays. These were selected due to their possible usefulness in solving the problem at hand or simply due to historical and background reasons.

2.2.1 Chord

Chord [23] is a fully decentralized peer-to-peer protocol that considers networks with high rates of membership changes. Communication costs, as well as the amount of information kept at the nodes, grow logarithmically with the number of nodes in the system. That is, each node stores routing information of about O(log N) nodes and takes O(log N) hops to perform lookups, rendering Chord able to support large systems, with reasonable performance measures.

Chord's structure is based on an identifier ring, where each node is assigned a key. Such assignment is performed by applying a hash function (typically SHA-1) to the node's ip-address. Data is then associated to chord nodes by applying the same hash function to the data identifier. From this process results a key, k, that is assigned to the node whose own key equals or follows k in the identifier ring. The chosen node is referred to as the *successor* of k.

In its membership information, each node maintains a table with at most m entries (with m being the identifiers' bit size), designated as the *finger table*. At node n, the first entry contains its own successor. The i^{th} entry stores information about the first node, n', that follows n by, at least, 2^{i-1} in the identifier space. In addition, a chord node also keeps a pointer to its immediate predecessor, allowing counter-clockwise navigation in the ring.

When node *n* needs to find the node responsible for a key *k*, it searches its finger table for a node whose identifier most precedes *k*. That node, *n'*, will have a clearer view of the identifier space in which *k* resides. Node *n* can now ask *n'* to do the same and return an even closer node to *k*. By applying this procedure repeatedly, *n* will eventually reach the successor of *k*.

To join the system, *n* initiates its *fingers* and its predecessor. Other nodes update their information accordingly and the successor of *n* is contacted to resolve the keys *n* will be responsible for. When several nodes join concurrently, it is hard to maintain every *finger table* updated. A "stabilization" protocol is thus introduced to help keep the successor pointers updated, which is enough to preserve the accuracy of lookups and, in time, to correct the finger tables.

To deal with failures, each node also keeps a list of its closest successors in the ring. When a node detects that its immediate successor failed, the next live node in the successors list is selected to replace it. The "stabilization" mechanism is then responsible for correcting finger table entries, as well as successors lists.

2.2.2 Pastry

Pastry[21] is a fully decentralized, fault tolerant, scalable, reliable and self-organizing peer-to-peer overlay. It adapts well to failures and is efficient and scalable in networks with up to 100,000 nodes. Unlike most DHTs, it tries to minimize communication latency by considering network locality and proximity metrics provided by the application level (e.g. number of hops).

Pastry's structure is based on an identifier ring. The identifier space ranges from 0 to $2^{128}-1$ and each node is assigned an identifier (*nodeID*) by applying a cryptographic hash function to its public key or ip-address.

Pastry nodes keep a *routing table*, a *neighborhood set* and a *leaf set*. The n^{th} row of the *routing table* contains nodes whose *n* first identifier digits equal the table owners'. The *neighborhood set* is used to maintain locality properties and contains information concerning close nodes. The *leaf set* is used in message routing and contains a set of nodes whose identifier range has the local *nodeID* as its mid-point.

A message is directly sent to its destination if its key is contained in the *leaf set*. Otherwise, the *routing table* is consulted to obtain the node whose identifier has the same prefix as the message's by, at least, one more digit. This way, the messages are always sent to a node closer

to the final destination, while traveling the minimum possible distance in the proximity space. The expected routing cost is O($log_{2^b}N$), with b a configuration parameter with typical value 4.

To join the overlay, a node X asks a known Pastry node - A - to send a request to the node with the closest identifier to X's - Z. All the nodes in the path from A to Z send their tables to Xso that it can initialize its own state tables and contact the nodes that need to be aware of its arrival. The total number of messages spent in a join is O($log_{2^b}N$).

When a node fails, the state tables need updating. To replace an entry in the *routing table*, another node in the same row is asked for its entry on the failed position. A failed node in the *leaf set* is replaced by contacting the node with the highest *nodeID* on the side of the failed node. Finally, to replace a node in the *neighborhood set*, other members of that set are contacted.

2.2.3 Self Correcting Broadcast in DHTs

In [6], two algorithms are presented to perform broadcasts in a *DKS* [1] network. Both provide coverage of all nodes in the network, without message duplicates, even when membership changes are frequent and routing information is outdated. The following paragraphs will focus on the *DKS* system and then present a brief summary of the algorithms.

DKS(N, k , f) [1] is an infrastructure for designing fully decentralized peer-to-peer applications. The parameters N (maximum number of nodes in the network), k (search arity) and f (fault-tolerance degree) characterize the overlay network. As an example, Chord [23] can be derived, to some extent, by *DKS* with $k = 2$.

Most DHTs rely on concurrent procedures to maintain routing information up to date - *active-correction*. This leads to unnecessary bandwidth consumption when lookups and insertions occur more frequently than membership changes. To minimize this drawback, *DKS* adopts a more passive approach - *correction-on-use* - by embedding information on lookup/insertion messages to allow detection and correction of outdated routing entries along their path. Intuitively, more useful traffic leads to more accurate routing tables.

When in the presence of frequent membership changes, other measures need to be considered. For instance, *correction-on-change*, which notifies all relevant nodes upon membership changes.

Like in Chord, both nodes and data are mapped to the identifier space using a hash function. Data is stored at its immediate successor on the identifier ring. In each lookup hop the identifier space is split into k equal parts, thus leading lookups to be resolved in no more than $log_k N$ hops.

Joins and leaves are similar as, in both cases, the node's successor needs to be contacted. Either to accommodate a new node or to process the exit of the departing one. Fault tolerance is achieved through replication and by maintaining, at each node, a list of its f+1 successors. This way, f nodes may fail without compromising the system's correct behavior.

Each node maintains a routing table containing (k-1)$log_k(N)$ entries and a pointer to its immediate predecessor in the identifier space. The table is divided in $log_k(N)$ levels, which, in turn, are divided into k equal intervals. Level one represents the whole identifier space. Each of the following levels represent only a k^{th} of the previous level. A representative node is assigned to each interval and it is the one contacted to resolve a key that falls in that interval's key range.

To broadcast a message using the first algorithm presented in [6], a node sends the message to the responsible nodes of every interval in its routing table. The iteration starts at the first level and, at each level, it is performed counter-clockwise. Each message contains the level and interval the sender is currently iterating through, and also a *limit*. These are used by the receiver to establish its own search range. The second algorithm can be perceived as an extension of the first. It additionally uses a *self-correction* technique in order to increase routing table correction.

2.2.4 Symphony

Symphony [12] is a DHT protocol that organizes nodes in a ring and provides each one with long distance links. It is stable, scalable, flexible and promises low latency lookups and cheap maintenance costs under network churn.

The overlay network is organized in a circular ring with key interval I = [0, 1]. Each node manages a sub-interval of I ranging from its own *id* to the *id* of its immediate predecessor and keeps two short links to its immediate neighbors. Unlike Chord [23] and others [21, 20, 25], there is no relationship between the number of links and the number of bits of the identifiers.

Each node also maintains $k \geqslant 1$ long distance links. Each targeting the node responsible for the point whose distance is x - random number belonging to I and drawn from a harmonic probability distribution function. Tests show that low latency routing in large networks is possible

with only four long distance links. Every node has a maximum of $2k$ incoming links. To draw x, a node needs to know N, the size of the network. Thus, an *Estimation Protocol* is executed to provide, at runtime, the knowledge of n to the nodes.

Lookups can be performed bi-directionally in the ring with average latency of $O(\frac{1}{k}log^2N)$. To do so, a node routes an incoming message to the incoming or outgoing link that minimizes the distance to the destination.

To join the network, a node chooses a random uniform number $x \in I$ as its *id* and contacts the node responsible for the sub-range that contains it. It obtains the size of the network from the *Estimation Protocol* and establishes its k long distance links. When a node leaves the network, its incoming and outgoing links are broken. Nodes that had outgoing links to the leaving node replace them with links to other nodes. On average, k incoming links need to be re-established. Since the establishment of each one of the k links takes $O(\frac{1}{k}log^2N)$ messages, both node joins and leaves have total average cost of $O(log^2N)$.

Nodes also maintain a *lookahead list* containing the identifiers of their neighbor's neighbors, allowing the reduction of the average lookup latency by half. Symphony tolerates failures of up to f nodes by keeping f additional links per node. Both unidirectional and bidirectional routing are possible and present an average lookup latency of $O(\frac{1}{k}log^2N)$. Symphony also provides run-time tuning of the number of links per node, enabling the regulation of the trade-off between the number of links and average lookup latency.

2.2.5 Structured Superpeers

This paper [15] describes a structured peer-to-peer system able to perform O(1) lookups. In fully distributed lookup systems, routing information is distributed uniformly among its peers, leading to high network latency as nodes have different processing and bandwidth capacities. This issue is addressed by exploring node heterogeneity, forming a hierarchical network with *superpeers* and choosing them based on their capabilities. This approach combines the advantages of centralized (lower lookup costs) and decentralized systems (scalability and reliability).

Nodes in this network are organized in a circular ring - the *outer ring*. Some of these nodes are selected to be *superpeers*, forming a smaller network - the *inner ring*. Superpeers maintain full knowledge of each other and each one is responsible for a fraction of the outer ring.

A node performs a lookup by contacting the *superpeer* responsible for its partition. If the query directs to a node in the same partition, the superpeer returns the result, if not, it forwards the request to the superpeer responsible for the partition containing the identifier. This scheme reveals constant lookup costs as, in the worst case, two nodes are contacted to resolve a lookup.

To join the network, a node contacts another already in the network in order to discover the superpeer responsible for its partition. After being contacted by the joining node, the superpeer updates its routing table with the new node. When a node from the outer-ring fails, the superpeer simply removes it from its table. If a super-peer fails, its partition might be assigned to others in the inner-ring or other node might be assigned to manage it. Failures are identified by other peers in the same ring, through the exchange of keep-alive messages.

Even though superpeers are selected taking their capabilities into account, they still have a limit. Therefore, when a node detects that its maximum capacity has been reached, it may share its load with another superpeer, or even with a node from its partition.

2.2.6 One-Hop Lookups

This project [7] presents a full-membership peer-to-peer lookup network. Other peer-to-peer structured overlays, try to maintain small routing tables at each node because they expect frequent membership changes. However, lookups have high latency as they require contacting several nodes in sequence. This system considers that maintaining full routing tables is viable, even in the presence of large networks with frequent membership changes.

Nodes in this overlay are arranged in an identifier ring and each one is assigned an uniformly distributed 128-bit identifier. The ring is divided into k equal adjacent intervals - ***slices***. Each *slice* is similarly divided into ***units***. Both these subdivisions have a leader: the mid-point successor of the range they cover. Each node has a predecessor and a successor, to whom they, periodically, send keep-alive messages. This approach has the disadvantage of possibly overloading the *slice* leaders. It's a drawback that could be minimized if *slice* leaders are chosen taking their bandwidth capacity into account.

In order to supply every node with full-membership, when a nodes detects a membership change it notifies its *slice* leader which, in turn, warns the other leaders about recent changes. The *slice* leaders aggregate received messages for a while and then dispatch a message to their

unit leaders. Finally, the *unit* leaders piggyback the notifications on their keep-alive messages to their successor and predecessor, in a way that the information flows from the *unit* leader to the boundaries of the *unit*, thus avoiding duplicates. This, together with an aggregation mechanism that minimizes the dispatch of small messages, leads to the efficient use of bandwidth.

2.2.7 Two-Hop Lookups

The peer-to-peer lookup protocol presented in [8] is able to route in two-hops. It is based on the structure of the one-hop scheme presented in [7] and summarized in section 2.2.6.

The one-hop scheme works well for systems as large as a few million nodes. However, for networks of larger sizes, the bandwidth requirements may become too large. Therefore, a scheme that scales to a larger network size is presented, one that lowers the bandwidth consumption by keeping only a fraction of the membership information at each node.

As in the one-hop scheme, nodes are organized in an identifier ring and each node is assigned an uniform 128-bit identifier. The network is also formed by ***slices***, ***units*** and their respective leaders. The particularity of this approach lies in the fact that every *slice* leader chooses l nodes from its partition for every other *slice* in the system (leading to k-1 sets).

The *slice* leader sends routing information about one group to exactly one other *slice* leader. This information is then disseminated to all members of that *slice* like in the one-hop scheme: from *slice* leaders to *unit* leaders and from these to the remaining nodes. This way, every node maintains routing information about exactly l nodes in every other *slice*. On top of this, each node also has routing information about every node in its own slice.

Node failures are handled similarly to the one-hop scheme, the only difference occurs in the information that a *slice* leader sends to the other *slice* leaders. In the one-hop scheme, each *slice* leader sends membership changes regarding every node in its *slice*. In the two-hop scheme, the message it sends to other *slice* leaders concerns only the l nodes "assigned" to the target leader.

Each node maintains a table with the closest nodes to every other slice, based on their network distance to itself (e.g., number of hops). To perform a query, the initiating node sends a lookup request to the closest node it knows from of the *slice* containing the key, thus minimizing the first hop latency. The chosen node then forwards the request to its destination.

2.2.8 Content-Addressable Network (CAN)

CAN [20] is a fully-distributed, scalable, robust and self-organizing distributed hash table. It provides a means to store, retrieve and delete data with low-latency communications.

The CAN architecture considers a virtual d-dimensional Cartesian coordinate space divided amongst its participants. Nodes are organized in an overlay, in a way that a given node only knows its immediate neighbors, i.e., responsible for the coordinate zones adjacent to its own.

For a node to join CAN, it generates a random point in the coordinate space and routes a join request to the node responsible for the zone that encompasses it. That node then divides its own zone in half and assigns one piece to the new node. The routing tables at the new node, the old node and the surrounding neighbors are updated accordingly. This process only affects O(d) nodes and is independent of the network size.

In the event of a node departure or failure, its zone is merged with the zone of one of its neighbors. If merging is not possible then the neighbor with the smallest zone temporarily takes over it. A zone-reassignment algorithm is then responsible for stabilizing the fragmentation of space. In order to enable failure detection, every node periodically sends keep-alive messages to its neighbors.

To store or lookup a piece of data, that data's key is mapped onto a point in the overall space and the node responsible for the zone that comprises it is selected. Routing is performed at each node by forwarding messages to the immediate neighbor that is closest to the destination, taking O($dN^{1/d}$) hops, with N the total number of nodes. Note that, in case of node failure, it is still possible to route messages to other neighbors.

Some design improvements are also proposed in [20]. Such as, increasing the dimensions of the coordinate space, which improves fault-tolerance (more next-hop alternatives) and reduces path length at the expense of a small increase in the routing tables; consider multiple coordinate spaces (*realities*) in a way that data is replicated in each one and nodes are assigned a zone in each reality, thus contributing to an improved data availability, fault-tolerance and reduced path length; tuning the routing metric to consider the IP-level topology, reducing path latency; assign multiple nodes to each zone, improving fault-tolerance; using different hash functions to map data to zones, in a way that the same information is stored at multiple sites; building the overlay taking topological concerns into account, by measuring the *RTT* to several physical landmarks;

caching and replication techniques in order to increase the availability of popular keys.

2.2.9 Kelips

Kelips [9] is a peer-to-peer DHT with constant lookup costs used to replicate file index information. It uses a model that increases the memory used to maintain routing information at each node - O($\sqrt{N}$). This aspect is motivated by the higher latency introduced by multi-hop routing, especially when nodes with low bandwidth capacity are involved.

Nodes are arranged in clusters, designated by *affinity groups*. The routing information maintained at each node is divided into three types of records. The first two types consist of routing information and some other data, like *round trip times* (RTT) or heartbeat counters, about nodes that belong to the same group and nodes from foreign groups, respectively. A third record type binds file names to their owners' ip-addresses, but only for the owners that are a part of the same affinity group. The owner of a file is designated by its *homenode*.

In order to keep a consistent system state, each piece of information stored has a heartbeat counter associated. This counter needs to be updated before a provided timeout occurs, otherwise the corresponding information is deleted. These updates are responsibility of the node whom the information is about.

System changes are disseminated using a protocol based on gossip. A node that wants to disseminate a message proceeds its execution in a series of rounds (at fixed time intervals). In each of these rounds the node chooses a few other contacts (nodes from its own group and from foreign groups) and forwards them the information. Target nodes are chosen by proximity, using the *RTT*.

Both file lookup and insertion are done in a similar way with constant complexity - O(1). The file name is first mapped into the corresponding *affinity group* using the same hash function employed in node-to-group assignment. To insert a file, or to look it up, the querying node contacts the closest node (smaller *RTT*) it knows from the target *affinity group*. This node finally routes the request to the relevant node.

2.2.10 Catadupa

Catadupa [13] is a full membership substrate designed to support peer-to-peer content-based routing. It focuses on the main premises of this particular routing problem - load balancing and false positives/negatives avoidance - while keeping good performance in mind.

This solution lies on two components: Catadupa and Turmoil. The former is responsible for the full membership/interests view maintenance while the later handles actual message delivery.

