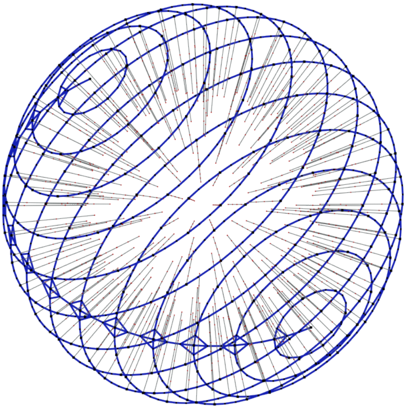
DISS ETH NO. 18573
TIK-Schriftenreihe Nr. 108

Routing on the Geometry of Wireless Ad Hoc Networks

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
Roland Flury



accepted on the recommendation of
Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Sándor P. Fekete, co-examiner
Prof. Dr. Leonidas J. Guibas, co-examiner

2009

Abstract

The routing of messages is one of the basic operations any computer network needs to provide. In this thesis, we consider wireless ad hoc and sensor networks and present several routing protocols that are tailored to the limited hardware capabilities of network participants such as sensor nodes. The constraint memory and computing power as well as the limited energy of such devices requires simplified routing protocols compared to the IP based routing of the Internet. The challenge is to build light routing protocols that still find good routing paths, such as to minimize not only the number of forwarding steps but also the energy consumption.

In the first part of this work we focus on the protocol design and analyze the properties of our routing algorithms under simplifying network models. In particular, we describe a location service that supports geographic routing even if the destination node is constantly moving. Such a location service is important as the geographic routing technique bases each routing step on the position of the destination node by repeatedly forwarding a message to the neighbor which is geographically closest to its destination. If there is no such neighbor, the message has reached a local minimum. This is a node at the boundary of a network hole around which the message needs to be led before it can continue its greedy path. We extend the classic notion of network holes to 3-dimensional unit ball graphs and propose several randomized recovery techniques to escape from local minima in such networks. In addition, we show that it is possible to forward messages greedily without ever falling in a local minimum. We do so by embedding the network into an higher-dimensional space such that there is a greedy path between any two nodes. Similarly, we describe a renaming technique in combination with small routing tables that ensures good routing paths not only for unicast, but also for anycast and multicast.

In the second part of this thesis, we examine the design of applications and come up with a programming technique to efficiently translate protocols to the limited hardware of sensor networks. We describe the slotted programming paradigm that fosters modular programming and decouples unrelated software components temporally. We demonstrate the advantages of our approach with two case studies: (1) an efficient clock synchronization module, and (2) an alarming module through which all nodes of a network can be awaken efficiently and reliably.

Zusammenfassung

Das Routing von Nachrichten ist eine Funktionalität, die jedes Computernetzwerk anbieten muss. In dieser Dissertation untersuchen wir Routing Protokolle für drahtlose Ad-hoc- und Sensornetzwerke, wo die Netzwerkteilnehmer, wie zum Beispiel Sensorknoten, oftmals sehr limitierte Hardware zur Verfügung haben. Dabei muss nicht nur die Rechen- und Speicherkapazität, sondern auch der Energiehaushalt von solchen Geräten berücksichtigt werden. Dies bedingt einerseits vereinfachte Protokolle im Vergleich zum IP-Routing, andererseits aber auch qualitativ gute Protokolle, welche gute Pfade finden um Energie zu sparen.

Im ersten Teil dieser Arbeit beschäftigen wir uns mit dem Design von Protokollen und analysieren mehrere Routingtechniken an vereinfachenden Netzwerkmodellen. Wir starten mit einem Positionsdienst, der geographisches Routing zu mobilen Knoten ermöglicht. Dies ist wichtig, da jeder Routingschritt auf den Koordinaten des Zieles beruht, indem die Nachricht jeweils zu dem Nachbar gesandt wird, der am nächsten zum Ziel liegt. Falls kein solcher Nachbar existiert, hat die Nachricht ein lokales Minimum erreicht. Dies ist ein Knoten am Rande eines Loches im Netzwerk, um welches die Nachricht geroutet werden muss. Wir erweitern die klassische Notation von Löchern zu dreidimensionalen Unit-Ball-Graphen und zeigen mehrere randomisierte Techniken auf, die aus lokalen Minima in solchen Netzwerken herausführen. Des Weiteren beschreiben wir eine virtuelle Einbettung von einem Netzwerk in einen mehrdimensionalen Raum, so dass zwischen allen Knotenpaaren ein direkter Pfad ohne Löcher besteht. Das Umbenennen von Knoten benutzen wir ebenfalls für eine Routingtechnik mit kleinen Routingtabellen, welche auch Multicasting und Anycasting unterstützt.

Im zweiten Teil erörtern wir die Konstruktion von Applikationen und präsentieren eine Programmiertechnik, die ein effizientes Implementieren von Protokollen für die limitierte Hardware von Sensorknoten erlaubt. Die beschriebene Technik fördert eine modulare Programmstruktur und separiert die verschiedenen Softwarekomponenten zeitlich, so dass die Komponenten unabhängig bleiben. Wir demonstrieren die Vorteile der Programmiertechnik anhand von zwei Beispielen: (1) einem Modul für energieeffiziente Uhrensynchronisation und (2) einem Alarmmodul, durch welches alle Knoten in einem Netzwerk effizient aufgeweckt werden können.

Acknowledgements

I look back at exciting years as a PhD student at ETH Zurich – last but not least because of the many people who supported me. In particular, I would like to thank my advisor Roger Wattenhofer for his guidance through the scientific jungle. Roger, I appreciate your patience with which you led me through the ups and downs of the past four years.

I would also like to thank my co-examiners Sándor Fekete and Leonidas Guibas for their willingness to serve on my committee board and work through this thesis.

Furthermore, my thanks also go to the members of *D*a *C*ool *G*ang. Stefan Schmid, my office mate number 1, thanks for not only sharing your office, but also your passion for running. I will never forget our nightly Sola Duo race. Roland Mathis, my office mate number 2, thanks for showing me the secrets of our servers and strengthen the Nidwalden-force in our group. Furthermore, I would like to thank Nicolas Burri for organizing all the coffee, Raphael Eidenbenz for helping to torture our DES students, Michael Kuhn for exploring our taste of music, Christoph Lenzen for beating everybody in chess, Remo Meier for knowing everything about Java, Johannes Schneider for surviving Iceland's summer storms, Benjamin Sigg for eating no killed animal, Jasmin Smula für dis Interässi a Schwizerdütsch, Philipp Sommer for synchronizing our motes, Pascal von Rickenbach for putting our nodes to sleep, Thomas Locher for teaching me how to use a Chinese dictionary, Yvonne Anne Pignolet for showing me how a real marriage works, Olga Goussevskaia for taking me to a favela disco in Rio de Janeiro, and Aaron Zollinger, Fabian Kuhn, Thomas Moscibroda, Keno Albrecht, and Regina O'Dell for the amusing table soccer matches.

Last but not least I would like to thank my parents Margrit and Peter and my two sisters Barbara and Regula who have supported me during my entire education in so many ways. Finally, my very special thanks are reserved for my darling Bea for the wonderful time we spent together.

Contents

1 Introduction **1**
1.1 Geographic Routing 2
1.2 Thesis Overview 3

I Protocol Design **5**

2 Routing in Mobile Networks **6**
2.1 Related Work 7
2.2 Model 9
2.3 Position Information 10
2.4 Lookup 13
2.5 Lazy Publishing 14
2.6 Concurrency 16
2.7 The MLS Algorithm 18
2.8 Analysis 20
2.9 Simulation 32

3 Routing in 3D Networks **35**
3.1 Random Walks 36
3.2 Notation and Model 37
3.3 Lower Bound 37
3.4 Towards 3D Routing Algorithms 40
3.5 Dual Graph 41
3.6 Routing on the Dual Graph 47
3.7 Simulation 49

4 Greedy Routing **51**
4.1 Related Work 52
4.2 Background, Results, and Approach 55
4.3 Greedy Embeddings of CUDGs 60
4.4 Simulation 63

5 Compact Routing with Any- and Multicast 67
5.1 Related Work 70
5.2 Definitions and Preliminaries 71
5.3 Dominance Net 72
5.4 Routing 77
5.5 Multicasting 81
5.6 Anycast 81
5.7 Distributed Dominance Net Construction 82

6 Conclusion 87

II Application Design 89

7 Simulation 90
7.1 sinalgo 91
7.2 Simulation modes 93
7.3 Mobility 93
7.4 Discussion 94

8 Slotted Programming 97
8.1 Related Work 99
8.2 Background 100
8.3 Slotted Programming 102
8.4 The slotos Reference Implementation 105

9 Slotted Clock Synchronization 110
9.1 Synchronized Transmission 111
9.2 Pipelined Synchronization 111
9.3 Initialization 112
9.4 Experiments 113
9.5 Discussion 113

10 Low-Power Signaling 115
10.1 Pipelining 116
10.2 Signaling of Binary States 116
10.3 RSSI vs Waves 117
10.4 Slotted Signaling 119
10.5 Test Application 120

11 Conclusion 127

1

Introduction

Wireless sensor networks and wireless mesh networks in general have received a lot of attention lately, last but not least because of their countless applications in various fields. The data collected by the nodes' sensors is valuable as it provides the basis to understand the monitored processes in more detail and react to predefined events. For example, we could imagine to equip our homes with sensors to monitor the usage of water and electricity to identify saving potentials. Similarly, the control of the heating and air conditioning may be driven by several environmental parameters such as the temperature inside and outside the building, humidity of the air, presence of people, or whether the doors and windows are open or closed. Independent of the application, the sensor readings only reveal their full potential if they are analyzed in a broader context, for example in combination with the readings of neighboring sensors. Therefore, the sensor nodes need to be equipped with a communication device to either share their sensor readings with close-by neighbors or to send the data to a central processing unit.

In this thesis, we will focus on sensors nodes that are equipped with a wireless communication device. In contrast to a wired solution, wireless communication allows for autonomous sensors if they are powered by a battery. On the one hand, such autonomous sensor nodes drastically simplify the deployment and reduce the installation cost as no wiring is necessary. On the other hand, however, the communication on wireless networks is much more involved and requires dedicated algorithms. Generally, the action of sending a message from a sender node to a target node is driven by a routing algorithm which guides messages through the network towards their destination. Being such an integral part of any network, there already exists a large diversity of routing algorithms, including the IP routing of today's Internet, communication protocols that connect robots exploring our solar system, and algorithms that ensure message delivery in ad hoc networks. The different needs and characteristics of the various networks impose many challenges, requiring appropriate routing techniques. Throughout this thesis, we examine several routing algorithms for large wireless ad hoc networks such as sensor and mobile ad hoc networks.

In contrast to the IP based Internet routing, which relies on large forwarding tables, routing algorithms for wireless ad hoc networks face not only the problem of unstable networks, but also that of rather limited network participants. The instability of the network may be caused due to mobility of the network nodes, or just by fluctuations of the wireless communication medium which is far more vulnerable than a wired network. The limitations on the network nodes are manifold, including hardware constraints such as small memory and low processing power, as well as power supply limitations.

1.1 Geographic Routing

Exploiting the geometry of the network to perform routing is a prominent approach to overcome the challenges posed by such limited ad hoc networks. *Geographic routing* protocols forward the packet to a neighbor geographically closer to the target, until the message reaches its destination. Thus, a requirement for geographic routing is that each node knows its own, as well as its neighbors' Euclidean coordinates. A node can learn its position through hardware support such as GPS. Alternatively, the position can be obtained through localization algorithms, of which a variety has been proposed in recent years, e.g. [16, 50, 73]. Furthermore, the position of the target node needs to be known, as each routing step is based on this information. Because learning the position of the target node may come at a certain cost, the sender node includes this information in the message for reuse in further routing decisions by the nodes along the route. The initial request for the position of a remote network node can be handled by a *location service* which determines the position of a node given its ID. Such services have been studied for static ad hoc networks [2, 24], and a probabilistic approach for mobile networks was presented in [45].

We define a geographic routing algorithm to base its decision solely on the position of the current node, the neighbors, and the destination, and we require the network nodes to be *memoryless*, i.e. not store any state for messages they see. This not only binds the routing state uniquely to the messages, but also removes an additional storage overhead from the nodes, which could limit the number of messages forwarded by a node if its memory is too small. As a matter of fact, the size of the memory is not the only challenge. The problem of storing message state is that this data arrives dynamically, and it is hard to predict how much of this data needs to be stored at any given time. Dynamic memory allocation would partially solve the problem, but introduces a computational overhead that many devices cannot afford. Consequently, the number of messages for which a node may store the state needs to be determined at compile time, jeopardizing routing success if more messages than anticipated need to be handled.

Another important property of geographic routing algorithms is that their decisions are only based on *local* information, which can easily be refreshed upon changes in the network. This stands in sharp contrast to routing algorithms that rely in some way on a global view of the network. Whereas these global routing schemes provide excellent routing paths, the construction of their routing information is rather expensive, and any change of the network may require a complete, network wide reconfiguration of the routing information. As a result, these routing algorithms are an excellent choice for *static* networks, but not for (wireless) ad hoc networks, where a continuous change of the network topology is unavoidable.

A key concept of geographic routing is *greedy forwarding*, where each node forwards a received message to the neighbor that is geographically closest to the target. This constitutes a very simple, yet efficient way of routing messages. Greedy routing, however, is not always successful in delivering the packet. When a packet reaches a node, whose neighbors are all further away from the target than the node itself, greedy forwarding fails, and we say that the message has reached a local minimum. Such local minima are especially common in sparse networks and in networks with holes, i.e. regions in the network where no network nodes are placed, and around which a message needs to be led. For 2-dimensional networks, *face routing* and several variants thereof are the most prominent solutions to escape local minima. In the greedy-face-greedy approach [13, 25], a message routes greedily until it gets stuck in a local minimum. It then routes along the face of the network hole until it finds a node closer to the target than the local minimum, from where it continues greedily. Techniques to proactively avoid routing voids are presented in [36] and a worst case optimal, but still average case efficient routing algorithm was obtained by constraining the range of the face routing in [59, 60]. Unless the the voids of the network are described through a boundary detection system [56], the latter two protocols require a planarized network graph to route along the faces.

1.2 Thesis Overview

In this thesis, we discuss several routing techniques that exploit the geometry of the underlying network to provide efficient routes. In the first part of this work we focus on the *protocol design* and analyze the proposed protocols under simplified network models. We start with the description of a location server for geographic routing that allows to deliver messages even if the destination node is constantly moving. Whereas we restrain our analysis to 2-dimensional networks for the location server, we also consider geographic routing in 3-dimensional networks where the recovery from local minima cannot be resolved deterministically using only local information. In particular, we extend the classic notion of network holes to 3-dimensional networks and describe several randomized recovery techniques to escape from local minima. Finally, we discuss two routing techniques that assign special coordinates to each network participant. On the one hand, we show that the coordinates can be assigned such that there is a greedy path between any two nodes. On the other hand, we describe a coordinate assignment that uses small routing tables to ensure good routes not only for the unicast, but also for anycast and multicast requests.

In the second part of this thesis, we shift our focus to the *application design* where we discuss the verification of such protocols and how the corresponding applications can be programmed for the limited hardware of sensor nodes. We first describe the simulation tool we used to examine the algorithms described in the first part. Afterwards, we propose a novel programming paradigm that eases the implementation of applications for sensor nodes. We conclude this thesis with two case studies that demonstrate the power of our programming approach: An efficient clock synchronization module and a low-power signaling module through which all nodes of a network can be alarmed efficiently.

Part I

Protocol Design

2

Routing in Mobile Networks

For systems where each node is equipped with a location sensing device, geographic routing has received much attention recently and is considered to be an efficient and scalable routing paradigm. However, geographic routing algorithms assume that the sender knows the position of the destination node. This introduces a high storage overhead if each node keeps track of the position of all other nodes. Even more challenging is the situation with mobile nodes: In a mobile ad hoc network (MANET), nodes might be moving continuously and their location can change even while messages are being routed towards them. Clearly, a node cannot continuously broadcast its position to all other nodes while moving. This would cause a high message overhead.

In the so-called *home-based approach*, each node is assigned a globally known home where it stores its current position. A sender first queries the home of the destination node to obtain the current position and then sends the message. This can be implemented using distributed or geographic hashing [81], and is a building block of many previous ad hoc routing algorithms, including [34, 80, 93, 98]. Despite of its broad usage, the home-based approach is not desirable, as it does not guarantee a good performance: The destination might be arbitrarily close to the sender, but the sender first needs to learn this by querying the destination's home, which might be far away. Similarly, a large overhead is introduced by moving hosts, which need to periodically update their homes, which might be far away. Even more important is the observation that the destination node might have moved to a different location by the time the message arrives. Thus, simultaneous routing and node movement require special consideration.

In this chapter, we present a routing framework called `MLS` (for *M*obile *L*ocation *S*ervice) in which each node can send messages to any other node without knowing the position of the destination node. The routing (lookup) algorithm works hand in hand with a publish algorithm, through which moving nodes publish their current location on a hierarchical data structure. We compare the length of the routes found by our protocol to the corresponding optimal routes and define the *stretch* to be the maximal factor by which the routes of our lookup algorithm are longer than the corresponding optimal routes. We formally prove that the stretch of the lookup algorithm is in $\mathcal{O}(1)$ and show in extensive simulation that the constant hidden in the $\mathcal{O}()$-notation is approximately 6. The amortized message cost induced by the publish algorithm of a moving node is in $\mathcal{O}(d \log d)$, where d is the distance the node moved. Again, our simulations show that the hidden constants of the $\mathcal{O}()$-notation for the publish overhead are small, the average being $4.3 \cdot d \log d$. Finally, `MLS` only requires a small amount of storage on each node. For evenly distributed nodes, the storage overhead is logarithmic in the number of nodes (with high probability).

We formally prove the correctness of `MLS` for concurrent lookup requests and node movement. That is, while a message is routed, the destination node might move considerably, but the lookup stretch remains $\mathcal{O}(1)$. To prove this property, we derive the maximum node speed v_{max}^{node} at which nodes might move. We express this speed as a fraction of v_{min}^{msg}, the speed at which messages are routed. Clearly, if $v_{max}^{node} \geq v_{min}^{msg}$, a message may not reach its destination at all. As a main result of this chapter, we show that `MLS` is correct if $v_{max}^{node} \leq v_{min}^{msg}/15$ in the absence of lakes[1]. I.e. we show that the lookup stretch remains in $\mathcal{O}(1)$ even though the destination node might move at a speed up to 1/15 of the message speed. To the best of our knowledge, this is the first work that determines the maximum node speed to allow concurrent lookup and node movement.

2.1 Related Work

Routing on ad hoc networks has been in the focus of research for the last decade. The proposed protocols can be classified as proactive, reactive, or hybrid. Proactive protocols distribute routing information ahead of time to enable immediate forwarding of messages, whereas the reactive routing protocols discover the necessary information only on demand. In between are hybrid routing protocols that combine the two techniques.

Much work has been conducted in the field of geographic routing where the sender knows the position of the destination. Face routing is the most prominent approach for this problem [13]. AFR [60] was the first algorithm that guarantees delivery in $\mathcal{O}(d^2)$ in the worst case, and was improved to an average case efficient but still asymtotically worst case optimal routing in GOAFR$^+$ [59]. Similar techniques were chosen for the Terminode routing [11], Geo-LANMAR routing [26], and in [36]. All of them combine greedy

---

[1] In the presence of lakes, v_{max}^{node} is reduced by a factor equal to the largest routing stretch caused by the lakes.

routing with ingenious techniques to surround routing voids. Georouting is not only used to deliver a message to a single receiver, but also for geocasting, where a message is sent to all receivers in a given area. All these georouting protocols have in common that the sender needs to know the position of the receiver.

If we consider a MANET, a sender node needs some means to learn the *current* position of the destination node. A proactive location dissemination approach was proposed in DREAM [10], where each node maintains a routing table containing the position of *all* other nodes in the network. Each node periodically broadcasts its position, where nearby nodes are updated more frequently than distant nodes. In addition to the huge storage and dissemination overhead, DREAM does not guarantee delivery and relies on a recovery algorithm, e.g. flooding.

An alternative to the fully proactive DREAM is the hybrid home-based lookup approach, as utilized in [34, 80, 93, 98]. However, this approach does not allow for low stretch routing, as outlined in the introduction.

Awerbuch and Peleg [8] proposed to use regional matchings to build a hierarchical directory server, which resembles our approach. However, to handle concurrent lookup and mobility, a *Clean Move Requirement* was introduced, which hinders nodes to move too far while messages are routed towards them. With other words, a lookup request can (temporarily) stop its destination node from moving. Furthermore, the lookup cost of [8] is polylogarithmic in the size of the network, which restrains scalability.

A novel position dissemination strategy was proposed by Li et al. in [63]: For each node n, GLS stores pointers towards n in regions of exponentially increasing size around n. In each of these regions, one node is designated to store n's position based on its ID. The lookup path taken by GLS is bounded by the smallest square that surrounds the sender and destination node. As outlined in [2], GLS cannot lower bound the lookup stretch and lacks support for efficient position publishing due to node movement. Xie et al. presented an enhanced GLS protocol called DLM in [99], and Yu et al. proposed HIGH-GRADE [101], which is similar to [64]. In contrast to GLS, DLM and HIGH-GRADE do not store the exact position of the corresponding nodes, which reduces the publish cost. Nevertheless, neither of them can lower bound the publish cost (e.g. due to the problem described in Figure 2.3) and they do not tackle the concurrency issue described above.

Abraham et al. proposed LLS [2], a locality aware lookup system with worst case lookup cost of $\mathcal{O}(d^2)$, where d is the length of the shortest route between the sender and receiver[2]. Similar to GLS, LLS publishes

---

[2] The quadratic overhead for lookups in LLS has two reasons. First, the proposed model utilizes an underlying routing algorithm such as GOAFR$^+$ with a quadratic worst case overhead. But even with a constant-stretch routing algorithm, the worst case lookup cost of LLS remains in $\mathcal{O}(d^2)$. This is due to a flooding technique that is interleaved with the spiral lookup to find a first pointer to the destination node. The authors

position information on a hierarchy of regions (squares) around each node. A lookup requests circles around the sender node with increasing radius until it meets one of the position pointers of the destination node, and then follows this pointer. `MLS` borrows some ideas from LLS and HIGH-GRADE, adding support for concurrent mobility and routing, improving the lookup to have linear stretch and bounding the publish overhead.

In the following sections, we present the `MLS` algorithm. We start with our model assumptions and the hierarchical lookup system through which messages are routed. Then, we discuss in more detail the routing of messages, and define a policy when a moving node needs to update its position data in Chapter 2.5. We introduce the issues of concurrent message routing and node movement in Chapter 2.6 and present the `MLS` algorithm in a concise and formal way in Chapter 2.7, such that we can prove its correctness in the sequel. Finally, we describe our simulation setup.

2.2 Model

For the analysis of our algorithm, we consider a world built of land and lakes. The nodes are distributed on the land areas, whereas no nodes can be placed on lakes. In order to allow for total connectivity, we assume that there are no islands, i.e. there are no disconnected land areas. The nodes are expected to contain a positioning system such as GPS, Cricket [89], cell tower or WLAN triangulation. Furthermore, each node is equipped with a communication module that provides reliable inter-node communication with minimal range r_{min}.

In our model, the nodes actively participate in ad hoc routing to deliver messages. To guarantee the reachability of any location on land, we consider a relatively dense node distribution and require that for any position p on land, there is a node at most $\lambda = r_{min}/3$ away. Furthermore, this invariant should hold over time while nodes are moving.

Using `MLS`, each node can send messages to any other node without knowing the position of the destination. To perform this task, `MLS` stores information about the nodes' whereabout in well defined positions (see Chapter 2.3). Then, the messages are routed to some of these special positions, where they learn about the current location of their destination. Because the positions

---

motivated the bounded flooding to overcome extreme situations due to the underlying routing algorithm. However, LLS allows worst case scenarios in which the destination can only be found through flooding. One such case can be observed when a node n_1 moves along the diagonal of its publish squares. In that case, n_1 can approach another node n_2 on the same diagonal up to an ϵ-distance without writing any new pointers into the lookup range of n_2. This is possible due to the delayed publish strategy of the moving nodes, and the limited and gird-aligned lookup range. This quadratic lookup overhead might be overcome relatively easily, but it is much harder to add support for concurrency to LLS, which was not considered in its current version.

of the intermediate destinations are known, this underlying routing might be performed by a geographic routing algorithm.

For the rest of this chapter, we abstract from this low level routing and assume the following communication capability: For any two positions p_s and p_t on land, a node s in the λ-proximity of p_s can send a message to a node in the λ-proximity of p_t. The time to route the message is bounded by $\eta \cdot |p_s p_t|$, where $|p_s p_t|$ is the Euclidean distance[3] between p_s and p_t. We assume that this underlying routing algorithm selects the shortest path if it has a choice.

Note that this underlying routing capability is orthogonal to the main routing problem discussed in this chapter, where the sender does not know the position of the receiver.

2.3 Position Information

In this section, we describe how each node t maintains a lookup system through which `MLS` routes messages to t. This lookup system is based on several layers, where the top layer is the smallest square of side length $\rho \cdot 2^M$ that encloses the entire world. M is dependent on the size of the world, and $\rho = \lambda/\sqrt{2} = r_{min}/(3\sqrt{2})$ is given by the radio range (see Chapter 2.2). In the following, we denote this square by level-M and write L_M. (Please refer to Table 2.1 for a summary of the notation used in this chapter.) Similar to a geographic hash table (GHT) [81], each node has a designated position on land where it stores directions to its position. But instead of storing its exact position, a node t only stores in which of the four possible sub squares it is located, as depicted in Figure 2.1. Recursively, each selected square contains a pointer to its sub square that surrounds t. Finally, the chain is broken when the size of the sub square reaches ρ. Thus, a message for node t can be routed along these pointers until it reaches the smallest square.

We use the term L_{M-1} to denote any of the squares received when L_M is divided into 4 sub squares of side length $\rho \cdot 2^{M-1}$. Recursively, L_i denotes any square of side-length $\rho \cdot 2^i$ that can be obtained by dividing a L_{i+1} square into its 4 sub squares. The recursion stops for L_0, which is a square of side length ρ. To denote the L_i that surrounds a specific node t, we use the notation L_i^t. Clearly, L_M^x is the same for all nodes x, namely the square that surrounds the entire world.

On each L_i^t for $i > 0$, node t has a well defined position where it stores in which of the 4 possible L_{i-1} it is located. We call this information **L**evel **P**ointer and write LP_i^t to denote the level pointer on L_i^t that points to L_{i-1}^t. Also, we write $^*\mathrm{LP}_i^t$ to denote the L_{i-1}^t where LP_i^t points.

---

[3] Note that if $|p_s p_t| \to 0$, the message would have to be delivered instantaneously. However, unless the sender and receiver are identical, at least one hop is necessary, which requires time. We can safely ignore this border case because this chapter presents a worst case analysis, where the shortest distance to be routed is λ.

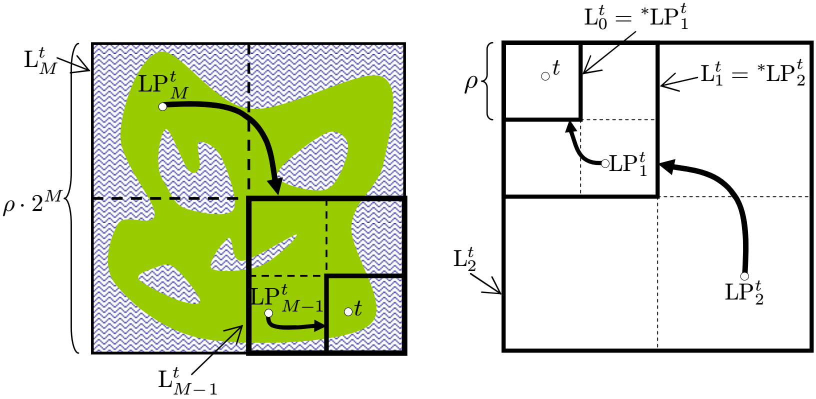


Figure 2.1: The left figure shows the entire world surrounded by L_M, the square of side length $\rho \cdot 2^M$. The land masses are filled with solid color (green), lakes are filled with waves. For each node t, L^t_i contains a level pointer LP^t_i that points to the sub square L^t_{i-1} of size $\rho \cdot 2^{i-1}$ that contains t. The right picture shows the smallest three levels around t and how each level for $i > 0$ contains a pointer that points to the next smaller level.

We have seen that every node t stores a LP^t_i on each of its levels L^t_i. This information needs to be hosted on a node somewhere in L^t_i. But because the nodes are mobile, we cannot designate a specific node on each L^t_i to store the LP^t_i. Therefore, we propose to store this pointer at a specific position p on land in L^t_i. Due to the minimal node density, we know that there exists at least one node in the λ-proximity of p, which we can use to store LP^t_i. If there are several nodes in the λ-proximity of p, we pick the one closest to p. The selected node then hosts LP^t_i until it moves away from p by more than λ. At that point, it passes on LP^t_i to its neighbor node closest to p, which must exist due to the minimal node density. Over time, the LP^t_i is not necessarily stored on the node closest to p, but by an arbitrary node in the λ-proximity of p.

Each node t stores a LP^t_i at a well defined position p_t on land within L^t_i, where p_t is determined through the unique ID_t of t. Any consistent hash function that maps the ID of a node onto a position on land can be used for this purpose, as long as the chosen positions are evenly distributed over the land area for different IDs.

One possible function is the following, where we use two hash-functions $H_1()$ and $H_2()$ to map the ID of t to real numbers in the range $]0, 1]$. p_t is determined as an offset $(\Delta x, \Delta y)$ from the top left corner of L^t_i. Δy is

L_i^t	Level that contains t with side-length $\rho \cdot 2^i$
$(\mathrm{L}_i^s)^8$	The 8 surrounding squares of L_i^s
LP_i^t	Level pointer on L_i for node t; points to L_{i-1}
$^*\mathrm{LP}_i^t$	The L_{i-1} where LP_i^t points to
δ_i^t	distance of a node t to $^*\mathrm{LP}_{i+1}^t$
FP_i^t	Forwarding pointer if $\mathrm{LP}_i^t \notin {}^*\mathrm{LP}_{i+1}^t$
$^*\mathrm{FP}_i^t$	The L_i where $^*\mathrm{FP}_i^t$ points
TFP_i^t	Temporary forwarding pointer, before a pointer to t is removed
$^*\mathrm{TFP}_i^t$	The L_i where TFP_i^t points
TTL_i	Time to live of a TFP_i
v_{max}^{node}	Max. speed of nodes
r_{min}	Min. communication range of a node
λ	Min. distance to a node from any land point
ρ	Side length of L_0; $\rho = \lambda/\sqrt{2} = r_{min}/(3\sqrt{2})$
M	L_M surrounds the entire world
α	When $\delta_i^t \geq \alpha \cdot \rho \cdot 2^i$, LP_{i+1}^t is updated
$\beta(\beta_T)$	Max. number of forwarding hops to reach LP_i^t from a FP_i^t (TFP_i^t)
γ	See Lemma 2.4
η	Routing overhead to route to a given position

Table 2.1: Nomenclature used throughout this chapter.

chosen such that, when only considering L_i^t, the fraction of land above Δy is $\mathrm{H}_1(\mathrm{ID}_t)$. Once Δy is fixed, we must choose Δx such that p_t lies on land. We concatenate the line-segments where p_t can be placed to a single line and determine Δx such that the length of the line left to p_t is $\mathrm{H}_2(\mathrm{ID}_t)$ of the total line length.

Given this mapping, any node can determine the potential position p_t where a node t might store a LP_i^t for any L_i^t. All the node needs to know is the ID of the receiver and the position of the lakes. Amongst others, this is necessary to route a message along the level pointers towards t: Once a message has been routed to a LP_i^t, the node hosting LP_i^t determines p'_t in $^*\mathrm{LP}_i^t$ and forwards the message to LP_{i-1}^t, which is located at p'_t.

We can already see that the number of levels only depends on the size of the deployment area and the transmission radius r_{min}. Therefore, every node needs to maintain only a constant number of level pointers. Because the positions of the level pointers are chosen randomly on the different levels, the storage overhead is balanced smoothly on the nodes if the nodes themselves are evenly distributed. This is an important property of `MLS`, and avoids overloading a few nodes with excessive amounts of data.

2.4 Lookup

Because messages are routed to a priori unknown positions, we denote them as *lookup requests*. When a sender node s wants to send a message to a destination node t, it issues a lookup request for node t, which encapsulates the message to be sent. So far, we have described where each node publishes its level pointers and how a lookup request is forwarded along the level pointers towards t once a first LP_i^t has been found. This section is devoted to the first phase of the lookup algorithm, which routes the lookup request to the first LP_i^t.

We propose a lookup algorithm that first searches t in the immediate neighborhood of s and then incrementally increases the search area until a LP_i^t is found. From there, the lookup request can be routed towards the smaller levels, as described in the previous section. Using this approach, we find t quickly if it is close to s. In particular, we prove that the lookup time is linear in the distance between s and t. As for the search areas, we use an extended version of the levels of node s, who issued the lookup request. For each L_i^s, we define $(\mathrm{L}_i^s)^8$ to be the 8 L_i squares adjacent to L_i^s.

In the very first step of a lookup, node s checks whether t is in its immediate neighborhood. In this case, the message can be sent directly to t. Otherwise, the lookup request is sent to L_1^s to check whether it finds a LP_1^t. If this is not the case, the lookup request is forwarded in sequence to the 8 squares of $(\mathrm{L}_1^s)^8$, where it tries to find LP_1^t. This step is repeated recursively: while the lookup request fails to find a LP_i^t on L_i^s and $(\mathrm{L}_i^s)^8$, it is forwarded to L_{i+1}^s and then in sequence to squares of $(\mathrm{L}_{i+1}^s)^8$, where it tries to find LP_{i+1}^t. A possible lookup path through the first 3 levels is depicted in Figure 2.2. Note that when the lookup request is forwarded through the levels of $(\mathrm{L}_i^s)^8$, the sequence is chosen such that the last visited L_i is contained in L_{i+1}^s, and the request is always forwarded to a neighboring L_i that shares an edge with the current L_i. (Skip L_i that are completely covered by lakes.)

In the first phase of the lookup algorithm, the lookup request is routed to a series of levels L_i, where it tries to find a level pointer LP_i^t. In the following lemma, we provide an upper bound for the time needed to search i levels. Note that this also gives an upper bound for the time needed to find LP_i^t on L_i, given that $\mathrm{LP}_i^t \in \mathrm{L}_i$.

Lemma 2.1. *The accumulated time for searching a level pointer of node t on the levels 0 through i is bounded by $\eta 2^{i+1}\rho(\sqrt{2} + 8\sqrt{5})$.*

Proof. When a lookup request for node t issued by a node s starts its search on L_j, it first queries for LP_j^t in L_j^s. From the lookup algorithm presented above, we know that the lookup request tries[4] to end its search of the lev-

---

[4] This fails, if L_{j-1}^s is the only sub-level of L_j^s covering land. In this case, the lookup request first needs to move into L_j^s. This additional overhead is well compensated on the previous level, where at least 3 L_{j-1}^s were not visited.

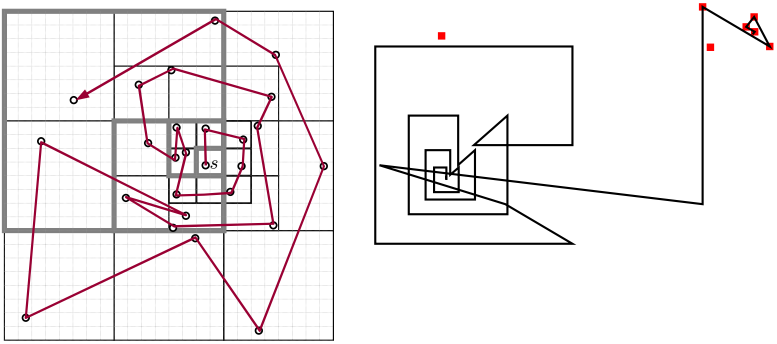


Figure 2.2: When s issues a lookup request for node t, the request is forwarded to the potential positions of a LP_i^t in $\mathrm{L}_i^s \cup (\mathrm{L}_i^s)^8$ for increasing i. The bold gray squares in the left image indicate the first 4 levels of node s. Note that we have only drawn the nodes visited by the chosen route. The right image shows a lookup path found by our simulation framework. The square dots indicate the LP of the destination node. Because no lakes were present in the lookup area, the lookup path is regular and draws quadratic shapes.

els L_{j-1} on a node in L_j^s. Thus, in the worst case, the request has to be routed over the diagonal of L_j^s to reach the potential place of LP_j^t, which implies a maximal route-time of $\eta 2^j \rho\sqrt{2}$. Then, to check for LP_j^t in $(\mathrm{L}_j^s)^8$, the lookup request is repeatedly sent to a neighboring L_i to which the previous L_i shares an edge. At the worst, this takes $\eta 2^j \rho\sqrt{5}$ for each of the 8 neighbors. Thus, the total time to query for LP_j^t on level j is at most $\eta 2^j \rho(\sqrt{2} + 8\sqrt{5})$, and the accumulated time to query i levels is bounded by $t \leq \sum_{j=0}^{i} \eta 2^j \rho(\sqrt{2} + 8\sqrt{5}) < \eta 2^{i+1} \rho(\sqrt{2} + 8\sqrt{5})$. □

Because L_M is the same square for all nodes, we are sure that a lookup request finds a level pointer for t at the latest on L_M^s [5]. In a later section, we give an upper bound on the time the lookup request needs to find a first LP_i^t based on the distance between s and t.

2.5 Lazy Publishing

In this and the following section, we analyze the implications of mobile nodes. In particular, this section focuses on the publish algorithm, through which every node keeps its level pointers up to date when it moves. Remember that

---

[5] This holds also under lazy publishing and in the concurrent setting, two concepts that are introduced in the following sections.

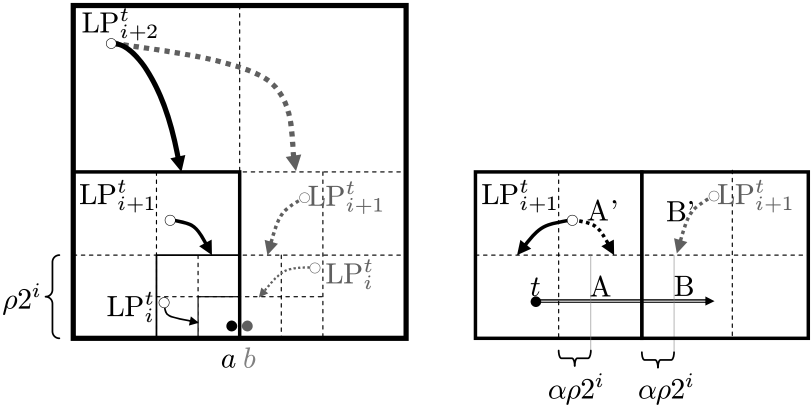


Figure 2.3: If a node t oscillates between the two points a and b in the left picture, immediate updating of its level pointers would cause an enormous amount of traffic. In black are the level pointers necessary if t is at a, the level pointers necessary at b are gray. Lazy publishing delays the updating of LP^t_{i+1} until node t has moved away from $^*\mathrm{LP}^t_{i+1}$ by more than $\alpha\rho 2^i$, as depicted in the right picture. Only when t moves across A, LP^t_{i+1} is updated to A'. Similarly, only when t crosses B, A' is deleted and B' is added.

every node t maintains a LP^t_i on each of its L^t_i for $0 < i \leq M$. Clearly, if a node t must update its LP^t_{i+1} as soon as it changes L^t_i, the publish cost might be extremely high. The left part of Figure 2.3 depicts a situation where many level pointers would have to be updated due to an arbitrarily small move of node t. If t oscillates between the two points a and b, and immediately sends messages to update the level pointers after moving an ϵ-distance, an enormous amount of traffic would be generated. To reduce this overhead, we employ lazy publishing, a concept that is similar to the lazy update technique utilized in [2].

Lazy publishing allows a node t to move out of $^*\mathrm{LP}^t_{i+1}$ up to a certain distance without updating LP^t_{i+1}, and reduces the overhead due to oscillating nodes. The following publish policy defines when a node t needs to update its LP^t_i.

We use the notation δ^t_i to denote the air distance of node t to $^*\mathrm{LP}^t_{i+1}$. Formally, δ^t_i is the shortest distance of t to any edge of $^*\mathrm{LP}^t_{i+1}$ if $t \notin {}^*\mathrm{LP}^t_{i+1}$. Otherwise, $\delta^t_i = 0$.

Definition 2.2. (PUBLISH POLICY) *When a node t has moved away from $^*\mathrm{LP}^t_{i+1}$ by more than $\alpha \cdot \rho \cdot 2^i$, it needs to update LP^t_{i+1}, such that LP^t_{i+1} points to the current L^t_i. Formally, a node t must update LP^t_{i+1} if $\delta^t_i \geq \alpha \cdot \rho \cdot 2^i$, for a fixed $\alpha \in \mathbb{R}^+$.*

We need to consider two cases while updating a LP^t_{i+1}: If the outdated LP^t_{i+1} is in L^t_{i+1}, it suffices to change the value of LP^t_{i+1} such that it points to L^t_i. In the right half of Figure 2.3, this is the case when t moves over the line marked A. However, if t has moved to a different L_{i+1}, the outdated LP^t_{i+1} needs to be removed and a new one must be added to L^t_{i+1}. An example of this case is depicted in Figure 2.3, when t moves over the line marked B.

The implementation of such an update is straight forward: Node t sends an *update* message to the outdated LP^t_{i+1} to change its value. If a new LP^t_{i+1} needs to be created, t sends a *remove* message to the outdated LP^t_{i+1} and a *create* message to the position of the new LP^t_{i+1}.

2.6 Concurrency

Clearly, the lookup algorithm presented in Chapter 2.4 does not support lazy publishing in its second phase, where the lookup request follows the level pointers in order to find a destination node t. So far, we have assumed that for $i > 1$, $^*\mathrm{LP}^t_i$ contains a LP^t_{i-1}. However, under the lazy publishing policy, LP^t_{i-1} might be outside $^*\mathrm{LP}^t_i$. We now derive modifications to the lookup and publish algorithms such that they support lazy publishing. At the same time, we introduce the issue of simultaneous lookup and publish requests.

2.6.1 Forwarding Pointer

`MLS` uses a forwarding pointer to guide a lookup request to LP^t_i. If $\mathrm{LP}^t_i \notin {}^*\mathrm{LP}^t_{i+1}$, $^*\mathrm{LP}^t_{i+1}$ contains a forwarding pointer at the location where the LP^t_i would be. This forwarding pointer points to the neighboring L_i that contains LP^t_i. In the following, we denote such a forwarding pointer by FP^t_i and write $^*\mathrm{FP}^t_i$ to denote the L_i where FP^t_i points. Figure 2.4 depicts a situation where a FP is necessary. Note that we restrict the value of α to the range $[0,\ 1[$ such that a FP^t_i might only point to an adjacent L_i. This simplifies our analysis, but does not restrain the final result, where α is chosen clearly smaller than 1 in order to maximize the allowed speed at which nodes may move.

With the forwarding pointers, a lookup request has an easy means to find the LP^t_i if $\mathrm{LP}^t_i \notin {}^*\mathrm{LP}^t_i$. However, this simplistic approach only works in a static setup, where all publish requests of node t have terminated before a lookup request queries for t. Consider again Figure 2.4 and suppose that t moves upwards. When $\delta^t_i > \alpha\rho 2^{i-1}$, t needs to send three messages: one to remove LP^t_i from [C], one to add a new LP^t_i in [D] and one to change the direction of FP^t_i in [B]. Because the messages might be delayed randomly by the presence of lakes, it is possible that a lookup request reads FP^t_i before it is updated, and then fails to find LP^t_i in $^*\mathrm{FP}^t_i$ because LP^t_i was already removed. This is a racing condition between a lookup request and the publish request which we need to avoid.

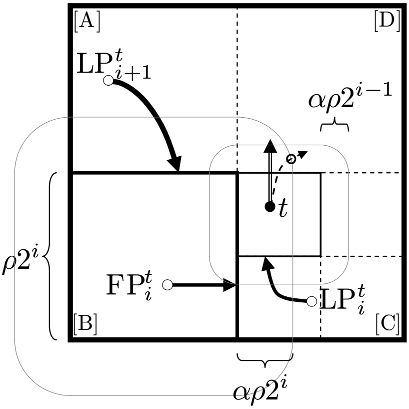


Figure 2.4: A node t is outside $^*\mathrm{LP}^t_{i+1}$, but does not need to update LP^t_{i+1}. Instead of removing its LP^t_i in $^*\mathrm{LP}^t_{i+1}$, t maintains a FP^t_i that points to the neighboring L_i there LP^t_i can be found. When t moves upwards along the indicated (solid) path, it eventually leaves $^*\mathrm{LP}^t_i$ by more that $\alpha\rho 2^{i-1}$ and needs to add a new LP^t_i in [D], update FP^t_i in [B] and remove the LP^t_i in [C]. To prevent a racing condition with a concurrent lookup, LP^t_i in [C] is not removed, but transformed to a TFP^t_i that points to [D].

2.6.2 Temporary Forwarding Pointer

Inspired by the fact that the racing condition in the previous example arose because of a LP_i^t that was removed too early, `MLS` does not remove LP_i^t immediately, but leaves behind a **T**emporary **F**orwarding **P**ointer, denoted TFP_i^t. Such a TFP_i^t points to the neighboring L_i where t was located when it decided to remove the LP_i^t. We will write $^*\mathrm{TFP}_i^t$ to denote the L_i where the TFP_i^t points.

To come back to Figure 2.4, t changes the LP_i^t in [C] to a TFP_i^t instead of removing it. A lookup request that follows the outdated FP_i^t then might find such a TFP_i^t instead of the expected LP_i^t and follows the TFP_i^t to finally find LP_i^t in [D].

At the time when LP_{i+1}^t is updated, the FP_i^t in $^*\mathrm{LP}_{i+1}^t$ becomes obsolete and needs to be removed. But deleting the FP_i^t could cause similar racing conditions with a concurrent lookup request as when a LP_i^t is removed. Therefore, `MLS` overwrites a FP_i^t with a TFP_i^t instead of removing the FP_i^t.

According to its name, a TFP does only exist for a limited time. TFP are automatically removed after a given time, which we denote TTL_i for a TFP_i^t. Thus, the lifetime TTL_i of a TFP_i^t depends on the value of i. We give constraints on the value of TTL_i while proving the correctness of `MLS`. Given that TTL_i can be determined statically, a TFP_i^t can be removed by the hosting node without any interaction of node t.

2.7 The MLS Algorithm

We have now gathered all parts of `MLS` and present it here in a concise form before proving its correctness and performance. As before, the algorithm comes in two parts, the publish and the lookup algorithm. The publish algorithm is executed permanently by each moving node as to maintain valid information on its hierarchy of levels, whereas the lookup algorithm is used to route a message to a given node.

2.7.1 MLS Publish

During the startup phase of node t, initialize all level pointers LP_{i+1}^t to point to L_i^t. While t is moving, it executes the protocol shown in Algorithm 2.1. In the initialization phase, t sends a message to the position of LP_i^t on each L_i. This message tells the receiving node to store LP_i^t, which points to L_{i-1}^t. We assume that there are no lookup requests for t during this initial phase.

While t is moving, it utilizes lazy publishing where it only performs an action on level i if δ_i^t (its distance to $^*\mathrm{LP}_{i+1}^t$) is above the given threshold (Line 1). If t updates LP_{i+1}^t, FP_i^t becomes obsolete and is changed to a TFP_i^t that points to L_i^t (Line 2). The simplest case arises when only the value of LP_{i+1}^t needs to be changed, because LP_{i+1}^t can still point to the

Algorithm 2.1: Publish protocol of node t

```
1  if $\delta_i^t \geq \alpha \cdot \rho \cdot 2^i$ then
2      if $i > 0$ then change $\mathrm{FP}_i^t$ in $^*\mathrm{LP}_{i+1}^t$ to $\mathrm{TFP}_i^t$
3      if $\mathrm{LP}_{i+1}^t \in \mathrm{L}_{i+1}^t$ then
4          change $\mathrm{LP}_{i+1}^t$ to point to $\mathrm{L}_i^t$
5      else
6          if $\mathrm{LP}_{i+1}^t \in {}^*\mathrm{LP}_{i+2}^t$ then
7              change $\mathrm{LP}_{i+1}^t$ to $\mathrm{FP}_{i+1}^t$ that points to $\mathrm{L}_{i+1}^t$
8          else if $\mathrm{L}_{i+1}^t = {}^*\mathrm{LP}_{i+2}^t$ then
9              change $\mathrm{LP}_{i+1}^t$ to $\mathrm{TFP}_{i+1}^t$ that points to $\mathrm{L}_{i+1}^t$
10         else
11             change $\mathrm{LP}_{i+1}^t$ to $\mathrm{TFP}_{i+1}^t$ that points to $\mathrm{L}_{i+1}^t$
12             change $\mathrm{FP}_{i+1}^t$ to point to $\mathrm{L}_{i+1}^t$
13         end
14         on $\mathrm{L}_{i+1}^t$, add $\mathrm{LP}_{i+1}^t$ that points to $\mathrm{L}_i^t$
15     end
16     if $i$ >0 and $\mathrm{LP}_i^t \notin \mathrm{L}_i^t$ then
17         add $\mathrm{FP}_i^t$ on $\mathrm{L}_i^t$ that points to $\mathrm{L}_i \ni \mathrm{LP}_i^t$
18     end
19 end
```

current L_i^t (Lines 3,4). This corresponds to the situation in the right part of Figure 2.3 where t crosses A. Otherwise, LP_{i+1}^t is added to the current L_{i+1} of t (Line 14). In between, we update the forwarding pointers according to the three different cases: (**1**) If the old LP_{i+1}^t is on the lookup-path because it is in $^*\mathrm{LP}_{i+2}^t$, it is modified to a FP_{i+1}^t (Lines 6,7). (**2**) If t returned to $^*\mathrm{LP}_{i+2}^t$, we replace the old LP_{i+1}^t with a temporary forwarding pointer to $^*\mathrm{LP}_{i+2}^t$ (Lines 8,9). The FP_{i+1}^t in $^*\mathrm{LP}_{i+2}^t$ is implicitly overwritten by Line 14. (**3**) In all other cases where t moves from one square to another in $(^*\mathrm{LP}_{i+2}^t)^8$, we replace the old LP_{i+1}^t with a temporary forwarding pointer to the new L_i and update FP_{i+1}^t, which is located in $^*\mathrm{LP}_{i+2}^t$ (Lines 11,12). Finally, if the new LP_{i+1}^t does not point to the L_i that contains LP_i^t, a forwarding pointer is added to L_i^t. This FP_i^t points to the L_i that contains LP_i^t (Lines 16, 17). This last case arises for example in Figure 2.4 when t moves along the dashed path, such that the publish algorithm is triggered at the circled area. In that case, a FP_i^t is added in square [D] and points to square [C], which holds LP_i^t.

When t needs to update a pointer (LP_i^t or FP_i^t) on an arbitrary L_i, t sends a command message to the node that hosts the pointer, where the command message indicates how the pointer should be modified. In order to create a new pointer on a L_i, t sends a create message to the node closest to the position p_t in L_i. If t sends a create message to set a LP_i^t or FP_i^t at position p_t, but there already exists a pointer (LP_i^t, FP_i^t or TFP_i^t) at this position, the existing pointer is overwritten.

Algorithm 2.2: Lookup protocol of node s to route to node t

```
1  if $t \in \mathrm{L}_0^s \cup (\mathrm{L}_0^s)^8$ then exit
2  for $i = 1$; true; $i{+}{+}$ do
3      if $P_i^t \in \mathrm{L}_i^s$ or $P_i^t \in (\mathrm{L}_i^s)^8$ then
4          $p = P_i^t$
5          break
6      end
7  end
8  Follow $p$ until $\mathrm{LP}_1^t$ is reached
9  Route to a node closest to an arbitrary point on land in $^*\mathrm{LP}_1^t$
10 Forward to $t$
```

2.7.2 MLS Lookup

A lookup request for node t issued by a node s is routed according to Algorithm 2.2. If the destination node t is in the same unit square or an immediate neighbor, node s and t can communicate directly over their radio. Because s needs to know all its neighbors in $\mathrm{L}_0^s \cup (\mathrm{L}_0^s)^8$ for routing, this situation can be detected immediately and the lookup stops (Line 1).

Then, for increasing size of the levels, s searches a pointer of t in L_i^s and then in the squares of $(\mathrm{L}_i^s)^8$. Note that the lookup request accepts any kind of pointer of node t, whether it is an LP_i^t, FP_i^t or TFP_i^t. Furthermore, remember from Chapter 2.4 that the squares of $(\mathrm{L}_i^s)^8$ need to be accessed in a given order.

In the second phase of the lookup algorithm, the lookup request is routed along the pointers until it reaches LP_1^t (Line 8). Because $^*\mathrm{LP}_1^t$ does not contain a LP_0^t, the lookup picks an arbitrary position p on land in $^*\mathrm{LP}_1^t$ and routes the lookup request to the node closest to p (Line 9). From that node, the lookup can be sent directly to t (Line 10).

2.8 Analysis

We devote this section to the analysis of `MLS`. In particular, we show that `MLS` works in a concurrent setup, where publish requests and lookup requests occur simultaneously. We prove that a lookup request finds its destination in $\mathcal{O}(d)$ hops, where d is the distance between the sender and the destination. Also, we show that the amortized cost for publishing the position data is $\mathcal{O}(d \log d)$, where d denotes the distance a node has moved. In order to prove these properties, we need to limit the maximum speed of nodes, denoted v_{max}^{node}. Throughout the proofs, we introduce different constraints on the value of v_{max}^{node}. After proving the correctness of `MLS`, we determine the maximum node speed that satisfies all these constraints.

We base the lookup performance on the distance $|st|$ between two nodes s and t. In the static case, this distance is well defined during an entire lookup operation. However, in the concurrent setting, both nodes, s and t, might be moving while a lookup request is executing, and the distance $|st|$ changes over time. In our analysis of the lookup algorithm, we determine the distance $|st|$ when a lookup request is issued and base the performance analysis on this value.

2.8.1 Lookup Analysis

For the first phase of the lookup algorithm, we show that a lookup request for a node t can be routed such that it finds a level pointer to node t. Then, for the second phase of the lookup algorithm, we prove that the lookup request can be routed along the pointers of t to finally reach t.

Lookup – Phase 1

We consider a lookup request for a node t issued by a node s, where $d = |st|$ is the distance between s and t at the moment when s issues the request. In this section, we prove that the time needed to find a first location pointer for t is in $\mathcal{O}(d)$. To start, we give a lower bound on TTL_i such that a lookup request cannot miss a pointer[6] to t due to concurrency. Then, we show that the lookup request meets a pointer to t at the latest while visiting $\mathrm{L}^s_{k+1} \cup (\mathrm{L}^s_{k+1})^8$ for a given k dependent on d.

Lemma 2.3. *Given that t is (and remains) located in $\mathrm{L}^s_i \cup (\mathrm{L}^s_i)^8$ and maintains a LP^t_i in this area, a lookup request issued by s finds a pointer to t at the latest while visiting $\mathrm{L}^s_i \cup (\mathrm{L}^s_i)^8$, if $\mathrm{TTL}_i \geq \eta 2^i \rho(\sqrt{2} + 8\sqrt{5})$.*

Proof. When t needs to relocate its LP^t_i, it sends a message m to create a new LP^t_i on L^t_i, which takes at most $\Delta t_m \leq \eta 2^i \rho \sqrt{2}$ time to arrive (traversing L^t_i). At the same time, t sends a message m' to transform the outdated LP^t_i to a FP^t_i or TFP^t_i. For this worst case analysis, we assume that m' is delivered instantaneously.[7]

A lookup request fails to find LP^t_i if it interleaves with these publish messages such that it arrives at the position p of the new LP^t_i before m, and if it reaches the position p' of the outdated LP^t_i after m'. In the worst case, t relocates LP^t_i into L^s_i at time T_0 and the lookup request visits p just before m arrives at $T_1 < T_0 + \Delta t_m$. The lookup request then continues its search in the squares of $(\mathrm{L}^s_i)^8$ and might choose the path such that it visits p' only on its last step, which is at the latest after $\Delta t_{lookup} \leq \eta 2^i \rho 8\sqrt{5}$ (see proof of Lemma 2.1). Thus, the lookup request reaches p' at the latest at $T_2 \leq T_1 + \Delta t_{lookup} < T_0 + \eta 2^i \rho(\sqrt{2} + 8\sqrt{5})$. Even if the outdated LP^t_i was

---

[6] A pointer to t is either a LP^t, a FP^t or a TFP^t.

[7] There might also be a message to update a potential FP^t_i, which is of no importance for this proof.

transformed to a TFP_i^t at T_0, the TFP_i^t exists at least until $T_0 + \mathrm{TTL}_i > T_2$ and the lookup request finds the TFP_i^t. (Note that if the outdated LP_i^t is transformed to a FP_i^t, the time during which p' hosts a pointer to t is even longer, because the FP_i^t is transformed to a TFP_i^t after t updates LP_{i+1}^t.) $\square$

In the static case where publish requests and lookup requests do not interfere, a lookup request finds a pointer to t at the latest while visiting $\mathrm{L}_{k+1}^s \cup (\mathrm{L}_{k+1}^s)^8$, if the side length of L_k is at least d ($d \leq \rho 2^k$). (We can argue that $s \in \mathrm{L}_{k+1}^s$ and thus t is at most d away from L_{k+1}^s. At the same time, the distance from t to any L_k outside $\mathrm{L}_{k+1}^s \cup (\mathrm{L}_{k+1}^s)^8$ is at least $\rho 2^k \geq d$. Because $\alpha < 1$, t must have created a LP_{k+1}^t in $\mathrm{L}_{k+1}^s \cup (\mathrm{L}_{k+1}^s)^8$.)

For the concurrent case, we weaken this result and show that the lookup request finds a pointer to t at the latest while visiting $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$, where $\rho 2^k \geq d > \rho 2^{k-1}$ and $\gamma \in \mathbb{N}^+$, if the maximum node speed is bounded by

$$v_{max}^{node} < \frac{1 - \frac{\alpha}{2} - 2^{-\gamma}}{\eta\sqrt{2}} \tag{2.1}$$

and the temporary forwarding pointers exist long enough:

$$\mathrm{TTL}_i \geq \eta 2^{i+1} \rho (1/\sqrt{2} + 8\sqrt{5}) \tag{2.2}$$

Note that increasing the value of γ results in a higher node speed, but a lookup request might need to search longer until it finds a first pointer to t. We keep $\gamma \geq 1$ as a parameter of `MLS` to tune its performance.

Lemma 2.4. *Consider a lookup request for t issued by s and let d be the distance $|st|$ at the moment when the request is issued. For any $\gamma \geq 1$, $k \in \mathbb{N}$ such that $\rho 2^{k-1} < d \leq \rho 2^k$, and if the Equations (2.1) and (2.2) hold, the lookup request finds a pointer to t at the latest on one of the levels in $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$.*

Proof. In a first step, we show that t has created a $\mathrm{LP}_{k+\gamma}^t$ in $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$ at the latest when the lookup request is issued. This property is necessary to ensure that the lookup request cannot arrive too early on $\mathrm{L}_{k+\gamma}$ and miss $\mathrm{LP}_{k+\gamma}^t$ because it is not yet created.

By definition, $s \in \mathrm{L}_{k+\gamma}^s$ and thus t is at most $d \leq \rho 2^k$ away from $\mathrm{L}_{k+\gamma}^s$. At the same time, the distance from t to any $\mathrm{L}_{k+\gamma-1}$ outside $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$ is at least $\rho(2^{k+\gamma} - 2^k)$. Furthermore, we know that t sent off a message to create a $\mathrm{LP}_{k+\gamma}^t$ in $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$ when it entered this region by more than $\alpha 2^{k+\gamma-1}\rho$ (lazy publishing). Thus, t moved at least a distance $\Delta d = \rho 2^k(2^\gamma(1 - \frac{\alpha}{2}) - 1)$ after sending off the message to create $\mathrm{LP}_{k+\gamma}^t$ and when the lookup request was issued.

At the limit, the update message to create $\mathrm{LP}_{k+\gamma}^t$ needs to be sent across $\mathrm{L}_{k+\gamma}$ which needs $\Delta t_{update} \leq \eta 2^{k+\gamma}\rho\sqrt{2}$. But the lookup request is only issued after t has moved Δd, which takes at least $\Delta t_{move} \geq \frac{\Delta d}{v_{max}^{node}}$. By Equation (2.1), $\Delta t_{move} > \eta 2^{k+\gamma}\rho\sqrt{2} \geq \Delta t_{update}$, which shows[8] that the lookup request is issued after a $\mathrm{LP}_{k+\gamma}^t$ has been created in $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$.

To conclude the proof, we show that a $\mathrm{TFP}_{k+\gamma}^t$ lives long enough to catch all cases where the lookup request does not find a $\mathrm{LP}_{k+\gamma}^t$. From Lemma 2.3 we know that the lookup request finds a pointer to t while t remains inside $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$. Before t can relocate $\mathrm{LP}_{k+\gamma}^t$ outside $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$, it must move at least $\Delta d' = \rho 2^k(2^\gamma(1+\frac{\alpha}{2})-1)$ after the lookup was issued, which takes at least $\Delta t \geq \frac{\Delta d'}{v_{max}^{node}} > \eta 2^{k+\gamma}\rho\sqrt{2}$ by (2.1). Thus, t might create a $\mathrm{TFP}_{k+\gamma}^t$ in $\mathrm{L}_{k+\gamma}^s \cup (\mathrm{L}_{k+\gamma}^s)^8$ at least Δt after the lookup request was issued. By Lemma 2.1, the lookup request finishes visiting the levels of $\mathrm{L}_{k+\gamma}$ at the latest after $\Delta t_{lookup} \leq \eta 2^{k+\gamma+1}\rho(\sqrt{2}+8\sqrt{5})$. Thus, to ensure that the $\mathrm{TFP}_{k+\gamma}^t$ has not expired, it must have lived for at least $\Delta t_{lookup} - \Delta t$, which holds by Equation (2.2). □

The previous lemma states that a lookup request can find a pointer to t. We now show that the lookup request meets another pointer to t when it is routed along a forwarding pointer (or temporary forwarding pointer). However, we need to restrain the maximum node speed to

$$v_{max}^{node} < \frac{\alpha}{\eta \cdot 2(3\sqrt{2}+\alpha)} \tag{2.3}$$

to ensure that this holds in all situations.

Lemma 2.5. *A lookup request that finds a* FP_i^t *or a* TFP_i^t *also finds a pointer to* t *in* ${}^*\mathrm{FP}_i^t$ *or* ${}^*\mathrm{TFP}_i^t$*, respectively, if the maximum node speed satisfies Equation (2.3).*

Proof. We need to show that **[a]** a pointer p' for t has been written in ${}^*\mathrm{FP}_i^t$ (${}^*\mathrm{TFP}_i^t$) before the lookup request following FP_i^t (TFP_i^t) arrives, and that **[b]** p' cannot expire before the lookup request arrives, if p' has been transformed to a temporary forwarding pointer. Let us denote the found FP_i^t (TFP_i^t) by p, the L_i that contains p by A, and the L_i where p points by B. Throughout the proof, we refer to the lines of the publish algorithm presented in Chapter 2.7.1.

The found pointer p was created by one of the Lines 2, 7, 9, 11, 12, or 17. If $p = \mathrm{TFP}_i^t$ was created by Line 2, it points to the L_i^t where t was located when t sent the message m to change the FP_i^t to p. By the Lines 16-18, we know that if $B = {}^*\mathrm{TFP}_i^t$ did not contain LP_i^t, t sent a message m' to create a FP_i^t in B. If $p = \mathrm{FP}_i^t$ was created by Line 17, $B = {}^*\mathrm{FP}_i^t$ contains LP_i^t by

---

[8] Note that we did not consider that the lookup request needs some time to visit the levels $1 \ldots k+\gamma-1$, because there are situations where this time is negligible small. Including this time would allow for a slightly better v_{max}^{node}, but unnecessarily complicate the proofs.

definition. If p was created by Line 7, 9, 11, or 12, t sent a message m' to create a LP_i^t in B at the same time while sending a message m to create p.

Because t sends the message m' to create the necessary pointer p' in B no later than m that creates p, the lookup request cannot arrive at the location of p' in B before m'. Otherwise, m' would have been sent over a sub-optimal route contradicting the triangle inequality. Therefore, condition [**a**] holds.

Consider the case where p is a TFP_i^t and T_0 is the time when t sent message m to crate p. Then, B (where p points) is the L_i where t was located at T_0, and B contains as p' either a LP_i^t or a FP_i^t (Lines 14, 16-18). Because t is located in B at T_0, it needs to move at least $\alpha 2^{i-1}\rho$ away from B until p' is changed to a TFP_i^t. Also, at T_0, t is at most $\alpha 2^i \rho$ away from A (Line 1), and therefore m arrives no later than $T_1 \leq T_0 + \eta 2^i \rho(\sqrt{2} + \alpha)$ to create $p = \mathrm{TFP}_i^t$. At the limit, the lookup request reads p just before it expires at time $T_2 = T_1 + \mathrm{TTL}_i$, and then moves to p' in B. Because the air-distance $p - p'$ is bounded by $2^i \rho\sqrt{8}$, the lookup request might arrive at p' no later than $T_3 \leq T_2 + \eta 2^i \rho\sqrt{8} \leq T_0 + \mathrm{TTL}_i + \eta 2^i \rho(3\sqrt{2} + \alpha)$. By this time, p' must not be expired, which requires that it was created after $T_4 > T_3 - \mathrm{TTL}_i$. Before T_4, t moved at most $\Delta d = (T_4 - T_0) \cdot v_{max}^{node}$. By Equation (2.3), $\Delta d < \rho 2^{i-1}\alpha$, which shows that t has changed p' to a TFP_i^t after T_4 and that p' cannot expire before the lookup request arrives.

For the second case where p is a FP_i^t, we use the fact that p is changed to a TFP_i^t if t moves away from A by more than $\alpha \cdot 2^i \rho$ (Lines 1,2), and therefore an update message m'' from t to p sent at T_0 arrives at the latest at $T_1 \leq T_0 + \eta 2^i \rho(\sqrt{2} + \alpha)$. If t moves out of B and modifies p' to a TFP_i^t at T_0, it sends a message m'' to p (Line 2 or 12). In the worst case, the lookup request reads p just before m'' arrives and visits p'. Because the air distance $p - p'$ is bounded by $2^i \rho\sqrt{8}$, the lookup request arrives at p' no later than $T_2 \leq T_1 + \eta 2^i \rho\sqrt{8} \leq T_0 + \eta 2^i \rho(\alpha + 3\sqrt{2})$. From Equation (2.2), we know that $\mathrm{TTL}_i \geq \eta 2^{i+1}\rho(1/\sqrt{2} + 8\sqrt{5})$. Because $\alpha < 1$, p' cannot expire before the lookup request arrives.

□

We have shown that a lookup request can find a pointer to t and that it can follow pointers to find new pointers. In the following, we give upper bounds on the time needed to find a LP_i^t after a lookup request has been routed to an arbitrary FP_i^t or TFP_i^t. We tackle this problem by limiting the maximum number of forwarding hops that the lookup request has to follow until it reaches LP_i^t. If the lookup request finds a FP_i^t, we show that it is routed to the corresponding LP_i^t in at most β forwarding hops, if the maximum node speed is bounded by

$$v_{max}^{node} \leq \frac{\alpha(\beta - 2)}{2\eta(\sqrt{2} + \alpha + \beta\sqrt{8})} \tag{2.4}$$

and $\beta > 2$. If the lookup request finds a TFP_i^t, we show that it is routed to the corresponding LP_i^t in at most β_T forwarding hops, if the maximum node speed is bounded by

$$v_{max}^{node} \leq \frac{\alpha 2^{i-1}\rho(\beta_T - 2)}{\eta 2^i \rho(\sqrt{2} + \alpha + \beta_T\sqrt{8}) + \mathrm{TTL}_i} \tag{2.5}$$

and $\beta_T > 2$. The values of β and β_T become two additional tuning parameters of `MLS` which influence v_{max}^{node} and the time a lookup needs to find its destination.

We will use the following helper lemma, which says that a node t needs to move at least a certain distance between consecutive updates to its LP_i^t.

Lemma 2.6. *Between successive updates to* LP_i^t*, node* t *moves at least* $\Delta d_{update} \geq \alpha 2^{i-1}\rho$.

Proof. When a node t updates LP_i^t, the new LP_i^t points to L_i^t, the L_i that contains t. Due to the publish policy (Definition 2.2), t only needs to update LP_i^t after it has moved out of L_i^t by $\alpha 2^{i-1}\rho$ and thus t must move at least Δd_{update} before it needs to update LP_i^t. □

Lemma 2.7. *Given a lookup request for a node* t *that has found a* FP_i^t*, and that* v_{max}^{node} *satisfies Equation (2.4), the lookup request can be routed to the corresponding* LP_i^t *in at most* $\Delta t \leq \beta\eta 2^i \rho\sqrt{8}$ *for a given* $\beta > 2$.

Proof. Node t maintains FP_i^t as long as $\delta_i^t < \alpha 2^i \rho$ (Lines 1,2 of the publish algorithm in Chapter 2.7.1) and t does not return to $^*\mathrm{LP}_{i+1}^t$ (Line 9). When t updates its LP_i^t, it also sends an update to FP_i^t, which takes at most $\Delta t_{update} \leq \eta 2^i \rho(\sqrt{2} + \alpha)$ time to arrive. When $\delta_i^t = \alpha 2^i \rho$, t updates LP_{i+1}^t and changes the FP_i^t to a TFP_i^t (Line 2). Thus, when the lookup request reads FP_i^t, it reads a direction that is at most Δt_{update} outdated.

Consider the case where a node t moved from level X to Y and sent an update m to FP_i^t, such that FP_i^t points to Y instead of X. If a lookup request reads FP_i^t before m arrives, it first visits XY, where it finds a TFP_i^t that points to Y [9]. By Lemma 2.5, this TFP_i^t has not yet expired.