Catadupa's network structure is based on an identifier ring modulo 2^{128} where each node's identifier is the hash of its ip and port. Like in some known DHTs [7, 8], the identifier space is partitioned into slices, each one with its own responsible node, designated here as the ***sequencer***. It is the node with the smallest key in the slice and is responsible for broadcasting arrivals of nodes whose keys fall in its slice range.

The broadcasts are performed every 30 seconds, period in which the sequencer aggregates and tags join requests with a sequence number. The aggregation mechanism allows the reduction of the communication overhead and allows compression. Each message is broadcast resorting to random-trees recursively built by subdividing, at every interior node, the identifier space into a number of sub-ranges (independent from the number of slices). The whole key space is considered initially and, for each sub-range, the node with the smallest key is chosen to process it, until only leaf nodes remain. A node that left the system is detected locally and only removed from the membership view of the node that tried to contact it.

Turmoil is the content-based routing layer. It is implemented on top of Catadupa and relies on its full-membership information in order to find, for a given message, all the nodes interested in its reception. To do so, Turmoil behaves similarly to Catadupa in the sense that the sender initiates the creation of a random-tree for each message. The difference lies on the fact that, an interior node, instead of choosing the node with the smallest key to handle each sub-range, it chooses the first node it knows to be interested in the message. Therefore, only the interested nodes will be part of the message dissemination tree.

As a backup measure, nodes periodically exchange view information, in an epidemic fashion, to recover from possibly missed Catadupa or Turmoil messages (due to failure or concurrent broadcasts). To allow this mechanism, nodes store the sequence numbers of the broadcasts they have received, in a way that allows them to assert which ones were missed.

Several other approaches for content-based routing have been proposed and, usually, nodes with similar interests are organized into overlay networks. Their maintenance negatively affects, among other properties, load balancing and low-latency dissemination. Therefore, instead of organizing nodes into overlays, this paper proposes a solution where different random trees are used in each message dissemination. This way, the load of finding other nodes with similar interests, as well as message forwarding, is balanced throughout the nodes. Finally, since full-membership view is required to support this solution, it is somewhat limited in terms of its scalable capabilities. Moreover, while allowing for correct solutions, the scheme proposed does not consider high network churn.

3. Design and Specification

The following chapter provides a contextualized and detailed description of the algorithm proposed to manage a geography-oriented network of nodes with partial membership awareness. In order to explicitly set the ground basis for this dissertation, section 3.1 describes the problem at hand. The following section (3.2) iterates through the various aspects of the solution adopted, providing, where needed, the rational behind each design decision.

3.1 Preamble

3.1.1 Context

The main goal of Participatory Sensing [2, 3, 22] inspired applications is to provide its users with a means to collect and share sensorial data from their physical surroundings. The monitoring of air quality [18] or of traffic and road conditions [10, 16, 5] are examples that illustrate this model's wide range of possibilities.

The main idea to retain from this application domain is that users are mainly interested in sensorial information from physically close locations and, therefore, are expected to mostly need to contact peers located nearby. It would then be interesting that applications based on these ideals could be supported by a communication middleware that offers low latency communications in physically delimited areas.

3.1.2 Problem

Considering a population of geographically referenced nodes, this dissertation addresses the challenge of how to allow each user to be made aware of the peers that lie within a certain physical distance. With decentralization, fault-tolerance and scalability as the main guidelines.

Motivated by this requisite, the notion of *neighborhood* is defined. It is a circular area centered at a node's geographical coordinates (latitude and longitude) encompassing the active peers whom that node is interested in knowing - its *neighbors*. These are stored at each node

in a local database, designated as the *neighborhood database*. Note that the priority is to store nodes that are, or have recently been, alive. The monitoring of dead neighbors can, conversely, be given less relevance. This is motivated by the fact that the usage expectations render the management of departed nodes to have little impact on the system's correction.

3.2 Solution Design

3.2.1 Overview

We propose an algorithm to manage an overlay network that reflects a strong geographical correlation between its participants, where each one features a physical location and is interested in knowing the active peers located up to a certain physical distance to itself.

To that end, the overlay network is defined over a two-dimensional plane where the location of each peer is determined by its latitude and longitude on the Earth's globe. The entire physical space is to be divided into smaller partitions so that the efforts of managing the entire network can be reduced to smaller, individual problems. Each partition is then managed by the peer that first joins in the area it encompasses.

Once a partition has a dedicated manager, designated as a *super-node*, it can then start to accommodate new arrivals. These new nodes are assigned as *sub-nodes* and need to be introduced to the peers that previously populated the area. To do so, the super-node responsible for that partition announces the new arrival to the relevant peers.

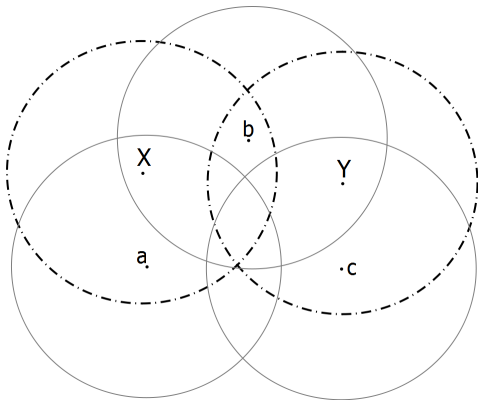


Figure 3.1: Network layout.

Figure 3.1 illustrates the general overview of the network layout. Nodes X and Y act as super-nodes and have their neighborhood boundaries represented by a dash-dot line. Sub-nodes a, b and c are also illustrated and a continuous line illustrates their neighborhoods.

The *announcement* of new arrivals is not perfect and might fail to inform every interested peer. A background *repair* mechanism is thus introduced, leading peers to periodically perform

pairwise exchanges of information in the attempt to recover from missed announcements. It also allows new nodes to acknowledge the peers that were, prior to their arrival, already in the vicinity.

Finally, a *promotion* mechanism was also adopted as part of the solution. Its purpose is to select and promote sub-nodes as replacements for super-nodes that leave the system. This stems from the fact that every space partition requires a managing super-node in order to maintain the network's correct behavior. Note that the handling of departed sub-nodes is somewhat less relevant, the network eventually "forgets" them.

3.2.1.1 Challenges

The maintenance of the proposed layout is not trivial, specially in a dynamic network with high churn rates [24], i.e., frequent membership changes. Several issues need to be carefully considered if the hierarchical node arrangement is to remain consistent through time. The following subsections iterate through these challenges, providing a general description for each one.

3.2.1.1.1 ***Spatial Coverage.***

The proposed algorithm aims to be suitable for a network of nodes that, ultimately, covers the surface of the planet. Users are expected to be located at any point in the Earth's sphere, even considering that most of the planet's surface is inhospitable (seas, deserts, forests, etc.) and that it is unrealistic to assume that the network would fully operate on these locations. Moreover, the population density is not uniform, some regions (e.g. large cities) are likely to have higher participant densities.

This heterogenic population distribution mainly affects the choice of how to partition the physical space. As mentioned before, this is a way to reduce the efforts of managing the entire network. This division could be implicit and performed "a priori", or it could be done dynamically, as the network is being assembled.

In the first, "a priori", alternative, one can consider a fixed grid composed by regular shaped partitions (e.g. rectangular, hexagonal) that would fit the Earth's surface. This approach has the advantage of enabling a disjoint coverage of the space, thereby avoiding any issues resulting from overlapping partitions (as will be mentioned later). However,

since the population density is not uniform, unnecessary efforts would be employed when pre-establishing the grid, specially for deserted areas. Moreover, a regular shaped division would also difficult the representation of the nodes' neighborhoods. Finally, the size and arrangement of the partitions should reflect population density in order to achieve a suitable and balanced coverage of the participants.

Partitioning can also be dynamic, performed as the network is being assembled. A new partition is created every time a node joins the network in an *orphan* location, i.e., a location that is not comprised by any existing partition. Its management is then delivered to the joining node, thereafter designated as *super-node*.

Super-nodes also have their area of interest (neighborhood). Therefore, it seems appropriate to match each partition with the neighborhood of the super-node managing it. Both are assigned circular shapes as these are the most intuitive to represent distances from a given point in space. Again, population density is not uniform, which should influence the size of the neighborhoods. In an attempt to balance their number of nodes, areas of higher density should expect smaller sized neighborhoods. Conversely, in less populated regions, larger neighborhoods should be considered.

The fact that partitions share the same shape as the neighborhoods of their managing super-nodes leads to a coverage issue. One that is related to the fact that it is not possible to, using circular shapes, completely cover an area without overlapping. This results in some loss of efficiency as the number of super-nodes needed to cover the network's entire area is higher than if the partitions were disjoint. Refer back to figure 3.1 for an illustration of this scenario. Note that the neighborhoods of super-nodes X and Y intersect and also that sub-node b has both of them in range.

In fact, any loss of efficiency resultant from this "super-node redundancy" results in an increased fault-tolerance. That is, considering an area of intersection, covered by more than one super-node, if one should fail, the sub-nodes lying there still remain with, at least, one super-node in range. This could be taken even further by increasing the minimum number of super-nodes that must be in range of each sub-node. Still, this matter is not extensively addressed in this dissertation and the imposed requisite is that every sub-node must, at least, have one super-node in range.

3.2.1.1.2 ***Super-node Concurrency.***

Two or more super-node arrivals are deemed concurrent if they occur approximately at the same time and the nodes involved belong to each other's neighborhoods. Referring back to the issue of "super-node redundancy" mentioned in section 3.2.1.1.1, "super-node concurrency" can be perceived as two, or more, super-nodes having such a large partition intersection that they are actually in range of each other.

When a node is joining the network, it needs to know which role it will play in the network - super-node or sub-node. That decision is dependant on whether or not there is a super-node already managing its location. The ability to correctly perform this decision determines the degree of super-node concurrency in the network.

A simple way to prevent super-node concurrency would be to centralize the information related to the spatial super-node coverage. This way, a single entity would monitor such information and there would be no inconsistencies as its privileged knowledge would always produce the correct decision. The problem with this approach is that it does not follow the decentralization mentality this dissertation strives to pursue.

On the other hand, for a fully distributed solution to achieve the correction level of a centralized solution, it would require the adoption of distributed consensus techniques, which are too complex and expensive in terms of bandwidth consumption.

Like "super-node redundancy", the fact is that "super-node concurrency" does not compromise the effectiveness of the network. This way, a possible solution is to mitigate the problem and accept that some node assignments might be based in an incorrect view of the super-node coverage and that "super-node concurrency" might actually occur. It consists in providing full-membership to super-nodes, enabling them to assert, to some degree, whether or not any given physical location is already being covered by one of its peers.

3.2.1.1.3 ***Membership Announcement.***

As users join the network, the neighborhood databases at each node should converge to an accurate membership view. To that end, one can endow the closest super-node with the responsibility of directly notifying every single interested peer. The problem is that such burden would probably prove to be too demanding in terms of bandwidth consumption.

This drawback requires a decentralized announcement solution that balances the load and minimizes the efforts of the super-node managing the new node's partition.

In order to further decrease the bandwidth consumed during the announcement process, it is necessary to avoid duplicates and *false positives*. Every node should only be notified of arrival events in its own neighborhood, and only once. On the other hand, it is also necessary to deal with announcement messages that might not reach their destination - *false negatives*.

Even though it would only provide probabilistic guarantees, one can also consider the adoption of an epidemic repair mechanism to solely deal with the membership dissemination or simply to fill in the gaps left the announcement process. Running in the background, it would consist in the periodic and pairwise exchange of information between peers.

Finally, considering that closely located users are expected to gather similar data, usually from the same locations, it is reasonable to assume, in many cases, high levels of redundancy. For this reason, a membership dissemination mechanism that would provide, at all times, a perfect knowledge of the peers lying in the vicinity is not mandatory and a "best-effort" approach might be sufficient to meet the requirements of most users.

3.2.1.1.4 ***Fault Tolerance.***

A node departure, voluntarily or not, has to be reflected in the system. The leaving node needs to be removed from the neighborhood databases to prevent these from growing unlimitedly. Therefore, a criteria that defines when and how to do it has to be adopted.

In the case of a leaving sub-node, the problem is confined as these only need to be removed from their neighbors' databases. Note that if a sub-node remains in its neighbors' databases some time after its death, no correctional issues affect the algorithm.

On the contrary, super-nodes require constant monitoring. The fact that super-nodes are responsible to handle arrival events in their partition introduces the need to quickly detect their failures and replace them with a suitable candidate. The selection of the replacement node should be achieved with minimal coordination overhead.

3.2.2 Architecture

The geography-driven partial membership algorithm proposed in this dissertation, denominated as ***Geodupa***, divides a network of georeferenced nodes into two hierarchical levels. Nodes assigned to the higher level are designated as **super-nodes** and each one has the responsibility to independently manage a partition of the space - typically its own neighborhood. A joining node is assigned as super-node if its location is not already being managed by another super-node. Conversely, nodes that join the network in a super-node's neighborhood are assigned as **sub-nodes** and compose the lower level.

Neighborhoods are a crucial logical component of a Geodupa network. The neighborhood of node ***A*** is a circular geographical area whose center is located at its coordinates. The radius of the neighborhood is a fixed parameter, measured in kilometers, whose value is common to every node and subject of consideration in section 5.2.4.1. If node ***B*** belongs to the neighborhood of node ***A***, i.e. is in its range, then Geodupa's efforts will eventually lead them to know each other.

The super-nodes that compose the higher-level form an independent network managed by Catadupa [13], a one-hop DHT described in section 2.2.10. This full-membership DHT ensures that all super-nodes know each other, providing them with full visibility over the spatial super-node coverage. Super-nodes also know every sub-node in their own neighborhood.

Sub-nodes form the lower-level of Geodupa. Unlike super-nodes, who have full knowledge of their peers in the higher-level, these only know the nodes that lie in their own neighborhoods.

Figure 3.2 illustrates the general overview of the node arrangement in Geodupa. Nodes *X*, *Y* and *Z* are the super-nodes and have their neighborhood boundaries represented by a dash-dot line. Sub-nodes *a* to *e* are, as expected, all in range of at least one super-node. It is clear that, for instance, nodes *a* and *d* belong to *X*'s neighborhood. The neighborhoods of the sub-nodes are not illustrated in the figure due to clarity reasons. However, the same kind of illustration used to represent the super-nodes' neighborhoods could be adopted.

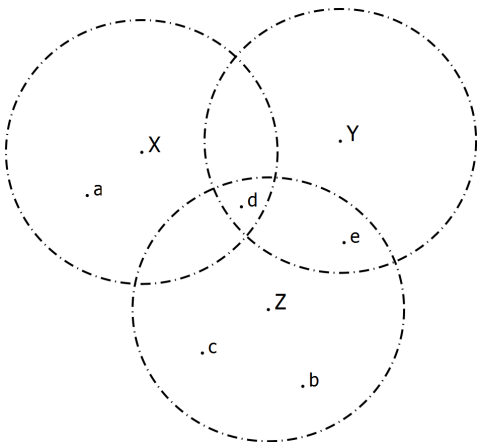


Figure 3.2: Node arrangement.

Ideally, every sub-node should have, at least, one super-node in range. This is important if they are to receive new arrival announcements and is also the ground basis to endow sub-nodes the possibility to reach nodes located beyond their neighborhoods. Actually, distant communication was not evaluated. Still, it could be achieved by, for instance, sending messages through the higher level, in which case super-nodes would act as routers.

Rational

The first impulse, when defining the network architecture, would be to design a solution where participating nodes all knew each other, enabling one-hop lookups throughout the entire network. The problem with this approach lies with the costs of maintaining full-membership databases at each node. These are expected to increase with the churn [24] and size of the network, leading to the scalability issues often associated with one-hop DHTs.

The solution adopted was designed with this problem in mind. Considering the *Participatory Sensing* premise that important information is closer, the network was divided into two levels and the membership scope at each node was reduced. Sub-nodes only know their neighbors, super-nodes know these plus every other peer in the higher-level. The size of each database is thus significantly reduced and any given node in the lower-level has no bandwidth overhead caused by membership changes outside its neighborhood.

3.2.3 Joins

For a node to join Geodupa it first sends a *join request* to its ***broker*** - a random super-node obtained through some external source. The *broker* is then responsible to determine to which level of the network the new node will connect itself to. It will be assigned as a super-node if its location does not belong to the neighborhood of any super-node. On the other hand, if the new node lies in the neighborhood of at least one super-node, it will join as a sub-node.

3.2.3.1 Sub-Nodes

If the *broker* determines that the new node has, at least, one super-node in range, it replies with the contact information of the closest one. The joining node then contacts the super-node received - its ***host*** - and waits for a reply containing its closest neighbor, which will

act as a seed for the first epidemic repair (section 3.2.5). Finally, to complete the joining process, the *host* triggers an announcement mechanism, known as *geographical multicast* (section 3.2.4), in order to disseminate the new node to its neighbors.