For each forwarding (FP_i^t or TFP_i^t), the lookup request has to move an air distance bounded by $2^i\rho\sqrt{8}$. Thus, the last lookup request relayed by X arrives in Y at most $\Delta t_{update} + 2\eta 2^i \rho\sqrt{8}$ after m was sent off. But during this time, t might have moved - and left behind yet other (possibly temporary) forwarding pointers.

In order to limit the routing time to $\beta\eta 2^i\rho\sqrt{8}$, the lookup request can follow at most β forwarding pointers until it reaches LP_i^t. Thus, the total time between sending m and when the lookup request reaches LP_i^t is $\Delta t_{tot} \leq$

---

[9] It is possible that $^*\mathrm{FP}_i^t$ does not contain a TFP_i^t that points to Y. We discuss this case later on.

$\Delta t_{update} + \beta\eta 2^i\rho\sqrt{8} \leq \eta 2^i\rho(\sqrt{2} + \alpha + \beta\sqrt{8})$. During this time, t moves at most $\Delta d \leq \Delta t_{tot} \cdot v^{node}_{max}$. By Equation (2.4), $\Delta d \leq \rho(\beta - 2)\alpha \cdot 2^{i-1}$ and causes at most $\Delta d/\Delta d_{update} \leq \beta - 2$ additional forwarding pointers by Lemma 2.6. Including the two forwarding hops to visit X and Y, the lookup request has to follow maximally β pointers, which takes at most $\beta\eta 2^i\rho\sqrt{8}$.

If $^*\mathrm{FP}^t_i$ does not contain a TFP^t_i that points to Y, then t must have returned to X and overwritten the TFP^t_i with a LP^t_i, a FP^t_i or a more recent TFP^t_i. Following such a FP^t_i or TFP^t_i short-cuts the path to LP^t_i, and the lookup request finds LP^t_i even faster. □

Lemma 2.8. *Given a lookup request for a node t that has found a TFP^t_i, and that v^{node}_{max} satisfies Equation (2.5), the lookup request can be routed to the corresponding LP^t_i in at most $\Delta t \leq \beta_T\eta 2^i\rho\sqrt{8}$ for a given $\beta_T > 2$.*

Proof. We distinguish if the found TFP^t_i was created due to [**a**] Line 9 or 11 of the publish algorithm in Chapter 2.7.1 or [**b**] by Line 2. For both cases, we consider the time T_0 when t sends a message m to create TFP^t_i. Also, if the lookup request is forwarded along a FP^t_i (TFP^t_i), it finds the next pointer to t at the latest after $\Delta t_{forward} \leq \eta 2^i\rho\sqrt{8}$, because a FP^t_i (TFP^t_i) points to a neighboring L_i. Therefore, it is sufficient to show that the lookup request reaches LP^t_i at the latest after β_T forwarding hops.

For case [**a**], t changed its old LP^t_i to a TFP^t_i because it has moved away from the L_i that contains LP^t_i by more than $\alpha 2^{i-1}\rho$. Thus, message m needs $\Delta t_{update} \leq \eta 2^i\rho(\alpha/2 + \sqrt{2})$ until it reaches the old LP^t_i. From Line 14 and Lemma 2.5, we know that the lookup request finds a pointer to t in $^*\mathrm{TFP}^t_i$. Also, a lookup request that reads the TFP^t_i just before it expires, arrives in $^*\mathrm{TFP}^t_i$ at the latest at $T_1 = T_0 + \Delta t_{update} + \mathrm{TTL}_i + \Delta t_{forward}$, where it might not find LP^t_i (created due to Line 14), because t has already moved away. The lookup request must catch up with t and find LP^t_i at the latest at $T_2 = T_0 + \Delta t_{update} + \mathrm{TTL}_i + \Delta t$. By this time, t has moved up to $\Delta d = (T_2 - T_0)v^{node}_{max}$, which is bounded by $\Delta d \leq (\beta_T - 2)\alpha 2^{i-1}\rho$ using Equation (2.5). From Lemma 2.6, we deduce that t has caused at most $\lfloor \frac{\Delta d}{\Delta d_{update}} \rfloor \leq \beta_T - 2$ (possibly temporary) forwarding pointers due to its motion, and the lookup request needs to follow at most a total of β_T pointers until it reaches LP^t_i.

We follow a similar argumentation for the second case [**b**], where t changed a FP^t_i to the found TFP^t_i. At T_0, $\delta^t_i = \alpha 2^i\rho$ and thus the message m needs $\Delta t_{update} \leq \eta 2^i\rho(\alpha + \sqrt{2})$ until it reaches the outdated FP^t_i. Case [**b**] is also different in that $^*\mathrm{TFP}^t_i$ might never contain a LP^t_i, but only a FP^t_i (Lines 16-18). Therefore, the lookup request might have to follow 2 pointers until it reaches LP^t_i, even if t does not move at all after T_0. Again, the lookup request must catch up with t and find LP^t_i at the latest at $T_2 = T_0 + \Delta t_{update} + \mathrm{TTL}_i + \Delta t$. By this time, t has moved up to

$\Delta d = (T_2 - T_0)v_{max}^{node}$, which is bounded by $\Delta d \leq (\beta_T - 2)\alpha 2^{i-1}\rho$ using Equation (2.5). From Lemma 2.6, we deduce that t has caused $\lceil \frac{\Delta d}{\Delta d_{update}} \rceil \leq \beta_T - 2$ (possibly temporary) forwarding pointers due to its motion. (Note that we needed to round up the number of forwarding pointers because t did not update LP_i^t at T_0.) Therefore, the lookup request needs to follow at most a total of β_T pointers until it reaches LP_i^t.

□

We are now ready to assemble the first pieces of the puzzle and show that a lookup request finds a first LP^t in bounded time.

Lemma 2.9. *Given that* v_{max}^{node} *satisfies the Equations (2.1), (2.3), (2.4) and (2.5), and* TTL_i *satisfies Equation (2.2) for fixed values of* $0 < \alpha \leq 1$, $\gamma \geq 1$, $\beta > 2$, *and* $\beta_T > 2$, *then, a lookup request for node* t *issued by node* s *finds a level pointer* LP^t *in* $\mathcal{O}(d)$ *time, where* d *is the distance* $|st|$ *at the moment when the request is issued.*

Proof. By combination of the Lemmas 2.4, 2.7, and 2.8: A first pointer p to t is found at the latest in one of the squares $\mathrm{L}_u^s \cup (\mathrm{L}_u^s)^8$, where $u = \lceil \log_2 \frac{d}{\rho} \rceil + \gamma$ (Lemma 2.4). The necessary time the lookup request needs to visit all these levels is bounded by $T_1 \leq \eta 2^{u+1}\rho(\sqrt{2} + 8\sqrt{5})$ (Lemma 2.1). If p is a FP_i^t, the lookup request reaches the corresponding LP_i^t in $T_2 \leq \beta\eta 2^i \rho\sqrt{8}$ (Lemma 2.7), and if p is a TFP_i^t, the lookup request reaches the corresponding LP_i^t in $T_3 \leq \beta_T \eta 2^i \rho\sqrt{8}$ (Lemma 2.8). For T_2 and T_3, $i \leq u$. The total time T to find a first LP_i^t is bounded by

$$\begin{aligned}
T &\leq T_1 + \max(T_2, T_3) \\
&\leq \eta 2^{\lceil \log_2 (d/\rho) \rceil + \gamma + 1}\rho(\sqrt{2} + 8\sqrt{5} + \max(\beta, \beta_T)\sqrt{2}) \\
&\leq d \cdot \underbrace{\eta 2^{\gamma+2}(\sqrt{2} + 8\sqrt{5} + \max(\beta, \beta_T)\sqrt{2})}_{constant} \\
&\in \mathcal{O}(d)
\end{aligned}$$

□

Lookup – Phase 2

For the second phase of the lookup algorithm, we need to show that once a lookup request has found a first LP_i^t, it can follow the pointers and find the destination node t. We start with another helper lemma stating that if LP_i^t points to L_{i-1} and t is at most $\alpha 2^{i-1}\rho$ away from L_{i-1}, then L_{i-1} contains a LP_{i-1}^t or a FP_{i-1}^t.

Lemma 2.10. *As long as* $\delta_i^t < \alpha 2^i \rho$, ${}^*\mathrm{LP}_{i+1}^t$ *contains a node that hosts either a* LP_i^t *or a* FP_i^t.

Proof. By inspection of the publish algorithm presented in Chapter 2.7.1. The only places where pointers are transformed to TFP_i^t are on the Lines 2, 9 and 11. On Line 2, a FP_i^t is removed, because LP_{i+1}^t will no longer point to the level that contains the FP_i^t. But this only happens when $\delta_i^t \geq \alpha\rho 2^i$ (Line 1).

Line 9 or 11 is executed when $\delta_{i-1}^t \geq \alpha\rho 2^{i-1}$. However, if Line 9 or 11 executes, we know from Line 6 that the LP_j^t that is overwritten is not in $^*\mathrm{LP}_{j+1}^t$. We conclude that the pointer for t in $^*\mathrm{LP}_{i+1}^t$ is transformed to a TFP_i^t iff $\delta_i^t \geq \alpha\rho 2^i$. □

The following lemma states that a lookup request that has reached a LP_i^t can be routed along LP_i^t and find LP_{i-1}^t.

Lemma 2.11. *Under the condition that* $i > 0$*, and all constraints on* v_{max}^{node} *and* TTL_i *are satisfied, a lookup request can follow* LP_{i+1}^t *and find* LP_i^t *after* $\Delta t \leq \eta 2^i \rho\sqrt{8}(1 + \max(\beta, \beta_T))$.

Proof. First, we show that $^*\mathrm{LP}_{i+1}^t$ contains a pointer p to t when the lookup request arrives. Then, we apply Lemma 2.7 and Lemma 2.8 to bound the time to find LP_i^t.

While $\delta_i^t < \alpha\rho 2^i$, $^*\mathrm{LP}_{i+1}^t$ contains a LP_i^t or a FP_i^t (by Lemma 2.10). At T_0, when $\delta_i^t \geq \alpha\rho 2^i$, the FP_i^t in $^*\mathrm{LP}_{i+1}^t$ is changed to a TFP_i^t (Line 2 of the publish algorithm in Chapter 2.7.1) and a message m is sent to change LP_{i+1}^t, where it arrives at $T_1 \leq T_0 + \eta 2^i \rho(\alpha + \sqrt{8})$. A lookup request that follows the outdated LP_{i+1}^t before m arrives reaches the TFP_i^t in $^*\mathrm{LP}_{i+1}^t$ at the latest at $T_2 = T_1 + \eta 2^i \rho\sqrt{8} \leq T_0 + \eta 2^i \rho(\alpha + 4\sqrt{2})$. By Equation (2.2), $\mathrm{TTL}_i > T_2 - T_0$, and thus the TFP_i^t does not expire before the lookup request arrives.

So far, we have shown that the lookup finds a pointer to t in $^*\mathrm{LP}_{i+1}^t$. Following LP_{i+1}^t to reach p takes at most $\eta 2^i \rho\sqrt{8}$. If p is a FP_i^t, the additional time to route to LP_i^t is bounded by $\beta\eta \cdot 2^i \rho\sqrt{8}$ (Lemma 2.7). If p is a TFP_i^t, the additional time to route to LP_i^t is bounded by $\beta_T \eta 2^i \rho\sqrt{8}$ (Lemma 2.8). Thus, the total time to LP_i^t is upper-bounded by $\Delta t \leq \eta 2^i \rho\sqrt{8}(1+\max(\beta, \beta_T))$. □

Using the previous lemma, we can show that a lookup request can be routed from a LP_i^t to LP_{i-1}^t until it reaches LP_1^t, from where it is forwarded to $^*\mathrm{LP}_1^t$. It remains to verify that the lookup request can be sent directly to t from within $^*\mathrm{LP}_1^t$, which we show under the constraint that

$$v_{max}^{node} < \frac{\sqrt{2} - \alpha}{\eta(\alpha + 4\sqrt{2})} \tag{2.6}$$

Lemma 2.12. *If* v_{max}^{node} *satisfies (2.6), a lookup request that has found* LP_1^t *can be sent directly to* t *from within* $^*\mathrm{LP}_1^t$.

Proof. When t moves out of $^*\mathrm{LP}_1^t$ by more than $\alpha\rho$, it sends an update message m to change LP_1^t at time T_0 (Definition 2.2). This message arrives at $T_1 \leq T_0 + \eta\rho(\alpha + \sqrt{8})$. A lookup request that reads LP_1^t just before m arrives, is forwarded to $^*\mathrm{LP}_1^t$. Because $^*\mathrm{LP}_1^t$ does not contain a LP_0^t, the lookup request is routed towards an arbitrary point on land in $^*\mathrm{LP}_1^t$, and might end up on a node u that is up to λ away from $^*\mathrm{LP}_1^t$ (Lines 8-10 of the lookup algorithm in Chapter 2.7.2). At the limit, this forwarding to u requires to traverse L_1^t, and the lookup request arrives at u at $T_2 \leq T_1 + \eta\rho\sqrt{8}$. By this time, t has moved $\Delta d \leq (T_2 - T_0)v_{max}^{node} < \rho(\sqrt{2} - \alpha)$ and is at most $\Delta d + \alpha\rho \leq \lambda$ away from $^*\mathrm{LP}_1^t$. Both, u and t are at most λ away from $^*\mathrm{LP}_1^t$, and the diameter of $^*\mathrm{LP}_1^t$ is $\rho\sqrt{2} = \lambda$. Thus, the total distance between u and t is at most $3\lambda = r_{min}$, which shows that u can forward the lookup request directly to t. □

Theorem 2.13. (LOOKUP) *Given that* v_{max}^{node} *satisfies the Equations (2.1), (2.3), (2.4), (2.5), and (2.6), and that* TTL_i *satisfies the Equation (2.2) for fixed values of* $0 < \alpha \leq 1$, $\gamma \geq 1$, $\beta > 2$, *and* $\beta_T > 2$, *then, a lookup request for a node* t *issued from a node* s *takes* $\mathcal{O}(d)$ *time to reach* t, *where* $d = |st|$ *is the distance between* s *and* t *when the request is issued.*

Proof. By Lemma 2.9, we know that the time to find a first level pointer is bounded by $T_1 \leq d \cdot \eta 2^{\gamma+2}(\sqrt{2} + 8\sqrt{5} + \max(\beta, \beta_T)\sqrt{2})$ Also, we know that this first level pointer is found at the latest on level $\lceil \log_2 \frac{d}{\rho} \rceil + \gamma$. From Lemma 2.11, we deduce that following the level pointers from level $\lceil \log_2 \frac{d}{\rho} \rceil + \gamma$ downto 1 takes

$$T_2 \leq \sum_{j=1}^{\lceil \log_2 d/\rho \rceil + \gamma} \eta 2^j \rho\sqrt{8}(1 + \max(\beta, \beta_T))$$

and the time to forward the lookup request from LP_1^t to t can be bounded using Lemma 2.12 to

$$T_3 \leq \eta\rho\sqrt{8}(1 + \max(\beta, \beta_T))$$

The total time until the lookup request reaches t is therefore bounded by

$$\begin{aligned} T_{lookup} &\leq T_1 + T_2 + T_3 \\ &\leq T_1 + \sum_{j=0}^{\lceil \log_2 d/\rho \rceil + \gamma} \eta 2^j \rho\sqrt{8}(1 + \max(\beta, \beta_T)) \\ &< T_1 + d\eta 2^{\gamma+2}\sqrt{8}(1 + \max(\beta, \beta_T)) \\ &\leq d \cdot \underbrace{\eta 2^{\gamma+2}(3\sqrt{2}(1 + \max(\beta, \beta_T)) + 8\sqrt{5})}_{constant} \in \mathcal{O}(d) \end{aligned}$$

□

2.8.2 Maximum Node Speed

While proving the lemmas in the previous section, we formulated constraints on the value of v_{max}^{node} and TTL_i. In this section, we collect all of these constraints and present the maximum node speed for which `MLS` is proven to work. The value of TTL_i only needs to satisfy Equation (2.2). Because enlarging TTL_i implies a reduction of v_{max}^{node} by Equation (2.5), we choose TTL_i as small as possible:

$$\mathrm{TTL}_i = \eta 2^{i+1} \rho(1/\sqrt{2} + 8\sqrt{5}) \tag{2.7}$$

To determine the maximum v_{max}^{node}, we need to fix the parameters $0 < \alpha < 1$, $\beta > 2$, $\gamma \geq 1$ and $\beta_T > 1$. The value of η is given indirectly by the topology and the underlying routing algorithm. While maximizing v_{max}^{node}, we opt to minimize T_{lookup}, the worst case time of a lookup request, which depends on the same parameters. Thus, we need to determine α, β, γ and β_T, such as to maximize v_{max}^{node} and minimize T_{lookup} under the constraints that the Equations (2.1), (2.3), (2.4), (2.5) and (2.6) are satisfied.

For $\gamma \to \infty$; $\beta \to \infty$, $\beta_T \to \infty$, we receive that $v_{max}^{node} \approx \frac{0.0845}{\eta}$ for $\alpha \approx 0.863$. This is an unreachable upper bound, because the maximum cost of a lookup request would grow to infinity. Clearly, there is a tradeoff between maximizing v_{max}^{node} and minimizing T_{lookup}. The higher we choose the maximum node speed, the longer is the worst-case time of a lookup request. But while T_{lookup} increases exponentially with γ, the value of γ barely increases v_{max}^{node}. Similarly, β and β_T have an impact on v_{max}^{node} while they are below 20, higher values only increase T_{lookup}. Twiddling the parameters, we found $\gamma = 1$, $\beta = 5$, and $\beta_T = 19$ to optimize the tradeoff between v_{max}^{node} and T_{lookup}. Using these values, v_{max}^{node} becomes maximal for $\alpha = 0.8$. We can now state our lower bound on the maximum node speed for which `MLS` is proven to work.

Theorem 2.14. *For $\alpha = 0.8$, $\beta = 5$, $\beta_T = 19$ and $\gamma = 1$, the nodes might move at speed $v_{max}^{node} \leq \frac{1}{15 \cdot \eta}$ without breaking* `MLS`.

Proof. By plugging the parameters into the Equations (2.1), (2.3), (2.4), (2.5) and (2.6). □

In the absence of lakes, $1/\eta$ can be interpreted as the minimum speed at which messages are routed by the underlying routing algorithm. Therefore, the above result shows that nodes might move at a speed that is only 15 times smaller than the routing speed, which is remarkably fast. If we consider real-world nodes such as the mica2 nodes from UC Berkeley, a data packet experiences around 50 ms delay while being forwarded by a node. Thus, a packet can be sent about 40 hops per second. If we assume that a message is forwarded around 10 meters per hop, the message speed reaches around 400 meters per second. In this setup and in the absence of lakes, the maximum node speed is bounded by $400/16 = 25$ meters per second (90 km/h, about 56 mph), which exceeds by far the node speed in typical network applications.

2.8.3 Publish Analysis

Last but not least, we need to consider the cost of the publish algorithm. While a node is moving, it continuously sends updates onto its different levels. This produces messages that need to be routed by the nodes of the system. Thus, bounding the message overhead of the publish algorithm is of critical importance to ensure that the overall routing cost induced by the moving nodes is reasonable. First, we derive the maximum cost of publishing to a level. Then, we determine the amortized message cost for publishing while a node is moving.

Until now, we made use of an underlying routing algorithm that can deliver a message to its destination in $\eta \cdot d$ time, when the air distance between the sender and destination is d. For the following, we assume that the number of routing hops needed to send a message over a distance d is proportional to the routing time and write $\tilde{\eta} \cdot d$ to denote the number of routing hops needed to route a message to a target that is d away.

Lemma 2.15. *The cost to publish on level-i is bounded by $\tilde{\eta} 2^{i+2} \rho(\sqrt{8} + \alpha)$ message-hops.*

Proof. By inspection of the publish algorithm in Chapter 2.7.1. We determine the message cost c of each individual line to pick the execution with maximum cost.

On Line 2, FP_i^t is in the neighboring level-i, which t just left by more than $\alpha \cdot 2^i$, thus $c_2 \leq \tilde{\eta} 2^i \rho(\alpha + \sqrt{2})$. For the Lines 4, 7, 9, and 11, the distance from t to LP_{i+1}^t is maximally $2^i(\alpha + \sqrt{8})$ and the message cost is $c_{4,7,9,11} \leq \tilde{\eta} 2^i \rho(\alpha + \sqrt{8})$. FP_{i+1}^t on Line 12 is contained in $(\mathrm{L}_{i+1}^t)^8$, but only as long as $\delta_{i+1}^t < \alpha 2^{i+1}$. Thus, the cost to send a message to FP_{i+1}^t is bounded by $c_{12} \leq \tilde{\eta} 2^i \rho(2\alpha + \sqrt{8})$. For Line 14, LP_{i+1}^t is contained in L_{i+1}^t, and therefore the cost is $c_{14} \leq \tilde{\eta} 2^i \rho \sqrt{8}$. Finally, for Line 17, t reaches FP_i^t in $c_{17} \leq \tilde{\eta} 2^i \rho \sqrt{2}$ hops, as $\mathrm{FP}_i^t \in \mathrm{L}_i^t$. The execution with maximum cost visits the Lines 2, 11, 12, 14 and 17. Summing up the corresponding costs results in the indicated number of hops.

□

Using an amortized analysis, we can now show that the expected message overhead induced by the publish method can be bounded.

Theorem 2.16. (PUBLISH)

The amortized message cost of a node induced by the publish method is at most $\mathcal{O}(d \log d)$ message hops, where d denotes the distance the node moved.

Proof. A node t updates its LP_i^t when $\delta_{i-1}^t \geq \alpha\rho 2^{i-1}$ (Definition 2.2). After an update to LP_i^t, $t \in {}^*\mathrm{LP}_i^t$ and thus t needs to move at least $\alpha\rho 2^{i-1}$ before it needs to issue another update for LP_i^t (Lemma 2.6). Therefore, t needs to update LP_i^t at most $d/(\alpha\rho 2^{i-1})$ times while moving a distance d. The cost to publish on L_i is maximally $\tilde{\eta}2^{i+2}\rho(\sqrt{8}+\alpha)$ (Lemma 2.15), and the total cost for updating L_i while moving the distance d is bounded by $d\tilde{\eta}8\rho(\sqrt{8}+\alpha)/\alpha$. The total publish cost is given by the sum of the cost of each level to which t has to publish. Because $\forall j : \delta_j^t = 0$ after the initialization phase of the publish method, t only needs to publish on L_i if $d \geq \alpha\rho 2^{i-1}$, and the total publish cost is bounded by $d\log_2(2d/\alpha)\tilde{\eta}8\rho(\sqrt{8}+\alpha)/\alpha \in \mathcal{O}(d\log d)$. Note that the total number of levels is M (Chapter 2.2) and the publish cost is further bounded by $dM\tilde{\eta}8\rho(\sqrt{8}+\alpha)/\alpha$. □

2.9 Simulation

We support our theoretical results with a series of simulations, through which we show that the average lookup time and publish cost are well below the worst case costs. Our simulation framework implements mobile nodes that select their route using the random waypoint model and maintain their hierarchical lookup data according to Chapter 2.7.1. However, being mainly a proof of concept, the simulation abstracts from the underlying routing[10], where we delay messages according to the route length, but do not simulate the node-to-node routing. Also, we simulate only a (random) sub-set of the nodes as to improve the run-time. As a consequence, the storage of pointers is performed by the framework and relieves us from ensuring the minimum node density.

The techniques of Le Boudec et al. [61] to obtain a stationary regime cannot be applied, because the nodes are not memoryless with respect to their lookup hierarchy. Therefore, we obtain a close to stationary regime through a long first phase, during which each node moves at least to its first waypoint before any lookup request is issued. Afterwards, the nodes issue lookup requests to randomly chosen nodes, where each node only sends another lookup request when the previous request arrived.

We ran the simulation with 5000 nodes issuing a total of 100000 lookup requests and repeated it for different node speeds and two different world maps. Map_1 corresponds to the world shown in Figure 2.1. The maximal routing stretch due to lakes is 10, the side-length is 5300 units and ρ was chosen to be 1 unit. Map_2 is of the same size, but contains no lakes at all. The average lookup stretch is shown in the left plot of Figure 2.5. For increased node speed (1 corresponds to v_{max}^{node}), nodes produce more TFP, which helps lookup requests to find a first pointer to their destination, which

---

[10] Remember that the underlying routing can route a message to a given *position*. This is orthogonal to our main goal, where the location of the destination node is unknown.

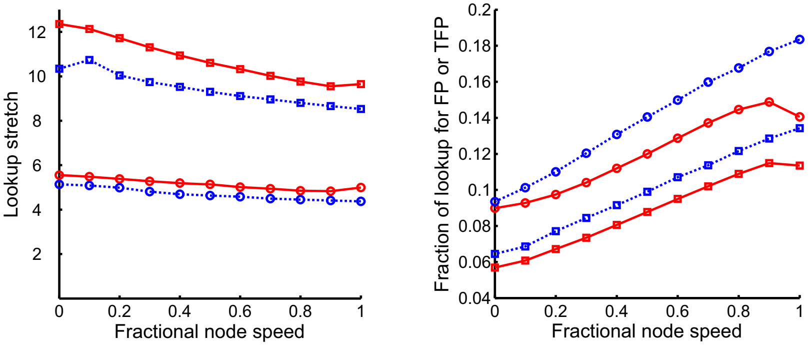


Figure 2.5: The left plot shows the lookup stretch depending on the node speed, where 1 is the maximum speed. The right plot shows the fraction of the time a lookup spends following FP and TFP. The solid lines indicate runs with Map$_1$ (a world as shown in Figure 2.1), the dotted lines show runs with Map$_2$ (no lakes). The results for the adapted fast-lookup are drawn with circles, the results for the sequential lookup with squares.

in turn decreases the lookup stretch. This fact is supported by the right plot of Figure 2.5, which shows that lookup requests spend more time following FP and TFP for increased node speed.

In order to obtain the average lookup stretch of approximately 6, we slightly modified the lookup algorithm: Instead of using the slow sequential search for a pointer in the 9 L_i of each level, the sender node s sends the lookup request to all 9 L_i in parallel. If a lookup request finds a pointer to the destination node t, it is immediately forwarded towards t and sends back a `FOUND` message to s. Lookup requests that do not find a pointer to t send back a `NOT_FOUND` message and die. s collects all the responses and only sends the lookup request to the next higher levels L_{i+1} if all responses are negative.

This fast-lookup approach reduces the lookup time by a factor 2 at the cost of increased message overhead. However, more than one copy of the lookup request might be routed towards t if no coordination between the 9 parallel lookup requests takes place. Consequently, each node needs some means to detect messages that were received several times.

For the publish requests, we measured the total message cost dependent on the distance a node moved (Figure 2.6). Using a least-square approach, we determined the average publish overhead to be $4.3 \cdot d \log d$, where d is the moved distance. This overhead is reasonably small and ensures that the network is not saturated with messages due to moving nodes.

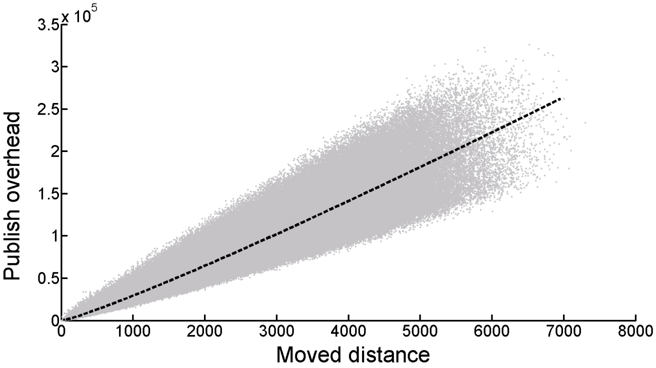


Figure 2.6: We captured over 600000 node motions during our simulation. This figure shows the publish overhead caused by a moving node plotted against the distance the node moved. The (least-square) average is indicated with the dashed line.

3

Routing in 3D Networks

We have seen that the geographic routing approach is very scalable as it is based solely on local information. In the preceding chapter, we have described a lookup system through which messages can even be delivered to nodes that are constantly moving. But all results we have seen so far are valid only for 2-dimensional networks. In this chapter, we examine how the geographic routing technique needs to be adapted to reliably deliver messages in 3-dimensional networks. In contrast to the previous chapter, however, we only consider the simpler case of static networks.

Clearly, messages can still be greedily forwarded towards their destination, which is likely to work fine in dense networks. But the recovery from local minima becomes more challenging, as the faces surrounding the network hole now expand in two dimensions and are much harder to capture. We show in Chapter 3.6.1 that it is indeed possible to describe the surface of network holes using only local information. Our approach is quite different from 2D as there is no equivalent to the planarization of a graph in 3D. Furthermore, we used the right/left hand rule in 2D to route along the 1-dimensional face, but there is no simple analogon in 3D to visit all nodes delimiting the surface of a hole. In fact, Durocher et al. have proved that there is no deterministic local routing algorithm for 3D networks that guarantees the delivery of messages if the nodes are not allowed to store any message state [30]. As an immediate consequence, there is neither a local memoryless algorithm to traverse all nodes on a given surface in a deterministic manner.

The proof in [30] has two parts: First, the authors show that the existence of a *k-local*[1] geographic routing algorithm for UBG implies the existence of a 1-local geographic routing algorithm for any connected graph. In the second part, they show that any deterministic 1-local routing algorithm can

---

[1] A *k-local* routing algorithm can base its routing decision on a k-neighborhood of the current node.

be defeated, disproving the existence of a k-local routing algorithm. In fact, even very simple graph structures do not allow for deterministic routing algorithms.

In the sequel, we present memoryless and local geographic routing protocols for 3-dimensional networks and compare them to other routing techniques. Unlike the deterministic greedy-face-greedy solutions in 2D, our approach applies a randomized recovery to lead messages out of local minima.

3.1 Random Walks

The non-existence of local, memoryless routing algorithms that deliver messages deterministically [30] has many direct and indirect consequences. Whereas it is possible to deterministically traverse a planar subdivision and report all nodes and faces [12], there is no corresponding algorithm in 3D. However, it has been shown in [27] that for *any* undirected graph, it is possible to assign each node a local ordering of its edges such that a routing algorithm can visit all nodes in $\mathcal{O}(n)$ time (deterministically!) by leaving a node through the edge succeeding the edge through which it entered. Unfortunately, the construction of the local edge-orderings requires a global view of the graph and has construction time cubic in the number of nodes. The greedy geographic routing we would like to apply to our 3D graphs is actually close to optimal - as long as the messages do not get stuck at local minima. But as there is also no deterministic recovery algorithm that could lead the messages out of local minima, we need to fall back to randomized recovery techniques which we describe breifly in this section.

A message moving around randomly in a network may seem very inefficient and too simplistic, but there is quite some work in this area indicating that random walks need not be as bad as it looks at a first glance. The two prominent models to capture a random walk on a graph $G = (V, E)$ are (1) the Markov chain, and (2) the flow in an electrical network obtained from G by replacing every edge by a resistance of 1Ω. See [68] for a survey of the topic. In the following, we use $n := |V|$ and $m := |E|$.

For our purpose, the hitting time H_{uv}, and the cover time C_G turn out to be most interesting. H_{uv} is the expected time until the random walk first visits vertex v when starting its walk at u, and C_G is the expected time needed to visit all nodes of G at least once. For arbitrary graphs, we have $C_G = \mathcal{O}(n \cdot m) = \mathcal{O}\big(n^3\big)$ [9], which poses also an upper bound on H_{uv}. The complete graph has optimal cover time $\Theta(n \log n)$, and the worst cover time of $\Theta\big(n^3\big)$ is obtained from the lollipop graph [38]. The 2D mesh has a non-optimal cover-time $\mathcal{O}\big(n \log^2 n\big)$, whereas the 3D mesh has optimal cover-time $\mathcal{O}(n \log n)$ [20]. Using the electrical resistance approach, the cover time of an arbitrary graph can be bounded to $C_G \leq 2mR_{span}$ [20], where R_{span} is the minimum resistance of a spanning tree in G. Since $R_{span} \leq n - 1$ for any graph, we obtain that $C_G \leq 2m(n-1)$. The hitting time H_{uv} seems to be intrinsically difficult to capture, but the somewhat related commute time κ_{uv}, the expected time to travel from u to v and back again, can be expressed by $H_{uv} < \kappa_{uv} = 2mR_{uv}$, where R_{uv} is the effective resistance between u and v [20].

A *random geometric graph* is obtained by placing n nodes uniformly at random in the unit square and connect two points if their distance is at most r. The minimal value of r such as to obtain a connected graph is subject of the percolation theory. It has been shown that if $r \geq r_{con} = \Theta\left(\sqrt{\log n/n}\right)$, the graph is connected w.h.p. [78]. Random geometric graphs with $r \geq \sqrt{8}r_{con}$ have optimal cover time of $\mathcal{O}(n \log n)$ [5]. Whereas this model seems appealing for wireless networks at the first moment, we need to keep in mind that the analysis only holds for $n \rightarrow \infty$. Furthermore, a connected random geometric graph requires so many network nodes that the graph tends to have no routing voids at all, causing no local minima to the greedy routing algorithm. In addition, wireless networks are obviously *not* random geometric graphs: There tend to be many holes in the network, where no nodes are deployed, which is ignored completely in this model.

3.2 Notation and Model

We now summarize the notation used to describe our 3D routing protocols before we show a lower bound on the routing stretch of *any* local routing algorithm in 3D. Whereas this lower bound applies for arbitrary graphs, we restrict our attention to unit ball graphs in Chapter 3.5. Unit ball graphs (short UBG) are the 3D equivalent to the unit disk graphs in 2D, and constitute a basic model for wireless networks where we assume that any two network nodes are connected iff their distance is below a certain threshold r_{max}, the maximal transmission radius[2]. W.l.o.g. we will assume that $r_{max} = 1$ unit. As usual, we describe the network as a graph $G = (V, E)$, where V is the set of network nodes, and E the set of connections between nodes. The number of nodes is denoted by $n := |V|$, and the number of edges by $m := |E|$. Furthermore, we use the notation $\mathcal{S}_r(v)$ to denote the sphere of radius r around a given node v, and the set of neighbors of a node v is abbreviated by $\mathcal{N}(v)$.

3.3 Lower Bound

We start by deriving a lower bound for the performance of geographic routing algorithms in 3 dimensions. The following theorem states that any random-

---

[2] We are aware that a UBG is a very simplistic model for wireless networks, where the transmission range is far from circular. Our main routing techniques presented in this work, however, are valid for real wireless networks.

ized geographic routing algorithm has at least a cubic stretch. (This lower bound would also hold for deterministic algorithms, which we know to not exist at all.)

Theorem 3.1. *Let d be the length of the optimal path between a given source and destination in a 3-dimensional network. There are networks for which the route found by any randomized geographic routing algorithm has expected length $\Omega\big(d^3\big)$.*

Proof. The proof idea is similar to the lower bound for 2-dimensions presented in [60]. We consider the following family of networks: For a positive integer r, construct a 3-dimensional graph as shown in Figure 3.1: First place nodes on the surface of a sphere with radius r such that the mutual distance between any two nodes is at least 2. Obtain a first set of surface-nodes from $S_1 := \{(r \cdot \sin(2i \cdot \arcsin(1/r)), 0, r \cdot \cos(2i \cdot \arcsin(1/r))) \mid i \in [0, \lfloor 0.5\pi/\arcsin(1/r)\rfloor]\}$. In Figure 3.1, these nodes are drawn as solid squares on the left boundary of the sphere. The remaining surface points, also drawn as solid squares, are obtained from this initial set: For each $(x, y, z) \in S_1$, add $\{(x \cdot \sin(2i \cdot \arcsin(1/x)), x \cdot \cos(2i \cdot \arcsin(1/x)), z) \mid i \in [1, \lfloor \pi/\arcsin(1/x)\rfloor]\}$ to an initially empty set S_2. As a second step, add intermediate nodes on the surface (drawn as small diamonds) that connect nearby surface nodes. (The math is nearly the same as for the surface nodes and is omitted.) Furthermore, append to each surface-node a line of $\lfloor (r-1)/2 \rfloor$ nodes. The distance between nodes on the line is 1, and the line is directed towards the center of the sphere. These line-nodes are represented with round (red colored) dots in Figure 3.1. Finally, select an arbitrary surface node w, and append to its line further nodes until the center of the sphere, node t, is reached.

Note that there is no edge between nodes on different lines, as the lines are mounted on surface-points at least 2 units apart, and the length of the lines is less than $r/2$. Furthermore, the number of points per line is $\Theta(r)$. To determine the number of surface nodes, we use (a) $\alpha/2 < \arcsin(\alpha) < 2\alpha$, and (b) $sin(\alpha) > \alpha/2$ for $\alpha \in [0, 1]$. For a surface-node in S_1 with a given x-coordinate, the number of surface-nodes added to S_2 is $\lfloor \pi/\arcsin(1/x)\rfloor > x$ for $x \geq 1.4$, using (a). Thus, we can bound $|S_2|$ by summing up the values of the x-coordinates of the nodes in S_1 using (a) and (b):

$$|S_2| \geq 2 \cdot \sum_{i=1}^{\frac{1}{2}\left\lfloor \frac{\pi}{2\arcsin(1/r)} \right\rfloor} r \cdot \sin(2i \cdot \arcsin\frac{1}{r}) \geq \sum_{i=1}^{r/2} i \in \Theta\big(r^2\big)\,.$$

The total number of nodes in the graph is $(|S_1| + |S_2|) \cdot \Theta(r) = \Theta\big(r^3\big)$.

We now route from an arbitrary node s on the surface to node t in the center of the sphere. An optimal routing algorithm routes along the surface until it hits w and then follows the line until it reaches t. The path on the surface consists of at most 2.5 surroundings of the sphere, requiring $\mathcal{O}(r)$

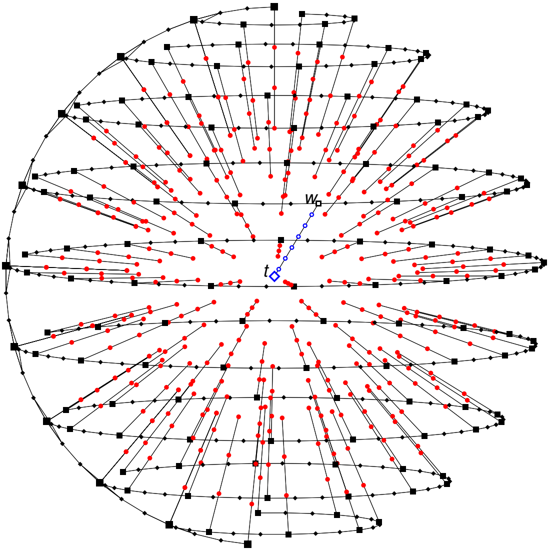


Figure 3.1: Lower bound graph for geographic routing algorithms. Nodes represented by solid squares lie on the surface of a sphere with mutual distance at least 2. Nodes printed as diamonds lie also on the surface and connect these points. The round (red colored) nodes lie on lines leading from the surface-nodes towards the center. A single dedicated surface-node w has an extended line leading to node t in the center of the sphere. This image was created with sinalgo, our simulator for network algorithms described in Chapter 7.

hops. The line contains at most r nodes to traverse, which results in a total cost of $\mathcal{O}(r)$ hops for the optimal algorithm.

A geographic routing algorithm, on the other hand, needs to find the surface-node w on whose line-end the destination node t is located. Since only local routing information is available, this can only be achieved by exploring the lines by descending from the surface-nodes until t is found. For any randomized routing algorithms, the adversary can attach the line leading to t to a random surface-node, requiring the algorithm to explore $\Omega\left(r^2\right)$ lines until it finds t, which requires $\Omega\left(r^3\right)$ hops, which shows that the expected routing stretch is at least cubic.

$\square$

3.4 Towards 3D Routing Algorithms

For our geographic routing algorithm, we use a *greedy-random-greedy* approach, short GRG, where the message is forwarded greedily until a local minimum is encountered. To resolve local minima, a randomized recovery algorithm kicks in. Unlike the deterministic face-routing in 2D, there is *no* deterministic recovery algorithm for 3D networks [30]. Therefore, our recovery technique is bound to be randomized.

Of course, the recovery part also needs to be memoryless and local, which immediately rules out flooding techniques[3] which could quickly find a node closer to the destination than the local minimum. We propose to use random walks (short RW), which constitute a very simple, yet surprisingly efficient recovery technique. We apply the following four techniques to ensure the performance of random walks:

(*1*) *Region Limited Random Walks:* When applying a RW to escape a local minimum at node u, the message is likely to explore large parts of the entire network until hitting a node v which is closer to the target than u, requiring $\mathcal{O}(n \cdot m)$ hops. In most cases, however, such an extensive exploration is not needed: Let v be the node closet to u such that v is closer to the target than u, and let k be the length of the shortest path connecting u and v. Then, exploring $\mathcal{S}_k(u)$ with a RW would have been sufficient in order to find v. As the value of k is not known, the recovery algorithm performs an exponential search by limiting the RW in sequence to $\mathcal{S}_{2^i}(u)$ with $i \in \{2, 3, \dots\}$ until a node closer to the target is found. For each sphere of radius r, the recovery algorithm performs $\mathcal{O}(r^6)$ RW hops, which corresponds to the cover time for the sparse subgraph contained in the sphere, bounding the recovery cost to $\mathcal{O}(k^6)$ hops (see Theorem 3.5).

(*2*) *RW on the surface:* Similar to the face routing in 2D graphs, we can further restrict the RW to nodes delimiting the hole which causes the local minimum and which needs to be surrounded. Chapter 3.6.1 describes how the nodes can *locally* determine the surface in a UBG, using the dual graph of Chapter 3.5.

(*3*) *Sparse Subgraph:* We have seen that for arbitrary graphs, $H_{uv} = 2mR_{uv} < 2mn$, and $C_G \leq 2mR_{span} < 2mn$. Both H_{uv} and C_G grow linearly in m and n, and we can improve H_{uv} and C_G by removing nodes in dense regions, and any edge that is not critical for the connectivity of the graph. We can achieve both points relatively easy by performing the RW on a *connected dominating set*, e.g. see [94], which implicitly also reduces the number of edges. Similarly, *topology control* algorithms build sparse subgraphs considering the network characteristics, and tend to be more stable [83, 97]. A

---

[3] Needless to say that we could implement a memoryless flooding algorithm where each node rebroadcasts a message whenever its TTL permits to do so, allowing multiple transmissions from the same node. However, such a scheme comes with an impractical overhead growing exponentially with the TTL.

truly sparse subgraph could be obtained by limiting the RW to an arbitrary *spanning tree*, reducing C_G and H_{uv} to $2n^2$. But as a spanning tree cannot be constructed locally, this approach is rather impractical for unstable networks. The *dual graph* presented in 3.5 is not only sparse, but also also fulfills the property that any $\mathcal{S}_r(u)$ contains at most $\mathcal{O}(r^3)$ dual nodes, limiting the cover time of a RW restricted to $\mathcal{S}_r(u)$ to $\mathcal{O}(r^6)$.
(4) Power of choice for RW: The cover time of a RW can be improved by not returning to the previous node, if applicable. I.e. when the message is sent from n to m, and if n is not the only neighbor of m, then m forwards the message to a random neighbor, but not n. This improvement derives from the power of choice for RW [6].

3.5 Dual Graph

We now describe the construction of a dual graph $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ of $G = (V, E)$, on which our routing schemes are based. The position of the dual vertices (short DV) is bound to the intersection points of a regular cubic (imaginary) grid, covering the entire space. The DV are only placed nearby network nodes in order to populate the grid in regions where G is present. Each DV is *owned* by exactly one nearby network node, relating $\widetilde{G}$ to G. The relation from G to $\widetilde{G}$ is a bit more involved, as the relation is not bijective. In order to switch from a node $v \in V$ to $\widetilde{G}$, v chooses the DV d closest to v. We will see that d is owned by either v or a neighbor of v.

Edges in $\widetilde{G}$ are only present between direct neighbors in the underlying grid. The dual graph is defined such that the connectivity of G is preserved in $\widetilde{G}$. I.e. a path between $\widetilde{u}, \widetilde{v} \in \widetilde{V}$ in $\widetilde{G}$ implies a path in the original graph between *own*$(\widetilde{u})$ and *own*$(\widetilde{v})$, where *own*(x) denotes the owner of x. Similarly, a path between $u, v \in V$ in G implies a path between the corresponding DV in $\widetilde{G}$. As a result, we can perform a virtual routing on $\widetilde{G}$, and execute the corresponding routing steps on G. Finally, $\widetilde{G}$ is *sparse* in the sense that each $d \in \widetilde{V}$ has constant out-degree, and $\|\widetilde{V}\|$ is linear in the volume of $\bigcup_{v \in V} \mathcal{S}_1(v)$, the volume G occupies. In the remainder of this section, we show the following theorem:

Theorem 3.2. *$\widetilde{G}$ is a sparse, connectivity preserving virtual graph of G, which can be constructed locally.*

The construction of $\widetilde{G}$ consists of the following two steps. First, each node determines the set of DV it owns. Then, the edges $\widetilde{E}$ are added to $\widetilde{G}$ such that two DV are connected iff they are direct neighbors in the virtual grid: $(x_1, x_2) \in \widetilde{E} \Leftrightarrow \|\overline{x_1 x_2}\| = \eta$, where η denotes the cell-side-length of the grid, whose value we determine in Chapter 3.5.2. This step basically requires each node to determine all neighboring DV owned by other nodes.

Algorithm 3.1: Construction of $\widetilde{G}$ (Code for node v)

```
1  Ownership Selection
2    foreach(DV $d \in \mathcal{S}_\rho(v)$)
3      if($\nexists u \in \mathcal{N}(v)$ s.t. $u$.ID $<$ $v$.ID$\wedge\|ud\| \leq \rho$)
4        $v$ selects $d$ as RDV
5    foreach($u \in \mathcal{N}(v)$)
6      $S := \{$DV $d \mid \mathsf{dist}(d, \overline{vu}) \leq h \ \wedge \ \|\overline{vd}\| < \|\overline{ud}\|\}$
7      remove from $S$ all DV already known to be RDV
8      TDV= TDV $\cup$ $S$
9    send TDV to $\mathcal{N}(v)$
10   drop multi-owned DV if $v$.ID $>$ ID of other owner
11 Connect
12   send(RDV $\cup$ TDV) to 3-hop neighborhood of $v$
13   Determine the edges adjacent to any owned DV
```

The construction of $\widetilde{G}$ is completely local, and each of the network nodes only knows a very limited local view of $\widetilde{G}$ at any time.

3.5.1 Ownership Selection

The DV are positioned only on specific positions in space, defined by the intersection points of a regular cubic grid. The set of possible positions for a DV is given by $(i\eta, j\eta, k\eta) \mid i, j, k \in \mathbb{N}$. Our algorithm will ensure (locally!) that *at most one* DV is added to $\widetilde{V}$ for any of these positions.

The ownership selection algorithm executed by each node $v \in V$ determines for each node the set of DV v owns, see Algorithm 3.1. It consists of two substeps. First, v determines its *regular dual vertices* (short RDV), for which v can determine statically whether it owns them (Lines 2–4). Node v chooses as RDV all DV which are most ρ away from v. In addition, for every selected RDV d, v may not have a neighbor u with a lower ID, whose distance to d is bounded by ρ. Formally, the set of RDV of node v is $\{d \mid d$ is a $DV \in \mathcal{S}_\rho(v) \ \wedge \ \nexists u \in \mathcal{N}(v)(u.\text{ID} < v.\text{ID} \wedge d \in \mathcal{S}_\rho(u))\}$. The exclusion of some nodes based on their ID is to ensure that each DV is owned by exactly one network node. In Figure 3.2, the sphere with radius ρ around v denotes the region where v selects its RDV. The value of ρ at least as large as to ensure that $\mathcal{S}_\rho(v)$ contains at least one DV, further information about ρ is given in Chapter 3.5.2. Please note that if a node $v \in V$ does not own any RDV, then the $DV \in \mathcal{S}_\rho(v)$ are owned by (direct) neighbors of v.

The second substep of the ownership selection ensures connectivity in the dual graph by adding additional DV to $\widetilde{G}$. In contrast to the first substep, there may be several nodes $v \in V$ that decide to own the same DV. The resolution of these conflicts is straight forward, but requires communication with the 1-hop neighborhood in G. Therefore, each node calls the DV selected in this substep *tentative dual vertices* (short *TDV*). To ensure connectivity, the *TDV* depend on the neighborhood $\mathcal{N}(v)$ of node v. For each neighbor u, node v determines all *DV* at most $h = \sqrt{3}\eta/2$ away from the line $\overline{vu}$ and closer to v than to u[4]. Then, v selects as *TDV* only the *DV* which it does not already own (as *RDV* or *TDV*), and are not already known to be regularly owned by a direct neighbor (Lines 5–8). Figure 3.2 depicts the region where node v chooses its *TDV* with a cylinder enclosing the line $\overline{vu}$. For each *RDV* node v keeps, it also remembers the *reason*, i.e. the node u due to which the *RDV* was considered in the first place.

For the removal of multiple owners for the same *TDV* introduced in this second step, each node v sends its *TDV* list to its neighbors (Line 9). Assume v selects *TDV* d due to its neighbor u, and also another node n choose d. Again, we resolve the conflict using the IDs, and let the lower ID win. Assume w.l.o.g. that $n.\text{ID} < v.\text{ID}$. Then, v needs to learn that it should drop its ownership of d. The value of η, and therefore the value of h, is chosen such that in such a situation, either (a) $n \in \mathcal{N}(v)$ or (b) $n \in \mathcal{N}(u)$, see Chapter 3.5.2 for more details. In case (a), v receives the *TDV* list from n directly and therefore learns about n's ownership of d. In the second case (b), u receives the *TDV* lists from both, v and n, and u can detect the conflict. u then determines whether u was v's *reason* for the conflicting *TDV* d, and if $n \notin \mathcal{N}(v)$. If (and only if) both conditions hold, u sends a *withdraw(d)* message to v, indicating that v should not own d.

After this second substep is completed, each *DV* (*RDV* and *TDV*) is sure to have exactly one owner $v \in V$. When the *DV* are connected as described in the introduction of this section, we are ready to state a first connectivity property of $\widetilde{G}$: If there is a path between two nodes $v_1, v_2 \in V$, there also exists a path between $d_1, d_2 \in \widetilde{V}$, where d_i is the closest *DV* to v_i. This property follows directly from Lemma 3.3, which is formulated for a single hop in G.

Lemma 3.3. *Given two neighboring nodes* $(v_1, v_2) \in E$ *and two dual vertices* $d_1, d_2 \in \widetilde{V}$ *s.t.* d_i *is the closest* DV *to* v_i*, we can ensure that* d_1 *and* d_2 *are connected in* $\widetilde{G}$ *by adding to* $\widetilde{V}$ *all* DV *at most* $h = \sqrt{3}\eta/2$ *away from the line* $\ell = \overline{v_1v_2}$.

Proof. Because the side length of the grid is η, we know that the d_i themselves are at most h away from v_i. This forms the base-case of our inductive proof. As for the induction step, we consider a point P that moves along ℓ from v_1 to v_2, and a *DV* x which is at most h away from ℓ. Let Q be the position of P when P leaves $\mathcal{S}_h(x)$. We show that when P moves out of $\mathcal{S}_h(x)$, there is a *DV* x' whose distance to Q is strictly smaller than h, and that x' is connected to x in $\widetilde{G}$.

---

[4] In case of equal distance, the node with smaller ID may own the *DV*.

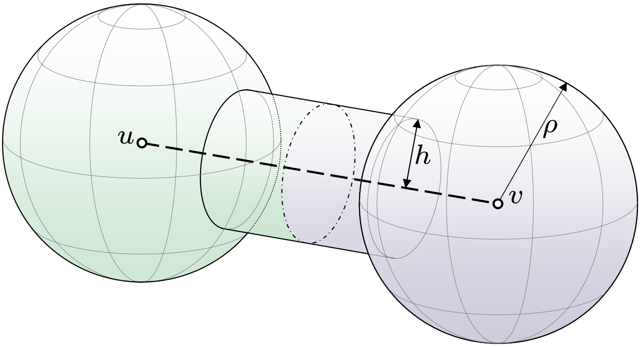


Figure 3.2: Dual vertices that are at most ρ away from a network node $v \in V$ are added to the dual graph $\tilde{G}$. The location of these DV is depicted by the spheres of radius ρ around u and v. To ensure connectivity, the dual graph also contains all DV whose distance from any edge $(u, v) \in E$ is bounded by $h = \sqrt{3}\eta/2$, illustrated with the cylinder of radius h around the edge (u, v).

Let C be the cube of the virtual grid (which has side length η and x as one of its corners) that surrounds Q. If Q is exactly the center of C, all 8 corners of C have distance at most h from ℓ. Therefore, all of them are added to $\widetilde{G}$, and there is a path to each of them starting from x. Thus, when P moves from Q to Q', ϵ closer to v_2, at least one of the 7 remaining corners is in $\mathcal{S}_h(Q')$, which we pick as x'. If Q is not the center of C, there is a corner a of C neighboring x, whose distance to Q is strictly smaller than h. Therefore, a is added to $\widetilde{G}$ and is suitable as x'. □

3.5.2 Tuning the Dual Graph

Before we describe the construction of the edge set $\widetilde{E}$, which connects any two DV that are direct grid neighbors, we make a short detour and give some insight in how to determine the values of η and ρ. Remember that η is the side length of the virtual grid and therefore the minimal distance between any two DV, and ρ is the radius of the sphere $\mathcal{S}_\rho(v)$ in which a node $v \in V$ searches for RDV. We select these two parameters such that for any edge $(d_1, d_2) \in \widetilde{E}$, there is a *short* path of at most 3 hops in G between $v_1 = own(d_1)$ and $v_2 = own(d_2)$, the corresponding owners. This will be a crucial property for routing algorithms simulated on $\widetilde{G}$, as they need to perform the actual routing hops on G.

We need to distinguish the following three cases: (***i***) d_1 and d_2 are both RDV, (***ii***) d_1 xor d_2 is RDV, and the other is TDV, and (***iii***) d_1 and d_2

are both *TDV*. From the ownership selection algorithm, we deduce that the maximal distance of a *RDV* from its owner is ρ.

In the first case (***i***), there should be an edge $(d_1, d_2) \in \widetilde{E}$ only if $(v_1, v_2) \in E$, requiring (***a***) $2\rho + \eta < 1$. Otherwise, it could happen that the path between v_1 and v_2 is of length $\mathcal{O}(|V|)$, which we wanted to avoid. For (***ii***), assume w.l.o.g. that d_1 is a *TDV* selected by v_1 due to its connection with r_1 (the reason for d_1), and that d_2 is a *RDV*. Then, the distance from v_2 to d_1 is at most $\rho+\eta$, and the distance from v_2 to $\overline{v_1r_1}$ is bounded by $\rho+\eta+h$. By requiring (***b***) $\rho+\eta+h \leq \sqrt{3}/2$, we ensure that $v_2 \in \mathcal{S}_1(v_1) \cup \mathcal{S}_1(r_1)$, and that v_2 is a neighbor of either v_1 or r_1. As a result, we can route from v_1 to v_2 either directly, or via r_1. For (***iii***), where v_i choose d_i as *TDV* due to a node r_i, there should be an edge $(d_1, d_2) \in \widetilde{E}$ only if v_1 or r_1 is neighbor of at least one of v_2 or r_2: $(\mathcal{N}(v_1) \cup \mathcal{N}(r_1)) \cap (v_2 \cup r_2) \neq \emptyset$. Otherwise, it could again happen that the path between v_1 and v_2 requires $\mathcal{O}(|V|)$ hops. Assume the situation where $(\mathcal{N}(v_1) \cup \mathcal{N}(r_1)) \cap (v_2 \cup r_2) = \emptyset$. Then, d_2 can be placed closest to $\overline{v_1r_1}$ if $\|\overline{v_1r_1}\| = \|\overline{v_2r_2}\| = 1$ and if v_2 and r_2 are placed on $\mathcal{S}_1(v_1) \cap \mathcal{S}_1(r_1)$. Then, the minimal distance between the two lines $\overline{v_1r_1}$ and $\overline{v_2r_2}$ is $1/\sqrt{2}$. As the d_i may be placed h away from their corresponding line, we need that (***c***) $1/\sqrt{2} - 2h > \eta$ s.t. $(d_1, d_2) \notin \widetilde{E}$. Thus, if $(d_1, d_2) \in \widetilde{E}$, this constraint ensures that there is a route from v_1 to v_2, either directly or via r_1 and r_2, requiring at most 3 hops.

Our goal is to maximize η such that $\|\widetilde{E}\|$ is as small as possible. Considering condition (c), we need to choose h as small as possible. In Chapter 3.5.1, we have seen that $h = \sqrt{3}\eta/2$ satisfied our requirements. In fact, choosing h any smaller could break the connectivity of $\widetilde{G}$. Using (c), we deduce that $\eta < 1/(\sqrt{2}+\sqrt{6})$. As for ρ, we would like to choose its value as large as possible to maximize the number of statically determinable *RDV*. Constraint (a) yields $\boldsymbol{\rho=0.37}$ for $\boldsymbol{\eta=0.258}$, which also satisfies condition (b). We have just shown the following lemma:

Lemma 3.4. *Given two* DV d_1*,* d_2 *from the dual graph, and* $v_i = \text{own}(d_i)$*, the corresponding owners of the* d_i*. Then, if* $(d_1, d_2) \in \widetilde{E}$*, i.e. the distance between* d_1 *and* d_2 *is* η*, there is a path from* v_1 *to* v_2 *in* G *of at most 3 hops.*

Proof. The lemma is satisfied by setting the values of η, ρ and h as described above. □

3.5.3 Connecting the Dual Graph

In the remainder of this section, we describe the construction of the edge set $\widetilde{E}$. The dual edges connect any two *DV* which are direct grid neighbors, which limits the outdegree of any *DV* to 6. For our local construction, this means that the owner $own(d) = v$ of a *DV* d needs to determine which of the

Algorithm 3.2: RW step (On node v, previous node p, $\mathcal{S}_r(s)$)

```
if(target ∈ $(v \cup \mathcal{N}(v))$)                    $v$ or one of its neighbors is the target
    deliver the message and return
if(number of hops for this sphere-size has been performed)
    r = 2r                                 Increase the sphere delimiting the region of the RW
$N$ := set of owners of $\tilde{\mathcal{N}}(v)$ s.t. $\forall n \in N : \|\overline{ns}\| \leq r$
                                                 Only select the owners which are in $\mathcal{S}_r(s)$
if($N = \{p\}$)                                  RW can only return to $p$, no choice
    send message to $p$
else                                                           Apply "power of choice"
    send message to random node in $(N \setminus p)$
```

6 potential neighbors of d exist. For each existing neighbor n, v determines the owner $own(n)$, and the path over which $own(n)$ can be reached in G.

We know from Lemma 3.4 that the owner $own(n)$ is at most 3 hops away from v. Therefore, it is sufficient if each node broadcasts[5] the set of DV it owns to its 3-hop neighborhood (Line 11 of Algorithm 3.1). Along the broadcasting path, every node that forwards the message can add its own ID, such that the receiver can easily determine the path (in G) to reach the sender. Please note that any node owns only $\mathcal{O}(1)$ DV, and therefore the storage overhead to remember the local view of the dual graph does not exceed the desired size.

Finally, we have seen all the pieces to state the proof of Theorem 3.2:

Proof of Theorem 3.2. From the ownership selection algorithm we deduce that the maximal distance between any DV and its owner is below 1. With other words, a network node $v \in V$ only owns DV that lie in $\mathcal{S}_1(v)$. Thus, any $u \in \widetilde{V}$ lies in $Q = \bigcup_{v \in V} \mathcal{S}_1(v)$. As the volume Q is composed of spheres of radius 1, there are no arbitrarily thin areas in Q, and we can conclude that Q contains $\mathcal{O}(Q)$ DV. In combination with the fact that the outdegree of any DV is at most 6, this shows that $\widetilde{G}$ is sparse.

The connectivity of the dual graph $\widetilde{G}$ is an immediate consequence of Lemma 3.4 and Lemma 3.3, and an algorithm to construct $\widetilde{G}$ locally has been presented in this section. □

---

[5] Broadcasting from a node v to its 3-hop neighborhood can be implemented quite efficiently without retransmissions in the following way: The message contains a TTL counter initially set to 3 and a variable s storing the sender v. Upon reception of a message m at node u, u first decrements the TTL. Then, u rebroadcasts the message only if the following three conditions hold. $TTL > 0$, $m.s \neq u$, and $m.s \notin \mathcal{N}(u)$ if $TTL = 1$.

3.6 Routing on the Dual Graph

Our simplest routing scheme, called `pureRW`, performs region limited random walks (see Chapter 3.4) until it finds the target. The spheres delimiting the regions to explore are centered at the sender node, and the random walk applies the "power of choice" technique. Note that `pureRW` does *not* perform any greedy routing steps at all.

The RW selects its next hop based on the sparse dual graph $\widetilde{G}$, the selection algorithm is described in Algorithm 3.2. If the sender node s does not own any DV itself, it first sends the message to its neighbor which owns the DV closest to s. (The existence of such a neighbor is given, see Chapter 3.5.1.) We use the following additional notation: For a node v, let $v.DV$ denote the set of DV owned by v, and $\widetilde{\mathcal{N}}(v)$ the set of DV which are neighboring a DV in $v.DV$, excluding the ones owned by v itself.

The number of random hops for a sphere of radius r is $\mathcal{O}(r^6)$, the cover time of the nodes contained in the sphere. The routing stretch of `pureRW` can be bounded as following:

Theorem 3.5. *Let S_{opt} be the stretch of the optimal geographic routing algorithm for 3D graphs. The expected stretch of* `pureRW` *is $\mathcal{O}({S_{opt}}^2)$.*

Proof. For any source-target pair (s, t), let k be the length of the optimal route between s and t, and j the smallest integer s.t. $2^j \geq k$. `pureRW` performs a RW limited to $\mathcal{S}_{2^i}(s)$ for $i \in \{2, 3, \ldots, j\}$. For any sphere of radius r, it performs $\mathcal{O}(r^6)$ random hops, which results in a total of $\mathcal{O}(k^6)$ random hops. As any optimal algorithm has cubic stretch requiring $\mathcal{O}(k^3)$ hops in the worst case (Theorem 3.1), the stretch of `pureRW` is $\mathcal{O}({S_{opt}}^2)$. □

Clearly, `pureRW` is not a practical routing scheme as its expected delivery time is just as bad as its stretch. Therefore, we try to route much more optimistically using the GRG approach, at the cost of that we are unable to analytically express its performance.

The greedy routing step of the GRG routing scheme selects the DV in $\widetilde{\mathcal{N}}(v)$ closest to the target and forwards the message to its owner (Line 3 of Algorithm 3.3). Initially, if the sender s of the message does not own any DV, it sends the message to its neighbor which owns the DV closest to s. The message is greedily forwarded until the target is found (Line 2), or a local minimum is reached (Line 5).

The recovery algorithm chosen by the GRG is either the region limited RW described in Algorithm 3.2, or a region limited RW that is also bound to the surface of the hole which needs to be surrounded.

Algorithm 3.3: Dual Greedy Step (On node v)

```
1 if(target $\in (v \cup \mathcal{N}(v))$)          $v$ or one of its neighbors is the target
2   deliver the message and return
3 select $d \in (\tilde{\mathcal{N}}(v) \cup v.DV)$ closest to the target
4 if($d$ is owned by $v$)                            $v$ is in a local minimum for the target
5   start recovery
6 else
7   send message to own($d$)
```

3.6.1 Routing on the Surface

We describe the surface of a hole in the network with the aid of the virtual 3D grid introduced in Chapter 3.5. The surface $\mathcal{S}$ is described by a list of grid-cubes which delimit the boundary between the hole and the network. Each node has only its local view of $\mathcal{S}$, which it determines using the presence, respectively absence of DV.

When the greedy routing described in Algorithm 3.3 reaches a local minimum at node v (Line 5), the DV d closest to the target is owned by v itself. Therefore, at least one of the 6 grid-neighbors of d is not present, otherwise, d would not be closest to the target. Let i be the non-present grid-neighbor which is closest to the target, i.e. i is an intersection point of the grid neighboring d, where no DV was placed. Then, the four grid-cubes delimited by the cube-edge $\overline{di}$ are part of the surface $\mathcal{S}$. The remaining grid-cubes describing v's view of $\mathcal{S}$ are obtained iteratively: Until $\mathcal{S}$ does not change anymore, v does the following for every grid-cube $q \in \mathcal{S}$: For every corner c of q for which v owns the corresponding DV, v determines $\{c_1, c_2, c_3\}$, the three corners adjacent to c on q. For each c_i, v adds to $\mathcal{S}$ the 4 grid-cubes delimited by the cube-edge $\overline{cc_i}$ *iff* no DV was placed on c_i. When the iteration stops, $\mathcal{S}$ contains the grid-cubes delimiting the surface around the local minimum, as seen by v.

Let $S\mathcal{N}(v)$ be the set of owners which own a DV lying on the corner of a grid-cube $q \in \mathcal{S}$, and exclude v from $S\mathcal{N}(v)$. This set describes the neighbors of v which also lie on the surface $\mathcal{S}$, and from which the RW picks an arbitrary node to forward the message to. If node v decides to forward the message to $u \in S\mathcal{N}(v)$, it needs to describe the surface to u. It does so by including each $q \in \mathcal{S}$ which has a corner owned by u. Upon receiving the message, u sets its initial view $\mathcal{S}$ of the surface to this subset and determines the remaining grid-cubes describing u's view of the surface by applying the iterative algorithm described above.

Thus, the description of the surface changes constantly, but remains strictly local. In fact, when v sends the surface description to u, it is possible that u determines yet more grid-cubes which touch DV of v as well. This can happen when the surface $\mathcal{S}$ touches v in two or more independent places, such that v cannot determine locally the relationship. In situations where u is the only node that knows that the surface bends back to v, we need to ensure that u sends its view of the surface to v from time to time. But this requires that we drop the 'power of choice" optimization presented in Chapter 3.4. Otherwise, we risk falling into an infinite loop, as the RW may not explore the entire surface.

3.7 Simulation

In order to validate our geographic routing algorithms for 3D networks, we performed a series of simulations in Sinalgo, our Java-based simulation framework for testing and validating network algorithms described in Chapter 7. We chose a fairly large simulation area of 20 × 20 × 10 units and deployed between 2000 and 40000 nodes to cover the range between very sparse and dense networks. In order to obtain more realistic networks, we first placed 100 randomly rotated and randomly positioned cuboids of 2 × 2 × 1 units in the simulation area. The cuboids were areas where no node could be placed, and they enforced holes in the network, such that, especially for dense networks, the messages could not be forwarded greedily without surrounding any holes.

Sparse graphs tend to be heavily twisted, which challenges our GRG routing algorithms with many local minima. To account for this fact, we performed more simulations for sparse networks, which can also be seen by the accumulation of samples for small n in Figure 3.3. For each initial deployment of n nodes, we first connected the nodes to a UBG, and kept only the *giant component*, the largest connected part of the network.

For each network, we selected 5000 random sender/target node pairs (s, t) and sent a message from s to t using the following five recovery algorithms when the message got stuck in a local minimum: RW on the dual, RW on the surface, RW on the Graph, bounded DFS on a spanning tree, and a bounded flooding. All RW were limited to exponentially growing regions (see Chapter 3.4), and all but the RW on the surface implemented the power of choice technique. The bounded DFS on an arbitrary spanning tree is *not* a local algorithm and was chosen for comparison. In that algorithm, we first built a spanning tree, and then perform a DFS on the tree, where the maximal depth to explore the graph increases exponentially. Finally, we implemented a recovery algorithm that uses flooding to escape a local minimum. The flooding relies on a mark-bits to avoid repetitions of the message, and is thus *not* memoryless. The TTL of the flooding message was incremented exponentially to obtain an optimal search time.

Figure 3.3 compares the overhead (measured in routing hops) of the five recovery algorithms. For ease of interpretation, we plotted against the overhead of the flooding algorithm, such that the y-axis shows how much more

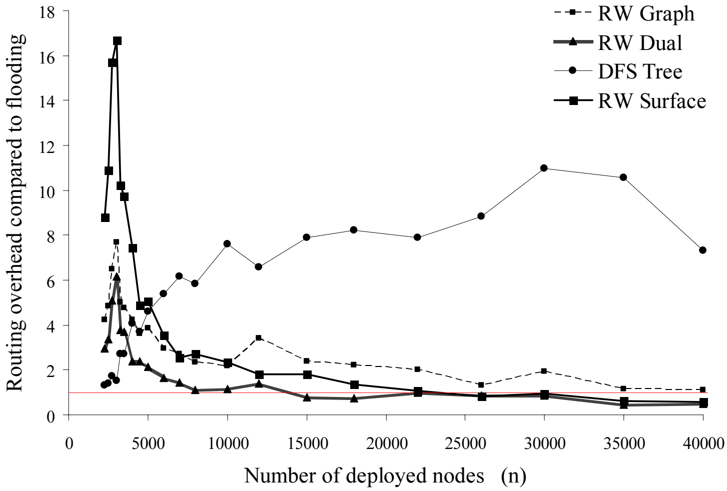


Figure 3.3: The overhead of our routing techniques compared against a *non-memoryless* flooding algorithm. The x-axis indicates how many nodes were deployed, the y-axis how much more routing-messages our routing algorithms induced.

overhead the other routing algorithms induced.

A first important observation is that limiting the RW to the surface of the hole does in fact not help at all. Unless the network is very dense, it tends to have a single huge face covering nearly the entire network. I.e. the holes in the network are nearly never completely closed and most of them are interconnected over the surface. As a result, the restriction to the face does not reduce the number of nodes to visit.

We can further observe that the overhead of the RW on the dual is below the overhead of the RW on the graph, which shows that we achieved our goals of obtaining a sparse network graph via the dual graph. The astonishing good performance of the DFS on the spanning tree for the sparse networks can be traced back to the fact that the nodes of these networks have very low degree, resulting in a tree-like network. For denser networks, however, the RW approaches perform much better. In particular, the RW on the dual and the RW on the surface perform even better than the flooding for very dense networks, as they operate on a sparser network.