To ensure the awareness between sub-nodes attended by different super-nodes, the final step is to disseminate the new node to all the super-nodes whose neighborhood intersects the new node's. That is, all the super-nodes whose neighborhood might contain sub-nodes interested in the new one. Super-nodes contacted this way proceed to send the new node to the neighbor that is closest to it.

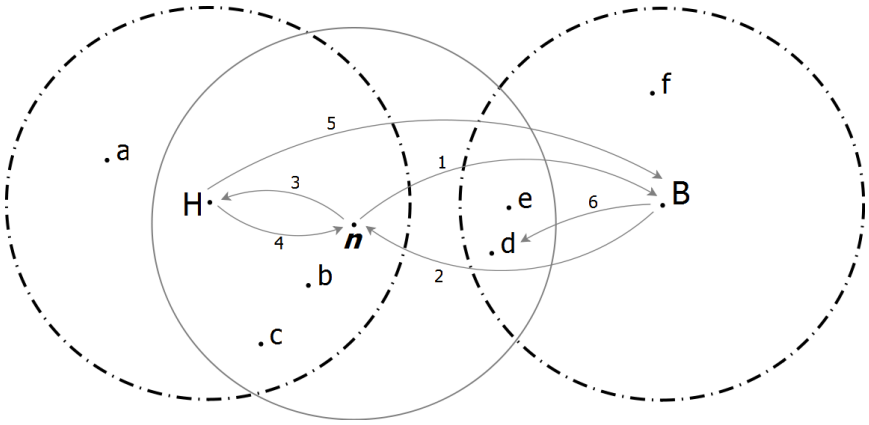


Figure 3.3: Sub-node ***n*** joining Geodupa

Figure 3.3 illustrates the joining process of a sub-node. The network is composed by super-nodes *H* and *B* and sub-nodes *a* to *f*. Again, due to clarity reasons, only *H*, *B* and *n* have their neighborhood boundaries illustrated.

The joining node - *n* - first sends a join request (**1**) to *B*, its *broker*, which determines that *n* has super-node *H* in range, thus replying with its contact information (**2**). The *host* is contacted (**3**) and replies with the contact information of *b* (**4**), to whom *n* can now resort to perform its first epidemic repair.

Since there is a super-node whose neighborhood intersects the neighborhood of the new node (in this case, the *broker* itself), *H* notifies it (**5**). *B* then notifies *d* (**6**), the closest it knows to *n*, thus enabling the interconnection between both super-nodes' neighborhoods as *n*) can now be aware of *d* and *e*.

3.2.3.2 Super-Nodes

If, on the other hand, the *broker* detects that the new node has no super-nodes in range, the reply informs the joiner to connect itself as a super-node. That is, the new node triggers the process of joining Catadupa - the DHT responsible for managing Geodupa's higher-level.

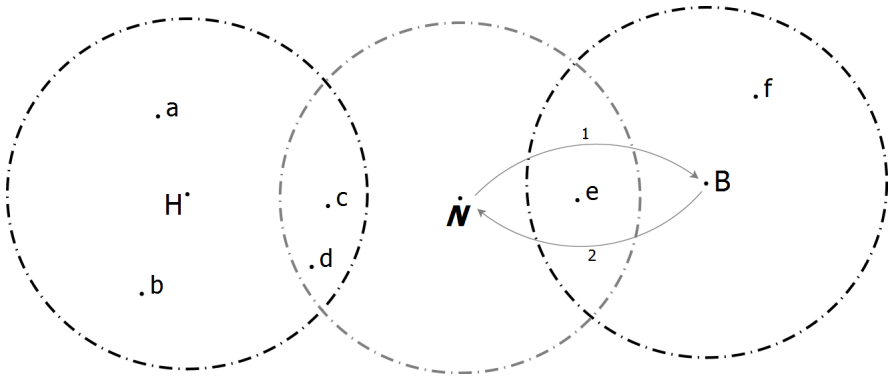


Figure 3.4: Super-node ***N*** joining Geodupa

Figure 3.4 illustrates the joining process of a super-node. The network is composed by super-nodes ***H*** and ***B*** and sub-nodes ***a*** to ***f***. The joining node - ***N*** - first sends a join request (**1**) to ***B***, the *broker*. In turn, it determines that ***N*** has no super-nodes in range. Therefore, its reply (**2**) induces the ***N*** into joining the higher-level, i.e., Catadupa.

When a new super-node joins the network, some sub-nodes could already exist that need to me made aware of it. For this reason, whenever an older super-node knows the new one, it checks if their neighborhoods intersect. If they do, then its neighborhood might contain some sub-nodes in range of the new super-node. It thus triggers the announcement of the new super-node to the lower-level, using the same multicast mechanism adopted to announce new sub-nodes, described in section 3.2.4.

Referring back to figure 3.4, when ***H*** and ***B*** receive the confirmation that ***N*** joined Catadupa, they determine that their neighborhoods intersect and both trigger the announcement mechanism that will eventually notify nodes ***c***, ***d*** and ***e***.

Discussion

The future role of a joining node depends on the *broker*'s knowledge of the super-node coverage. The *broker* selects the super-node it knows that is closest to the new one. If they are in range of each other, the new node will connect as a sub-node. If not, it will be a super-node.

The fact is that the *broker*'s database, due to the consistency issues involved in maintaining the full-membership, might not accurately reflect the super-node spatial coverage. This introduces a drawback to the criteria adopted. That is, upon the receival of a join request, the *broker* might not yet be aware that a super-node exists in the neighborhood of the joining node. Its answer to would then lead the joining node into connecting as a super-node, thus resulting in "super-node concurrency", i.e., two super-nodes coexisting in each other's neighborhood.

Some changes to the joining process were considered in order to minimize this phenomenon. Namely, instead of basing its decision solely on its own view of the higher-level, the *broker* could contact other super-nodes in order to receive their input regarding super-node coverage.

Another way to minimize "super-node concurrency" would be not to accept the *broker*'s response unconditionally. Instead, the new node could send a new join request to the super-node that the *broker* recommended. This would lead to an iterative process that would end when the join request reached a super-node that considered itself as the closest.

Finally, a third alternative was considered. When a super-node joins, it obtains the membership information of its peers in the higher-level. With this knowledge, it could check if its own neighborhood contains any of the others. It it does, a *demotion* mechanism could be introduced so that one, or more, concurrent super-nodes would be relegated to sub-nodes.

Although it would be possible to reduce "super-node concurrency", these changes would lead to an increased complexity and traffic overhead. The fact is that this issue can be mitigated. That is, in terms of efficiency, it is in fact an undesirable problem. Since partitions overlap, the number of super-nodes required to cover the entire physical space is higher than if concurrency was excluded. However, it has no impact in the algorithm's correctness and, as mentioned before, can even be seen as a way to increase fault-tolerance. Besides, considering that Catadupa's membership management is quite effective in providing super-nodes with an accurate and updated knowledge of their peers, the result is a somewhat insignificant percentage of new nodes being erroneously assigned as super-nodes (as observed in section 5.2.3). For these reasons, the mentioned modifications were not adopted and the joining process was kept as simple as possible.

3.2.4 Announcement

For super-nodes to announce other nodes into the lower level, a *geographical multicast* is used. It is performed through the creation of distributed random trees composed by the neighbors of the node being announced.

To perform it, the super-node starts by dividing, into four quadrants, the square that bounds the entire neighborhood of the target node. For each quadrant, it selects one random node from its own database and delivers it the responsibility to continue the multicast in that quadrant. Note that the nodes selected must also belong to the neighborhood of the node being announced.

Each selected node also subdivides its entrusted quadrant into four sub-quadrants, selecting a node it knows for each one. This recursive subdivision ends when an intermediate node determines that the nodes left to notify in its quadrant are less than a parameterizable *fanout* (typically equal to 4). In which case the remaining nodes are notified directly.

Figure 3.5 illustrates the multicast process. The network is composed by super-node ***H*** and sub-nodes ***a*** to ***i***. Sub-node ***n*** is being announced. For illustrative purposes, the fanout used in this example is equal to 2, meaning that the intermediate nodes will subdivide the quadrants until the number of remaining nodes is 1 or 0.

To start the multicast process, in step 1, H slices n's neighborhood into four quadrants and, for each one, selects a random node (b, e, g and h) and delivers them the responsibility to process their corresponding quadrants. In step 2, g and h do not know any other nodes in the quadrants they were entrusted. The subdivision ends. Node b actually knows two nodes that lie in its quadrant - H and a. However, they are not eligible to receive the announcement: H already knows n and a is not in n's neighborhood. Finally, node e determines that there are four eligible nodes in its assigned quadrant and proceeds with its subdivision. Nodes c and i are selected. In step 3, while i does not take any further action (zero nodes left in the assigned quadrant), node c determines that f and d still need to be informed. c subdivides the quadrant and randomly selects d to deal with the remaining space. Finally, in step 4, d computes the pool of eligible target nodes and realizes that only f is left. Since the fanout was set to 2, d simply notifies f directly and the multicast ends.

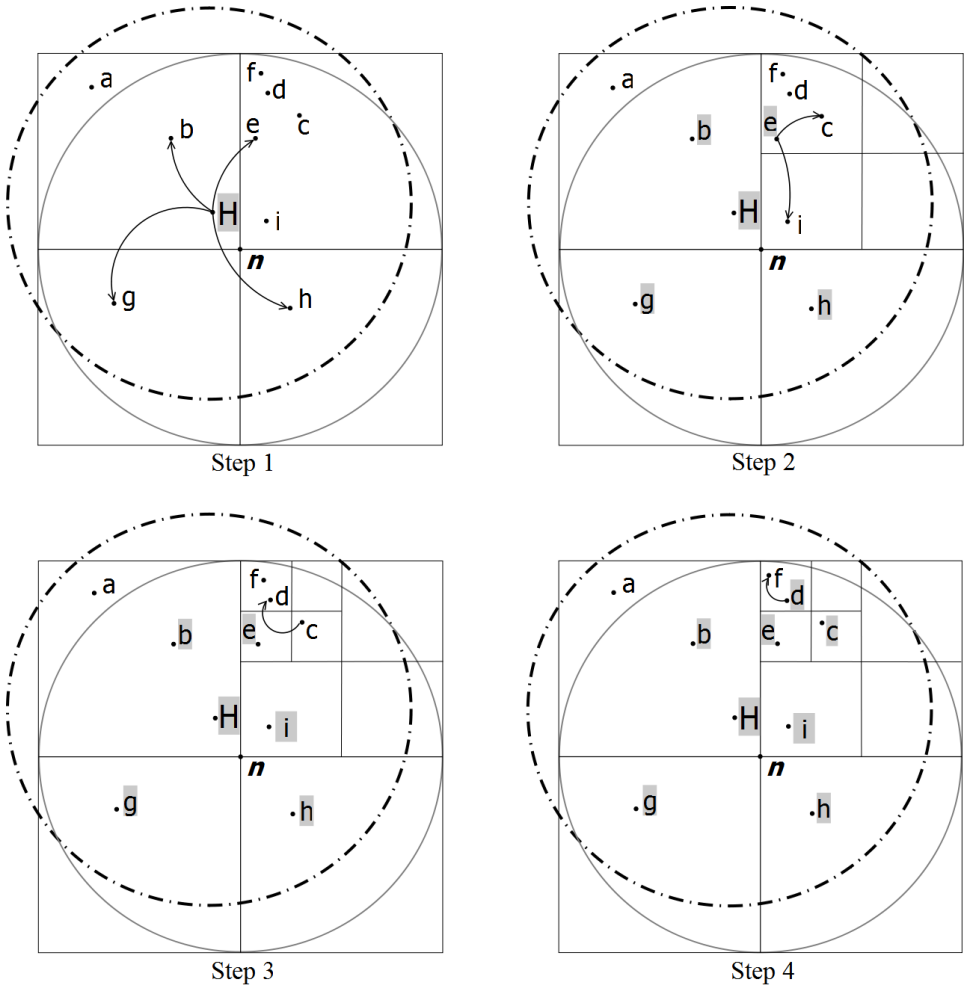


Figure 3.5: Announcement of node ***n*** using the *geographical multicast* (*fanout* = 2).

Optimization

The effectiveness of the geographical multicast is dependent on the knowledge that the initiating super-node has from the neighborhood of the node being announced. For instance, if the announced lies near the neighborhood border of the super-node, then the first step of the multicast would probably fail to inform every relevant quadrant. Figure 3.6 illustrates such case. Super-node H triggers n's announcement and is responsible to inform one node for each quadrant of its neighborhood. However, since their belonging nodes lie outside H's neighborhood, it would not be able to reach the two rightmost quadrants, i.e., nodes e, f and g.

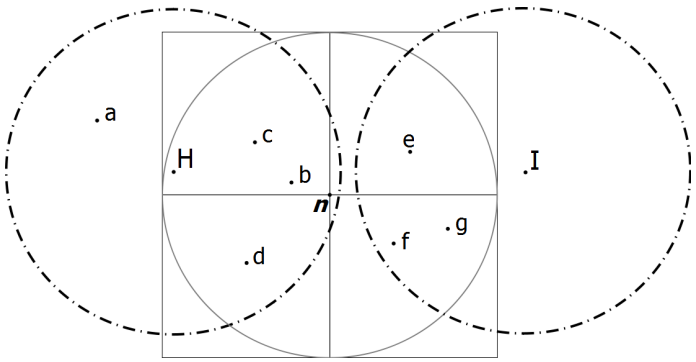


Figure 3.6: Example of the multicast need for optimization.

To minimize the occurrence of such event, the initiating super-node does not start the multicast process right away. It consults its database and selects the neighbor closest to the node to announce, relegating the multicast responsibilities to it. This is motivated by the fact that closer nodes have larger neighborhood intersections. In figure 3.6, H would contact b to start the multicast. Nodes e, f and g are thus more likely to be included in the multicast tree.

Due to the limitation described above, as well as the constant membership changes in the network, this mechanism is not 100% accurate. That is, some nodes might not be notified. To address this problem, section 3.2.5 introduces a background epidemic repair mechanism that allows nodes to recover from missed announcements.

3.2.5 Epidemic Repair

The goal of the epidemic repair mechanism is to fill in the gaps possibly left by the announcement of new nodes (section 3.2.4). It also serves for newly joined nodes to acknowledge the membership status of their neighborhoods before their arrival.

At fixed time intervals, each node (super-node or sub-node) selects a random neighbor from its database and sends it the set of neighbors in its range. The selected node then replies with a similar set of nodes. However, this one does not contain the nodes that the requester initially sent, i.e., nodes that both previously knew. When this 2-way interaction ends, both participants are left with a similar membership view of their neighborhoods' intersection.

An example of this process is illustrated in figure 3.7. Super-nodes X and Y form the higher-level while sub-nodes a to g compose the lower-level. In an effort to maximize the clarity of the figure, only nodes X, Y, a and b have their neighborhood boundaries illustrated.

The epidemic repair is performed between sub-nodes ***a*** and ***b***. Before their interaction, *a* knows *Y*, *b*, *d* and *c*. *b* is aware of *X*, *f*, *c* and *e*. To initiate the process, *a* sends *b* a message (**1**) containing *c* and *d*. Since node *b* already knew *c*, it only adds *d* to its database. Note that *b* was not aware of *a* and thus also stores it in the database. To complete the repair process, *b* replies (**2**) with a set of its own neighbors in range of *a* (that were not in message **1**). Only one node satisfies these conditions: *e*. The epidemic repair finishes, leaving the databases of the nodes *a* and *b* with the same knowledge of their neighborhoods' intersection.

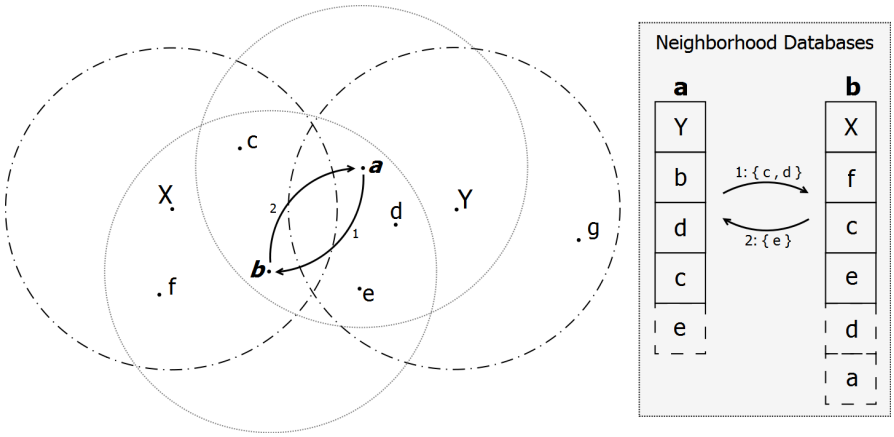


Figure 3.7: Nodes **a** and **b** epidemically repairing their databases.

Clock Synchronization

In Geodupa, each participant tags each neighbor with a ***time-stamp*** marking the last moment in time it believes that node was active. This way, each node is able to remove old entries from its neighborhood database by periodically checking their freshness. Neighbors whose *time-stamp* is older than a given time threshold are thus removed.