4

Greedy Routing

There is a huge body of research on routing schemes, and we have already seen several versions of *geographic routing* in the preceding chapters. Even though greedy forwarding leads to nearly optimal routing paths on average, the routing protocols must ensure to properly circumvent voids in the network, enlarging the routes considerably. We now present a radically different approach to overcome this limitation of geographic routing by embedding the network into an higher-dimensional space such that there is a greedy path between any two nodes. Such coordinate assignments are called *greedy embeddings* as they allow for a purely greedy routing technique. Consider Figure 4.1 where we have two embeddings of the same network in the Euclidean plane. For instance, there is no greedy path from vertex 2 to vertex 6 in the left embedding. As the greedy algorithm forwards the message to

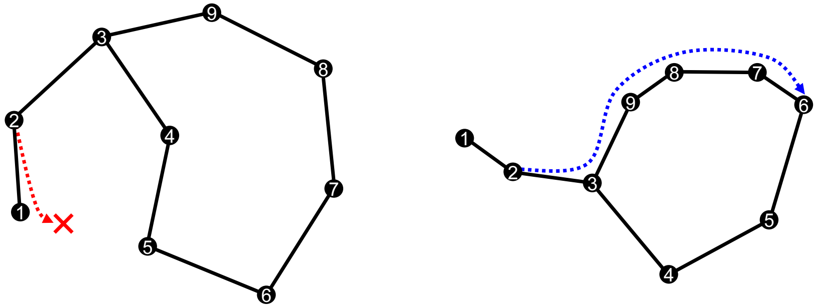


Figure 4.1: Embeddings of networks need not be greedy by default (left network), but a greedy embedding can be constructed (right network).

the neighbor which is closest (in the Euclidean L_2-norm) to the destination, it gets stuck at vertex 1. We solve this problem by constructing a greedy embedding for which the greedy forwarding scheme always succeeds. The right figure shows such a greedy embedding which ensures a greedy path between any two vertices. In our example, the greedy algorithm finds the path $2-3-9-8-7-6$ using 5 hops. The optimal path, however, is $2-3-4-5-6$ using only 4 hops. Similar to the previous chapters, we capture this routing overhead through the *stretch*, which is the ratio of the greedy path-length and the optimal path-length. E.g. in our example, the path from 2 to 6 has stretch of 5/4. The fact that a greedy embedding may use long routes is a problem of critical importance that is solved in this chapter; we show that our greedy embedding introduces only a constant stretch.

In a nutshell, our approach is as follows: Given the connectivity information of a wireless network, modeled as a combinatorial unit disk graph (CUDG) we assign each node a polylog-dimensional virtual coordinate such that greedy routing is always provably successful, between any source-destination pair, e.g. see Figure 4.1. In contrast to previous work our greedy routing provides a bounded stretch (see Figure 4.7 for experimental results supporting this claim). Previous work on greedy routing was done for *geometric* UDGs where an embedding of the network is known (i.e. each vertex knows its coordinates). We consider the more intricate problem of greedy routing on CUDGs, which are UDGs *without any position information.* We believe that our approach is valuable for real world networks where position information is not readily available. Our main result is a polynomial-time algorithm that embeds CUDGs into $\mathcal{O}(\log^2 n)$-dimensional space, permitting greedy routing with constant stretch. To the best of our knowledge, this is the first greedy embedding with stretch guarantees for this class of networks. Our main technical contribution involves extracting, in polynomial time, a constant number of isometric and balanced tree separators from a given CUDG. We do this by extending the Lipton-Tarjan separator theorem for planar graphs to CUDGs. Our result can be adapted to a compact routing scheme with routing tables of size $\mathcal{O}(\log^3 n)$. Furthermore, the proposed approach extends to general graphs as well, with a slightly higher overhead. We support our results by extensive simulations and show that the constant overhead is on average only about 10%; only few routes have a stretch above 1.5.

4.1 Related Work

Papadimitriou and Ratajczak conjectured that any planar 3-connected network has a greedy embedding into 2 dimensions [75]. They also showed that any planar 3-connected network has a greedy embedding in 3 dimensions, for which an embedding algorithm was described in [22]. The Papadimitriou-

Ratajczak conjecture was settled by Moitra and Leighton [72] who present a polynomial-time algorithm for constructing a greedy embedding of a given 3-connected planar graph. In this chapter, we study the embedding of a *combinatorial* UDG where only the connectivity information but no position information of the vertices is given. The absence of such geometrical information eliminates the possibility of planarizing the network and using a triangulation technique. The embedding of a UDG in combination with routing has been studied widely [16, 43, 80, 86], but none of these approaches guarantees a greedy embedding. In fact, finding an exact embedding is NP-hard [14] and cannot be approximated arbitrarily well [58]: If non-neighboring vertices must have a distance larger than 1, it can be shown that there may be neighbors with distance $\sqrt{3/2}$. The best known approximation algorithm for this problem is described in [77] and may induce distances in $\mathcal{O}\big(\log^{2.5} n\big)$. Even if each vertex knows the exact distance to any of its neighbors or the exact angles at which the neighbors are located, the problem of finding an exact embedding remains NP-hard [4, 16].

As a way of avoiding these difficulties and finding embeddings for CUDGs, anchor based routing was introduced, e.g. [37, 41, 96]. The main idea of these routing schemes is that a few vertices are elected as anchor nodes and each vertex stores the length of its shortest path to all anchor nodes. The set of these distances can be seen as a virtual coordinate on which greedy routing can be applied. To ensure a greedy embedding, however, $\Theta(n)$ anchors need to be chosen in the worst case [96], which renders this approach unattractive.

Kleinberg proposed a greedy embedding of arbitrary networks into the hyperbolic plane [52], which was improved by Eppstein et al. such that coordinates of the embedding only need $\mathcal{O}(\log n)$ bits [33]. As of this writing, such greedy embeddings into hyperbolic space are only known for trees, i.e. given a general network, only a spanning tree thereof is embedded. The reduced connectivity information of a spanning tree has severe consequences for the routing performance: Close-by vertices in the network may be far away on the tree on which the routing takes place, introducing a linear worst-case stretch. To the best of our knowledge, we present the first greedy embedding with guaranteed sublinear stretch. For CUDGs, a generalization of planar graphs, our embedding guarantees a constant stretch and for general graphs, the stretch is in $\mathcal{O}(\log n)$.

Compact routing is one of the most prominent routing paradigms. Compact routing studies the tradeoff between the efficiency of a routing algorithm (i.e. the stretch) and its space requirements for the routing tables. For general graphs, there is a stretch-k routing algorithm with an average routing table size of $\mathcal{O}\Big(k^3 n^{1/k} \log n\Big)$ and $\mathcal{O}(\log n)$ bit labels[76]. Renaming (labeling) of the vertices is a widely used technique to reduce the routing table size. In fact, any routing algorithm that does not rename the vertices and requires a stretch below 3 may need routing tables of $\Omega(n)$ bits on general

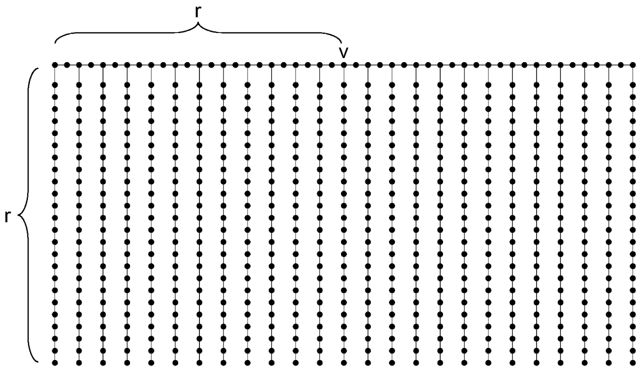


Figure 4.2: Most compact routing results are analyzed for the general class of doubling metrics and provide excellent results if the metric has a small constant doubling dimension. Unfortunately, many reasonable metric spaces induced by even common network models do not have a constant doubling dimension. For example, consider the hop-metric induced by the UDG shown in this figure: A ball of radius $2r$ around v covers all vertices, whereas $\sqrt{n}$ balls of radius r are needed. Note that the if we take the nodes in the network to be embedded in the plane as shown, then the induced Euclidean metric has constant doubling dimension. But, for network routing the hop-metric may be more relevant.

graphs [42]. Many impressive compact routing results have been developed for constant doubling metrics with doubling dimension α. These networks have the property that the set of vertices in any ball of radius $2r$ can be covered by at most 2^α balls of radius r. We refer the interested reader to Chapter 5 for an overview of the results in this area. Note that many graph classes, including UDGs, do not have a constant doubling dimension (see Figure 4.2).

Greedy routing schemes, such as the one presented here, are close relatives to compact routing schemes: as the greedy routing decision is based solely on the coordinates of the neighbors, this neighborhood information could be stored locally, which is equivalent to the routing table in compact routing. Using this transformation, we show that our greedy routing for CUDGs can be transformed to a compact routing scheme with $\mathcal{O}(\log^3 n)$ bit routing tables, beating any known compact routing scheme for this class of networks.

4.2 Background, Results, and Approach

The focus of this chapter is on *combinatorial unit disk graphs.* For points p and q in Euclidean space we use $|pq|$ to denote the Euclidean distance in L_2 norm between p and q. A graph $G = (V, E)$ is a *unit disk graph (UDG)* if there is an embedding $\phi : V \rightarrow \mathbb{R}^2$ of the vertices of G into the Euclidean plane such that $\{u, v\} \in E$ iff $|\phi(u)\phi(v)| \leq 1$. The embedding ϕ is called a *realization* of G. UDGs are widely used as models of wireless networks and this is what motivates our focus on this class of graphs. A UDG may be specified by its realization and in such a setting coordinates of all vertices are known. Alternately, a UDG may be specified as a combinatorial object, e.g., a collection of vertices and a collection of edges. In such a setting the neighbors of each node are known, but no geometric information such as node coordinates, pairwise Euclidean distances, etc. are known. These two specifications are fundamentally different from a computational point of view. Given a realization of a UDG, it is trivial to construct a combinatorial representation of it; on the other hand given a combinatorial representation of a UDG, it is impossible (unless P = NP) [14] to compute its realization. We work in the latter setting, in which we are given a UDG merely as a combinatorial object, with no recourse to any geometric information. To emphasize this we refer to these graphs as *combinatorial* UDGs (CUDG). This makes our approach robust to situations in which geometric information is missing or is only partially available or is erroneous.

Let $f : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ be a function that assigns to each pair of points in $\mathbb{R}^d$ a non-negative real. For any vertex v, let $N(v)$ denote the set of neighbors of v. A *greedy embedding* of an undirected graph $G = (V, E)$ into the space $(\mathbb{R}^d, f)$ is a mapping $\phi : V \rightarrow \mathbb{R}^d$ such that for any pair $s, t \in V$ of distinct vertices, there exists a vertex $u \in N(s)$ such that $f(u, t) < f(s, t)$. Greedy embeddings formally characterize embeddings for which the greedy routing algorithm will guarantee message delivery and were first defined in this manner by Papadimitriou and Ratajczak [75]. These authors were concerned with greedy embeddings into $\mathbb{R}^2$ and the function f was taken to be the Euclidean distance between pairs of points. In our case, d will typically be poly-logarithmic in n, the number of vertices of G, and f will be a "min-max" function that turns out to be a natural "distance" measure for us because the space $\mathbb{R}^d$ into which we embed G is obtained by "gluing" together a bunch of lower dimensional subspaces. In each subspace, an estimation of the distance is obtained through the L_∞ norm, and the overall distance is obtained by taking the minimum of these estimated distances. Note that even though using the `min-max` function is not as natural as using the L_1, L_2 or the L_∞ norm, it is computationally as easy to deal with as any of these norms. Formally, the `min-max` is defined as follows. Let c be a factor of d and let $s = (s_1, s_2, \ldots, s_d)$ and $t = (t_1, t_2, \ldots, t_d)$ be two points in $\mathbb{R}^d$. For each j, $0 \leq j < d/c$ let

$$D_j = \max_{c \cdot j+1 \leq i \leq c \cdot (j+1)} |s_i - t_i|.$$

Then the function $\texttt{min-max}_c$ is defined as

$$\texttt{min-max}_c(s,t) = \min_{0 \leq j < d/c} D_j.$$

The $\texttt{min-max}_c$ function essentially views the space $\mathbb{R}^d$ as a composition of $\frac{d}{c}$ c-dimensional spaces, takes the L_∞ norm of the projections of s and t into those spaces, and finally takes the smallest of those L_∞ distances as the "distance" between s and t. Often c will either be irrelevant or be understood from the context and we will usually write $\texttt{min-max}_c$ as `min-max`.

Let ϕ be a greedy embedding of G into $(\mathbb{R}^d, f)$. An st-path $P = (s = v_1, v_2, \ldots, v_k = t)$ in G is called a *greedy st-path* (or just a *greedy path*, if s and t are clear from the context) if for each i, $1 \leq i < k$, $v_{i+1} = \text{argmin}_{u \in N(v_i)} f(u,t)$. In other words, among all neighbors of v_i, the vertex v_{i+1} is the neighbor that is closest to the destination t. A greedy embedding ϕ of G into $(\mathbb{R}^d, f)$ is said to have *stretch* ρ if for all distinct $s, t \in V$ and for all greedy st-paths P, $|P| \leq \rho \cdot d_G(s,t)$. Here $|P|$ denotes the number of hops in path P and $d_G(s,t)$ denotes the shortest path hop-distance between s and t in G. We will use the latter notation widely throughout the remainder of this chapter with different graphs serving as the subscript of "d" to denote the shortest hop-distance between pairs of vertices in that graph.

4.2.1 Results

Our main result is a polynomial-time algorithm for constructing a low-dimensional greedy embedding with constant stretch of a given CUDG. Our algorithm constructs, for any given n-vertex CUDG, a greedy embedding into $(\mathbb{R}^d, \texttt{min-max})$, where $d = \mathcal{O}(\log^2 n)$. Furthermore, each coordinate of each vertex has size $\mathcal{O}(\log n)$ bits, implying a total of $\mathcal{O}(\log^3 n)$ bits for each vertex-label. Therefore our solution can be easily transformed into a constant-stretch, labeled, *compact routing scheme* for UDG hop-metrics that uses labels of size $\mathcal{O}(\log^3 n)$ per vertex[1].

The main ingredient of our approach is the construction of small-size constant-stretch *tree covers*. Given a graph $G = (V, E)$, a *tree cover* of size k and stretch ρ is a family $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of spanning subtrees of G such that for every $u, v \in V$, there is a tree T_i such that $d_{T_i}(u,v) \leq \rho \cdot d_G(u,v)$. A

---

[1] To obtain the compact routing scheme, each node locally stores the labels of its neighbors in a routing table. By only embedding an MIS of the network and by routing on this virtual topology, only $\mathcal{O}(1)$ neighbors need to be stored.

tree cover $\mathcal{T}$ with stretch 1 is called an *exact* tree cover[2]. The main technical contributions are as follows.

- We show that every CUDG has an $\mathcal{O}(\log n)$-size tree cover with constant stretch and that such a tree cover can be computed in polynomial time even for CUDGs. This is comparable to the result of Gupta et al. [47] who show how to construct a constant-stretch $\mathcal{O}(\log n)$-size tree cover for planar graphs.

- The above tree cover result is obtained via an extension to CUDGs of the Lipton-Tarjan Separator theorem for planar graphs [66]. The algorithm implied by the Lipton-Tarjan Theorem makes explicit use of a planar embedding of the given planar graph. Similarly, recent work by Chen et al. [21] constructs separators for UDGs, but again with explicit use of a given UDG realization. As mentioned in Chapter 4.1, recovering a realization of a CUDG is intractable and our result is the first to show that Lipton-Tarjan type separators can be constructed for UDGs even without any geometric information.

Our approach works for any class of graphs for which small-size, small-stretch tree covers can be constructed. For example, via the result of Awerbuch and Peleg (Lemma 6.8 in [7]) we can compute an $\mathcal{O}(\log n)$-stretch greedy embedding of arbitrary graphs into $(\mathbb{R}^d, \texttt{min-max})$, where $d = \mathcal{O}(\log^3 n)$.

4.2.2 Overall Approach

At a high level, our algorithm, which we call `GREEDY-EMBED`, consist of the three steps of Algorithm 4.1, which are depicted in the Figures 4.3–4.6. The following theorem about algorithm `GREEDY-EMBED` drives the rest of this chapter.

Theorem 4.1. *Algorithm `GREEDY-EMBED` takes as input an n-vertex graph G and returns a greedy embedding of G into $(\mathbb{R}^d, \texttt{min-max}_b)$ with stretch ρ, where $d = k \cdot c \log n$ and each coordinate of each vertex uses $\log n$ bits.*

Proof. Let s and t be an arbitrary pair of vertices in G and let $T_i \in \mathcal{T}$ be a tree containing a shortest st-path, among all trees in $\mathcal{T}$. Then $d_{T_i}(s,t) \leq \rho \cdot d_G(s,t)$. Let u be the neighbor of s along the unique, simple st-path in T_i. Then the L_∞ distance between $\phi_i(u)$ and $\phi_i(t)$ is smallest over all neighbors of s and over all trees. The $\texttt{min-max}_b$ function will therefore lead the message to vertex u. Now the message is at a node that is at most $\rho \cdot d_G(s,t) - 1$ hops

---

[2] In literature on tree covers, see e.g., Awerbuch and Peleg [7], the trees in the tree cover are not required to be spanning. Furthermore, the size of a tree cover is defined as the maximum number of trees that a vertex participates in. For the purposes of greedy routing, it is more convenient to require all trees to be spanning and therefore the size of a tree cover can simply be defined as the number of trees in the collection.

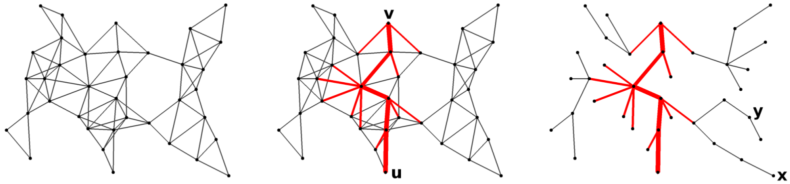


Figure 4.3: We consider the network on the left. In a first step, we find a separator which ensures that any connected component after removing the separator has size at most 2/3 of the total network size (middle figure). Note that this separator is a shortest path connecting u and v along with the 1-hop neighborhood of this path. From this separator tree, we grow a BFS-style tree by repeating the following two steps until we have a spanning tree (right figure). (1) Determine all vertices adjacent to the tree but not yet contained in the tree. (2) Attach each vertex found in (1) to one of its neighbors already on the tree. On this top level, we only ensure good routing paths from one side of the separator to the other. For instance, the vertices x and y in the right bottom corner are 5 hops apart on the tree even though their graph distance is 2. To fix this issue, additional trees are built on the components.

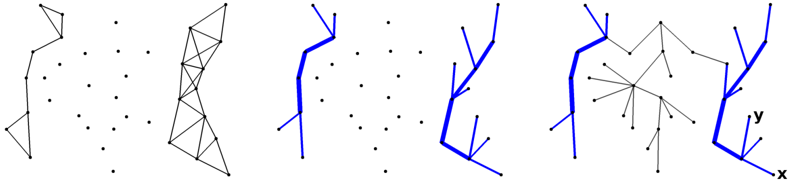


Figure 4.4: After removing the first separator, we obtain several connected components (left figure), for each of which we find a separator (middle figure). In this case, the separator already covers all vertices of the components s.t. the tree of the components is equal to the separator. In the final step, all trees constructed in this step are connected by reinserting the removed separator and connecting the 3 trees. This gives the second tree of the tree cover. Note that for larger networks, the recursion would continue for the components created by the separators found in this step. Also notice that this tree connects the two vertices x and y much better.

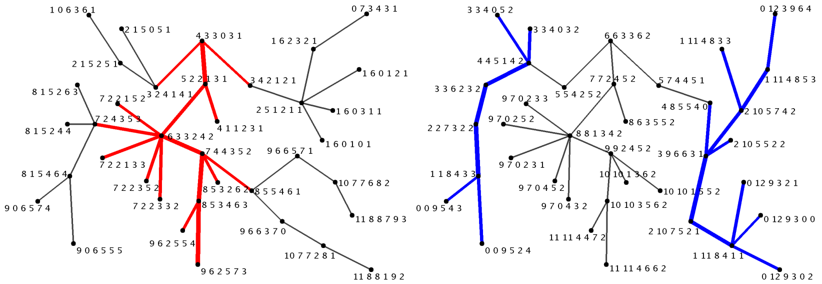


Figure 4.5: Given the tree cover, each tree is embedded in a $\mathcal{O}(\log n)$-dimensional space. Note that each coordinate uses at most $\log n$ bits, resulting in coordinates of size $\mathcal{O}(\log^2 n)$ bits. In our example, 6 dimensions per tree are enough.

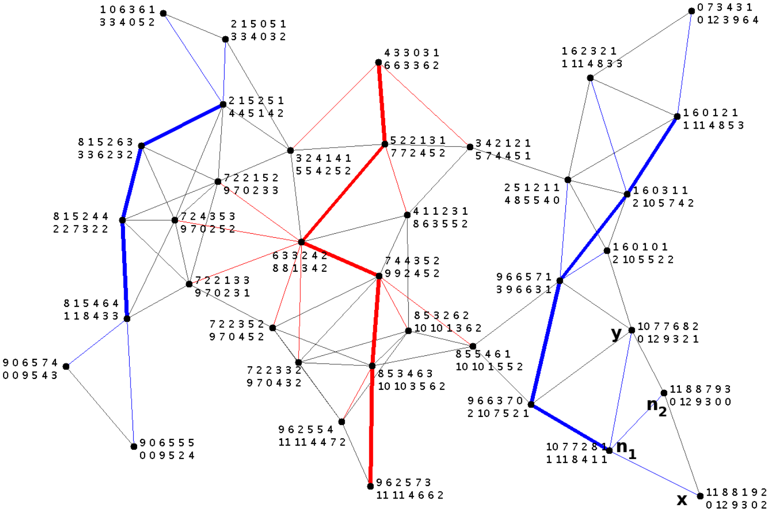


Figure 4.6: Finally, the coordinates assigned for the trees are combined to a single coordinate for each vertex. Using at most $\mathcal{O}(\log n)$ trees, the size of the coordinates is bounded by $\mathcal{O}(\log^3 n)$ bits. To send a message from x to y, x compares the label of y to its neighbors n_1 and n_2 and forwards to the closer neighbor. The comparison is done tree by tree, e.g. for n_1 we have $L_\infty(10\,7\,7\,2\,8\,1, 10\,7\,7\,6\,8\,2) = 4$ for the first tree and $L_\infty(1\,11\,8\,4\,1\,1, 0\,12\,9\,3\,2\,1) = 1$ for the second tree, giving an estimated distance of 1 from n_1 to y. Similarly, the estimated distance from n_2 to y is 1 as well. Therefore, x forwards the message to either of its neighbors.

```
Algorithm 4.1: GREEDY-EMBED($G$)
Step 1
    Construct a tree cover $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of $G$ with stretch $\rho$ (Figures 4.3
    and 4.4).
Step 2
    Embed each tree $T_i$ isometrically into $\mathbb{R}^b$, where $b = c \cdot \log n$ for some constant
    $c$. For this we use a simple and well-known algorithm due to Linial et al. [65],
    Theorem 5.3. We note that using this algorithm guarantees that every
    coordinate of every vertex uses at most $\log n$ bits. Let $\phi_i(v)$ denote the
    coordinates of vertex $v$ obtained by constructing the embedding of $T_i$
    (Figure 4.5).
Step 3
    Output $\phi := \phi_1 \cdot \phi_2 \cdots \phi_k$, the "concatenation" of the mappings $\phi_i$. In other
    words, $\phi(v)$ consists of $k \cdot c \log n$ coordinates, obtained by writing the
    coordinates in $\phi_1(v)$, followed by the coordinates in $\phi_2(v)$, followed by the
    coordinates in $\phi_3(v)$, and so on (Figure 4.6).
```

from the destination t. Continuing this argument we see that the message will be greedily routed to t in at most $\rho \cdot d_G(s,t)$ hops. $\square$

For arbitrary graphs we utilize the above theorem as follows. Awerbuch and Peleg [7] have shown that every graph has a tree cover of stretch $\mathcal{O}(\log n)$ and size $\mathcal{O}(\log^2 n)$ and such a tree cover can be computed in polynomial time. This yields $\rho = \mathcal{O}(\log n)$ and $k = \mathcal{O}(\log^2 n)$ and leads to the following corollary.

Corollary 4.2. *There is a polynomial time algorithm that can compute for any n-vertex graph an* $\mathcal{O}(\log n)$*-stretch greedy embedding into* $(\mathbb{R}^d, \texttt{min-max})$ *where* $d = \mathcal{O}(\log^3 n)$. *Each coordinate of each vertex uses* $\mathcal{O}(\log n)$ *bits.*

In the next section we show our main technical result: for any CUDG we can compute in polynomial time a constant-stretch $\mathcal{O}(\log n)$-size tree cover.

4.3 Greedy Embeddings of CUDGs

We now show how to construct a constant-stretch tree cover $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ with $k = \mathcal{O}(\log n)$ of any given CUDG $G = (V, E)$. To obtain a tree cover of G we use a recursive algorithm inspired by the approach of Gupta et al. [47], based on *isometric separators*; Gupta's algorithm yields a constant-stretch $\mathcal{O}(\log n)$-size tree cover for any planar graph. For any graph $G = (V, E)$ a vertex-subset $V' \subseteq V$ is called a *1/3-2/3 separator* of G if a largest connected component in $G \setminus V'$ has size at most $\frac{2}{3} \cdot |V|$. Such "balanced" separators have played a fundamental role in algorithm design for a variety of problems [82]. Given a graph $G = (V, E)$, a *k-part, isometric*

Algorithm 4.2: `CUDG-SEPARATOR`(G)

```
Step 1
    Pick an arbitrary vertex $r$ of $G$ and construct a breadth-first search tree $T$ rooted at $r$.
Step 2
    For any $u \in V$, let $P(u)$ denote the (unique) path in $T$ from $r$ to $u$ and let $\mathtt{N}(P(u))$ denote the set of vertices in the closed neighborhood of $P(u)$. In other words, $\mathtt{N}(P(u))$ contains all vertices in $P(u)$ along with vertices that have a neighbor on $P(u)$. For every pair of vertices $u, v \in V$ construct $G_{uv}$ by deleting from $G$ the vertices in $\mathtt{N}(P(u)) \cup \mathtt{N}(P(v))$. Terminate this step successfully on finding a $G_{uv}$ such that every connected component in $G_{uv}$ has size at most $2|V|/3$.
Step 3
    From $P(u)$ construct a tree $S_1$ by taking each vertex $v$ in $\mathtt{N}(P(u)) \setminus P(u)$ and connecting $v$ to an arbitrary neighbor in $P(u)$. Similarly construct $S_2$ from $P(v)$ and return $S_1$ and $S_2$.
```

separator of G is a family $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of subtrees (not necessarily spanning) of G such that

1. $S = \cup_i V(S_i)$ is a 1/3-2/3 separator for G. Here $V(S_i)$ is the vertex set of the tree S_i.

2. For each i and each pair of vertices $u, v \in V(S_i)$, $d_{S_i}(u, v) = d_G(u, v)$. In other words, each of the subtrees S_i contain the shortest paths between their constituent vertices and hence are isometric to the restriction of G to $V(S_i)$.

This definition is due to Gupta et al. [47]. For a CUDG G we obtain a 2-part tree separator that is not quite isometric, but preserves distances approximately. Specifically, we show that for some constant $\rho \geq 1$, $d_{S_i}(u, v) \leq \rho \cdot d_G(u, v)$ for all $u, v \in V(S_i)$. We will call such a family of separators a 2-*part stretch-ρ separator.*

Theorem 4.3. *Every CUDG has a 2-part, stretch-3 separator and such a separator can be computed in polynomial time.*

The proof of the above theorem is due to Sriram Pemmarju and can be found in the conference publication [40]. It implies Algorithm 4.2, which we call `CUDG-SEPARATOR`, for computing a 2-part, stretch-3 separator of G. In a sense the proof of the above theorem should be thought of as a "proof of correctness" for Algorithm 4.2 and it shows that Step 2 of `CUDG-SEPARATOR` will always terminate successfully.

The reason Algorithm 4.2 returns S_1 and S_2 rather than just $P(u)$ and $P(v)$, is worth mentioning here. For planar graphs, the paths $P(u)$ and $P(v)$ separate T into several disconnected components. Since a UDG is not planar, it is possible to "hop" between the components by following edges that cross $P(u)$ or $P(v)$. However, UDGs have the nice property that if a pair of edges cross then at least one pair of end points of the crossing edges are neighbors. Therefore, if a path P is crossed by an edge $\{u, v\}$ then either u or v is a neighbor of a vertex in P. Therefore, by removing S_i, the path P_i along with its neighborhood, we ensure that the components are really disconnected and S_i separates the given graph.

The tree cover of G can now be constructed by recursively applying algorithm `CUDG-SEPARATOR` for finding a 2-part, stretch-3 separator of G. Consider the trees S_1 and S_2 returned by the above algorithm. For each $i = 1, 2$, grow a BFS-style tree T_i from S_i in the following way: Initially, set $T_i = S_i$. While T_i is not spanning, repeat the following two actions: (1) Determine the set of nodes N that are adjacent to T_i, but not yet on T_i. (2) For each $n \in N$, connect n to an arbitrary neighbor on $T_i \setminus N$.

T_1 and T_2 are the first two trees in the tree cover being constructed. Now recurse on the connected components in G_{uv}. More specifically, suppose that the connected components in G_{uv} are $C_1, C_2, \ldots, C_t$ and suppose that for each C_i we find $\{S_1^i, S_2^i\}$, a 2-part, stretch-3 separator and obtain the corresponding trees $\{T_1^i, T_2^i\}$. The collection $\{T_1^1, T_1^2, \ldots, T_1^t\}$ is a forest and we arbitrarily add vertices and edges to this forest to obtain a spanning tree of G. We similarly extend the forest $\{T_2^1, T_2^2, \ldots, T_2^t\}$ to a spanning tree of G, thus obtaining two more spanning trees for each level of the recursion.

Theorem 4.4. *Given an n-vertex CUDG G, there is a polynomial-time algorithm that computes a constant-stretch tree cover $\{T_1, T_2, \ldots, T_k\}$ of G with $k = \mathcal{O}(\log n)$. In addition, the stretch of the tree cover is upper-bounded by 4.*

Proof. The size of the tree cover is in $\mathcal{O}(\log n)$ as the separator ensures a $1/3-2/3$ cut such that each recursion step operates on a connected component whose size is reduced at least by a factor $2/3$ compared to the previous step. A naive algorithm finds such a separator by testing each node pair u, v in Step 2 of Algorithm 4.2, requiring at most $\mathcal{O}(n^2)$ tests for each level of the recursion. The test itself is trivial and can be done in $\mathcal{O}(n)$ operations. The overhead for the construction of corresponding BFS trees and the assembly of the forests to a spanning tree is in $\mathcal{O}(n)$. Thus, the overall complexity for building the tree cover is polynomial in n.

As for the stretch, consider an arbitrary non-neighboring pair s, t of vertices in G. There are two cases depending on whether or not s and t are separated by a separator S_i constructed by Algorithm `CUDG-SEPARATOR`. In the following, we use the fact that S_i is a shortest path P_i along with nodes in the neighborhood of P_i.

1. Vertices s and t are not separated in any level of the recursion. In this case s and t together lie on a separator S and we can show that $d_S(s,t) \leq 3 \cdot d_G(s,t)$. We need to analyze three cases: (a) If both, s and t lie on P, the connection along S is optimal (as P is a shortest path). (b) Assume w.l.o.g. that only s lies on P and t is attached to an arbitrary neighbor y on P. Then, $d_P(s,y) \leq d_G(s,t)+1$, otherwise P would not be a shortest path between s and y. (c) Similarly, if s and t are attached to x and y on P, respectively, $d_P(x,y) \leq d_G(s,t)+2$. As a result, $d_S(s,t) = d_P(x,y)+2 \leq d_G(s,t)+4 \leq 3 \cdot d_G(s,t)$ for non-neighboring s and t.

2. Suppose that s and t are separated by a separator $S_1 \cup S_2$ in some level of the recursion. Then, the shortest path between s and t intersects either S_1 or S_2 in a point u. W.l.o.g. assume that it intersects S_1. We now consider the tree T of the tree cover which was built from S_1 and show that $d_T(s,t) \leq 3 \cdot d_G(s,t)+2$ using an argument similar to Theorem 5.1 of Gupta et al. in [47]. Let s' (t') be the vertex on P_1 closest to s (t) on the tree T. For our argument, we distinguish two cases: (a) if the optimal path from s to t passes through a point u on P_1, we have that $d_T(s,s') \leq d_G(s,u)$ and $d_T(t,t') \leq d_G(t,u)$. In addition, $d_T(s',t) \leq d_T(s,s') + d_G(s,t) + d_G(t,t')$. Combining the inequalities, we get that $d_T(s,t) = d_T(s,s') + d_T(s',t') + d_T(t',t) \leq 3 \cdot d_G(s,t)$. ($b$) If the optimal path between s and t does not pass through any vertex of P_1 (this is possible as we do not have a planar network), then u is a node neighboring a node in P_1. In that case, we can bound the path to reach a first node in S_1 by $d_T(s,s')-1 \leq d_G(s,u)$ and $d_T(t,t')-1 \leq d_G(t,u)$. $d_T(s',t')$ can be bounded as in the first case, and we obtain that $d_T(s,t) \leq 3 \cdot d_G(s,t)+2 \leq 4 \cdot d_G(s,t)$ for non-neighboring s and t.

In both cases, there is a tree T along which the distance $d_T(s,t)$ is at most 4 times longer than the optimal distance $d_G(s,t)$ on the graph G.

□

Theorem 4.5. *There is a polynomial time algorithm that can compute for any n-vertex CUDG an* $\mathcal{O}(1)$*-stretch greedy embedding into* $(\mathbb{R}^d, \texttt{min-max})$ *where* $d = \mathcal{O}(\log^2 n)$. *Overall, each coordinate uses* $\mathcal{O}(\log^3 n)$ *bits.*

Proof. The claim follows directly from Theorem 4.1 and Theorem 4.4. □

4.4 Simulation

Extensive simulations on large, randomly generated networks show that on average our embedding algorithms provide extremely low stretch routes, the average stretch being much smaller than the worst case guarantees proved

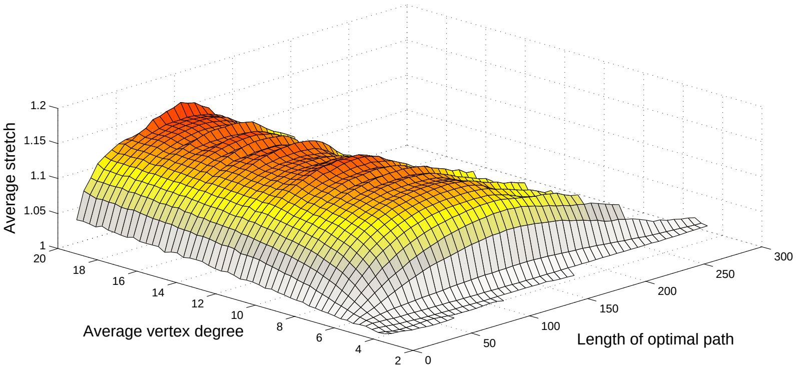


Figure 4.7: Average stretch as a function of the average vertex degree and the length of the optimal path. Note that our greedy embedding guarantees an upper bound on the stretch for any UDG. In practice, the stretch is always below this bound and depends not only on the network topology, but also on the length of the optimal route.

in the previous sections. We considered a range from very sparse to dense networks for each of which we sampled random source and destination vertex pairs. The average routing overhead is around 10% and the worst stretch we ever encountered is 3. Of course, such a simulation is much weaker than our formal proof (Theorem 4.5) on the maximum stretch. However, it provides a good approximation for the average case and also helps to validate the overall approach.