The management of the *time-stamps* is performed locally, in the sense that each participant maintains its own set (one for each neighbor) stored in its neighborhood database. Still, these can be influenced by other nodes.

Initially, when a node joins the network, its *host* tags it with a *time-stamp* marking the time of arrival. Every node that receives its announcement stores it with the time-stamp provided by the *host*. Thereafter, other peers locally update that node's *time-stamp* whenever they directly contact it or whenever they receive a more recent *time-stamp* for that node, during the epidemic repair.

The fact that nodes exchange *time-stamps* introduces the issue of clock synchronization, often associated with distributed systems. Keeping the clocks at every node perfectly synchronized is a difficult and complex task. The fact is that there is no strict need that motivates the adoption of mechanisms that pursue a synchronization with, say, millisecond precision. Note that if clocks are allowed to deviate even by a few minutes, in the worst case scenario, some neighbors are removed slightly sooner, or later, than they should.

As mentioned in section 3.2.1.1 (under "Membership Announcement"), the expected redundant nature of the information allows us to consider a "best-effort" approach when dealing with management of the neighborhood databases. That is, a node does not require to know, at every moment, exactly which active peers lie in its neighborhood or which ones have already departed. Therefore, the issue of clock synchronization was mitigated. Nodes are only expected to be connected to the Web and synchronized with the "Internet Time".

3.2.6 Departures

When a node fails to communicate with one of its peers, it proceeds to remove the unresponsive node from its neighborhood database. No further actions are performed, thus resulting in the absence of overhead when dealing with node departures. The drawback of this approach is that a node might persist in several databases long after its departure, only being removed when an attempt to contact it is made. To address this problem, each node locally maintains, as mentioned in section 3.2.5, a *time-stamp* associated with each one of its neighbors. This enables every node to, periodically, discard the neighbors that are thought to be inactive for a certain amount of time - the *neighbor TTL*.

Discussion

As an alternative to this simplistic solution, a more proactive behavior could be adopted. For instance, failure events could be disseminated throughout the network so that others could also remove the failed node from their databases. This way, ideally, no one would ever try to contact the departed node again. This would seem a suitable approach.

However, besides the obvious traffic overhead, another, more fastidious problem, would have to be addressed. It is related to the inability to assert if an unresponsive peer has actually

left the system or is simply suffering from connection problems. This uncertainty could thus lead to the incorrect announcement of departed nodes. The complexity and intricacies of this issue turned the balance in favor of a deferred handling of failures, where each node eventually "forgets" its dead neighbors.

3.2.7 Promotions

When a super-node leaves the network, the region it previously occupied and the sub-nodes that lie there are possibly left *orphan* of super-nodes. A *promotion* mechanism is thus presented. It detects the departure of super-nodes and replaces them with suitable sub-nodes.

Every sub-node in the network periodically receives evidence that a super-node exists in its neighborhood. It might be direct evidence, when communicating with a super-node, or indirect, during the epidemic repairs or when node announcements triggered by super-nodes in range are received. Whenever such evidence is received, the moment in time it occurred is recorded. This allows every sub-node to periodically check how long it has been since the last time it "heard" from a super-node in its range. If this time interval is greater than a parameterizable threshold - the **suspicion timeout**, the node suspects that there are no super-nodes in its neighborhood.

One of the possible evidences, the announcement, is triggered when a new node joins. Consequently, in the absence of new arrivals, there are no announcements, thus significantly reducing the evidence flow. This might cause some sub-nodes to be unaware of one or more super-nodes in the area, eventually leading to erroneous promotions. Therefore, in order to prevent super-nodes from being idle for long periods of time, each one announces itself if it has not performed any announcement for a pre-established period - **heart-beat timeout**.

When a sub-node becomes suspicious, it triggers a process that will eventually lead one of its neighbors (or even itself) to replace the last super-node it knew. Intuitively, the best candidate would be the sub-node closest to the failed super-node, simply because its neighborhood would cover most of the region left *orphan*.

The first step of the promotion process is to compile the set of promotion candidates. To do so, the suspecting node consults its database and selects the neighbors closer than itself to the last super-node it "heard" from, ordering them taking into account their distance to the departed super-node. Then the suspecting node sends a promotion request to the first candidate

(the closest). If it rejects the request, the suspecting node skips to the next candidate. This iterative process terminates when one of the candidates accepts to be promoted, is promoting or has already been promoted to super-node. If none of the candidates answers successfully, the suspecting node attempts to promote itself.

A node that receives a promotion request will only accept to promote if it also suspects it was left orphan. If not, it denies the promotion request. Possibly, there are several nodes executing the promotion protocol. This eventually results in the same node being contacted several times by different suspecting nodes. Thus, a node might be promoting or already promoted when a promotion request arrives, in which case its response would lead the requesting node to acknowledge it as a super-node and to cancel its side of the promotion protocol.

When a node decides to promote, i.e., join the higher-level (Catadupa), it does not do so right away, it only promotes after a certain period of time - **promotion tolerance**. It is a way of preventing concurrent promotions as, during this interval, a promoting node will cancel its intentions if evidence of an active super-node arrives. Note that also the nodes that are trying to promote a neighbor will also stop doing so if super-node evidence is received.

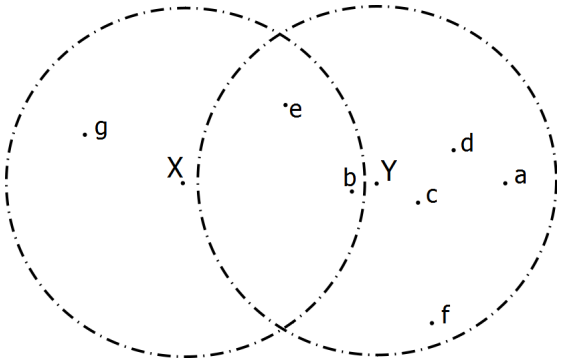
Figure 3.8 illustrates the promotion process. Figure 3.8a presents a network composed by super-nodes *X* and *Y* and sub-nodes *a* to *g*. In figure 3.8b, *Y* leaves the network and sub-nodes *a*, *c*, *d* and *f* become *orphan*. The first one to detect it is sub-node *a*. It thus triggers the promotion protocol and selects *b*, *c*, and *d* as the possible candidates. Note that these are ordered considering their distance to ***Y***. The first promotion request is sent to *b*, the closest. It is denied because *b* is not suspicious, it knows an active super-node in range - *X*. Sub-node *a* skips to ***c***. Since it has also meanwhile detected that it was left *orphan*, the reply is positive and it joins Catadupa. The final layout of the network is illustrated in figure 3.8c.

Motivation

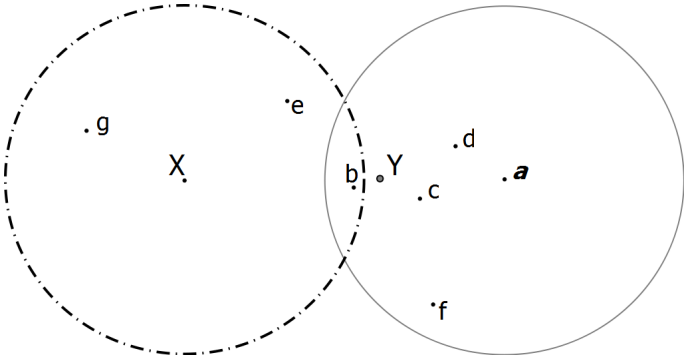
One of Geodupa's premises is that every sub-node should contain, at least, one super-node in its neighborhood. This is motivated by the fact that super-nodes are the means to reach farther nodes and because they are responsible to attend and announce new arrivals in the network.

When a super-node leaves the network, the partition it previously managed is possibly left *orphan* of super-nodes, and so will the sub-nodes that lie there. In order to cover that region again, it would be possible to simply wait for a node to join in an *orphan* location as it would

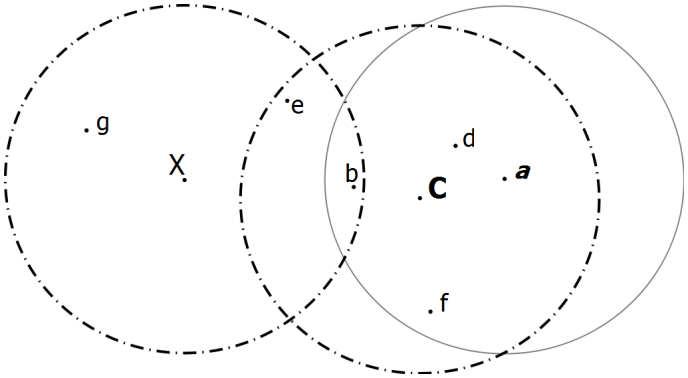
be assigned as a super-node. While this solution would eventually solve the problem, there are no guarantees as to when. That is, node arrival is probabilistic and, therefore, the time a region is left orphan might be significant jeopardize the algorithm's behavior. This problem was thus solved using sub-node promotion, a more proactive approach.



(a) Before the departure of super-node ***Y***



(b) ***Y*** leaves the network. ***a*** starts the promotion protocol.



(c) ***C*** is promoted to super-node

Figure 3.8: Node ***c*** promoting to super-node.

4. Implementation

The following chapter presents and describes the implementation process for the solutions proposed in chapter 3. First, we state this chapter's objectives (section 4.1) and provide a general description of the platform used to simulate and evaluate the algorithm (section 4.2). The following section (4.3) briefly presents the implementation of Catadupa and section 4.4 finally describes the implementation of Geodupa in detail.

4.1 Objectives

In order to determine that Geodupa achieves the proposed objectives for this dissertation, it is necessary to evaluate its operating behavior under conditions as realistic as possible.

Ideally, Geodupa would be implemented in a "real-life" scenario, where each node would actually represent a Personal Computer. This would allow a realistic evaluation as network conditions would be genuine, e.g., churn rates, network problems, bandwidth capacity, communication latency, geographic constraints, user behavior, etc. However, evaluating Geodupa in such a scenario is not feasible in the available time frame.

To get around this problem, we resorted to a platform that enables the simulation of a network of nodes, as well as its inherent conditions and variables, on a single physical computer. The main objective of this chapter is then to describe the adopted simulation environment and the way the algorithm's design was translated to suit it.

4.2 Simulation Platform

In the scope of this dissertation, a simulation platform is a programming environment in which it is possible to emulate a "real-life" operation process, over time, of a network of several nodes, with a good degree of fine control. It provides a means to analyze and collect statistics of distributed coordination algorithm's, such as Geodupa, by enabling the representation of nodes and their behavior, as well as the communication channels used by nodes to exchange messages.

There two main execution models used in simulation environments for distributed systems, one based in *cycles* and the other in *discrete events*.

In the **cycle-based model**, time elapses in a constant manner and is divided into equally sized slots. At every slot change, every node executes its pending tasks, which translates into high concurrency and synchronization. Conversely, this leads to the existence of idle cycles, in which no actions are performed. The fact that nodes often perform their actions at the same simulation time leads to the absence of a certain amount of lag that would be desirable to allow the actions of nodes to intercalate. The main advantage of this simulation model is that it easily accommodates very large quantities of nodes as it does not require the constant re-ordering of the event priority queue (as does the *discrete-event model*, mentioned next).

In the **discrete-event model**, the network has its operation reflected in a sequence of distinct chronological events used to model internal node tasks and network packets. This sequence is represented as a priority queue, shared by all entities in the system, in which the events are ordered by their starting time. The simulation unwinds by constantly removing and executing the first event in the priority queue. This requires the reordering of the priority queue whenever a new task is added. Being a computationally heavy operation, this restrains the simulation of very large numbers of nodes. In this model, time elapses at irregular hops as the simulation clock skips to the scheduled start time of the next event. Moreover, the execution of an event is an atomic operation, i.e., it is instantaneous from the simulation's point of view.

The simulation platform adopted to model and evaluate Geodupa, designated as ***SimSim***, follows a *discrete-event* model. It is more suitable to simulate concurrent events as tasks from different nodes are intercalated in an event priority-queue. Considering the temporal ordering of node tasks, this allows a finer evaluation of highly distributed systems composed by several nodes executing concurrently as it maximizes the diversity of node behavior.

SimSim operates over a Java Virtual Machine and acts as a middle-agent between it and Geodupa (figure 4.1). It allows the simulation of several thousands of nodes, each one with its own independent behavior. The modeling of a node's behavior is achieved by its ability to execute small tasks. These can be scheduled to execute only once or periodically, at fixed time intervals. A task is always associated with its owner, a piece of code and a scheduled time of execution.

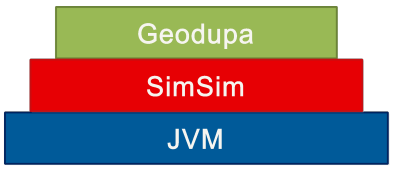


Figure 4.1: Execution model.

Coordination between nodes is achieved by their ability to send and receive asynchronous messages. In fact, a synchronous mode of communication is made available by *SimSim* but it was not exploited as it relies on threading capabilities, which would limit the size of the simulated network.

SimSim allows the usage of two distinct **network models**: *Orbis* and *Euclidean*, with the main difference between them being the way communication latency is dealt with.

Orbis relies on a core graph of nodes that conceptually represents a simplification of the Internet. Simulation nodes are connected to the outer nodes of this graph (perceived as ISP routers) and feature a pre-established discrete communication cost (e.g., 5ms, 10ms). Latency between two peers is computed by summing up the cost to reach their corresponding router nodes, plus the cost of the shortest path between both routers inside the core graph. *Orbis* is then more suitable to simulate classical P2P algorithms, with no geographical relation.

Conversely, in the ***Euclidean*** model, communication latency between peers is calculated taking their position into account. To that end, every node features 2-dimensional coordinates, defined a priori, and latency is proportional to the geometrical distance that separates them. In our case, the *Euclidean* model is obviously more suitable as Geodupa addresses group communication in an geography-oriented environment. Therefore, *SimSim* was parameterized to operate under its conditions and the geometrical coordinates of each node were matched with their geographical coordinates (latitude and longitude). This way, latency is, as expected, higher between farther nodes and lower between closer nodes.

4.2.1 Overview of the Simulation Environment

Figure 4.2 presents a class diagram composed by the fundamental components of *SimSim*. It is only intended to provide a global overview, not a formal and exhaustive description.

To perform a simulation, a ***Simulation*** object is instantiated. It is the simulation's "control center" as it can monitor and control the execution process and, from it, new nodes are created. Moreover, some of the key entities belong to it: the ***Network*** is an abstract class extended by the network type classes (*Euclidean* or *Orbis*); the ***Scheduler*** is responsible for managing the simulation time and the event priority queue; ***Traffic***, a communication statistics collector; and ***Random***, responsible for introducing the necessary unpredictability of "real-life" scenarios.

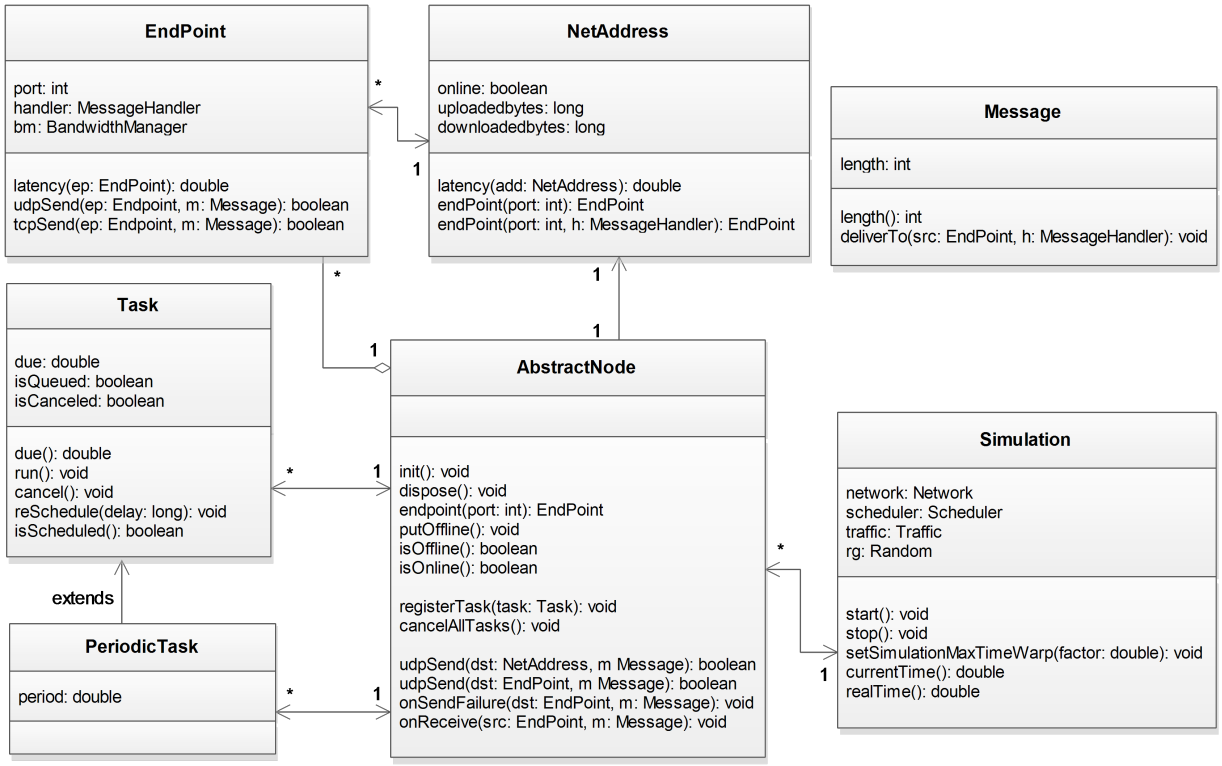


Figure 4.2: *SimSim*'s general class diagram.

Actions in *SimSim* are implemented as tasks. These might be executed only once (***Task***) or periodically (***PeriodicTask***). They may be used by nodes to schedule the execution of some task (e.g., message dispatch) or by other entities to, for instance, gather statistics.

AbstractNode is an abstract class meant to be extended by simulation nodes. In order to communicate with others, each one contains a ***NetAddress*** (perceived as the node's IP address). Messages are represented by class ***Message*** and their exchange is done resorting to the ***EndPoint*** class, representing generic transport endpoints by associating the *NetAddress* to a port.

Actual message reception and delivery is done separately by two distinct methods. The drawback of this approach is that, on request-reply scenarios, the code that dispatches the request is separated from the reply handling. Moreover, it is necessary to implement a *receiver* method for each possible reply. Even if just a layout problem, this intuitively decreases code simplicity and readability, specially when dealing with highly complex algorithms.

To address this inconvenient, *SimSim* provides an alternative higher-level communication construction that easily associates each request with the handling of its possible replies and the

measures to take in the event of a dispatch failure. This way, the method that deals with the reception of messages - *onReceive(Message m)* - is only exploited to handle actual requests, not replies. Procedure 1 presents the behavior adopted by both sides (requester and replier) in pseudo-code.

Procedure 1 SimSim - Request/Reply procedure.

```
1  send RequestMessage(...) to other
2      ↳onFailure
3          // deal with delivery failure
4      ↳onReply( ReplyMessage_1 )
5          // process ReplyMessage_1
6      ...
7      ↳onReply( ReplyMessage_n )
8          // process ReplyMessage_n

1  ↳onReceive( RequestMessage(...))
2      // process RequestMessage
3      reply ReplyMessage
```

4.3 Catadupa

Geodupa relies on a one-hop full-membership DHT, Catadupa [13] section 2.2.10, to manage the super-nodes in the higher-level. This section provides a general overview of its given implementation on *SimSim*. Figure 4.3 presents its simplified class diagram. The goal is not to formalize the implementation of Catadupa in detail but only to provide a general idea.

4.3.1 Entities

4.3.1.1 CatadupaDB

The ***CatadupaDB*** class stores every node in the Catadupa network. These are internally arranged in different data structures, depending on whether they are online or offline. Live nodes are kept in a *List<CatadupaNode>*, dead nodes in a *Set<CatadupaNode>* and, finally,

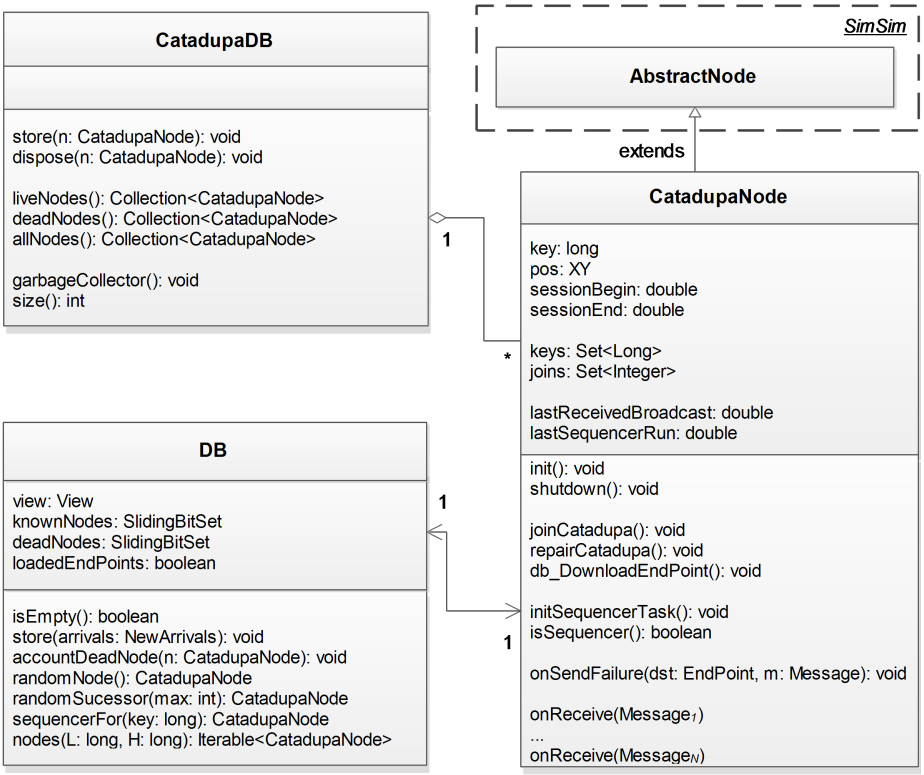


Figure 4.3: Catadupa's entity class diagram.

a *Map<key: Long, value: CatadupaNode>* stores every node and is used as a way to easily obtain a node (live or dead) using its key.

When a node leaves the network, it is thereafter treated as a dead node. It is only actually removed from the database when all other nodes mark it as dead. This "garbage collection" is implemented in a *PeriodicTask*, thus executing at fixed time intervals.

4.3.1.2 CatadupaNode

Nodes in the Catadupa network are represented by class ***CatadupaNode***. At creation time, each one is stored in the *CatadupaDB* (4.3.1.1) being assigned a unique *long* key. Its state variables are then initialized, in particular, its own local database, represented by class *DB* (4.3.1.3).

The several different actions performed by a *CatadupaNode* are implemented using *SimSim*'s ability to schedule and perform tasks. For instance, to join Catadupa, a new node first executes a *PeriodicTask* that requests other Catadupa nodes for their membership databases,

until one replies successfully. It then instantiates another *PeriodicTask* to actually send join requests to the relevant *sequencer*. It executes until the joining node receives an announcement broadcast containing itself, confirming its successful connection to Catadupa.

4.3.1.3 DB

Class representing the individual node databases. It keeps track of the broadcasts already seen (*view*), the nodes believed to be active (*knownNodes*) and the ones presumed dead (*deadNodes*).

Since Catadupa provides full-membership, there is a problem in representing the information at each node. That is, the fact that every node must maintain the membership information of all its peers might stem performance issues and constrain the size of the simulated network.

As a way to minimize the load, Catadupa only maintains a global membership database (*CatadupaDB*) and uses bit masks (*SlidingBitSet*) to reflect inter-node membership awareness. This way the awareness a node has from a given peer is encoded in a single bit (1 - aware, 0 - not aware). Note that, overtime, the masks at each node are progressively being filled as Catadupa drives all its participants to know each other. This way, it is possible to compress the past using a counter that represents the total of bits set to 1 before the first 0.

This membership management allows nodes to maintain only a small fraction of information, compared to what they would if *CatadupaNode* object references were to be used.

4.4 Geodupa

Figure 4.4 illustrates the class diagram for Geodupa's implementation on *SimSim*. Note that the class *GeodupaNode* extends the *CatadupaNode* in a way that the later was kept unchanged, preserving code extensibility. Whenever a new node is created, it is instantiated as a *GeodupaNode*. Variables and methods from its superclass (*CatadupaNode*) are only exploited when, and if, it becomes a super-node, in which case it is also stored in the *CatadupaDB*. This is motivated by the desire to minimize the modifications to the provided Catadupa implementation as it is intended as an independent block of Geodupa. Super-nodes are thus stored in the *GlobalDB* (as Geodupa nodes) and in the *CatadupaDB* (as Catadupa nodes).

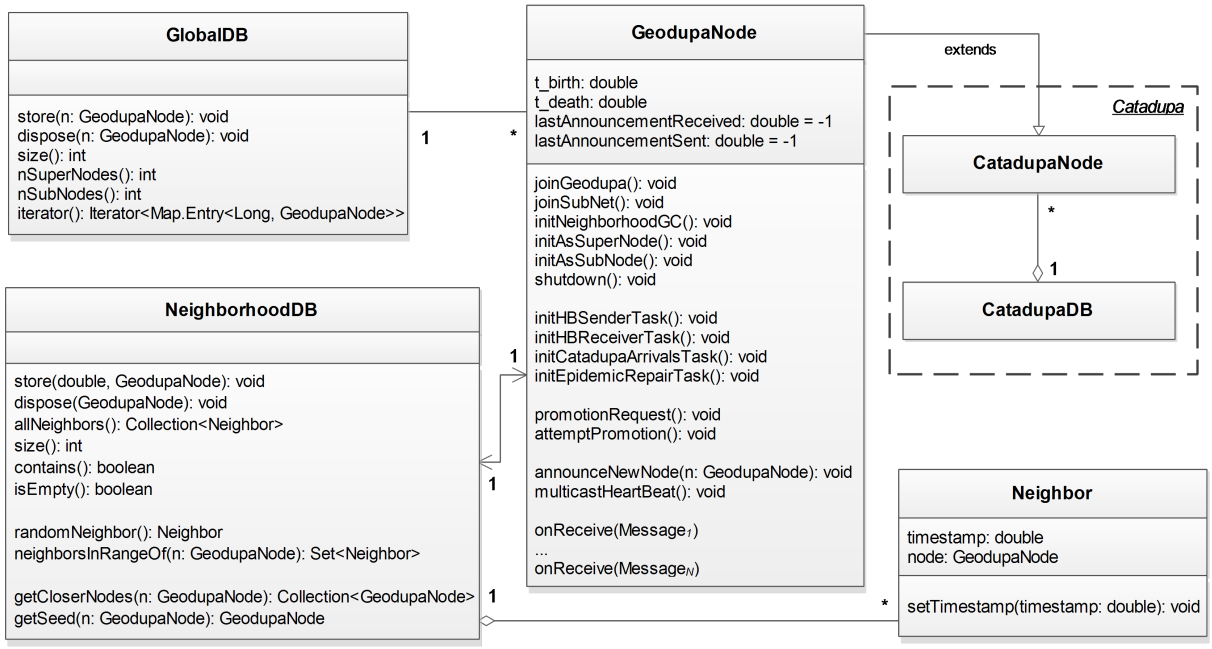


Figure 4.4: Geodupa's entity class diagram.

4.4.1 Entities

4.4.1.1 GlobalDB

This class serves to store every node in the system, regardless of its level (super-node or sub-node). Nodes are kept in a static *HashMap<key: long, value: GeodupaNode>*, made publicly accessible in order to facilitate the gathering and processing of statistics. Like in Catadupa, every *GeodupaNode* is assigned a unique *long* key when it is stored in the *GlobalDB*.

4.4.1.2 GeodupaNode

Class containing the behavior specification of every node in Geodupa. It extends the class *CatadupaNode* thus inheriting the ability to behave as a Catadupa node, i.e., as a super-node.

When a new node is created, an empty *NeighborhoodDB* (section 4.4.1.4) is initialized and the node is stored in the *GlobalDB* (4.4.1.1). The following step is to generate the geographic coordinates (latitude-longitude) that would, in "real-life" scenarios, be obtained by GPS or some

other external source. To do so, a set of **cartesian** coordinates (x, y, z) is first randomly generating on the Earth's surface, modeled as a uniform sphere with radius equal to 6371 Km. These coordinates are actually used to calculate (*Euclidean*) distances between nodes. Once the set of cartesian coordinates has been generated, they are then converted to **geographic** coordinates.

Since the Earth's surface is mostly covered by water, it makes no sense to generate nodes in its total. Therefore, the coordinates that represent a water location are rejected and a new set is generated, until it falls on land. To do so, a third set of coordinates is created for each node. It is obtained by projecting its geographic coordinates in a 2-dimensional plane, which results in a set of **display** coordinates (x, y). These are then placed on top of figure 4.5. If they lie on the green zone (land), they are accepted. They are otherwise rejected.

The projection adopted was the *Sinusoidal Projection*, an "equal-area" projection that maintains the relation between areas proportional to the reality. Note that the process of verifying whether or not a node is located on land is only approximate. It is not based on any formal measurements and is only intended as a way to reduce the number of nodes in the network (to a more realistic number) in a somewhat illustrative way.

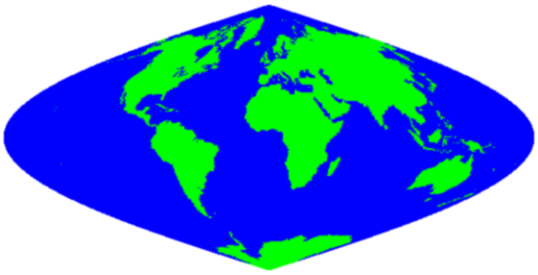


Figure 4.5: Earth's Sinusoidal Projection.

The representation of a node's circular neighborhood is done by defining a point - the center - and a line segment - the radius. The center is matched with the geographical coordinates of the node and the radius is measured from the center of the neighborhood up to a certain distance, a *double* common to all nodes and subject of observation in section 5.2.4.1.

Finally, as mentioned in 4.2, communication between nodes is achieved through the asynchronous exchange of messages. To that end, *GeodupaNode* implements the interface *GeodupaSocketHandler* (in annex, listings A.1) that defines which messages a node must be prepared to receive. Again, when sending a message, it is possible to easily define a handler to deal with the possible replies. To that end, the handler must extend the class *GeodupaReplyHandler* (in annex, listings A.2), overriding the methods relevant to each distinct interaction.

4.4.1.3 Neighbor

A *Neighbor* object represents a node in the neighborhood database. It contains a reference to the corresponding *GeodupaNode* and a time-stamp (*double*) marking the last time that the database owner presumes it was active. The main method this class provides is the *setTimeStamp(double timestamp)* invoked to update the time-stamp of a *Neighbor* whenever the storage of some previously know node is attempted and the time-stamp received is more recent.

4.4.1.4 NeighborhoodDB

This class represents the neighborhood database present at each node. It stores neighbors and provides methods to manage them. Actual storage is done in a *TreeMap<key: long, value: Neighbor>*. The *key* of a given *Neighbor* is equal to the key of the *GeodupaNode* it represents.

When storing a previously unknown *GeodupaNode*, a new *Neighbor* is created and tagged with the received time-stamp. On the other hand, if the *GeodupaNode* to store is already in the database, the *Neighbor* that represents it has its time-stamp updated if the one received is more recent. Procedure 2 describes this in pseudo-code.

Procedure 2 Storing a new neighbor.

```
−store( timestamp, node)
  if node ∈ NeighborhoodDB
     neighbor ← getNeighbor( node )
     if timestamp > neighbor.ts
        neighbor.ts ← timestamp
  else
     new_neighbor ← Neighbor< timestamp, node >
     save new_neighbor
```

To prevent the unlimited growth of the databases, a *PeriodicTask* executes the method *garbageCollector*. Every neighbor's time-stamp is then periodically analyzed and if it represents a moment in time older than a given time threshold (double: *Neighbor_TTL*), the corresponding *Neighbor* is removed from the database. Finally, the exchange of *Neighbor* sets during epidemic repairs is done resorting to method *neighborsInRangeOf*.

4.4.2 Protocol

This subsection iterates through the several distinct node interactions of Geodupa, presenting the entities involved in each one and their behavior. The messages exchanged and their data members are listed in annex A.2.

4.4.2.1 Joins

The process of joining a new node to Geodupa involves three instances of the *GeodupaNode* class - a **joiner**, a **broker**, and a **host** - each one playing a different role. The communication scheme of this process is illustrated in figure 4.6. Remember that the *joiner* only contacts its *host* when it is designated as a sub-node.

The *joiner* first sends a *JoinGeodupaRequest* message to the *broker* containing a reference to itself. The reply (message *JoinGeodupaReply*) informs the *joiner* which level of Geodupa to join. If it is to be a sub-node, the message also contains a reference to its *host*. The process of joining Geodupa is shown in pseudo-code in Procedure 3 and implemented in a *PeriodicTask* that executes until the node joins the network (either as a sub-node or as a super-node).

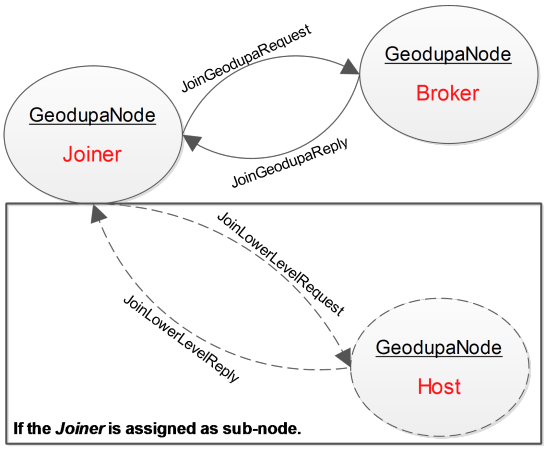


Figure 4.6: Join process.