We used the sinalgo simulator described in Chapter 7 to construct the desired network topologies which all live in an area of size 100 × 100 units. For each network, we deployed n vertices at random positions, constructed the UDG with a transmission radius of 1 unit, and kept the giant connected component for the embedding. Varying n from 10,000 to 60,000 yields networks in the desired density and size range. In the following, we characterize the resulting networks by their average degree of the vertices, which grows linearly with n. With over 2,000 networks and 50,000 random source and destination pairs, the simulation analyzed over 10^8 routing paths with an average length of 74 hops.

Figure 4.7 shows the average stretch as a function of the network density (the average degree of the vertices) and the length of the optimal path for which the samples were taken. It is interesting to note that the average stretch partially depends on the length of the chosen route: close vertex pairs tend to have at least one tree connecting them nearly optimally, yielding a

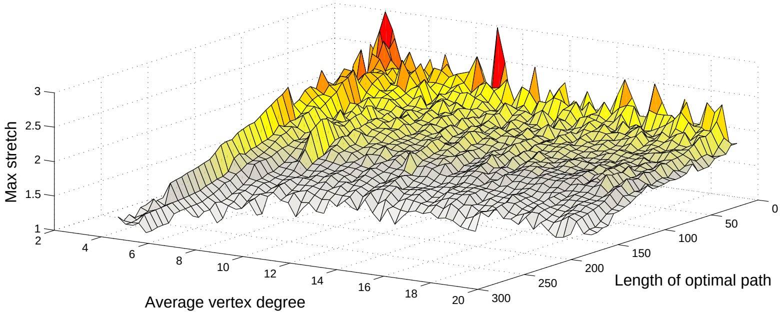


Figure 4.8: Maximum stretch found over all sampled routing paths.

low stretch for short routes. Longer routes, on the other hand, have a higher chance that the greedy algorithm first needs to travel along one or several trees which do not connect optimally (or close to optimal) to the destination, resulting in an increased stretch. While Figure 4.7 seems to indicate that the average stretch falls as the length of optimal paths increases, this decrease may be due to a sampling bias: in our experiments distant s-t pairs show up relatively infrequently and as a result our experiments may be encountering costly situations rarely, as well.

The density of the network is another key parameter for the average stretch of our greedy routing algorithm. For sparse networks, the tree cover tends to include many optimal paths resulting in a nearly optimal stretch. With increasing density, the trees miss more and more shortcuts, the average stretch grows. This growth, however, is stopped when an increased connectivity only adds additional edges but no additional shortcuts. Figure 4.7 shows that this critical density is reached with an average vertex degree of approximately 10.

Figure 4.8 shows the maximum stretch we have encountered for any of the 10^8 sampled routing paths. The highest peaks stem from the shortest routes, for which even a short detour may result in a high stretch factor. However, we know from Figure 4.7 that on average, only very few routes suffer such a high stretch.

A comparison of our simulation results with related work is rather difficult. For instance in [52], Kleinberg illustrated his work with a set of very small networks of 50 vertices, not covering the challenging networks where the routing stretch may be linear in the network size. Kuhn et al. [59] compared several greedy routing algorithms for UDGs *with* position information.

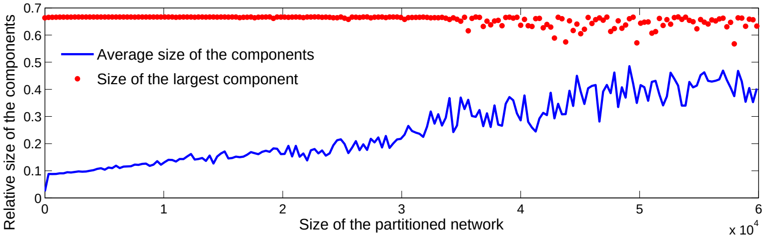


Figure 4.9: When removing a separator, one or several connected components remain in the network. This plot shows the relative size of these components, compared to the network which was separated.

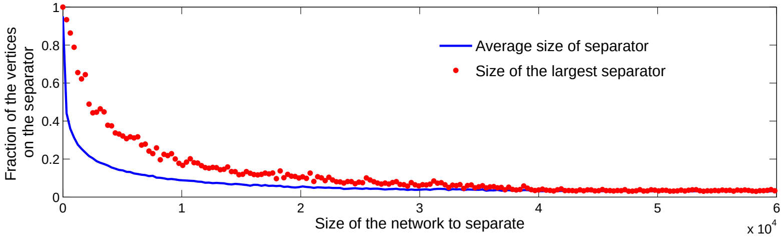


Figure 4.10: To construct our greedy embedding, we recursively partition the network into several smaller networks by removing a separator. In this plot, we show the number of vertices on the separator in function of the network for which the separator was built.

Their simulation for varying network sizes indicates that the average stretch for sparse networks is well above 3 for small networks and increases linearly with the size of the network.

For completeness, we also include some statistics on the size of the separators and the size of the connected components that we obtained during the simulation. Figure 4.9 shows that indeed, after removing a separator, none of the connected components had a size larger than 2/3 of the original network. It is interesting to note that the average size of the components increases with the density, which shows that in sparse networks, we tend to obtain several components and in the dense networks only 2 or 3. In Figure 4.10, we plotted the size of the separators against the size of the network for which the separator was built. We can see that for sparse networks, the separator tends to remove between 10% and 30% of the vertices. For all other networks, the size size of the separator is approximately constant around 5%.

5

Compact Routing with Any- and Multicast

We finish this first thesis part on algorithm design with a routing scheme that is slightly different from the previous routing techniques in that the nodes are allowed to store small routing tables. In return, the presented routing scheme not only supports unicast transmissions, but also offers efficient anycasting and multicasting. Thereby, we not only consider how fast a message is delivered, but also how much overhead the algorithm generates. For instance, optimal routing is straightforward if each node stores in its routing table the optimal path to every other node in the network. But this comes at a cost of $\mathcal{O}(n)$ routing entries, which easily exceeds the memory capacity of hardware limited network devices.

The multicasting feature may be extremely helpful to execute a query only on a subset of nodes. E.g. consider a fire detection system where a coordinator first determines the set of nodes measuring a critical state. In subsequent rounds, it may only need to survey these nodes. Talking

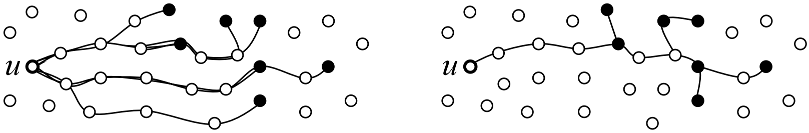


Figure 5.1: Support for efficient multicasting is crucial. If node u sends a separate message to all destination nodes marked with solid circles (left image), it causes a much higher message overhead than if it utilizes an efficient multicast algorithm (right image).

to each of them individually introduces a much higher message overhead than efficiently multicasting a single message, see Figure 5.1. The gain of multicasting becomes even more substantial when the receiver set tends to be clustered. In contrast to multicasting, anycasting becomes interesting when a node needs to send some data to any node of a given set. In the example of the fire detection network, this may arise when a node measuring a critical state wants to inform one of several coordinators. The anycasting algorithm picks the coordinator as to minimize transmission cost.

In addition to the routing tables, each node is assigned a special *label* that encodes routing information. This is similar to the greedy embedding of Chapter 4 where we assigned a special set of coordinates to each node. But in contrast to the greedy embedding, we now talk about labels instead of coordinates as the labels themselves cannot be used to deliver messages. Furthermore, remember that the size of the coordinates in the greedy embedding was in $\mathcal{O}\big(\log^3 n\big)$ bits for CUDGs and even in $\mathcal{O}\big(\log^4 n\big)$ bits for general graphs. The labels used in this final routing scheme only require $\mathcal{O}(c \cdot \log n)$ bits for a small constant c that depends on the network topology. Thus, by adding a routing table on each node, we can drastically reduce the size of the labels. But just as important as the size of the labels is the size of the routing tables and the overhead of the message delivery. We will show that our routing tables are relatively small and prove our scheme to provide close to optimal unicasting, and constant approximations to anycast and multicast. In addition, our assumption about the capabilities of the nodes is minimal: we only require that each node has a unique ID and that it can communicate with its direct neighbors.

Unfortunately, the often studied UDG or UBG connectivity models are not really appropriate for wireless networks because they do not consider perturbations of the wireless medium, e.g. caused by obstacles, and assume the transmission radius to be perfectly circular. However, to ensure applicability of the proposed algorithms, the analysis should be on a connectivity model that characterizes wireless networks as accurately as possible. Clearly, any network connectivity can be described using a general graph. But taking into account the limited range of wireless devices, we observe that nodes are mostly connected with other nodes in their proximity, which results in a far more regular connectivity graph than a general graph. Although close-by nodes may be out of communication range due to obstacles, a node is typically highly connected to nodes in its surroundings. In other words, even in environments with many obstacles, the total number of mutually independent[1] neighbors of a node is likely to be small [84].

We exploit the slightly more general formulation of this property that takes into account the multi-hop nature of the network: Given a h-hop neighborhood of a node v, it is possible to cover all these nodes with only a small

---

[1] Two nodes are independent if they are not (direct) neighbors.

number of $\frac{h}{2}$-hop neighborhoods. Similar to before, this property states that in any h-hop neighborhood, one can pick only a relatively small subset of nodes with pairwise distance larger than $\frac{h}{2}$. Figure 5.2 shows the deployment of 39 sensor nodes in our office building and how the nodes are connected [31]. In this network, the 2-hop neighborhood of *any* node can be covered by at most five 1-hop neighborhoods. The figure shows the 2-hop neighborhood of node v, which is covered by four 1-hop neighborhoods.

These observations motivate to characterize the network connectivity by $\mathcal{A}$, the maximum number of half-sized neighborhoods that are necessary to cover any given neighborhood. This number is expected to be quite small for most wireless networks, e.g. a small two-digit number. Being the only parameter to describe the network, we can adjust $\mathcal{A}$ to apply our analysis to virtually any wireless network topology, including obstacle-rich deployments in 3D and worst case scenarios. Furthermore, because the presented routing algorithms are self-adaptive and need *not* to know the value of $\mathcal{A}$, they are good for any network. It is only the performance of the routing algorithms that depends on the value of $\mathcal{A}$.

In the sequel we describe the network connectivity by a weighted graph $G = (V, E)$ where each network node is mapped to a node of the graph, and a weighted edge is present between any two nodes within mutual transmission range. We observe that the coverability property described above matches the definition of a constant doubling metric[2] on G (see Definition 5.2). The distance metric $\mathcal{M}$ associated with G corresponds to the metrization of G using the cost-function $\mathrm{d}_{\mathcal{M}}(a, b)$ which assigns to each pair $(a, b) \in V \times V$ the cost of the least cost path between a and b. Our analysis holds for a more general class of networks, where each link may be assigned a cost, e.g. the number of retransmissions needed to send a message over the link. Setting all link costs to 1, we obtain the hop-metric discussed above.

Definition 5.1 (Ball). *Given a node $v \in V$, the ball $\mathcal{B}_v(r)$ with radius r denotes the set of nodes with distance at most r from v: $\mathcal{B}_v(r) = \{u | \mathrm{d}_{\mathcal{M}}(v, u) \leq r\}$.*

Definition 5.2 (Constant Doubling Metric). *A graph $G = (V, E)$ fulfills the doubling metric property if any ball $\mathcal{B}_v(r)$ can be covered [3] by a constant number of balls $\mathcal{B}(\frac{r}{2})$ with half the radius: For $r \geq 0$ and $\forall v \in V$: $\exists U \subseteq \mathcal{B}_v(r)$ such that $|U| = \mathcal{O}(1)$ and $\mathcal{B}_v(r) \subseteq \bigcup_{u \in U} \mathcal{B}_u(\frac{r}{2})$. If $|U|$ is bounded by 2^α for a constant α, we say that the metric associated with G has* doubling dimension *α.*

We point out that the value of $\alpha = \lceil \log_2 \mathcal{A} \rceil$ is quite small for most wireless networks, e.g. around 3 or 4.

---

[2] A metric assigns to each node-pair $(a, b) \in V \times V$ a cost satisfying non-negativity, identity of indiscernibles, symmetry and triangle inequality.

[3] A ball B is *covered* by a set of balls $\{b_1, \ldots, b_n\}$ if $\forall u \in B : \exists i$ such that $u \in b_i$.

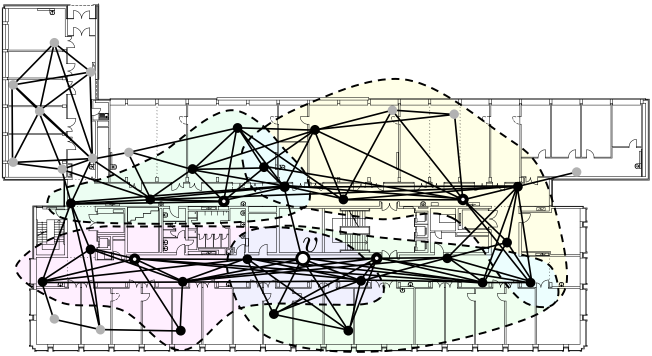


Figure 5.2: The figure shows a deployment of 39 nodes in an office building and how they are connected. The nodes in black denote a 2-hop neighborhood of node v. These nodes can be covered by four 1-hop neighborhoods indicated with the dashed lines and rooted at the nodes with the small white dot.

5.1 Related Work

One milestone in the area of compact routing schemes was laid by Peleg and Upfal in [76], where they examine the trade-off between the efficiency of a routing algorithm and its space requirements. They present a stretch-k routing algorithm for general graphs with an average routing table size of $\mathcal{O}\left(k^3 n^{1/k} \log n\right)$ bits and $\mathcal{O}(\log n)$ bit labels. The renaming (labeling) of the network nodes is a widely used technique to reduce the routing table size. In fact, any routing algorithm that does not rename the nodes and requires a stretch below 3 may need routing tables of $\Omega(n)$ bits [42]. For constant doubling metrics, we know that the stretch is above $9 - \epsilon$ if the routing tables size is $\mathrm{o}\left(n^{(\epsilon/60)^2}\right)$ [53].

In his recent work [91], Talwar described a $(1+\epsilon)$ stretch routing scheme for α-doubling metrics with routing tables of $\mathcal{O}(1/(\epsilon\alpha))^{\alpha} \log^{2+\alpha} \Theta$ bits and label-size $\mathcal{O}(\alpha \log \Theta)$ bits where Θ is the diameter of the network. This work was improved by Chan et al. in [19] by reducing the storage overhead to $(\alpha/\epsilon)^{\mathcal{O}(\alpha)}(\log \Theta)(\log \Delta)$ bits per node and a label size of $\mathcal{O}\left(\alpha \log\left(\epsilon^{-1}\right)\right) \log \Theta$ bits, with Δ denoting the maximal degree of any node.

Slivkins presented two improved compact routing schemes in [88]. The first uses $(\epsilon)^{-\mathcal{O}(\alpha)}(\log \Theta)(\log \Delta)$ bits per routing table and $\mathcal{O}\left(\alpha \log\left(\epsilon^{-1}\right)\right) \log \Theta$ bits per label, whereas the second scheme uses $\epsilon^{-\mathcal{O}(\alpha)}(\log \Theta)(\log \log \Theta)(\log n)$ routing table bits and $2^{\mathcal{O}(\alpha)}(\log n) \log\left(\epsilon^{-1} \log \Theta\right)$ bits for each label.

In their seminal work [3], Abraham et al. presented the first compact routing scheme with $\lceil \log n \rceil$ bit labels and routing tables of $\epsilon^{-\mathcal{O}(\alpha)} \log n \log(\min(\Theta, n))$ bits. This work emphasized on scale-freedom, i.e. independence of Θ. Dropping this constraint, their technique easily yields $\mathcal{O}(\Theta)$ bit labels. In addition, the authors provide a scale-free name-independent routing scheme, including the matching lower bounds.

Our work does not quite achieve the bounds of [3], our routing tables are of size $\mathcal{O}\left(\frac{1}{\epsilon}\right)^{\alpha} (\log \Theta)(\mathcal{O}(\alpha) + \log \Delta)$ and the routing labels require $2\alpha\lceil \log \Theta \rceil + \lceil \log n \rceil$ bits[4]. This small overhead allows us to build an *all-in-one* routing scheme that not only supports unicasting (Chapter 5.4), but also provides constant approximations to anycasting (Chapter 5.6) and multicasting (Chapter 5.5). Furthermore, we present a distributed algorithm to construct the labeling and routing tables in Chapter 5.7.

5.2 Definitions and Preliminaries

We first define some further terms and state properties of doubling metrics that will turn out to be handy. We start with the definition of a $\rho-$net, which is closely related to maximal independent sets and dominating sets.

Definition 5.3 ($\rho-$net)**.** *A subset U of the node-set V of a graph $G = (V, E)$ is a $\rho-$*net *if each node in V has distance at most ρ to at least one node in U, and the mutual distance between any two nodes in U is strictly larger than ρ. Formally, a set $U \subseteq V$ is a $\rho-$*net *of $G = (V, E)$ if*

$\forall v \in V : \exists u \in U : \mathrm{d}_{\mathcal{M}}(v, u) \leq \rho$ *and*

$\forall u_1, u_2 \in U : \mathrm{d}_{\mathcal{M}}(u_1, u_2) > \rho$*.*

We say that a node $u \in \rho-$net covers the nodes contained in $\mathcal{B}_u(\rho)$. Thus, each node of the network is covered by at least one $u \in \rho-$net. In the sequel, we denote the nodes of a $\rho-$net as **net-centers** of the $\rho-$net.

Given a constant doubling metric, we know that for every ball $\mathcal{B}_v(r)$, it is possible to cover all nodes in $\mathcal{B}_v(r)$ with 2^{α} balls of half the radius. However, this is only a feasibility statement and does not give an upper bound on the number of $\mathcal{B}(\frac{r}{2})$ that may be deployed to cover $\mathcal{B}_v(r)$. The following property gives an upper bound on the number net-centers of $\frac{r}{2}-$net a ball $\mathcal{B}_v(r)$ may cover.

Property 5.4 (Sparseness)**.** *For $x \in \mathbb{N}^0$, each ball $\mathcal{B}_v(2^x\rho)$ covers at most $2^{(1+x)\alpha}$ nodes from an arbitrary $\rho-$*net *on the same graph with constant doubling dimension α.*

Proof. By recursively applying the definition of the doubling metric, we observe that it is possible to cover $\mathcal{B}_v(2^x\rho)$ with at most $2^{(1+x)\alpha}$ balls of radius $\frac{\rho}{2}$. Assume we know such a covering C with $|C| \leq 2^{(1+x)\alpha}$.

---

[4] Note that even for a huge network of 10^6 nodes, diameter 10^4, and $\alpha = 4$, the routing labels are still below the size of an IPv6 address and fit in the tiny messages of today's sensor nodes.

In order to see that $\mathcal{B}_v(2^x\rho)$ covers at most $2^{(1+x)\alpha}$ net-centers $U = \{u_1, u_2, ..., u_{|U|}\}$ from any $\rho-$net, we first observe that the distance between any two distinct nodes in U exceeds ρ. Secondly, by the definition of a cover, each $u_j \in U$ is covered by at least one ball from the cover C. Because the balls in C have radius $\frac{\rho}{2}$ and the distance between any two u_j is more than ρ, any $c \in C$ covers at most one $u_j \in U$. Thus, $|U| \leq |C| \leq 2^{(1+x)\alpha}$, which completes the proof. □

Along with this upper bound on the number of net-centers covered by a ball, we can give an upper bound on the number of net-centers of a given $\rho-$net that may cover any given node v. In particular, we are interested in the maximum number of net-centers of a $\rho-$net that cover a node v, if each net-center u covers an extended ball $\mathcal{B}_u(2^x\rho)$.

Property 5.5 (Dominance)**.** *Given a $\rho-$net on a α-doubling metric represented by the graph $G = (V, E)$ where each net-center u covers $\mathcal{B}_u(2^x\rho)$ with $x \in \mathbb{N}^0$, then any node $v \in V$ is covered by at most $2^{(x+1)\alpha}$ net-centers from the $\rho-$net.*

Proof. If v is covered by a net-center u, then u is at most $2^x\rho$ away from v. Therefore, it is sufficient to show that $\mathcal{B}_v(2^x\rho)$ contains at most $2^{(x+1)\alpha}$ $\rho-$net-centers, which follows from Property 5.4. □

Corollary 5.6 (Dominance II)**.** *Given a $\rho-$net on a α-doubling metric represented by the graph $G = (V, E)$ where each net-center u covers $\mathcal{B}_u(i\rho)$ with $i \in \mathbb{R}^+, i > 1$, then any node $v \in V$ is covered by at most $2^{2\alpha}i^\alpha$ net-centers from the $\rho-$net.*

5.3 Dominance Net

Due to our minimal assumptions about the nodes' capabilities (unique ID and communication with direct neighbors), our routing algorithms need some means to characterize the network.

We propose to obtain this information through a *dominance net*, which is a hierarchic locality-preserving decomposition of the network. In a nutshell, the dominance net is built of several layers of $\rho-$nets with exponentially increasing radius ρ, each of which covers the entire network. In the sequel, we will show how to build a dominance net with the following properties:

1. The number of layers is at most $1 + \lceil \log \Theta \rceil$, where Θ is the diameter of the network.[5]

2. Each node is dominated by at most $\mathcal{O}(\log \Theta)$ nodes.

---

[5] Throughout this chapter, log stands for the binary logarithm.

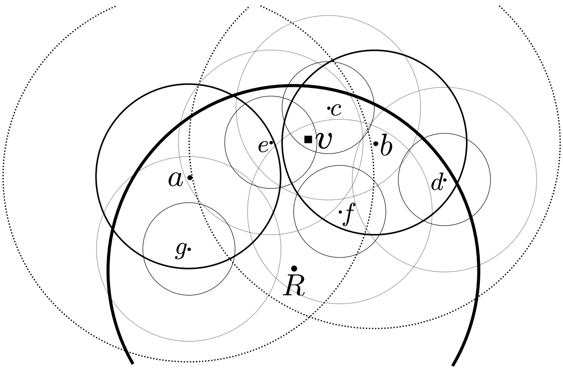


Figure 5.3: A small dominance net with only three levels. R is the root and therefore the only net-center of level-2. It covers all nodes of the network. The net-centers of level-1 are $\Gamma_1 = \{a, b, R\}$ and $\Gamma_0 = \{c, d, e, f, g, a, b, R\}$ for level-0. (Note that some nodes are net-center on several levels.) The solid circle around each net-center indicates its coverage area, whereas the dashed circle indicates the extended coverage area. Note that the coverage area may have arbitrary shape – the circular coverage area is only used for this schematic representation.

3. The parent-tree induced by the dominance net allows for an unique distance labeling with label-size $\mathcal{O}(\log \Theta)$ and stretch at most $\sqrt{6}$.

4. Adding routing tables of $\mathcal{O}(1/\epsilon)^{\alpha} \log \Delta \log \Theta$ bits to the nodes allows for a routing scheme with stretch $1 + \epsilon$. (Δ is the maximum degree of any node.)

We assume w.l.o.g. that the smallest distance between any two nodes is 1, that the diameter of the network is given by Θ, and $\boldsymbol{\vartheta} = 1 + \lceil \log \Theta \rceil$. Furthermore, we use $G = (V, E)$ to denote the graph induced by the network, whose metrization has constant doubling dimension α.

5.3.1 Dominance

For building the dominance net, we construct a hierarchy of $\rho-$nets on G with $\rho = 2^i$, where i is chosen from the range $\{0, 1, 2, 3, ..., \vartheta - 1\}$. In the following, we call the the $(2^i)-$net the **level-*i*** of the hierarchy, and we denote its net-centers by $\boldsymbol{\Gamma_i}$. Note that on level-$(\vartheta - 1)$, a single (arbitrary) node of the network is elected as net-center. We call this node the **root** of the hierarchy. An algorithm to construct these $\rho-$nets in a distributed manner is presented in Chapter 5.7.

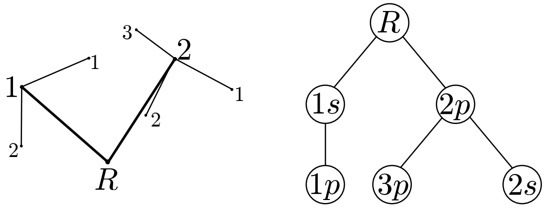


Figure 5.4: The left picture shows the dominance-tree corresponding to the dominance-net of Figure 5.3. The tree on the right is the cover-tree of node v, a tree-representation of v's label. At depth i, this tree contains the enumeration-values of the net-centers of level-$(\vartheta - i - 1)$ which cover v. The letter p indicates a primary net-center, s a secondary net-center.

We define the dominance on this $\rho-$net hierarchy in the following way: A net-center $\gamma_i \in \Gamma_i$ is dominated by a net-center $\gamma \in \Gamma_{i+1}$ iff $\gamma_i \in \mathcal{B}_\gamma(2^{i+1})$. Thus, each net-center (except the root) is dominated by at least one net-center of the next higher level.

5.3.2 Naming Scheme

Given the dominance-net, we name the net-centers in the following way: Each net-center v except for the root node selects exactly one[6] of its dominators to be its *parent* $\boldsymbol{\mathcal{P}(v)}$. This results in a dominance-tree with depth $\vartheta - 1$. Each parent enumerates its children and informs each of them about the assigned enumeration value. The naming scheme is defined recursively: The root has an empty name, while any other net-center obtains its name by appending its enumeration value to its parents name.

Figure 5.3 shows a sample dominance net with only three levels. The left picture of Figure 5.4 shows the corresponding dominance-tree with the assigned enumeration values. The name of a net-center is obtained by walking from the corresponding node of the dominance-tree towards the root and concatenating the enumeration values of the visited nodes. E.g. the name of net-center g is `R:1:2` whereas the name of c is `R:2:3`. The uniqueness of the names is guaranteed due to the tree structure, where each net-center is identified through its parent and the enumeration value assigned by the parent.

Because any net-center may be parent of at most $2^{2\alpha}$ net-centers (Property 5.4), the enumeration values can be represented with at most 2α bits. Thus, the name may grow by 2α bits on each level, which results in a maximum name size of $2\alpha\lceil\log\Theta\rceil = \mathcal{O}(\log\Theta)$ bits for net-centers on level-0.

---

[6] For performance reasons, each net-center may choose the closest net-center of the next higher level.

5.3.3 Dominance Labeling

Based on the dominance-net and the corresponding naming scheme, each node v of the graph G assigns itself a label $\boldsymbol{\mathcal{L}(v)}$ of size $\mathcal{O}(\log\Theta)$ which allows for efficient routing, multicasting, anycasting, and distance labeling. In the remainder of this section, we show the following theorem:

Theorem 5.7 (Compact Labels)**.** *The nodes of G can be assigned unique labels of size $\mathcal{O}(\log\Theta)$ allowing approximate distance queries of stretch $\sqrt{6}$ between any two nodes.*

We use an *extended dominance net* for the labeling, where a net-center $\gamma \in \Gamma_i$ covers all nodes in $\mathcal{B}_\gamma(2^{i+1})$, cf. Figure 5.3. This extension results in an overlap of the net-covers, which allows to rule out borderline effects where two close-by neighbors are not covered by common net-centers. Given this extended dominance set, each node v determines the net-centers by which it is covered. Essentially, the label of v contains the name of all net-centers that cover v.

In order to obtain the desired label size for a node v, we cannot simply store all names of the covering net-centers: By construction, there are at most ϑ levels, each of which has at most $2^{2\alpha}$ net-centers covering v (Property 5.5). From the previous section, we know that the naming for the net-centers uses at most $2\alpha(\vartheta-i-1)$ bits on level-i, which bounds the maximum label length to $\sum_{i=0}^{\vartheta} 2^{2\alpha}2\alpha(\vartheta-i-1) = \mathcal{O}(\vartheta^2) = \mathcal{O}(\log^2\Theta)$. To receive the $\mathcal{O}(\log\Theta)$ label size, we make use of the following lemma which shows that only partial net-center names need to be stored for each covering net-center.

Lemma 5.8 (Parental Cover)**.** *If a net-center $c \in \Gamma_i$ covers a node v in the extended dominance net, then its parent $\mathcal{P}(c) \in \Gamma_{i+1}$ covers v as well.*

Proof. The net-center c on level-i only covers nodes in $\mathcal{B}_c(2^{i+1})$ in the extended dominance net. Therefore, $\mathrm{d}_{\mathcal{M}}(v,c) \leq 2^{i+1}$. Because the parent $\mathcal{P}(c)$ of c is chosen from the ordinary[7] dominance net and $\mathcal{P}(c) \in \Gamma_{i+1}$, $\mathrm{d}_{\mathcal{M}}(c,\mathcal{P}(c)) \leq 2^{i+1}$. Using the triangle inequality we receive that $\mathrm{d}_{\mathcal{M}}(v,\mathcal{P}(c)) \leq \mathrm{d}_{\mathcal{M}}(v,c) + \mathrm{d}_{\mathcal{M}}(c,\mathcal{P}(c)) \leq 2^{i+2}$. The claim follows because $\mathcal{P}(c)$ covers $\mathcal{B}_{\mathcal{P}(c)}(2^{i+2})$ in the extended dominance net, which includes v. □

5.3.4 Cover Tree

The label $\mathcal{L}(v)$ of a node v is the flat representation of a cover tree that efficiently describes all net-centers that cover v. This tree is rooted at the root node. The body of the cover tree is defined recursively: At depth i, the tree contains the net-centers $\gamma \in \Gamma_{\vartheta-i-1}$ that cover v. Each γ in the cover tree is connected to its parent $\mathcal{P}(\gamma)$. This is possible because we know from

---

[7] *Ordinary* means not in the extended dominance net.

Lemma 5.8 that if γ covers v, then also $\mathcal{P}(\gamma)$ covers v and therefore must be present in the cover tree.

Because the name of a net-center is defined as the recursive concatenation of the enumeration values of its parents in the dominance net (see Chapter 5.3.2) and the same parenthood persists in the cover tree, each node of the cover tree only needs to store the enumeration value of its corresponding net-center. The name of a net-center c in the cover tree can be obtained by prefixing the enumeration value of c with the name of $\mathcal{P}(c)$, which is obtained recursively. Therefore, each node of the cover tree carries at most 2α bits, see Figure 5.4 for an example. We also know from Property 5.5 that there are at most $2^{2\alpha}$ net-centers per level covering node v. Because the depth of the tree is at most $\vartheta - 1$, we deduce that the cover tree holds at most $\alpha 2^{2\alpha+1}\vartheta$ bits. The serialization of the cover tree to a flat data structure is straightforward and can be done introducing only two additional bits per net-center, which leads to a label size of $(2\alpha + 2)2^{2\alpha}\vartheta \in \mathcal{O}(\log \Theta)$ bits.

In the remainder of this discussion, we rely on a slightly enhanced node labeling where each node v not only stores the net-centers by which it is covered, but also indicates their type. We distinguish two types of net-centers depending on their distance to v: A net-center $c \in \Gamma_i$ is called **primary** if $v \in \mathcal{B}_c(2^i)$. Otherwise, the net-center is called **secondary**, i.e. a secondary net-center covers v only in the extended dominance net, whereas a primary net-center covers v already in the ordinary dominance net. This additional bit per net-center does not significantly increase the size of the cover tree, which becomes at most $(2\alpha + 3)2^{2\alpha}\vartheta = \mathcal{O}(\log \Theta)$ bits. This proves the first part of Theorem 5.7.

5.3.5 Distance Approximation

To obtain the distance between two nodes a and b given their labels $\mathcal{L}(a)$ and $\mathcal{L}(b)$, we determine the smallest level-i for which a and b have at least one common net-center. Let $\rho = 2^i$ be the ordinary coverage radius of level-i and $C \subseteq \Gamma_i$ the set of common net-centers on level-i.

We immediately get a lower bound on the distance between a and b by observing that if $\mathrm{d}_{\mathcal{M}}(a, b) \leq \rho/2$, there exists a net-center $c \in \Gamma_{i-1}$ that covers both a and b. Therefore, $\mathrm{d}_{\mathcal{M}}(a, b) > \rho/2$. For the remainder of this proof, we need to consider the following three cases only:

1. If there is a net-center $c \in C$ s.t. c is primary for both a and b, then $\mathrm{d}_{\mathcal{M}}(a, b) \leq 2\rho$ and therefore $\mathrm{d}_{\mathcal{M}}(a, b) \in (\rho/2, 2\rho]$. This property holds due to the triangle inequality: Because a and b are primary, $\mathrm{d}_{\mathcal{M}}(a, c) \leq \rho$ and $\mathrm{d}_{\mathcal{M}}(b, c) \leq \rho$, which implies that $\mathrm{d}_{\mathcal{M}}(a, b) \leq \mathrm{d}_{\mathcal{M}}(a, c) + \mathrm{d}_{\mathcal{M}}(c, b) \leq 2\rho$.

2. If all net-centers $c \in C$ are secondary for a and b, $\mathrm{d}_{\mathcal{M}}(a, b) \in (\rho, 4\rho]$. Again, the upper bound is given by the triangle inequality: $\mathrm{d}_{\mathcal{M}}(a, c) \leq$

2ρ and $\mathrm{d}_{\mathcal{M}}(b,c) \leq 2\rho$, and therefore $\mathrm{d}_{\mathcal{M}}(a,b) \leq 4\rho$. The lower bound stems from the fact that any primary net-center of a would cover b if $\mathrm{d}_{\mathcal{M}}(a,b) \leq \rho$, and vice versa.

3. Finally, if there is at least one net-center $c \in C$ s.t. c is primary for either a xor b, then $\mathrm{d}_{\mathcal{M}}(a,b) \in (\rho/2, 3\rho]$. W.l.o.g. assume that c is primary for node a. Then $\mathrm{d}_{\mathcal{M}}(a,c) \leq \rho$ and $\mathrm{d}_{\mathcal{M}}(c,b) \leq 2\rho$. Therefore, $\mathrm{d}_{\mathcal{M}}(a,b) \leq \mathrm{d}_{\mathcal{M}}(a,c) + \mathrm{d}_{\mathcal{M}}(c,b) \leq 3\rho$.
Note: If for either a xor b there is no $c \in C$ s.t. c is primary for the node, then $\mathrm{d}_{\mathcal{M}}(a,b) \in (\rho, 3\rho]$. The increase in the lower bound holds because no primary net-center of a xor b covers the other node, which is only possible if $\mathrm{d}_{\mathcal{M}}(a,b) > \rho$.

Given the interval $(r_1, r_2]$ for the possible values of $\mathrm{d}_{\mathcal{M}}(a,b)$, we set $\mathrm{d}_{\mathcal{M}}(a,b) = \sqrt{r_1 r_2}$, the geometric mean of the two bounds. The maximum factor by which the approximation is off from the actual distance is $\sqrt{r_2/r_1}$. Therefore, our labeling scheme suffers from a maximum stretch in the third case, which is at most $\sqrt{6}$. This concludes the proof of Theorem 5.7.