If the new node is assigned as a super-node, it triggers the process that will eventually connect it to Catadupa. Nodes that already occupied its neighborhood need to be notified. To that end, the class *CatadupaNode* was slightly modified in a way that, whenever a super-node receives a Catadupa membership broadcast, it stores (in a *List*) the new arrivals whose neighborhood intersects its own. A *PeriodicTask* is then responsible to announce these new super-nodes to the lower level, clearing the *List* at each execution to avoid repeated announcements.

On the other hand, if the *joiner* is assigned as a sub-node it sends a *JoinLowerLevelRequest* to the *host* suggested by the *broker*. The reply (message *JoinLowerLevelReply*) contains a reference to a *seed* node with whom the new node can perform its first epidemic repair.

Procedure 3 Joining Geodupa.

```
/* JOINER */
–joinGeodupa()
  repeat
    seed ← CatadupaDB.randomSeedNode()
    send JoinGeodupaRequest<self> to seed
      ↳onFailure()
        continue
      ↳onReply( JoinGeodupaReply<level, +host> )
        if level equals "higher"
          join Catadupa
        if level equals "lower"
          joinLowerLevel(host)
    sleep 45 seconds
  until Joined

/* BROKER */
↳onReceive( JoinGeodupaRequest<joiner> )
  closest ← DB.getClosestSuperNode( joiner )
    if inRange(closest, joiner)
      reply JoinGeodupaReply<"lower"> to joiner
    else
      reply JoinGeodupaReply<"higher", closest> to joiner
```

Using message *SeedDispatch*, the *host* then notifies every super-node it knows (if any) whose neighborhood intersects the new node's. These in turn notify the neighbor that is closest to the new node, using a *NeighborArrival* message (section 4.4.2.2). Finally, the *host* triggers the multicast mechanism to announce the *joiner* to its neighbors. Procedure 4 describes the behaviors of the *joiner* and the *host*, in pseudo-code.

4.4.2.2 Announcements

To announce a node, several instances of the class *GeodupaNode* are involved. Namely, a ***root node*** that starts the process; ***inner nodes*** that receive, process and forward announcements and, finally, the ***leaf nodes*** which are notified directly and have no further responsibilities.

Procedure 4 Joining the lower level (as sub-node).

```
/* JOINER */
 –joinLowerLevel( host )
   send JoinLowerLevelRequest<self> to host
        ↳onFailure()
            rejoin Geodupa
        ↳onReply( JoinLowerLevelReply<seed> )
            initAsSubNode( seed )

/* HOST */
 ↳onReceive( JoinLowerLevelRequest<joiner> )
   seed ← NeighborhoodDB.getSeedNode( joiner )
   reply JoinLowerLevelReply<seed> to joiner
   foreach super_node in SuperNodeDB
    if areasIntersect(super_node, joiner)
      send SeedDispatch<joiner> to super_node
   multicast joiner
```

Figure 4.7 illustrates an example of the distributed tree generated while announcing a new node. The *root* dispatches the announcement to four different nodes (one for each quadrant). While the three leftmost *inner* nodes continue with the subdivision of their assigned quadrants, the rightmost only knows two nodes in its assigned quadrant. It thus treats them as *leaf* nodes by notifying them directly.

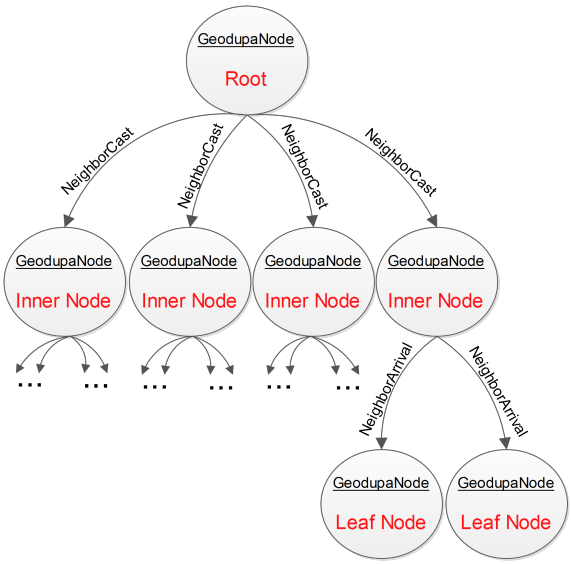


Figure 4.7: Announcement process.

Depending on the destination node (*inner* or *leaf*), the message types involved are different. A ***NeighborArrival*** is used to directly dispatch the announcement to *leaf nodes*. It contains references to the *host* and to the node being announced, along with the time-stamp marking the beginning of the announcement process. The ***NeighborCast*** contains the information necessary to assemble the distributed tree. Namely, a *level*, a *quad* and a *path*. It also encapsulates a *NeighborArrival*, which enables *inner* nodes to acknowledge the node being announced.

The *level* represents the actual level of the distributed tree and it is incremented before a *NeighborCast* is dispatched. The *path* is the list of nodes already notified, i.e., where the message has already passed. It helps to avoid false-positives as the message is never forwarded to a node contained in the *path*. Finally, the *quad* is an instance of class *Quad*, which remits this discussion to the introduction of two auxiliary classes: *Quad* and *NeighborRange*.

Class ***Quad*** represents a rectangle in the physical space. It resorts to the *Rectangle2D.Double* Java object to perform the necessary geometric operations (e.g., *divide()*, *containsNode()*).

An instance of the class ***NeighborRange*** represents an interval in the node identifier ring defined by a random pair of boundaries - low (L) and high (H). It can be perceived as a mask from which the *NeighborhoodDB* can produce a set of nodes. The random nature of these boundaries allows the representation of a different set every time this class is instantiated, thus contributing to the randomness of the distributed tree.

Procedure 5 presents, in pseudo-code, the multicast process. A *inner* node that receives a *NeighborCast* first processes the *NeighborArrival* within in order to add the node being announced to its *NeighborhoodDB*. The *Quad*, representing the region the *inner* node is responsible for, is then dealt with. First, the neighbors left to notify are computed by creating a new *NeighborRange*. It provides the method *nodeList* that returns the set of nodes whose physical location is contained in a given *Quad*. From this set, nodes that have already been notified (contained in *path*) are removed.

If the number of nodes left to notify is greater or equal to 4, the *Quad* is subdivided into 4 equally sized *subquads*. Like before, a set of candidates is computed for each one *Quad* and, from each set, a node is randomly selected to deal with it. Next, a *NeighborCast* is created for each *subquad*. The level of the received *NeighborCast* is incremented by one and associated to each new message. The current *inner* node is appended to the *path* and the messages are finally dispatched. Each receiving node then recursively handles the *Quad* it was entrusted.

Conversely, if the number of nodes left to notify in the current *Quad* is less than 4, they are treated as *leaf nodes* and directly notified by dispatching of a *NeighborArrival* message to each one. This message is processed by attempting the storage of the *host* and the announced node.

Procedure 5 Geographical multicast in Geodupa.

```
↳onReceive NeighborCast( quad, path, level, NeighborArrival<host, timestamp, node>)
1  send NeighborArrival<host, timestamp, node> to self
2  nodes_left ← NeighborRange.nodeList( quad )
3  remove path from nodes_left
4  if nodes_left.size ⩾ 4
5     subquad1..4 ← quad.divide()
6     foreach subquad in { subquad1..4 }
7        candidates ← NeighborRange.nodeList( subquad )
8        foreach n in candidates
9        if n ≠ node ⋀ inRange(n, node) ⋀ n ∉ path
10          send NeighborCast< subquad, path∪self, level+1, NeighborArrival<host, timestamp, node> > to n
11             ↳ onFailure
12                continue
13             ↳ onSuccess
14                break
15 else
16    foreach n in nodes_left
17       if n ≠ node ⋀ inRange(n, node)
18          send NeighborArrival<host, timestamp, newnode> to n

↳onReceive( NeighborArrival<host, timestamp, node> )
1  NeighborhoodDB.store( host∪node, timestamp )
```

4.4.2.3 Epidemic Repair

The epidemic repair process is implemented in a *PeriodicTask* and involves two instances of *GeodupaNode*: the ***requester*** and the ***replier***. The exchange of information is performed by encapsulating sets of the class *Neighbor* in two messages - *RepairRequest* and *RepairReply*.

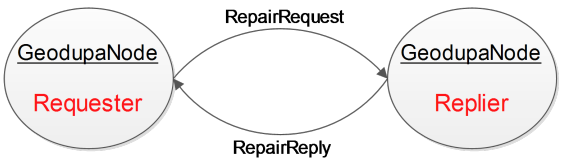


Figure 4.8: Epidemic Repair process.

Geodupa nodes are periodically taking the initiative to execute the epidemic repair process. Figure 4.8 illustrates this process. The *requester* obtains, from its *NeighborhoodDB*, the set of neighbors in range of the *replier* and dispatches it in message *RepairRequest*. The *replier* then

compiles the set of nodes in range of the *requester*, that were not contained in the request, and sends it in message *RepairReply*. The behaviors of both nodes are described in procedure 6. Note that the *requester* side is implemented in a *PeriodicTask*.

Procedure 6 Epidemic repair process.

```
/* REQUESTER */
–repairGeodupa()
  ↻onTimer( REPAIR_PERIOD)
    other ← NeighborhoodDB.getRandomNeighbor()
    Neighbor[]A ← NeighborhoodDB.neighborsInRangeOf( other)
    send RepairRequest< self, Neighbor[]A > to other
      ↳onFailure()
        continue
      ↳onReply( RepairReply<Neighbor[]B> )
        NeighborhoodDB.store(Neighbor[]B)

/* REPLIER */
↳onReceive( RepairReply< requester, Neighbor[]A > )
  foreach neighbor ∈ Neighbor[]A
    NeighborhoodDB.store(neighbor)
  Neighbor[]B ← NeighborhoodDB.neighborsInRangeOf( other)
  remove Neighbor[]A from Neighbor[]B
  reply RepairReply<Neighbor[]B> to requester
```

4.4.2.4 Departures

As mentioned in section 3.2.6, the departure of nodes is dealt in a simple, while effective, way. Departed nodes are removed from a given node's *NeighborhoodDB* when that node attempts to directly contact them. Additionally, nodes are also removed from a node's *NeighborhoodDB* when their time-stamps are older than a given time threshold. This behavior is implemented in a *PeriodicTask* and, at each run, this task resorts to the method *garbageCollector* provided by the *NeighborhoodDB*. Procedure 7 presents the contents of this method in pseudo-code.

Procedure 7 Neighborhood garbage collector.

```
–garbageCollector()
  foreach neighbor
    if currentTime – neighbor.timestamp > NEIGHBOR_TTL
      discard neighbor
```

4.4.2.5 Promotions

Whenever a super-node leaves the network, the promotion mechanism comes into play in order to readily find a suitable replacement. It is triggered by a sub-node that does not receive evidence of super-nodes in its range for a certain period of time. Figure 4.9 illustrates the instances of the GeodupaNode class involved in this procedure and the messages exchanged. Note that several candidates may be contacted until one accepts to promote itself.

A node is able to suspect that no super-nodes cover its location by keeping reference to the last super-node in range it "heard" from and a *double* (***lastAnnouncementReceived***) with the last moment in time it did. It is updated during the epidemic repair or whenever a *NeighborCast* initiated by a super-node in range is received.

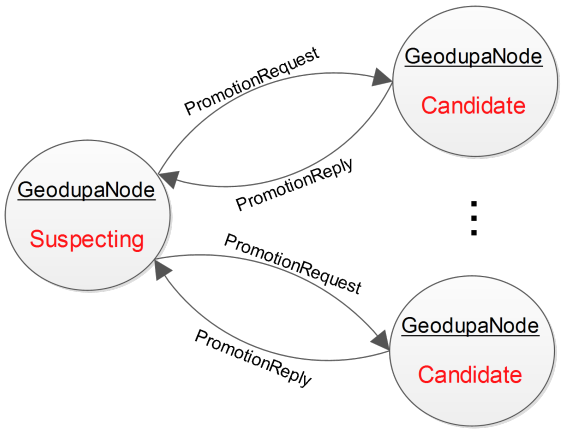


Figure 4.9: Promotion process.

Every super-node periodically checks when was the last moment in time it sent an announcement (***lastAnnouncementSent***). If it occurred before a given time threshold then it simply announces itself so that other nodes acknowledge its presence and delay their suspicion.

Procedure 8 presents the pseudo-code of the *PeriodicTask*s implementing both behaviors. Super-nodes checking when was the last moment in time they sent an announcement. Sub-nodes checking for the last time they received one.

The actual promotion process is presented in procedure 9. The suspecting node compiles the set of candidates to be promoted - nodes closer to the last "seen" super-node - and sends a *PromotionRequest* to the first one. If it replies unsuccessfully, the next candidate is contacted. This continues until one candidate accepts to be promoted or there are no more candidates, in which case the suspecting node promotes itself. Nodes that receive the request only accept to promote if they are also suspecting. In order to minimize super-node concurrency, the promoting node does not readily join Catadupa. Instead, it schedules a *Task* to execute after a pre-determined time delay - the *promotion tolerance*. This allows it to stop promoting if, meanwhile, it receives evidence of a super-node in the vicinity.

Procedure 8 Promotion mechanism - announcement checking.

```
/* SUPER-NODES */
–checkLastAnnouncementSent()
  ↺onTimer( HEART_BEAT_TIMEOUT)
    if currentTime − lastAnnouncementSent > HEART_BEAT_TIMEOUT
      announce self

/* SUB-NODES */
–checkLastAnnouncementReceived()
  ↺onTimer( SUSPITION_TIMEOUT)
    if currentTime − lastAnnouncementReceived > SUSPITION_TIMEOUT
      start promotion process
```

Procedure 9 Promotion protocol.

```
–promotionRequest()
  candidates ← NeighborhoodDB.getPromotionCandidates( lastSuperNodeAcknowledged)
  loop
    if candidates isEmpty
      join Catadupa
      stop
    else
      node ← candidates.removeFirst()
      send PromotionRequest<> to node
        ↳onFailure()
          continue
        ↳onReply( PromotionReply<answer> )
          if answer equals "ACCEPTED"
            lastAnnouncementReceived ← currentTime
            lastSuperNodeAcknowledged ← node
            stop
          else
            continue

↳onReceive( PromotionRequest <> )
  if currentTime − lastAnnouncementReceived > SUSPITION_TIMEOUT
    reply PromotionReply< "ACCEPTED" >
    wait PROMOTION_TOLERANCE
    join Catadupa
  else
    reply PromotionReply< "REJECTED" >
```

5 . Experimental Evaluation

This chapter describes the procedures involved in the validation of Geodupa. The goal is to experimentally perceive its behavior under different parameter configurations.

We first present, in section 5.1, the network configurations that will enable the experimental study of Geodupa. Next, in section 5.2, the algorithm's behavior is observed by defining and evaluating some relevant metrics .

5.1 Preamble

This section establishes the ground basis for the evaluation of Geodupa. Namely, the simulation area on which the network will be assembled (section 5.1.1); the network global size and the influence of the neighborhood radius on the number of super-nodes necessary to cover the physical space (section 5.1.2); finally, in section 5.1.3, we establish the network configuration adopted to validate Geodupa, considering the several relevant metrics presented in section 5.2.

5.1.1 Simulation Area

Ideally, the area of simulation would comprise the entire land surface of planet Earth which, as one would expect, would involve large numbers of nodes. The fact is that the computational power available is unable to support simulations of this nature in an acceptable time frame. For this reason, Geodupa is only evaluated on a sample area, designated as the **simulation area**, so that the network size can be significatively reduced.

The creation of new Geodupa nodes was then restricted to a rectangle on the Earth's surface whose area equals 25.000 km^2 (approximately a quarter of the area of Portugal). The process of assigning coordinates to nodes was thus inverted. That is, a random set of geographical coordinates (latitude, longitude) is first generated in the simulation area. These are then converted to the cartesian coordinates (x, y, z) used to calculate distances between nodes.

5.1.2 Network Size

The global size of the network is closely related to the network churn [24], a phenomenon characterized by the recurring arrival and departure of nodes. The churn model of Geodupa is defined by two probabilistic distributions, one *Exponential* distribution to model the delay between node arrivals (**arrival rate**), and one *Weibull* distribution (*shape* = 1,8) to obtain the **session duration** of each node, with a maximum of 8 hours.

Various simulations were then run in order to observe the impact of these variables on the overall network size. Two session durations and three arrival rates were considered. The results are presented in table 5.1 and show that either increasing the arrival rate or the session duration results in an approximately proportional increase of the network's size.

Session Duration (h)		Arrival Rate (nodes/s)	Network Size (avg. # of nodes)
Mean	Max.		
2	4	0.1	656
		0.5	3284
		1	6606
4	8	0.1	1309
		0.5	6567
		1	13220

Table 5.1: Geodupa's global network size.