5.4 Routing

We present a single destination routing algorithm (*SDR*) which will be used as a building block for multicasting and anycasting. In SDR, a message needs to be forwarded from a sender node s to a single target node t. The stretch $\boldsymbol{S_{\mathcal{A}}}$ of a routing algorithm $\mathcal{A}$ is defined as $S_{\mathcal{A}} = \max_{s,t \in V} \frac{\mathrm{d}_{\mathcal{A}}(s,t)}{\mathrm{d}_{\mathcal{M}}(s,t)}$ where $d_{\mathcal{A}}(s,t)$ is the length of the path found by the routing algorithm. Clearly, we desire the stretch to be as small as possible, but this comes at a certain cost. In our approach, the stretch is coupled with the routing table size, i.e. lowering the stretch induces bigger routing tables. Our unicast routing result is summarized in the following theorem, where Θ is the diameter of the network, and $\boldsymbol{\Delta}$ stands for the maximum degree of a node.

Theorem 5.9. *For a fixed ϵ with $0 < \epsilon \leq 2$, SDR routes messages with stretch $(1+\epsilon)$ such that the chosen path by SDR is at most $(1+\epsilon)\mathrm{d}_{\mathcal{M}}(s,t)$. The header size of each message is at most $2\alpha\lceil\log\Theta\rceil + \lceil\log n\rceil$ bits, and the routing tables stored at each node require only $\mathcal{O}\left(\frac{1}{\epsilon}\right)^{\alpha}(\log\Theta)(\mathcal{O}(\alpha) + \log\Delta)$ bits.*

The routing scheme is based on the node labeling introduced in Chapter 5.3.4 and is quite simple: The sender node s extracts from $\mathcal{L}(t)$ the name $\mathcal{N}$ of an arbitrary primary net-center $\gamma \in \Gamma_0$ that covers the target node t. This name $\mathcal{N}$ and the ID of t serve as header information in the message, which uses at most $2\alpha\lceil\log\Theta\rceil + \lceil\log n\rceil$ bits (see Chapter 5.3.2). Remember that $\mathcal{N}$ is a concatenation of the enumeration values of γ and its ancestors in the dominance tree. Therefore, $\mathcal{N}$ encodes the names of exactly one net-center (primary or secondary) per level that covers t (Lemma 5.8). Let us denote these net-centers by $\{c^0, c^1, c^2, c^3, \ldots, 2^{\vartheta-2}\}$, where c^i is the net-center on level-i. The forwarding mechanism works as follows: Each node that receives a message finds the net-center c^i on the lowest level for which it has routing information and forwards the message in this direction. This is repeated until the message hits a direct neighbor of t which sends the message directly to t based on its neighborhood list.

In order to support this forwarding scheme, each node needs to store how to reach some of the net-centers in its surroundings. For this purpose, each net-center $\gamma \in \Gamma_i$ advertises itself to $\mathcal{B}_\gamma(\eta 2^i)$ with $\boldsymbol{\eta} = \frac{8}{\epsilon} + 6$. I.e. every node in this ball stores how to reach γ on an optimal path. In fact, it suffices to store the neighbor node lying on the optimal path. This corresponds to the shortest path problem (in a restricted area) and can be obtained by distributed versions of the Dijkstra algorithm. Thus, the routing table of each node v stores how to reach any net-center $\gamma \in \Gamma_i$ iff $\mathrm{d}_{\mathcal{M}}(v, \gamma) \leq \eta 2^i$. This can be seen as a mapping from the *name* of the net-center γ to the neighbor of v which lies on the shortest path to γ. Before we show in Chapter 5.4.2 that the routing information can be stored efficiently, we prove that SDR finds good routing paths.

5.4.1 $(1+\epsilon)$-Stretch Routing

We first observe that once the message has reached a net-center covering t, it can be forwarded to a net-center covering t on a lower level:

Lemma 5.10 (Net-Center Hopping)**.** *Given a net-center $c^i \in \Gamma_i$ that covers the destination node t, the routing table of c^i contains entries for all net-centers $c^{i-x} \in \Gamma_{i-x}$ covering t. This holds for $x \leq \lfloor \log_2 \frac{\eta-2}{2} \rfloor$.*

Proof. Suppose that $c^j \in \Gamma_j$ with $j < i$ is a net-center of t. Due to the routing table construction, all nodes in $\mathcal{B}_{c^j}(\eta 2^j)$ have routing table entries to c^j. Using the triangle inequality, we know that $\mathrm{d}_{\mathcal{M}}(c^j, c^i) \leq 2^{j+1} + 2^{i+1}$. Therefore, c^i is sure to have a routing table entry if $\mathrm{d}_{\mathcal{M}}(c^j, c^i) \leq \eta 2^j$, which holds if $x = i - j \leq \lfloor \log_2 \frac{\eta-2}{2} \rfloor$. □

The following proof of the routing stretch describes the worst case scenario where the message visits the net-centers $\{c^i,\ c^{i-x},\ c^{i-2x},\ \ldots\}$, where c^i is the net-center towards which s forwards the message, cf. Figure 5.5.

Proof. (*Routing Stretch*) Let c^i be the net-center on the lowest level for which s has routing information. Following the routing algorithm, the message is first forwarded to c^i, from where it can be forwarded to c^{i-x} with $x = \lfloor \log_2 \frac{\eta-2}{2} \rfloor$ (Lemma 5.10). This step is repeated until the message reaches

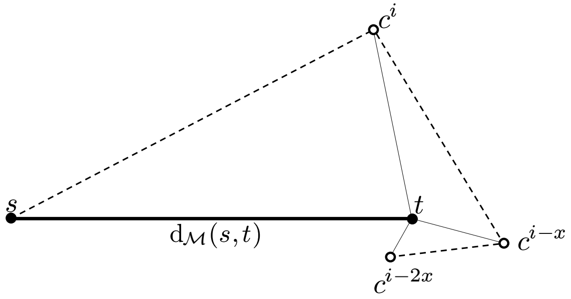


Figure 5.5: Schematic illustration (out of scale) of the worst case scenario where a message to be sent from s to t first visits c^i, then c^{i-x} and so on, until it reaches a level-0 net-center covering t which directly delivers the message.

a net-center on level-0, which directly delivers the message. Thus, the total distance $\mathrm{d}_{\mathrm{SDR}}(s,t)$ of SDR is bounded by

$$\mathrm{d}_{\mathcal{M}}\left(s,c^{i}\right)+\mathrm{d}_{\mathcal{M}}\left(c^{i},c^{i-x}\right)+\mathrm{d}_{\mathcal{M}}\left(c^{i-x},c^{i-2x}\right)+\cdots+1$$

Using the triangle inequality, we obtain that $\mathrm{d}_{\mathcal{M}}\left(s,c^{i}\right)\leq\mathrm{d}_{\mathcal{M}}(s,t)+2^{i+1}$ and for $j\geq 0:\mathrm{d}_{\mathcal{M}}\left(c^{i-jx},c^{i-(j+1)x}\right)\leq 2\cdot 2^{i-jx}+2\cdot 2^{i-(j+1)x}$. Therefore,

$$\begin{aligned}
\mathrm{d}_{\mathrm{SDR}}(s,t) &\leq \mathrm{d}_{\mathcal{M}}(s,t)+4\sum_{j=0}^{\infty}2^{i-jx} && (5.1)\\
&= \mathrm{d}_{\mathcal{M}}(s,t)+4\sum_{j=0}^{\infty}\frac{2^{i}}{2^{\lfloor\log_2\frac{\eta-2}{2}\rfloor j}} && (5.2)\\
&\leq \mathrm{d}_{\mathcal{M}}(s,t)+2^{i+2}\sum_{j=0}^{\infty}\left(\frac{4}{\eta-2}\right)^{j} && (5.3)\\
&\leq \mathrm{d}_{\mathcal{M}}(s,t)+2^{i+2}\frac{\eta-2}{\eta-6} && (5.4)
\end{aligned}$$

We obtain (3) from (2) by observing that $2^{\lfloor\log_2\frac{\eta-2}{2}\rfloor j}\geq 2^{j\log_2\frac{\eta-2}{4}}=((\eta-2)/4)^{j}$. To obtain (4), note that the sum in (3) sums the elements of a geometric series with factor $\frac{4}{\eta-2}=\frac{\epsilon}{2+\epsilon}<1$.

Because s does not have a routing table entry for c^j with $j<i$, we deduce that $\mathrm{d}_{\mathcal{M}}(s,t)>(\eta-2)2^{i-1}$. This holds because $\mathrm{d}_{\mathcal{M}}\left(c^{i-1},t\right)\leq 2\cdot 2^{i-1}$ and c^{i-1} advertises itself to $\mathcal{B}_{c^{i-1}}(\eta 2^{i-1})$. Therefore, s has a routing entry to c^{i-1} if $\mathrm{d}_{\mathcal{M}}(s,t)\leq(\eta-2)2^{i-1}$.

Putting together the two results, we obtain that the routing stretch of SDR is

$$\begin{aligned} S_{\mathrm{SDR}} &\leq \frac{\mathrm{d}_{\mathrm{SDR}}(s,t)}{\mathrm{d}_{\mathcal{M}}(s,t)} \leq 1 + \frac{2^{i+2}\frac{\eta-2}{\eta-6}}{2^{i-1}(\eta-2)} \\ &= 1 + \frac{8}{\eta-6} = 1 + \epsilon. \end{aligned}$$

This constitutes the proof for the routing stretch statement in Theorem 5.9. □

Throughout the proof, we assumed that the message visits the net-centers $\{c^i, c^{i-x}, c^{i-2x}, \dots\}$, which is the worst case scenario. A considerable performance boost can be achieved in the average case if the routing algorithm tests in each step for a closer net-center. The key idea is that while the message is being routed towards c^j, it tests after each hop whether the current node can route towards a closer net-center c^k with $k < j$. If this is the case, the message immediately routes towards c^k. This produces a shortcut towards t, reducing the routing path and therefore the stretch.

5.4.2 Compact Tables

The construction of the routing table is similar to the cover tree presented in Chapter 5.3.4 with the main difference that each net-center $\gamma \in \Gamma_i$ covers an area which depends on ϵ, namely $\mathcal{B}_\gamma((\frac{8}{\epsilon}+6)2^i)$. Because of this mutable coverage radius, we need to restate Lemma 5.8:

Lemma 5.11 (Parental Cover II). *If a node v has routing information about net-center $\gamma \in \Gamma_i$, then v also has a routing entry to $\mathcal{P}(\gamma)$ unless γ is the root node.*

Proof. Node v has routing information about $\gamma \in \Gamma_i$ iff $v \in \mathcal{B}_\gamma(\eta 2^i)$ with $\eta = \frac{8}{\epsilon}+6$. Because $\mathrm{d}_{\mathcal{M}}(\gamma, \mathcal{P}(\gamma)) \leq 2^{i+1}$ and $\mathrm{d}_{\mathcal{M}}(\gamma, v) \leq \eta 2^i$, $\mathrm{d}_{\mathcal{M}}(v, \mathcal{P}(\gamma)) \leq (\eta+2)2^i$ using the triangle inequality. Therefore, it is sufficient to show that $(\eta+2)2^i \leq \eta 2^{i+1}$ such that $v \in \mathcal{B}_{\mathcal{P}(\gamma)}(\eta 2^{i+1})$. The inequality holds because $\eta = \frac{8}{\epsilon}+6 \geq 2$. □

Using the same arguments as in Chapter 5.3.4, we can show that all net-centers for which a node v needs to store routing information can be be stored in a tree, where each tree node corresponds to one net-center. Each node of the tree only holds the enumeration value of the corresponding net-center γ and the routing information on how to reach γ. Recall that the enumeration value fits in 2α bits and the routing information is the neighbor node which lies on the optimal path between v and the corresponding net-center. Thus, the routing information uses at most $\log \Delta$ bits, where Δ is the maximum degree of a node.

Each node v needs to keep routing information for at most $2^{2\alpha}\left(\frac{8}{\epsilon}+6\right)^{\alpha}$ net-centers per level (Corollary 5.6). Because there are at most ϑ levels, the tree has at most $2^{2\alpha}\left(\frac{8}{\epsilon}+6\right)^{\alpha}\vartheta$ nodes, each of which needs to store $2\alpha+\log_2\Delta$ bits. We have already noted that such a tree can be stored in a flat data structure adding only two bits per node, which results in a total routing table size of $(2\alpha+\log_2\Delta+2)2^{2\alpha}\left(\frac{8}{\epsilon}+6\right)^{\alpha}\vartheta=\mathcal{O}\left(\frac{1}{\epsilon}\right)^{\alpha}(\log\Theta)(\mathcal{O}(\alpha)+\log\Delta)$ bits. This concludes the proof of Theorem 5.9.

5.5 Multicasting

We now have developed all tools that allow for efficient multicasting from a sender node s to a set U of receivers. In a nutshell, the sender s approximates a minimum spanning tree (MST) on the set $s \cup U$ using Kruskal's algorithm and then routes the message along this tree towards all receivers. Note that in contrast to the centralized multicasting presented in [95], our approach is distributed. We show the following result:

Theorem 5.12. *Consider a network $G=(V,E)$ on which a dominance net with the associated labeling and routing tables was created. Then, any sender node $s\in V$ can multicast messages to any set $U\subset V$ with constant stretch. The cost associated with the multicasting is at most $12(1+\epsilon)$ times the cost of an optimal multicasting algorithm, which knows the entire network topology.*

Proof. We need to show that the path along the MST approximation is at most $12(1+\epsilon)$ longer than the optimal path, which is given by a minimum Steiner tree (MStT) on the set $s\cup U$ and the remaining nodes as Steiner points. The stretch is composed of three parts: First, a MST on $s\cup U$ is a 2-approximation of the corresponding MStT. This result was shown by Kou et al. in [54]. The $(1+\epsilon)$ part is caused by the SDR routing scheme, which is responsible to forward the message along the tree. Lastly, the construction of the MST is based on the stretch-$\sqrt{6}$ distance labeling. As a result, Kruskal's algorithm may not choose the shortest, but up to a factor 6 longer edges in each step, which results in a MST approximation at most 6 times longer than the MST. □

5.6 Anycast

As for the special case of anycasting, where a message has to be routed to exactly one node of a given node set U, we provide a constant stretch algorithm to deliver the message:

Theorem 5.13. *The node labeling from Chapter 5.3.4 and SDR from Chapter 5.4 allow for a $6(1+\epsilon)$-approximation to anycast.*

Proof. Based on the distance labeling, pick the node $u \in U$ which seems closest to the sender node s and send the message to u. Because of the stretch of the distance labeling, this approach may pick a receiver that is up to 6 times further away from s than the optimal receiver. The $(1+\epsilon)$ factor stems from the SDR routing scheme. □

5.7 Distributed Dominance Net Construction

In this last section, we describe an efficient distributed algorithm to build the $\rho-$nets that constitute the dominance net. Recall from Chapter 5.3.1 that the dominance net consists of ϑ $\rho-$nets with exponentially increasing ρ chosen from $\{1, 2, 4, ..., 2^{\vartheta-1}\}$.

To start, we would like to point out that a *centralized* algorithm to build a $\rho-$net on $G = (V, E)$ is straightforward: Greedily pick an arbitrary $v \in V$, add v to Γ and remove all nodes in $\mathcal{B}_v(\rho)$ from V. Repeat until V is the empty set. By construction, the distance between any two net-centers is longer than ρ and each node $v \in V$ is covered by at least one net-center.

For our *distributed* algorithm, we exploit the fact that a $\rho-$net is a maximal independent set (MIS) for the ρ metric closure[8] $G^\rho = (V, E^\rho)$. This immediately leads to a simple distributed algorithm to create a $\rho-$net: create a ρ metric closure where each node $v \in V$ has all nodes in $\mathcal{B}_v(\rho)$ as direct neighbors. Then, run a distributed MIS algorithm on the closure and pick the nodes in the MIS to be the net-centers of the $\rho-$net. There exists a broad assortment of distributed MIS algorithms, e.g. there is an elegant randomized algorithm with expected running time in $\mathcal{O}(\log n)$ by Luby [69]. More recently, Kuhn et al. [57] described a deterministic MIS construction for bounded independence graphs with running time $\mathcal{O}(\log \Delta \log^* n)$.

The above algorithm is realistic for small values of ρ. However, when ρ approaches Θ, the nodes tend to have huge neighborhoods, consisting of nearly all other nodes of the network. This may quickly exceed the memory capabilities of simple network nodes, especially when considering large networks. To overcome this issue, we present not only a relatively fast, but also *memory conservative* algorithm to construct the $\rho-$net hierarchy.

5.7.1 Sequential $\rho-$nets

Algorithm 5.1 describes in high-level pseudocode the steps each node v of the network performs in order to create the dominance nets. At the end of the algorithm, each node knows for which $\rho-$nets it is a net-center, and it already holds the complete routing table for the SDR routing scheme presented in Chapter 5.4. In a nutshell, the algorithm is recursive and performs the following two steps to build the $\rho-$net of level-i.

---

[8] The *n metric closure* of a graph $G = (V, E)$ is the graph $G^n = (V, E^n)$ with $E^n = \{(u, v) | u, v \in V \wedge u \neq v \wedge \mathrm{d}_{\mathcal{M}}(u, v) \leq n\}$

Algorithm 5.1: Dominance Net (Code for node v)

```
state = active
Main()
    $N$ = nodes in $\mathcal{B}_v(1)$
    Build MIS with neighbor set $N$
    if $v \notin$ MIS then
        state = passive
    else
        Inform $\mathcal{B}_v(1)$                                  $v$ is a net-center of level-0
        Add routing entries to $v$ in $\mathcal{B}_v(\eta)$                     $\eta = \frac{4}{\epsilon} + 3$
        JoinMIS(1)
    end
JoinMIS($i$)
    $\rho = 2^i$
    $N$ = active nodes at most $\rho$ away
    Build MIS with neighbor set $N$
    if $v \notin$ MIS then
        state = passive
                    $v$ is not covered if $\nexists$ net-center $u$ of level-$i$ s.t. $v \in \mathcal{B}_u(\rho)$
        while $\exists w \in \mathcal{B}_v(\frac{\rho}{2})$ s.t. $w$ not covered do
            $P$ = arbitrary uncovered node in $\mathcal{B}_v(\frac{\rho}{2})$
            Send EXC($i$) to $P$ and wait for answer
        end
        Send ACK($i$) to $N$
    else
        Inform $\mathcal{B}_v(\rho)$                                  $v$ is a net-center of level-$i$
        Add routing entries to $v$ in $\mathcal{B}_v(\eta\rho)$
        Collect ACK($i$) from all neighbors $N$
        JoinMIS($i + 1$)
    end
ReceiveMessage(EXC($i$) from $u$)
    state = excited, $\rho = 2^i$
    Add temporary routing entries to $v$ in $\mathcal{B}_v(\rho)$
    $N$ = excited nodes at most $\rho$ away
    Build MIS with neighbor set $N$
    if $v \in$ MIS then
        state = active
        Inform $\mathcal{B}_v(\rho)$                                  $v$ is a net-center of level-$i$
        Add/validate routing entries to $v$ in $\mathcal{B}_v(\eta\rho)$
        JoinMIS($i + 1$)
    else
        state = passive
        Remove temporary routing entries to $v$ in $\mathcal{B}_v(\rho)$
    end
    Send ACK(state) to $u$
```

1. Approximate a ρ−net given the $\frac{\rho}{2}$−net, i.e. build a MIS on the ρ metric closure of Γ_{i-1}, the independent nodes join Γ_i. Note that this

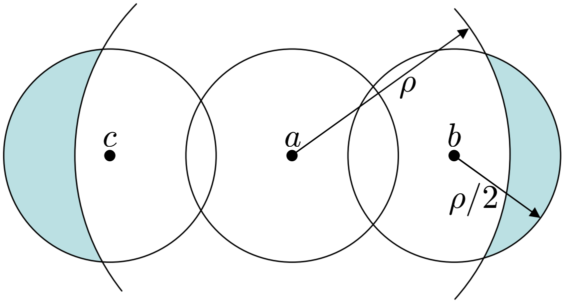


Figure 5.6: A MIS on the ρ metric closure on the net-centers of a $\frac{\rho}{2}-$net does not necessarily cover all nodes of the network: Suppose that the $\frac{\rho}{2}-$net consists of $\{a, b, c\}$. One possible MIS on these 3 nodes is to pick a, which covers b and c, but not the shaded areas. Note that coverage areas may have arbitrary shape and need not be circular, as drawn in this example.

MIS only guarantees that the net-centers of the $\frac{\rho}{2}-$net are covered, but it does not ensure coverage of all nodes of the network.

2. Add additional net-centers to Γ_i until all nodes $v \in V$ are covered by at least one $\gamma \in \Gamma_i$.

We obtain the desired memory relaxation by building the $\rho-$nets sequentially on top of each other. Initially, all nodes participate in a MIS construction on a 1 metric closure of the network, which results in the $1-$net (Lines 2–4). Then, for any level-i, the ρ metric closure is constructed only with the net-centers Γ_{i-1} and induces at most $2^{2\alpha}$ neighbors per node, independent of the network size (Property 5.4). The downside of this approach is a longer running time and that the MIS on the net-centers of level-$(i-1)$ does not necessarily cover all nodes of the network, see Figure 5.6 for an example. This requires a second phase, where additional net-centers are added until full coverage is obtained.

The recursive call to join the next higher MIS is handled on the Lines 13–15. Note that only nodes that are net-centers on level-$(i-1)$ participate in the MIS construction for level-i. If node v does not make it into the MIS of level-i, it becomes passive (Line 17) and is responsible that all nodes in $\mathcal{B}_v(2^{i-1})$ are covered (Lines 18–22). This covering algorithm works in several rounds, where v participates as long as $\mathcal{B}_v(2^{i-1})$ contains uncovered nodes. In each round, each net-center whose $\mathcal{B}(2^{i-1})$ contains uncovered nodes picks an uncovered node and sets its state to *excited* by sending an `EXC` message. The set of excited nodes build a MIS on their ρ metric closure (Lines 32–33), and the independent excited nodes join the $(2^i)-$net (Lines 35–38). The temporary routing entries added on Line 31 and, if the node does not make it into the MIS, removed on Line 41 enable the MIS algorithm to exchange messages on the optimal path between any two neighbors in the corresponding metric closure.

For simplicity, we did not address the following issues in the pseudocode of Algorithm 5.1.

- If a node v receives several `EXC` requests in the same round, it executes Lines 30–42 only once, but finally acknowledges on Line 43 to all nodes that sent the request.
- The net-centers need to obtain their enumeration values (see Chapter 5.3.2) in order to construct their labels. Because these labels are incomplete until all $\rho-$nets are constructed, the routing table entries (constructed on Lines 9, 25, 31, 37) are built using a partial labeling.
- The algorithm requires a rough synchronization between consecutive MIS constructions, such that a MIS construction only starts when the previous one finished. This can be obtained during the construction of the metric closure for the next MIS (Lines 14 and 32), where any node can ask the others to wait. To support this synchronization, each node that joined the $\rho-$net waits until all of its neighbors (in the metric closure) have terminated the coverage procedure (Line 26).
- Whenever $N = \emptyset$ on Line 14, node v trivially joins the MIS and tests whether it covers all nodes of the network. If this is the case, v becomes the root of the network and stops the recursive call to JoinMIS().
- Before the first MIS can be constructed, the nodes need to determine the lowest cost associated with any edge of the network, as to determine the scaling factor that scales this cost to 1.

Theorem 5.14. *Algorithm 5.1 constructs a dominance net as described in Chapter 5.3.1. Furthermore, the algorithm requires to build a total of at most* $\vartheta(1+2^{3\alpha}) = \mathcal{O}(\log\Theta)$ *MIS, which gives an upper bound on its running time.*

Proof. We show the first property by induction on the levels: Initially, all nodes participate in a MIS election on the 1-metric-closure, which corresponds to building a MIS on the network where only the edges with cost 1 are considered. By construction, the resulting MIS is a $1-$net.

Given the points Γ_{i-1} of a $2^{i-1}-$net, the algorithm first approximates Γ_i by a MIS on the 2^i metric closure of Γ_i. By construction, $a, b \in \Gamma_i \implies \mathrm{d}_{\mathcal{M}}(a, b) > 2^i$. Because only nodes that are not covered by Γ_i may become excited (Line 19), the addition of any excited node to Γ_i does not break this property. Furthermore, the excited nodes added to Γ_i in one round are independent with respect to the 2^i metric closure on the set of excited nodes (Lines 32–34). Therefore, the minimum distance between net-centers of level-i is preserved. Because every $\gamma \in \Gamma_{i-1}$ repeats the completion process until all nodes in $\mathcal{B}_\gamma(2^{i-1})$ are covered (Line 18), Γ_i finally covers all nodes and is a valid 2^i−net.

As for the running time, we show that the algorithm may have to build at most $1+2^{3\alpha}$ MIS per level: The first MIS is built on Line 15, the remaining ones on Line 33. Therefore, we need to show that a node v may execute the while-loop of Line 18 at most $2^{3\alpha}$ times until all nodes in $\mathcal{B}_v(2^{i-1})$ are covered. We first note that an excited node P (Line 19) may not join the MIS if another excited node Q with $\mathrm{d}_{\mathcal{M}}(P,Q) \leq 2^i$ joins. Therefore, any excited node Q that may interfere with P must lie in $\mathcal{B}_v(3 \cdot 2^{i-1})$, and the net-center that excited Q in $\mathcal{B}_v(2^{i+1})$. By Property 5.5, there are at most $2^{3\alpha}$ net-centers of level-$i-1$ in $\mathcal{B}_v(2^{i+1})$ and therefore at most $2^{3\alpha}-1$ excited nodes that may interfere with P. Just looking at P, we know that either P or one of its interfering nodes is selected in each round of the while loop. Thus, after at most $2^{3\alpha}-1$ rounds, P can join the MIS, which shows that at most $1+2^{3\alpha}$ MIS are necessary per level. Because there are at most ϑ levels, the number of MIS to construct is bounded by $\vartheta(1+2^{3\alpha})$. □

6

Conclusion

In the preceding chapters we have described several routing protocols for wireless ad hoc networks. We started with a location service for mobile networks which is a prerequisite for any geographic routing approach when the nodes may change their position over time. The described protocol is novel in the sense that we can show that messages are successfully delivered even if the destination node is constantly moving. The formalism introduced to prove this property, however, is based on rather strong model assumptions such as a dense node distribution, reliable communication, and the knowledge of the position of lakes. In fact, by abstracting from the underlying routing layer that forwards a message to a given position, we have hidden one of the most difficult tasks in dynamic networks. For example, consider Figure 6.1 which depicts two networks A and B that are connected through two edges e_1 and e_2 and assume that a message needs to be routed from a node within A to a node within B. In a worst case scenario, the two components are always connected either by e_1 or e_2, but when the message is at a_1, e_1 is not

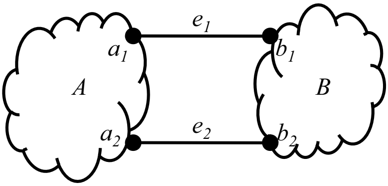


Figure 6.1: In a dynamic scenario, the two links e_1 and e_2 may come and go in a worst case fashion preventing communication between the two network components A and B.

available. Similarly, if the message is at a_2, the connection to b_2 is broken. Thus, even though the two networks are constantly connected, it is hard to route the message to its destination. Consequently, any protocol that wishes to provably ensure message delivery either needs to be able to detect and handle such situations or it needs to be analyzed under a network model that prohibits such dynamic scenarios. But this raises the question on what model should be chosen such that the model is simple enough to analyze the protocols as well as realistic enough to represent the most important properties of wireless networks. Of course, the construction of simplifying models is a tightrope walk and may not be possible at all without losing some important aspects. For instance, the UDG connectivity model abstracts from temporary link failures and the non-circular transmission range of physical radios.

Despite of the theoretical nature of our results and their analysis under somewhat strong models, we believe them to be a valuable starting point for further discussions and we hope that they may lay the basis for protocols tailored to more aspects of wireless networks, analyzed on more realistic models. For instance, many geometric routing algorithm for wireless ad hoc networks were studied solely for 2-dimensional networks. In Chapter 3, we described several randomized recovery techniques to escape local minima in 3-dimensional networks, which are by far more complex to handle than in the 2-dimensional counterparts. Even though the fine grained analysis and the detection of the face was examined only under the simplified UBG connectivity model, the results hold also for general graphs. Similarly, we showed that any CUDG has a greedy embedding with constant stretch into $\mathcal{O}\big(\log^2 n\big)$ dimensions. Without the simplified network model, our approach is still valid, but requires $\mathcal{O}\big(\log^3 n\big)$ dimensions for the embedding and may experience a stretch in $\mathcal{O}(\log n)$. The compact routing result in Chapter 5 was based on a networking model that seems to match the properties of a wireless network better than the circular transmission radii of a UDG. But the construction of the labels and the routing tables suffers from the same problem as the construction of our greedy embedding: the labels cannot be computed strictly locally and they hardly support network dynamics. Thus, there are still many open questions which we hope to see resolved in the coming years.

Part II

Application Design

7

Simulation

The application development for distributed systems introduces new challenges compared to the traditional application design. Most importantly, the actors of a distributed setup often need some means of coordination to collaborate on a given task. Depending on the quality of the communication medium, a broad range of potential failures needs to be handled to guarantee a smooth operation, increasing the complexity of this coordination. Even in the absence of such low-level failures, however, the design of distributed applications remains challenging as it is often hard to foresee the interaction patterns between a (possibly large) set of concurrently executing actors. While hand-crafted toy examples give a basic insight into the application behavior, they are limited to small systems and tend to be cumbersome. Computer aided simulations can help to quickly grasp the functioning of a given protocol also in large settings, offering the possibility to verify the desired behavior under a variety of operational conditions.

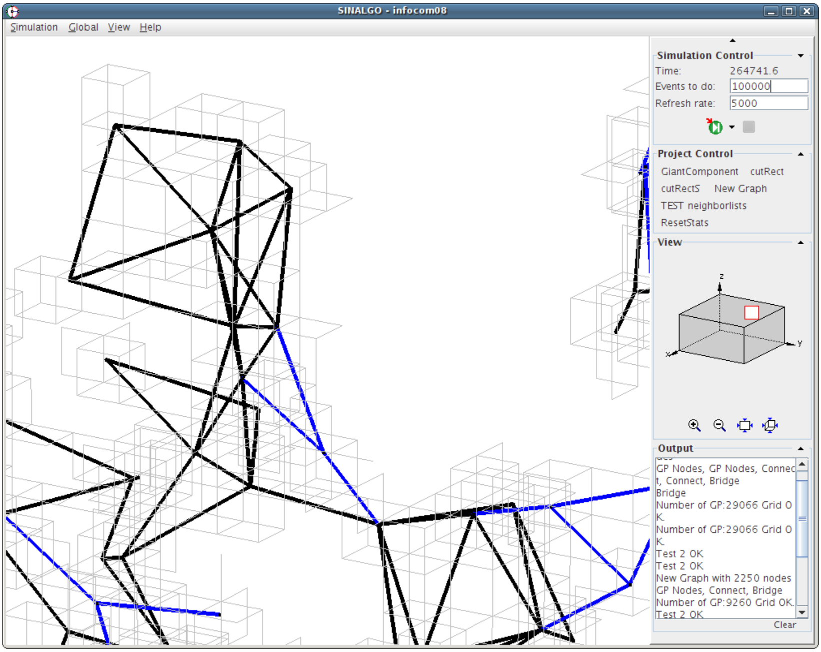
In this thesis, we consider wireless sensor networks which can be seen as large distributed systems: the sensor nodes collaborate to exchange messages and perform the assigned surveillance tasks. For the design and validation of protocols for such sensor networks, we have developed sinalgo, a simulator for network algorithms that not only supports the development process, but is also suitable to run large scale simulations. We have used a preliminary version of sinalgo for the simulation of the `MLS` algorithm described in Chapter 2.9. The simulations of the random walks on the dual graph in Chapter 3.7 and the greedy embedding in Chapter 4.4 have been performed with the final version available on sourceforge [39]. In all of our projects, the customizable visualization of sinalgo was helpful to understand and to perfect the algorithms before launching large scale batch simulations.

7.1 sinalgo

sinalgo offers a node-centric view of the network where each node may send a message to a given neighbor or to all its neighbors, react to received messages, and set timers to schedule an action in the future. Similarly to real sensor nodes, our software nodes are passive by default and act only upon receiving a message or when one of their timers fires. Therefore, the core task of sinalgo is to schedule message deliveries, fire timers, and advance the simulation time. For more complex simulation scenarios, additional simulation models can be hooked up as needed. For instance, the simulation abstracts from the hardware layer and provides no real MAC layer. To simulate message loss sinalgo provides several plug-ins to drop messages randomly or based on a interference models such as the SINR model. The connectivity models define when two nodes are in communication range, UDG or QUDG are the most prominent examples. Finally, mobility models describe how nodes change their position over time, influencing indirectly the connectivity and interferences models.

To allow for the greatest possible flexibility, each node holds its own instance of a mobility, connectivity, and interference model. Thus, sinalgo can simulate networks consisting of nodes with heterogeneous properties. In addition to the described models, each node also implements a radio reliability model which may drop messages ad libitum. This model can be useful to test an algorithm under the conditions of a lossy network without requiring the computational overhead of a complete interference model.

The implementation of a given network protocol in sinalgo consists of the following two steps: First, the properties of the network need to be defined by setting the appropriate models for the connectivity, mobility, and the interference. Furthermore, the desired amount of nodes must be deployed in the simulation area according to a given placement model. As sinalgo already ships with an implementation of the most common models, this initial step should not require much work. In the second step, the behavior of the nodes must be specified by implementing their reaction to received messages and fired timers. In addition, it must be defined how the progress of the algorithm is visualized and how the user can interact with an ongoing simulation. As for the visualization, nodes can be drawn with user defined shapes and colors and may also display text. Figure 7.1 shows a screen shot of sinalgo while developing the routing algorithms on 3D networks of Chapter 3. This visualization tool not only enabled us to understand in more detail the construction of the dual graph, but also facilitated the implementation and debugging of the simulation. To simplify user interaction at runtime, each node offers a context sensitive pop-up menu through which arbitrary actions on the node may be initiated. Furthermore, the application code can be modified even at run-time via hot-code replacement as sinalgo is written entirely in Java.



Time: 264741.6
Events to do: 100000
Refresh rate: 5000

```
GP Nodes, GP Nodes, Connec
t, Connect, Bridge
Bridge
Number of GP:29066 Grid O
K.
Number of GP:29066 Grid O
K.
Test 2 OK
Test 2 OK
New Graph with 2250 nodes
GP Nodes, Connect, Bridge
Number of GP:9260 Grid OK.
Test 2 OK
```

Figure 7.1: sinalgo in action showing the dual graph (light cubes) of a 3D network (bold lines). The customizable visualization can be used to display complex constructs while the GUI's rotation and zoom functions help to see the third dimension. The right panel contains the simulation control, custom buttons for additional operations and logging output; it also shows which part of the entire network is currently being displayed.