5.1.2.1 Super-node coverage

Previously, we observed the churn influence on the global network size. Now the focus shifts to the radius of the neighborhoods and how it influences the number of super-nodes necessary to completely cover the simulation area. To do so, the churn model is set to the configuration presented in table 5.2, producing a network of about 10.560 nodes, and the simulation area is adjusted to 25.000 Km^2 (as described in section 5.1.1).

Several simulation runs were then performed in order to determine the impact of the neighborhood radius on the size of the higher level. The experimental results are presented in table 5.3 and clearly show that, as the neighborhood radius increases, the amount of super-nodes needed to cover the simulation area decreases. The remaining nodes are then assigned as sub-nodes.

Session Duration (s)		Arrival Rate (nodes/s)	Network Size (avg. # of nodes)
Mean	Max		
4	8	0.8	10560

Table 5.2: Churn model adopted to validate Geodupa.

Neighborhood Radius (km)	Neighborhood Area (km^2)	Super-Nodes (#)	
		Optimal	Observed
2,5	19.6	1273.2	1970
5	78.5	318.3	714
10	314.2	79.6	197
20	1256.6	19.9	64
40	5026.5	5.0	21

Table 5.3: Impact of the neighborhood radius on a 25.000 Km^2 area.

The **optimal** number of super-nodes is the minimum amount necessary to completely cover the space. It is obtained by dividing the total simulation area by the area of a single neighborhood: $\frac{A_{total}}{A_{neigh.}}$, with $A_{neigh.} = \pi r^2$. However, the number of super-nodes observed is higher than the optimal. Moreover, since the area of the neighborhoods quadruple by doubling the radius, it would be expectable that the number of super-nodes also decreased by a factor of 4. We observed that these only decrease by an average factor of 3. Both these results suggest a loss of efficiency in covering the physical space, which can be justified by the following reasons:

- It is not possible to completely cover a rectangular area with circles without overlapping.
- Super-node concurrency also contributes, to some extent, to neighborhood overlapping.
- Neighborhoods that intersect the boundaries of the simulation area do not contribute with their entire areas to the coverage of the space.

In order to obtain an approximate number for the super-nodes necessary to completely cover the simulation area, we derived an empirical expression (presented in 5.1) that returns values with a 10% margin of error, based only on the values experimentally obtained (table 5.3). The constants presented were introduced and adjusted in order to approximate the number of super-nodes returned to the number of super-nodes experimentally observed. Note that it was only tested and remains valid for a radius interval ranging from 2,5 to 40 Km and for an area of 25.000 Km^2. For other configurations, these constants would have to be readjusted.

$$\#SuperNodes = \frac{A_{total}}{\pi \times r^{1.65}} \times 1.2 \tag{5.1}$$

.

5.1.3 Setting the configuration

Ideally, Geodupa would be evaluated under several network conditions as it would be interesting to obtain its behavior under various neighborhood sizes (by varying the churn model or the neighborhood radius). The fact is that the time frame available does not allow it and, from now on, the process of validating Geodupa considers only one network configuration.

The churn model has already been established in table 5.2. It is defined by an arrival rate of 0,8 nodes/sec and an average session duration of 4 hours (8 hours max.). The result is a network composed by about 10560 nodes, on average.

The simulation area, mentioned in section 5.1.1, has 25.000 Km^2 and the neighborhood radius equals 10 Km. These, in combination with the churn model adopted, result in a network with about 197 super-nodes and, consequently, about 10363 sub-nodes. Through simulation, we observed that each neighborhood contains, on average, 125 nodes.

From now on, the collection of statistics is performed by executing simulation runs that last for 16 hours, a sufficient amount of time for the metric values to stabilize. The goal is to evaluate the behavior of Geodupa in stable conditions, therefore, the collection of statistics is not performed during the first 8 hours (*warmup* period) of simulation.

5.2 Evaluation Metrics

We now iterate through the various aspects related to the behavior Geodupa. The membership dissemination mechanisms are evaluated in subsection 5.2.1. The excess of entries in the neighborhood databases is observed in 5.2.2. Section 5.2.3 discusses super-node concurrency and how to minimize it. Finally, section 5.2.4 considers Geodupa's bandwidth requirements.

5.2.1 Membership Dissemination

This section evaluates the usefulness of the mechanisms used to disseminate membership information: the *geographical multicast*, used to announce new nodes (section 5.2.1.1), and the *epidemic repairs* (5.2.1.2). Section 5.2.1.3 finally presents the motivation to combine both.

5.2.1.1 Announcements (*geographical multicast*)

The dissemination of a new arrival is performed by readily multicasting it to the relevant peers. The problem with this approach is that it is not 100% accurate, i.e., not all the interested peers receive the announcements they should. Besides, taking the neighborhoods' management into account, namely the need to constantly revalidate the knowledge of the surrounding peers (remember the time-stamp based philosophy from section 3.2.5), this mechanism is not expected to alone provide Geodupa nodes with an accurate and updated view of their neighborhoods.

The simulation runs performed using only the multicast mechanism to disseminate membership information show that, on average, each node only knows **20%** of its active neighbors. Moreover, since the membership view is so inaccurate, we observe an average multicast error of **64%**, meaning that each node announcement only reaches about one third of the target peers.

These results render the multicast not suitable to alone be responsible for the membership management of Geodupa. The fact is that the its main contribution is the rapid announcement of new nodes (as will be observed in section 5.2.1.3). Also, it does not enable them to acknowledge previously existing neighbors, leading the neighborhood database of a node to only contain neighbors that joined Geodupa subsequently.

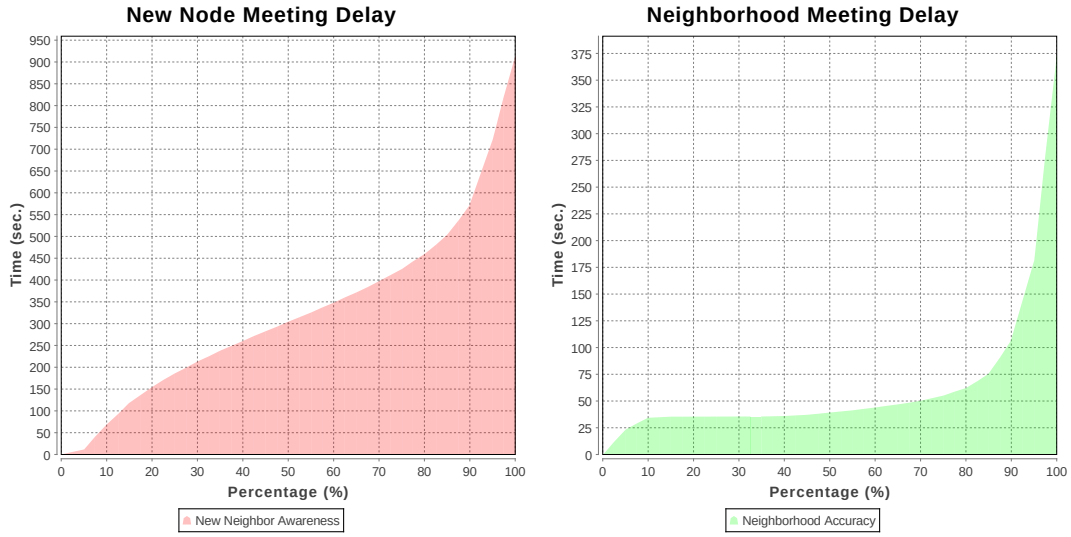
5.2.1.2 Epidemic Repair

Previously, we evaluated the neighborhood accuracy of Geodupa nodes using only the multicast mechanism. The same notion is now applied and the epidemic repair mechanism's ability to alone manage the membership information is questioned.

Remember, from section 3.2.3, that the multicast mechanism is used to perform two distinct tasks when announcing new nodes. First, there is the announcement of a new sub-node to the

lower level, triggered by its *host*. Second, new super-nodes are announced by other super-nodes when their neighborhoods intersect. Disabling these will expectably result in a longer delay for new sub-nodes and super-nodes to be respectively acknowledged by their neighbors. Note that the multicast mechanism is still used by super-nodes to announce themselves (*heart-beats*). This instance of the multicast was not disabled as it relates to the promotion process and doing so would lead to a significant increase in the percentage of concurrent promotions.

The simulation performed considered a 2 minute epidemic repair period and revealed a neighborhood accuracy of around **95%**. This result suggests that Geodupa could in fact operate resorting only to epidemic repairs. However, we expect that it would be slow to disseminate new nodes. To support this assumption, figure 5.1 presents the time necessary for a new node to be progressively acknowledged by its neighbors (5.1a). For future comparison, it also presents the opposite, i.e., the delay verified for a new node to know its neighborhood (5.1b).



(a) New node. (b) Neighborhood.

Figure 5.1: Membership dissemination delay (Epidemic repairs only).

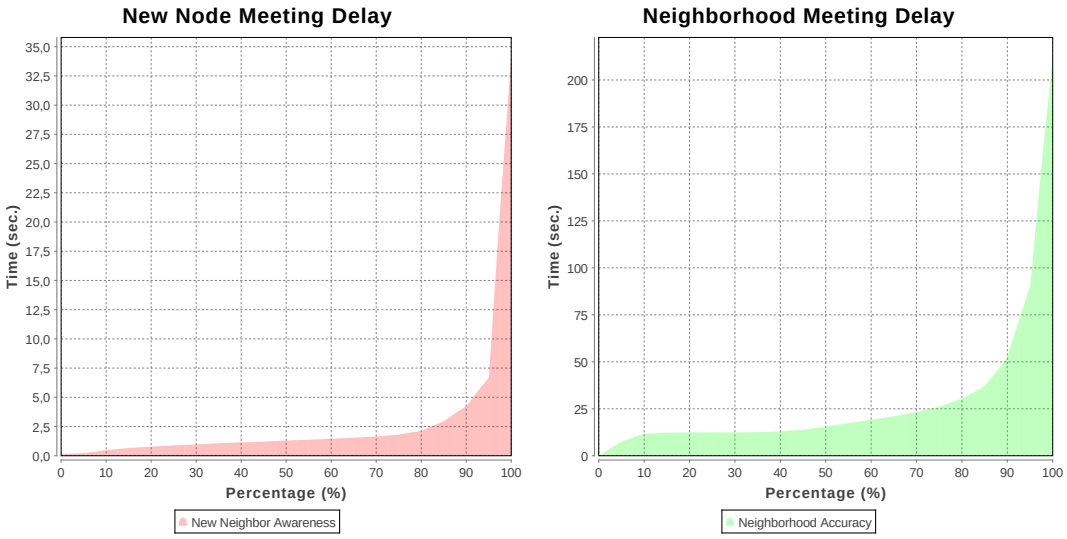
As expected, the time necessary for a new node to be disseminated is high. It takes, on average, almost 900 seconds (15 minutes) for a node to be known by its entire neighborhood. Even if increasing the epidemic rate would result in a faster dissemination, these results suggest that combining the epidemic repair with the multicast mechanisms seems an approach to consider. The following section (5.2.1.3) addresses this.

5.2.1.3 Multicast and Epidemic Repair

We previously determined that both the multicast and the epidemic repair mechanism are not quite suitable to correctly and efficiently manage the membership dissemination. This section thus evaluates Geodupa using both mechanisms. The multicast was enabled in all the procedures it is involved in and every node performs an epidemic repair every 2 minutes.

We observed an average neighborhood accuracy of about **98%**. The multicast correctness also improved when comparing to the multicast-only approach. Its error decreased to around **25%**, meaning that about three quarters of the relevant nodes are immediately notified whenever a new neighbor arrives. This is justified by the fact that the epidemic repair mechanism's ability to recover the past and missed announcements results in a more accurate neighborhood view, which in turn leads to the ability to build multicast trees that span more nodes.

Figure 5.2 presents the membership dissemination time delays. As expected, the most visible improvements reside in the new node dissemination delay (5.2a). Moreover, since new nodes are readily announced, their notified neighbors will select them to repair their databases in a shorter time delay, allowing the new node to progressively acknowledge its neighborhood more rapidly (5.2b) than in an epidemic-only approach (5.1b).



(a) New node. (b) Neighborhood.

Figure 5.2: Membership dissemination delay (Multicast and Epidemic repairs).

We thus conclude that combining the multicast mechanism with epidemic repairs is a suitable approach to endow Geodupa nodes with an accurate and fresh view of their neighborhoods.

Finally, note that one could still increase the rate at which nodes perform the epidemic repairs in order to obtain even better results. Due to time restrictions, this matter was not properly evaluated, however, its consequences are somewhat obvious: better results at the expense of higher bandwidth consumption.

Moreover, even if not implemented, there are actually a few other ways that could enhance the epidemic repairs, specially to allow new nodes to know their previously existing neighbors in a quicker way. For instance, once a new node joined, it could perform epidemic repairs more frequently in order to rapidly acknowledge its neighborhood. As time elapsed, these could be executed with progressively larger intervals until a stable rate was reached. Another way would be to also consider the physical distance as a way to influence the selection of the repair partner. That is, a node could choose closer neighbors more frequently as their neighborhoods are more alike and would thus provide a better contribution.

5.2.2 Database Excess

The neighborhood database management leads to the appearance of some excess nodes. Remember, from 3.2.6, that a neighbor is only discarded whenever it is unsuccessfully contacted or when its *time-stamp* is older that a given time-threshold - the *neighbor TTL*. This results in the existence of some database entries corresponding to neighbors that have already departed.

Table 5.4 presents the nodes' neighborhood status for three different values for the *neighbor TTL*: 40min., 20min. and 5min. The average neighborhood accuracy is presented and the size of the nodes' databases is compared with the real number of nodes in each neighborhood. This way it is possible to determine which *neighbor TTL* brings them closer together.

Neighborhood TTL (min.)	Neighborhood Accuracy (%)	Neighborhood Size (#)	
		Databases	Real
40	99.5	149 (+16%)	125
20	98.4	134 (+7%)	
5	58	72 (-42%)	

Table 5.4: Impact of varying the *neighbor TTL*.

The *neighbor TTL* clearly influences the size of the neighborhood databases. For a *TTL* of 40 min., we verify an excess of nodes of about 16%, meaning that only 84% of the database entries represent active neighbors. In order to reduce the excess, the *neighbor TTL* was reduced to 20 min. so that old entries are removed faster. This leads the excess to drop to 7%. Finally, for a *neighbor TTL* of 5 min., the databases are smaller than the real neighborhoods, meaning that the neighbors are being removed too fast. In fact, 42% of the active neighbors are absent.

Clearly, reducing the *neighbor TTL* contributes to reduce the excess of nodes. However, if it is set too low, the neighborhood accuracy is negatively affected as some active neighbors are removed prematurely. Under the network conditions set in 5.1.3, our recommendation for the *neighbor TTL*, of the ones considered, is 20 min. It minimizes the excess and still allows nodes to maintain a high neighborhood view accuracy.

5.2.3 Super-node concurrency

Super-node concurrency is a phenomenon that may occur due to one of two situations: when a new node is incorrectly assigned to the higher level by the joining process, or when a node erroneously promotes itself. While the first is dependant on the awareness that super-nodes have from each other, i.e. Catadupa's membership management, the second one essentially depends on three parameters: *promotion tolerance*, *heart-beat timeout* and *suspicion timeout*.

Promotion tolerance is the time it takes for a node to actually promote itself, from the moment it decided it was going to. ***Heart-beat timeout*** is the maximum amount of time that a super-node, in the absence of new arrival events to announce, waits before it announces itself to the lower level. Finally, the ***suspicion timeout*** is the maximum amount of time that a sub-node waits for evidences of active super-nodes in range before triggering the promotion mechanism.

In order to evaluate the impact of these variables on super-node concurrency, we decided to establish a fixed *suspicion timeout* of 6 minutes as it is a reasonable amount of time for a sub-node to be without super-nodes in range. The values for the *heart-beat timeout* and *promotion tolerance*, as well as the simulation results, are presented in table 5.5. Note that all these parameter values are somewhat arbitrary and were chosen experimentally.

Heart-beat Timeout (min.)	Promotion Tolerance (sec.)	Concurrency (%)		
		Promotions	Joins	Total
3	60	45.7	4.5	34.8
	120	27.3	3.1	16.5
2	60	11.0	3.9	8.5
	120	3.8	5.1	5.8

Table 5.5: Super-node Concurrency.

The first result to highlight is the fact that the percentage of **concurrent joins**, of Catadupa's responsibility, remains somewhat constant around 4%. An acceptable result, showing that Catadupa effectively endows its participants with an accurate membership view.

As for the concurrency resulting from the promotion mechanism, note that the *heart-beat timeout* plays a sensitive role. Changing from 3 to 2 minutes produces a substantial drop in the percentage of **concurrent promotions**. Since super-nodes announce themselves more frequently, their surrounding sub-nodes are more likely to receive at least one announcement every 6 minutes (*suspicion timeout*). One can thus tune the *heart-beat timeout* in order to obtain an even lower concurrency, at the expense of an increased bandwidth consumption.

Increasing the *promotion tolerance* also leads to a concurrency reduction. However, it also leads to a significant reduction in the overall number of promotions. This is caused by the fact that, while sub-nodes are waiting to promote, they are more likely to have their promotion plans canceled by new super-nodes joining the *orphan* area. Although this is not a correctional problem, it goes against the main motivation of the promotion mechanism: avoid the necessity to wait for new nodes to cover *orphan* areas.

In conclusion, the values obtained suggest that it is possible to manipulate the percentage of super-node concurrency by tuning the mentioned parameters. Note that, due to time limitations, this evaluation considered only one value for the *suspicion timeout*. Still, we expect that increasing this parameter will result in less concurrent promotions, as long as the other parameters are adjusted accordingly. However, like increasing the *promotion tolerance*, this also leads to a reduction in the overall number of promotions.

5.2.4 Bandwidth Consumption

This section presents some considerations on the bandwidth required from Geodupa nodes. In section 5.2.4.1 we first present an empirical study determining the impact of the neighborhood radius on the overall expected bandwidth. Section 5.2.4.2 then confirms the expectations by presenting the overall bandwidth consumptions considering different values for the radius.

5.2.4.1 Influence of the neighborhood radius

This dissertation strived to design an algorithm that, by restricting node visibility to geographical neighborhoods, reduces the cost of full-membership solutions. To assert if the solution proposed actually succeeds, we must first determine the influence of the neighborhoods' size in the overall bandwidth consumption, as a way to obtain the one that minimizes it.

The bandwidth required from each node is closely related to the amount of peers they need to know. It is thus related to the churn phenomenon, particularly with the arrival rate. A node that monitors many arrivals consumes more bandwidth than one who monitors less. To that end, we start by defining the percentage of the total arrival rate (T_T = 0,8 nodes/s) that concerns each node. This percentage has to be derived for both levels of Geodupa.

Since nodes in the **higher-level** need to be aware of every other peer in that level, their cost is related to the fraction of the global arrivals related to new super-nodes: $T_H = T_T(\frac{\#Nodes_{super}}{\#Nodes_T})$.

In turn, nodes belonging to the **lower-level** only need to monitor arrival events occurring in their neighborhoods, leading to a fraction of the arrival rate equal to the area percentage that each neighborhood represents in the total area: $T_L = T_T(\frac{Area_{neighb.}}{Area_T})$.

After establishing the arrival rate concerning each level, it is necessary to multiply it by the cost of disseminating a new node - $\boldsymbol{U}$. Obtaining this value is not trivial as node dissemination involves several procedures (announcements, epidemic repairs), leading to its variation. Moreover, there is still the uncertainty introduced by communication overhead (packet headers, handshakes, acknowledgements).

The goal of this section is not to obtain the cost itself, but to obtain an idea of how the neighborhood radius influences the total cost. We thus consider only the cost of disseminating a node in an optimal way. Any kind of dissemination redundancy or communication overhead is

not considered. The cost (U) is therefore set to 22 Bytes: 6 for the endpoint (ip + port) plus 16 for the geographical coordinates (latitude + longitude). Note that in this evaluation we present only the upload cost as the download is quite similar [1].

The total average upload cost demanded from each Geodupa node is thus obtained by expression 5.2. $\boldsymbol{T}_T$ is the total arrival rate (0,8 nodes/s); $\boldsymbol{U}$, the minimal upload cost (22 Bytes).

$$U_{total} = [\underbrace{T_T(\frac{\#Nodes_{super}}{\#Nodes_T})}_{\mathrm{T}_H} + \underbrace{T_T(\frac{Area_{neigh.}}{Area_T})}_{\mathrm{T}_L}] \times U \tag{5.2}$$

From section 5.1.2.1, the number of super-nodes can be obtained, with a 10% margin of error, by an empirical expression derived by observing the size of the higher-level as a function of the radius. Since it was only tested for values between 2,5 Km and 40 Km, only this range is considered when obtaining the overall cost of Geodupa. Figure 5.3 presents the results.

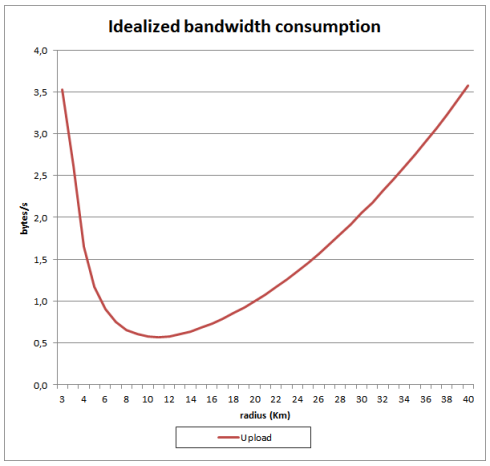


Figure 5.3: Behavior of the overall cost in function of the neighborhood radius.

For a smaller radius, there are many super-nodes maintaining full-membership. This results in a high overall cost composed mainly by the cost of maintaining the higher-level. Since neighborhoods are composed only by a few sub-nodes, the bandwidth necessary to maintain them is lower. Ultimately, the radius would be so small that the network would only be composed by super-nodes, in which case the overall cost would be similar to the cost of maintaining a Catadupa network.

[1] The multicast balances the upload/download ratio by the usage of random trees. In the epidemic repairs, the bandwidth that a given node uses to upload information, is the same that other node uses to download it.

As the radius increases, the number of super-nodes decreases and so does the cost of maintaining the higher-level. Conversely, since each node has progressively more neighbors, the cost of maintaining full-membership neighborhoods in the lower-level increases.

For a larger radius, since the number of super-nodes is low, the higher-level contributes less to the overall cost. On the other hand, larger neighborhoods lead to a higher lower-level cost and to its predominance in the overall cost. Ultimately, the neighborhood radius would be big enough so that only one super-node existed. In which case the cost of the higher-level would tend to zero and the lower-level would eventually reach a full-membership scenario.

We thus conclude, by observing figure 5.3, that the neighborhood radius that minimizes the overall cost of Geodupa, for the network configuration considered (section 5.1.3) and under optimal dissemination conditions, equals approximately 10 Km.

5.2.4.2 Overall bandwidth consumption

The previous section (5.2.4.1), determined the influence of the neighborhood radius in the overall expected cost and established an approximate one that expectably minimizes Geodupa's overall cost. We now confirm it experimentally by approximately considering communication overhead (packet headers, handshakes, acknowledgements) in message exchange.

Three simulation runs were performed in which only the traffic accounted by nodes that joined after the *warmup* period (8 hours) was considered. Table 5.6 presents the upload average bandwidth consumption categorized by level. For the lower-level we further distinguish between multicast announcements and epidemic repairs. The join and promotion protocols are not presented as their cost is insignificant (always lower than 0,15 B/s and 0.01 B/s, respectively). They are, however, accounted in the lower-level total.

Neighborhood Radius (Km)	Higher-Level	Lower-Level			Total
		Multicast	Repairs	Total	
2.5	40.1	3.22	3.38	6.7	46.8
10	10.9	6.32	21.65	28.1	39.0
20	7.2	13.01	78.95	92.1	99.3

Table 5.6: Geodupa's average upload consumption (Bytes/s).

From the three values considered, it is clear that the 10 Km neighborhood radius actually represents a minimum for the total cost of Geodupa. As expected, the cost for a 20 Km radius is mainly introduced by the lower-level. Conversely, for a 2,5 Km radius, the cost to maintain the higher-level prevails.

Note, however, that the total cost for the 20 Km radius was expected to be lower than the one for the 2,5 Km (from figure 5.3). In fact, we also expected a steeper drop when changing from 2,5 Km to 10 Km. Such expectations are not observed due to the rapid growth verified in the cost of the lower-level. While the cost of the higher-level decreases almost proportionally with the radius, the cost of the lower-level increases exponentially. This is mainly caused by the inefficiency of the epidemic repairs and suggests that the best radius is actually smaller. Through experimental evaluation, we in fact determined that the radius that minimizes the overall cost of Geodupa is around **7 Km**. It results in an average upload consumption of about **34 B/s**.

The repair inefficiency is caused by the fact that, at each interaction, both repair participants exchange every single neighbor they know in each other's neighborhoods. This leads the cost to grow rapidly with the neighborhoods' size. These results suggest that, besides the improvements suggested before (section 5.2.1.3), the epidemic repair mechanism should also be enhanced in a way that the amount of information exchanged could be reduced. For instance, instead of sending every single neighbor in range of the repair partner, each node could send only a subset, in which newer and/or closer neighbors could be given priority.

To conclude this section, we finally consider the overall cost of a Catadupa network operating under the same network conditions set in section 5.1.3. It can be perceived as a Geodupa network composed only by super-nodes with no notion of geographical neighborhoods and all aware of each other. The overall upload cost observed is about **65 B/s**, confirming that maintaining full-membership for 10560 nodes (on average) results in a higher average bandwidth consumption than the one verified for Geodupa with a 7 Km neighborhood radius - **34 B/s**. The reduction verified (approximately **48%**) is expected to become even more significant for larger networks. The results obtained clearly show that it is possible to use Geodupa in order to achieve less bandwidth requirements than a full-membership solution, as long as the neighborhood radius minimizes the overall cost.

6. Conclusion

In a world where mobile devices play an ever increasingly important role, a new application domain has been gaining more notoriety. Designated as *Participatory Sensing*, it relies on the technological evolution of mobile devices, and their increasing ability to collect sensorial information, as a way to form wide-area user networks that monitor the physical world. These networks need an underlying infrastructure so that users can access and share the collected information.

Despite the ongoing breakthroughs on mobile technology, processing and energy capabilities are still relatively limited to support a network composed solely by mobile devices that independently collect and share information. For this reason, we considered a network of Personal Computers to act as servers to the mobile devices and share the information collected by them. It is a way to solve the resource limitation problem while also maintaining the *Participatory Sensing* community driven philosophy.

This dissertation thus presented an algorithm to act as the underlying substrate for a network of *Participatory Sensing* driven applications, running on the user's Personal Computers. Designated as Geodupa, it divides the network into two hierarchical levels composed by super-nodes (higher level) and sub-nodes (lower level). While super-nodes know every peer in the higher level, every sub-node is only concerned with the ones in its close surroundings.

Every node has the ability to perform one-hop lookups in its neighborhood as they are all aware of its neighbors. Such awareness is provided by the announcement of new arrivals and an epidemic repair mechanism. Through simulation we observed that these mechanisms perform effectively, even if their assurances are only probabilistic.

By restricting node visibility to geographical neighborhoods it is not only possible to achieve low latency communication between closely located nodes, but also to reduce the costs of membership dissemination as a new node arrival only generates traffic in a fraction of the network.

The ability to reach farther nodes was not discarded as each sub-node must have at least one super-node in range. Since super-nodes are connected in a full-membership network (managed by Catadupa) spanning the entire network physical area, the communication between nodes located far away is still possible, even if not in a single hop.

6.1 Main Contributions

- A partial membership algorithm designed to manage a network of georeferenced nodes that supports applications based on the premise that important information is closer.
- An implementation of the proposed algorithm, on a simulation environment, in order to enable its experimental evaluation and future work assessment.
- A behavior appraisal through experimental evaluation, taking several relevant metrics into account.

6.2 Future work

Even though the main objective of this dissertation was fulfilled, the fact is that some work is still in need of consideration. Namely:

- Introduction of new features to the algorithm proposed. For instance, the ability to operate with variable neighborhood radiuses or provide nodes with some awareness of their distant peers to enable lower latency distant lookups.
- Updating existing features as a way to improve the algorithms's efficiency in terms of its bandwidth requirements. Namely, the epidemic repair mechanism.
- A performance evaluation concerning both the introduction of new features and the impact of the enhancements to the existing ones.

A. Annex

A.1 Message Handlers

```
interface GeodupaSocketHandler {
  void onReceive(Socket call, JoinGeodupaRequest m);
  void onReceive(Socket call, JoinLowerLevelRequest m);
  void onReceive(Socket call, SeedDispatch m);
  void onReceive(Socket call, NeighborsRepairRequest m);
  void onReceive(Socket call, NeighborCast m);
  void onReceive(Socket call, NeighborArrival m);
  void onReceive(Socket call, PromotionRequest m);
}
```

Listing A.1: Handler for received messages.

```
class GeodupaReplyHandler{
    public void onFailure() {}
    public void onReply(Message m) {}
    public void onReply(Socket call, Message m) {}

    public void onReply(JoinGeodupaReply m) {}
    public void onReply(JoinLowerLevelReply m) {}
    public void onReply(NeighborsRepairReply m) {}
    public void onReply(PromotionReply m) {}
}
```

Listing A.2: Handler for message replies.

A.2 Geodupa Messages

Message Type	Data Members
JoinGeodupaRequest	GeodupaNode requester;
JoinGeodupaReply	boolean supernode; GeodupaNode host;
JoinLowerLevelRequest	GeodupaNode requester;
JoinLowerLevelReply	boolean accept; GeodupaNode seed;
SeedDispatch	GeodupaNode seed;
NeighborCast	int level; Quad quad; List<GeodupaNode> path; NeighborArrival payload;
NeighborArrival	GeodupaNode host; GeodupaNode node; double timestamp;
NeighborsRepairRequest	GeodupaNode requester; Set<Neighbor> neighbors;
NeighborsRepairReply	Set<Neighbor> neighbors;
PromotionRequest	- *EMPTY* -
PromotionReply	boolean accept;

Table A.1: Messages used in Geodupa.

Bibliography

[1] L. O Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS (N, k, f): a family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2003.

[2] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B Srivastava. Participatory sensing. In *World Sensor Web Workshop*, pages 1–5, 2006.

[3] Andrew T. Campbell, Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, Hong Lu, Xiao Zheng, Mirco Musolesi, Krist Fodor, and Gahng-Seop Ahn. The rise of People-Centric sensing. *IEEE Internet Computing*, 12(4):12–21, 2008.

[4] Shane B. Eisenman, Emiliano Miluzzo, Nicholas D. Lane, Ronald A. Peterson, Gahng-Seop Ahn, and Andrew T. Campbell. BikeNet: a mobile sensing system for cyclist experience mapping. *ACM Trans. Sen. Netw.*, 6(1):1–39, 2009.

[5] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: using a mobile sensor network for road surface monitoring. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 29–39, Breckenridge, CO, USA, 2008. ACM.

[6] A. Ghodsi, L. O Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-correcting broadcast in distributed hash tables. In *Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 1, pages 93–98, 2003.

[7] A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *9th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 18–21, 2003.

[8] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, page 9, San Francisco, California, 2004. USENIX Association.

[9] I. Gupta, K. Birman, P. Linga, A. Demers, and R. Van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. *Lecture Notes in Computer Science*, pages 160–169, 2003.

[10] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: a distributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 125–138, Boulder, Colorado, USA, 2006. ACM.

[11] M. Kaashoek and David Karger. Koorde: A simple Degree-Optimal distributed hash table. In *Peer-to-Peer Systems II*, pages 98–107. 2003.

[12] Gurmeet Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world, 2003.

[13] S. Mata, J. L Martins, S. Duarte, and M. Mamede. Análise do custo e da viabilidade de um sistema P2P com visibilidade completa. In *INForum*, 2009.

[14] Emiliano Miluzzo, Nicholas Lane, Shane Eisenman, and Andrew Campbell. Cenceme - injecting sensing presence into social networking applications. In *Smart Sensing and Context*, pages 1–28. 2007.

[15] A. T Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *The Third IEEE Workshop on Internet Applications*, 2003.

[16] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336, Raleigh, NC, USA, 2008. ACM.

[17] R. Murty, A. Gosain, M. Tierney, A. Brody, A. Fahad, J. Bers, and M. Welsh. CitySense: a vision for an urban-scale wireless networking testbed. In *Proceedings of the 2008 IEEE International Conference on Technologies for Homeland Security, Waltham, MA*, 2008.

[18] E. Paulos, R. Honicky, and E. Goodman. Sensing atmosphere. In *Workshop on Sensing on Everyday Mobile Phones in Support of Participatory Research*, 2007.

[19] Eric Paulos and Tom Jenkins. Urban probes: encountering our emerging urban atmospheres. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 341–350, Portland, Oregon, USA, 2005. ACM.

[20] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. CAN - a scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, San Diego, California, United States, 2001. ACM.

[21] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for Large-Scale Peer-to-Peer systems. In *Middleware 2001*, pages 329–350. 2001.

[22] Katie Shilton, Jeffrey Burke, Deborah Estrin, Mark Hansen, Nithya Ramanathan, Sasank Reddy, Vids Samanta, Mani Srivastava, Ruth West, and Jeffrey Goldman. Participatory sensing: A Citizen-Powered approach to illuminating the patterns that shape our world. *Resources*, 2009.

[23] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, San Diego, California, United States, 2001. ACM.

[24] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, 2006.

[25] B. Y Zhao, J. Kubiatowicz, and A. D Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *Computer*, 74:11–20, 2001.