Algorithm 7.1: Synchronous simulation round

```
1 Increment Time                    The time advances by 1 unit
2 PreRound                          Simulation specific initializations for this round
3 Mobility                          Move each node according to its mobility model
4 Connectivity                      Recompute the connections
5 Interference                      Drop messages that experience too much interference
6 On each node:
7     Receive messages
8     Handle fired timers
9 PostRound                         Simulation specific cleanup code for this round
```

These two tools turned out to be convenient for testing special cases and different versions of an algorithm.

7.2 Simulation modes

The sinalgo simulator offers two simulation modes: a synchronous round based simulation and an asynchronous event driven simulation. In the latter, each event (a message reception or the firing of a timer) executes at a well defined time. sinalgo only needs to temporally order these events and execute them sequentially. Whereas an asynchronous simulation mode reflects well the asynchronous nature of wireless networks, it also breaks up the algorithm in its atomic pieces, making it harder to grasp the overall picture. For this reason, algorithms are often designed and analyzed in a round based model, where the algorithm advances in rounds. In this model, the time advances in discrete steps from round to round and the nodes perform their actions synchronously during these rounds. I.e. a message sent in round i arrives at the earliest in round $i + 1$. Algorithm 7.1 shows the actions sinalgo performs in each round: After initializing the round in Line 2, the mobility and connectivity models move and reconnect the nodes and the interference model drops messages if necessary (Lines 3–5). After this step, sinalgo iterates over all nodes to signal message receptions and the firing of timers (Lines 7,8). Please note that the order in which the nodes are processed is not important as messages sent within this round will arrive at the earliest in the following round. Finally, the round is concluded on Line 9 to perform cleanup and logging operations specific to the simulation.

7.3 Mobility

The simulation of mobile nodes is quite involved if continuous paths should be analyzed, especially in combination with interference and connectivity models which are based on the geometry of the network. Therefore, sinalgo offers only a simplified node mobility where the nodes move in discrete steps. In the synchronous simulation mode, the nodes are moved according to their mobility model at the beginning of each round (Line 3 of Algorithm 7.1) such that the following connectivity check can reestablish connections. If necessary, a finer resolution of the moved paths can be obtained by advancing the time in each round by a smaller amount (Line 1). Consequently, messages need more rounds until they are delivered and are therefore tested several times for interference and dropped connections. For most simulations of mobile networks, however, the default resolution is enough, as messages may also be dropped through the reliability model, simulating message loss due to arbitrary effects. In contrast to the synchronous mode, sinalgo does not support mobility in the asynchronous simulation mode. A discrete mobility could be implemented, but we did not want to break the purely event driven scheme.

The simulation of mobile networks is computationally involved as each node constantly needs to validate its connections and establish links to nodes that have moved into its communication range (Line 4). For general connectivity models, all node pairs need to be compared, costing $\mathcal{O}\big(n^2\big)$ operations, where n is the number of nodes. But as we mostly consider geometric networks where the connectivity depends on the distance between nodes, we can bring down this overhead to $\mathcal{O}(n \cdot d)$, where d is the expected node degree. To achieve this performance gain, sinalgo stores the nodes according to their position such that range queries can be answered efficiently. For 2D networks, the deployment area is subdivided into a quadratic grid with a cell side length of r_{max}, the maximum transmission range of the nodes. For each of these grid cells, sinalgo remembers which nodes are located within the cell. Therefore, all neighbors of a node located in a cell c are either in c or one of the 8 cells adjacent to c. Similarly, 3D networks are subdivided into cubic grids of side length r_{max} such that a set of potential neighbors can be found efficiently.

7.4 Discussion

sinalgo is a simulation tool for testing and validating wireless network protocols. It is written in Java and ships with a set of common operational modes such that the initial overhead to prototype a protocol is low. For a basic simulation, it is sufficient to implement only the behavior of the nodes before gradually fine tuning the simulation by customizing the simulation models, the run-time environment, or the visualization. To obtain an efficient framework, sinalgo operates on a message passing model that abstracts from the physical properties of hardware devices. As a result, sinalgo only provides limited support for the simulation of MAC layers. For simulations that should consider the entire networking stack, the ns-2 simulator for wireless networks [74] is a widely used option. In contrast to sinalgo, however, ns-2 focuses on the simulation and provides less support for the development process, e.g. there is only a trace-based visualization that displays the interactions after the simulation finished. The simulation tool most similar to sinalgo is Shawn [55] which also abstracts from the networking layer to focus on the protocol implementation. Shawn is written in C++ and outclasses sinalgo in terms of performance when simulating huge networks. On the contrary, sinalgo offers more visualization options and interaction with an ongoing simulation.

Independent of the simulation tool in use, we need to keep in mind that no simulation replaces a real experiment, especially in wireless networks where the operational conditions often change over time. In addition, it is often hard to predict how good simulation results match reality, especially if the simulation was based on simplistic models. For instance, sinalgo offers the convenient round based simulation mode where nodes advance synchronously to perform their actions in parallel. This viewpoint helps to describe the algorithms in a simpler form hiding the asynchronous nature of distributed hardware. However, it raises the question on how such a synchronous protocol can be implemented on hardware after a successful simulation. In fact, the transition from a simulation prototype to the hardware implementation is challenging as any adaptation of the code must be verified to not break the overall behavior. The TOSSIM emulator [62] for TinyOS applications avoids this problem by using the same code base for the simulation as well as for the device. To generate the simulation, TOSSIM requires a special compilation procedure which redirects hardware calls to the simulation core of TOSSIM which then simulates the corresponding interrupts to keep the system running. Whereas this tool is great to emulate code prior to deployment, it requires the protocols to be written in nesC and the application must already be tailored to a specific hardware platform. But both, the C language and the platform dependence are highly undesirable for prototyping and testing new protocols, rendering high level simulation tools such as sinalgo attractive.

To facilitate the transition between a simulation and the implementation for the hardware, we have studied how to port synchronous protocols to asynchronous hardware such as our sensor nodes. In fact, the synchronous approach not only facilitates simulations, but would also be handy for the device software as the different steps of the protocols are automatically separated and synchronized, giving the execution an implicit structure. We present first steps towards this goal in the remaining chapters of this thesis. In particular, we propose a novel programming paradigm that challenges the traditional asynchronous applications with a synchronized execution environment where the nodes perform their operations in a series of synchronous time slots. Even though our system does not work exactly as the synchronous simulation mode described in Algorithm 7.1, we believe it to be powerful enough to ease the implementation of protocols that were initially designed in a synchronous context.

8

Slotted Programming

After the design and simulation of novel protocols for wireless networks, they need to be implemented on the hardware devices. This final step often requires a major redesign of the program code used for the simulation, as hardware specific constraints need to be taken into account. In this chapter, we focus on the software development for wireless sensor networks and present a novel programming style to facilitate this rather involved task.

In general, software for distributed systems tends to be "*heavy*", consisting of several components and layers that interact with each other in a non-trivial way to cope with various transient or permanent failures and most likely also some kind of dynamics. The icing on the cake is the whole process of software development, e.g. issues such as debugging between remote nodes. But programming embedded systems is no picnic either. Embedded systems ask for "*light*" software that does not waste resources such as memory, processing power, or energy. In addition, the systems often need to meet tight run-time requirements and guarantee a predictable execution. Again the software development cycle is tedious, as the programmer does not have direct access to the hardware, but must go through cross-compilation and cross-debugging.

Wireless sensor networks clearly inherit all the problems from networked distributed systems as well as embedded systems. To make matters worse, running "heavy" distributed systems software on "light" embedded systems hardware sounds like an insurmountable contradiction at first. However, sensor networks can be done. While early deployments have suffered in terms of bad reliability, high energy consumption, or bad scalability, more recent developments overcome these issues with novel algorithms such as energy-efficient MAC protocols [51, 90, 100] which can be tuned to suit a given application best.

The work to build such energy efficient protocols and applications shall not be underestimated. Current energy efficient implementations need to be optimized across the entire application. This comes at the price of a non-modular design. Adding further functionality may require drastic changes in code, and even small adaptations to parameters or changes of the environment may trigger a serious investment in re-testing.

Not surprisingly, there is a trend towards simplifying and standardizing the programming environment. Some proposals abandon the idea of energy efficient hardware altogether, and instead advocate a Linux or Java VM framework [15, 49, 85]. Some other proposals [51] envision an IP layer hiding a general purpose low-power networking stack. Clearly, all these efforts are fine for rapid prototyping. However, as we will argue later in this chapter, these abstractions are not suitable for real-world implementations. For instance, the energy efficiency of many applications could be improved by an order of magnitude when writing special purpose communication protocols. In addition, the abstractions hide the complexity of the underlying implementation. This is dangerous especially for time sensitive tasks, where hardware devices such as the CPU or the radio must be available exactly at a given time.

In this chapter we outline the slotted programming paradigm which supports both modularity and energy-efficiency. This programming approach is not orthogonal to the current trend of providing high level abstractions. It rather extends the abstractions with a predictable execution scheme, giving back full control to the application developer. Our concept is simple: We consider the different tasks of a sensor node such as clock synchronization, routing, topology control, sensing, or code updates. Each of these tasks is given time slots during which it performs its operations in parallel on all nodes. The time slots need to be assigned in a non-overlapping way such that at any time, at most one task executes its code. If a job has not terminated at the end of its assigned time slot, it must be suspended and may only continue its execution during its next time slot. Thus, tasks do not interfere with each other, and they do not interrupt each other. Therefore, each task can be implemented as an independent *module* which can be exchanged easily. Indeed, alternative modules for the same task can be tested and compared within the same application. Moreover, different modules can communicate with incompatible packet formats, e.g. IPv6 packets vs. AM-packets, as they will not disturb each other. In fact, we will demonstrate in Chapter 10 that it is even possible to use the radio in radically different ways, e.g. packets vs "waves". All in all, we believe that our approach fosters modular programming where modules can be exchanged easily. In addition, we show that also modular applications allow for highly optimized and energy efficient code. For instance, our clock synchronization described in Chapter 9 turns on the radio only for 0.06% of the time even though it receives and sends one message every 32 seconds.

8.1 Related Work

The application development for sensor networks combines techniques from embedded systems, networking, and also real time systems. To support this delicate task, dedicated operating systems were built already in the early days. The most prominent OS is TinyOS [92], which was a purely event driven framework supporting many hardware platforms. Over time, TinyOS was extended with new features to ease application development and reduce programming errors. Most recently, a safe type-system [23] and support for multi-threaded programming [71] were added.

However, the quest for an optimal programming environment was not only led by TinyOS. For instance, the MANTIS OS [1] supported preemptive multi-threading well before TinyOS did, and Contiki [28] is based on lightweight protothreads [29]. The notion of virtual memory was introduced in the t-kernel [46] to protect the operating system from being corrupted by the user application, and SOS [48] supports the dynamic loading of modules, avoiding the static linking at compile time. All operating systems above are based on the low level C programming language or a dialect thereof, but there have been efforts to support Java [15, 49, 85], requiring the nodes to run a virtual machine to execute the byte-code.

Besides these low-level OS developments, explicit support for energy aware programming was proposed. Whereas Nano-RK [35] is a real-time operating system with static resource allocation, the pixie OS [67] allows for a dynamic resource allocation at runtime, predicting the energy draw of the operations in an online fashion. As the radio is among the primary energy consumers, much work has also been put into building energy efficient communication protocols. In asynchronous low power listening schemes such as WiseMAC [32] or B-MAC [79], each node periodically samples the radio for packet announcements. Whenever a node wishes to send a message, it first sends a rather long preamble to announce the forthcoming packet. X-MAC [17], SCP-MAC [100], RI-MAC [90] and [51] outline several approaches to reduce this overhead. TP-MAC [44] is probably the most related work to our signaling described in Chapter 10. The authors of [44] propose a pipelined wake-up schedule but do not consider interference. We extend this idea and provide an interference resilient signaling which we also tested on real hardware. To the best of our knowledge, this work is the first to describe and implement a fully functional wake-up signaling that is resilient to interference.

To reduce the learning curve and simplify the development of sensor network applications, TinyOS and other operating systems support multi-threaded programs. Furthermore, low power listening protocols can be used to provide a basic MAC layer to the application developer. In fact, Hui and Culler [51] envision a IPv6 stack on top of such a MAC layer to hide the underlying networking to the application developer. We embrace this trend towards a standardized communication framework and believe it to be perfect for prototyping sensor network applications. However, when it comes to the development of real-world applications, we would like to express our reservation to this approach for several reasons. Firstly, the execution of a multi-threaded application is hard to predict. As a result, the development and also debugging is much more involved as some errors may occur only sporadically under a specific preemption. Similarly, the introduction of a networking abstraction such as a low power listening MAC introduces a rather unpredictable use of the CPU and the radio device, possibly delaying other tasks. Thus, the developer does not have complete control over the application, which makes it hard to ensure a bug-free program. Last but not least, the resulting applications are by far not as energy efficient as they could be. For instance, the task of collecting a message from every node once a minute causes a duty-cycle of 0.65% with the IPv6 network stack of [51]. In Dozer [18], a data gathering protocol with a dedicated MAC layer, a message was collected every 30 seconds. The duty-cycle for this task was reported to be only 0.167%, reducing the overhead by a factor of 8. However, this improvement comes at the price of a higher latency and a completely asynchronous application that is hard to maintain. For comparison, our clock synchronization algorithm described in Chapter 9 receives and sends one message every 32 seconds and requires an average duty-cycle of only 0.06%. Of course, these numbers cannot be compared directly, but they indicate that there is still a lot of potential for saving energy. With the slotted programming paradigm, we demonstrate that it is indeed possible to write modular applications that are also energy efficient.

8.2 Background

Access to the different hardware components such as the radio module, timers, flash chip, or the sensors is often done through software arbitration or the explicit knowledge that no other code fragment is using the desired component. Clearly, the latter approach is not only prone to undetected conflicts, but also renders code reuse a nightmare. But software arbitration alone is not enough to write modular applications as we outline in the following points.

Real time applications The load of the CPU is hard to predict, as other pieces of the application may require computational complex tasks at any time. As a result, tasks and synchronous events may execute with a non-negligible delay, which poses problems to the implementation of time critical applications where some actions should happen exactly at the prescribed time. This is in sharp contrast to classic real time systems, where the tasks should execute within a larger time window before a given deadline. For instance, even the simple task of enabling the radio on two neighboring nodes at exactly the same time is hard to achieve, as any of the nodes may be occupied with other tasks and therefore delay the power-on command.

A common approach to suppress this problem is to introduce a guard-time, which specifies how much before the arranged time the radio should be turned on. Apart from the fact that it is hard to predict good guard-times, applications use more energy than necessary and also gain in complexity.

Alternatively, the radios could be powered on in an asynchronous code block, ensuring instantaneous execution if no other asynchronous event interferes. However, implementations which rely on preemptive code execution (e.g.[18]) are quite hard to maintain, as any modification or extension may affect any other part of the system. As such asynchronous systems are often very complex and even the slightest anomaly in the code can lead to misbehavior such as memory corruption or even a deadlock, we believe that this approach should be avoided whenever possible.

Multi-threaded operating systems are not improving the situation, as they even explicitly permit preemption of tasks, delaying operations even further. Prioritization of the threads does not resolve the problem, as the thread with the lower priority still experiences a delay.

Modularity In most sensor network scenarios, each network node needs to fulfill several tasks, e.g. clock synchronization, sense the environment, process the measurements, and disseminate the sensed data. To obtain a highly optimized application that uses as little energy as possible, the usual approach is to combine the required tasks as well as possible. In our example, this could mean that the messages required for the clock synchronization are piggybacked onto the data gathering messages or vice versa. Whereas such an implementation can be very energy efficient, it is also highly specific to the given problem instance. Therefore, any modification or extension needs to take into account the entire application, and partial code reuse in a different application is cumbersome.

Incomplete algorithm design We have already argued that any two tasks running in parallel may interfere by reducing the responsiveness of the node. Similarly, the software arbitration may temporarily block the access to a hardware resource such as the radio module while another task is using it. Also, if two hardware components happen to be connected to the CPU through a shared bus, at most one of them may be accessed at any time. This is the desired behavior, but it is often difficult or even impossible to predict such conflict patterns. Therefore, algorithms for sensor nodes are normally designed under the simplifying assumption of immediate access to the hardware, ignoring the fact that another task may run in parallel competing for the same resources. Adapting these algorithms to the existing hardware con-

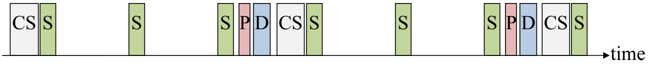


Figure 8.1: A slot assignment on a sensor node with 4 tasks: Periodically sample a sensor [S], process the sensor data [P], disseminate the processed data [D], and run a clock synchronization algorithm [CS].

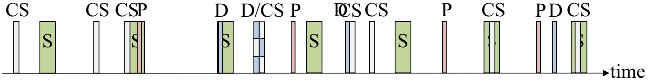


Figure 8.2: In an uncoordinated execution model where tasks are not temporarily decoupled, several collision patterns need to be taken into account.

strains is often a challenge, especially for time critical applications. Even if a delay may be tolerable, the unpredictable timing may introduce new side effects which must be verified to not break the original algorithm.

Configuration conflicts A wrongly configured hardware device not only provokes unpredictable responses but may even hang a sensor node. If several tasks require conflicting configurations, the application must be careful to always load the appropriate configuration. If a task interrupts the execution of another task due to asynchronous execution or multi-threading, this may not be possible at all.

8.3 Slotted Programming

The above mentioned issues arise because several tasks of a sensor node may execute simultaneously. With the slotted programming paradigm, we introduce a temporal arbitration between the tasks to resolve the described problems, and at the same time, slotted programming fosters code modularity. In the following, we introduce the slotted programming paradigm and show how the above issues are addressed.

In a nutshell, the slotted programming approach decouples the different tasks of a sensor node to render them as independent as possible such that each task can be implemented as a self-contained software module. The decoupling is achieved through temporal separation of the different tasks, assigning each module a time slot during which it may have exclusive access to all resources. Figure 8.1 shows a possible slot assignment on a sensor node

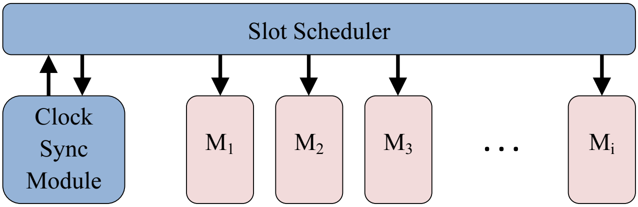


Figure 8.3: Schematic view of a slotted system: The slot scheduler starts and stops the software modules, the clock synchronization module is responsible for obtaining the network time.

with 4 distinct tasks. This is in sharp contrast to an uncoordinated execution model, where tasks my collide and experience unpredictable side effects as illustrated in Figure 8.2.

8.3.1 The Basics

The slotted programming paradigm builds on a synchronous execution model. It requires that all nodes have a global notion of time, i.e. they need to run some kind of clock synchronization algorithm. This network time is used to schedule the execution of the software modules, such that the same software module executes simultaneously on all nodes.

Each task of a sensor node is implemented as an independent software module providing the desired functionality. E.g. there may be a clock synchronization module, a data gathering module, a sampling module, and a data processing module. To obtain the temporal decoupling with other software modules, each software module must ensure that its code executes only within an assigned time slot and that it causes no side effects outside this slot.

The software modules are then integrated into the slotted system by allocating time slots for each of them. Whereas arbitrary complex schedules can be built, simple periodic schedules similar to the one shown in Figure 8.1 are already sufficient for most applications. The only restriction for the overall schedule is that there may be no region where slots overlap, otherwise the temporal decoupling would be broken. Depending on the scheduling complexity, this property can already be tested at compile time. Alternatively, a run-time check may be applied.

The basic support for slotted programs is provided by two components: A clock synchronization module and a scheduler module. The latter executes the desired schedule by signaling each module the start and end of its time slots. Figure 8.3 illustrates a schematic view of a slotted system. Interesting to note is the tight coupling of the clock synchronization module and the slot scheduler: Whereas the slot scheduler depends on the network time to schedule the modules, the clock synchronization module itself is scheduled by the slot scheduler. This circular dependency is no problem as soon as the node is roughly synchronized, but special care needs to be taken while a node is not synchronized. See Chapter 8.4.2 for more details on this topic.

8.3.2 Guidelines for Slotted Programming

Given an operating system that provides the basic functionality for slotted programming (clock synchronization and a slot scheduler), the development of a sensor node application boils down to the following three steps:

(A) *Modularization of the application.* In this preliminary software design step, we identify the tasks which can be implemented as self-contained software modules. This first step intentionally leaves open a broad range of design decisions such as to not restrict the programming paradigm to a narrow field of applications. The focus of this step, however, should be on the reusability of the individual modules and on the temporal decoupling of the implemented tasks.

(B) *Implementation of each individual module.* First, define the time slots during which the given module is allowed to operate. Whenever possible, this choice should be parametrized such that an overall schedule can be obtained easily when assembling several modules (see below). The implementation of the module must ensure that its code only runs between the startSlot() and stopSlot() events generated by the slot scheduler.

(C) *Determine an overall schedule.* The integration of the desired modules requires a non-overlapping scheduling of all time slots. Ideally, each module parametrizes the execution time of its time slots, such that the global schedule can be obtained by tuning these parameters. When the modules adapt their schedule dynamically at runtime, this task may quickly become quite complex. But as periodic schedules are sufficient for most applications, a valid schedule can often be found quickly.

8.3.3 Discussion

With the slotted programming paradigm, we foster modular programs for sensor nodes. The key component is the temporal decoupling of modules which guarantees that at any time, at most one module is running its code without being preempted. This decoupling lays the basis for the following properties of slotted programming:

- During the time slot of a module, full access to all hardware resources is granted to the module, it is neither blocked nor delayed by other tasks. This also includes the hardware configuration which is guaranteed to be consistent during the entire slot. Overall, the programmer of the module can count on the timely availability of the hardware, which allows for simpler and more efficient implementations using less energy.

- Each module can be implemented and tested independently as there are no side effects from other tasks. This greatly supports the software development, as it is much easier to code and test several small pieces instead of writing and testing an entire application altogether.

- The reuse of a module in a different applications is straightforward as each module is supposed to work independently of the context. The only thing that may need to be adjusted is the scheduling of the time slots.

8.3.4 Limitations

Whereas the above properties are desirable for all applications, the slotted programming paradigm has its limits. For instance, if a sensor node must constantly perform some action, e.g. sample a sensor at 100Hz, other modules cannot be scheduled without conflicting with the sensing task. In fact, there is no clean solution to this fundamental problem as a complete decoupling of the modules is impossible. We believe, however, that in the case of such a scenario, the slotted programming is still useful. On the one hand, the application developer may be more aware of the existing conflicts, which helps to predict possible side effects. On the other hand, the conflicting module can be designed to reduce its actions during foreign time slots to a minimum, shifting the non-time-critical operations to its assigned time slots.

8.4 The slotos Reference Implementation

To demonstrate the potential of slotted programming in practice, we developed slotos, an operating system that supports slotted programming. In general, any operating system can be adapted to provide this functionality. For our reference implementation, we chose to extend TinyOS 2.1, an event driven operating system widely used for sensor networks. It runs on many hardware platforms and is continuously improved and extended with new features. Being single-threaded in its default version, TinyOS is a perfect candidate for applying the slotted programming paradigm.

In slotos, the *slot scheduler* is responsible to invoke the execution of the software modules according to their time slot reservations. To simplify the scheduling of periodic time slots, slotos provides a periodic slot management

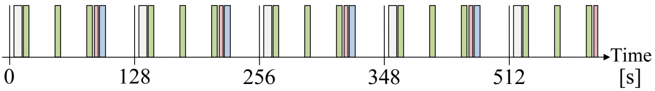


Figure 8.4: The slot scheduler repeats a partial schedule generating a periodic execution of the allocated time slots.

Interface 8.1: Slot scheduler interface

events

```
  void init()
  void startSlot(slotID, slotCmd, syncQuality, syncStatus)
  void stopSlot(slotID)
```

commands

```
  slotID addTimeSlot(startTime, length, slotCmd)
  void modifyTimeSlot(slotID, startTime, length, slotCmd)
  bool testSchedule()

  void stopScheduling()
  void continueScheduling()
  ⋮
```

---

with an allocation window of 128 seconds. Each module allocates its time slots within this allocation window which is repeated periodically by the slot scheduler to obtain the overall schedule, see Figure 8.4 for an example. With this scheduling approach, slotos facilitates the allocation of periodic time slots which we believe to be sufficient for most applications. The size of the allocation window is currently set to 128 seconds, but it may be set to whatever fits best the application needs. For more sophisticated schedules, the dynamic time slot reservation of slotos may be used to reorganize the schedule at run-time.

Interface 8.1 sketches the interface of the slot scheduler. During the `init()` event, which is called upon booting the node, each module allocates its time slots. `addTimeSlot()` is used to reserve an additional time slot, and `modifyTimeSlot()` reschedules an existing time slot. Whenever a time slot starts, the `startSlot()` event is signaled on the corresponding module indicating an estimate of the current synchronization quality. Finally, the end of the slot is signaled by the `stopSlot()` event.

8.4.1 Policy Enforcement

The temporal decoupling of the modules is the key of slotted programming. Whereas a static schedule can be checked at compile-time to have no overlapping regions, dynamically generated schedules can only be verified at run-time. slotos implicitly detects overlapping time slots and refrains from scheduling them simultaneously. Instead, it provides a best-effort service delaying the start of the later time slot until the end of the active time slot. It is important to note that this error handling approach should not be exploited as a feature of the slot scheduler. It is rather a last resort to guarantee a continuous execution of all reserved time slots, based on the assumption that the broken schedule is only of temporal nature.

In addition to a non-overlapping schedule, the temporal decoupling also requires that the software modules operate only during their assigned time slots. This property, however, is much harder to verify or even enforce, as the modules should be allowed to execute arbitrary actions within their time slots. For instance, a module may set a timer to fire outside its time slot or initiate a split phase command whose callback returns only after the time slot terminates. Whereas the timer issue can be addressed by cancelling timers that are set outside the current time slot, callbacks from the hardware are device specific and often hard to predict. The current implementation of slotos does not try to detect or avoid activity outside the assigned time slots and requires the module developer to adhere strictly to the time constraints. Thus, slotos does not enforce slotted programming but only provides a suitable execution environment.

8.4.2 Clock Synchronization

Clock synchronization is an integral part of slotos; it ensures that the software modules are scheduled simultaneously on all nodes. Its task is to maintain the clocks synchronized with neighboring nodes and react to topology changes which may leave the node unsynchronized for an arbitrary amount of time.

As clock synchronization is implemented as a self-contained module, it needs to be scheduled by the slot scheduler, but the slot scheduler itself relies on the synchronization. slotos supports two approaches to break this circular dependency.

(A) The clock synchronization module may temporarily turn off the slot scheduler. During this time, no other module is scheduled to execute and the synchronization module is free to access the radio for an extended time to receive synchronization messages. Once an approximate synchronization is available, the slot scheduler can be turned on again.

(B) It is often undesirable to completely shut down the slot scheduler as this also stops any other activity on the sensor node, including the

Interface 8.2: Clock synchronization interface

event

void timeChanged()

commands

time hardwareToNetworkTime(hwTime)
time networkToHardwareTime(netTime)
time networkIntervalToHardwareInterval(netDT)

quality getSynchronizationQuality()

sensing tasks for which it was deployed. While the measured data may be useless if the node is not synchronized with its neighbors in some scenarios, many applications rely on a continuous stream of measurements, making it impossible to turn off the slot scheduler. But if the clock synchronization module can only access the radio during its assigned time slots, it may never receive synchronization messages if the neighboring nodes have a different notion of time. To accommodate such scenarios, slotos explicitly permits the clock synchronization module to break the temporal modularity and listen on the radio also outside its assigned time slot. But this violation of the slotted programming paradigm needs to be taken into account by all modules that are scheduled during this time. In particular, other modules should refrain from using the radio or turning it off. For that purpose, the `startSlot()` command informs the modules about the current state of the synchronization. Fortunately, this restriction does not really limit the functionality of the node any further, as communication with neighboring nodes is likely to fail anyways while it is not synchronized.

The choice between the two approaches depends on whether the sampling or the synchronization is more mission critical. Clearly, the second approach is not as clean as we would like, but it is as modular as possible for the given application requirements.

In either case, the clock synchronization module must provide the interface shown in Interface 8.2. Whenever the network time is modified, `timeChanged()` needs to be called such that the slot scheduler can adapt the schedule. Additionally, several methods to convert between hardware and network time and a method to query the current synchronization quality should be provided.

8.4.3 Timers

It is often the case that two neighboring nodes wish to wake up simultaneously to exchange messages or perform other operations. Slotted programs support such interactions as the software modules are scheduled simultaneously on all nodes. Thus, it is sufficient for a module to set a timer to be waken up at the arranged time. To obtain a truly decoupled system, the modules should respect the following guidelines when setting timers:

- Most importantly, timers may only be set to fire within the current time slot. If a timer should fire during a later time slot, the module should remember this and start the timer only at the beginning of the desired time slot. Thanks to this restriction, timers never need to be updated to reflect a modified network time because no timer is active when the clock synchronization module resets the network time.

- Whenever possible, timers should be started relative to the start time of the current time slot. This results in a more precise timing than when using offset timers, as their fire time depends on the time when they are set. In TinyOS, this can be done using the `startOneShotAt()` method of the Timer interface.

- Interactions with other nodes should be arranged based on the common network time. The timers in slotos, however, operate on the local time of the node, requiring a conversion from the network time to the corresponding local time. The methods offered by the synchronization module can be used for this conversion, see Interface 8.2.

- To account for fluctuations of the clock synchronization, a module may apply guard times to ensure that it does not miss an arranged meeting. In the case of a scheduled message transmission, the receiver wakes up a bit earlier and the sender sends the message a bit later to ensure that the two meet. The estimation of the clock synchronization passed on in the `startSlot()` event may be used to adaptively set the guard times.

In the following we demonstrate the advantages of the slotted programming approach at two applications. We start by describing the design and implementation of a slotted time synchronization module in Chapter 9. In Chapter 10, we describe a novel signaling technique which can be used to propagate alarms efficiently and reliably through a network. While the signaling technique itself is of interest, the simplicity of its integration into our test application demonstrates the power of slotos.

9

Slotted Clock Synchronization

As clock synchronization is a central part of slotos and slotted programming in general, we now describe in more detail the clock synchronization module that comes with slotos. Being implemented itself as a software module, we use this synchronization module to demonstrate the advantages of the slotted programming paradigm.

Due to its simplicity, we provide a variant of the FTSP algorithm [70] where an elected root node dictates its time to the remaining nodes. In contrast to the original FTSP algorithm, we apply a simplified version thereof where a dedicated root node dictates its time to the remaining nodes. Nodes in the immediate neighborhood of the root node learn the network time directly from the root node. Once a node is synchronized, it disseminates synchronization messages itself such that nodes not directly connected to the root synchronize indirectly. Each node selects a single parent node to which it synchronizes, resulting in a tree algorithm. Furthermore, our synchronization algorithm relies on a static root node. Whereas our implementation is not really fine-tuned and experiences an average synchronization error of 44μs per hop, this is already sufficient for many applications. In fact, as our hardware timers are based on a rather slow 32.768 kHz oscillator, it is hard to obtain a synchronization error below one time unit (30μs). Overall, we focused on obtaining an energy efficient solution rather than a perfectly synchronized system.

slotos offers a time window of 128 seconds to schedule the execution of the software modules. Our clock synchronization module reserves 4 time slots of 1 second in regular intervals such that the module is launched every 32 seconds, see Figure 9.1. Within these assigned time slots, the module may perform any actions as the slotted programming guarantees that no other task interferes. In our case, the module only needs to receive a sync message from its parent and to broadcast a sync message for its children.

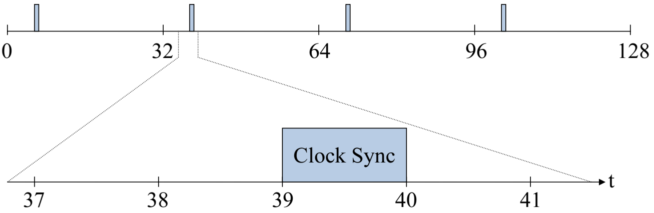


Figure 9.1: The default clock synchronization module of slotos allocates 4 time slots of 1 second. The time slots are spread over the allocation window of the slot scheduler such that the module is invoked every 32 seconds. The remaining time slots can be used arbitrarily by other software modules.

9.1 Synchronized Transmission

In order to avoid collisions, each node chooses its transmission time of the sync message randomly within the assigned time slot. In each transmitted message, the sender includes the seed value of the random number generator that will be used to send forthcoming sync messages. From this information, the receiving child can predict when its parent sends the next sync message and enable its radio just for the required time period. The child can even predict the arrival of a sync message when it has missed several messages in between. This is possible as we use a circular random number generator where the generated number is used as the seed for the following draw.

Synchronization fluctuations render it impossible for the receiver to wake up just in time to receive the message. We apply an adaptive guard time which is decreased for each successful reception and increased otherwise. This leads to a close to optimal message reception overhead, ensuring low energy consumption.

9.2 Pipelined Synchronization

When sending the sync message at a random time within the assigned slot, a child may forward its sync message before receiving the sync message from its parent. As a result, the sync message sent by the child may be inaccurate as it is based on a synchronization which is at least 32 seconds old. To overcome this issue, our implementation pipelines the transmission along the synchronization tree. The synchronization module achieves this by dividing its time slot into k cells of equal length (see Figure 9.2) and restricting the transmission of the sync message to one specific cell. This cell assignment is based on the number of hops the node is away from the root node (on the synchronization tree). A node which is h hops away from the root node

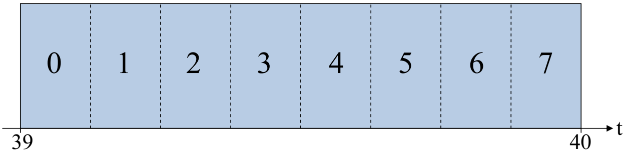


Figure 9.2: The synchronization module divides its time slot into k logical cells (in our case, $k = 8$). The root node broadcasts its sync message in the first cell with ID 0. The immediate children of the root receive this message and transmit their sync message within the second cell with ID 1. In general, a node that is h hops away from the root receives a sync message from its parent in the cell $(h - 1) \bmod k$ and sends its sync message in the cell $(h \bmod k)$.

chooses to send its sync message in the cell $(h \bmod k)$. Note that nodes with a distance of k or more hops to the root reuse cells already assigned to nodes much closer to the root.

The choice of k is driven mainly by two constraints. On the one hand, k should be chosen as small as possible such that the sync messages can be spred over a longer cell, reducing the probability of collisions. On the other hand, if a node is $i \cdot k$ hops away from the root (with $i \in \mathbb{N}^+$), the node sends its sync message in the cell with ID 0, but receives the sync message from its parent only in the last cell with ID $k - 1$, not achieving the desired pipelining. Thus, the larger k is, the fewer nodes break the pipelining. In the current implementation of slotos , we have set $k = 8$ to ensure pipelining for our sample networks.

9.3 Initialization

When a node is (at least roughly) synchronized to its parent node, the above techniques are applied to keep the node synchronized. But after booting or when the node has lost its synchronization parent, a special initialization step is required to find a suitable parent. As the node has no a priori knowledge about the network time, the techniques from Chapter 8.4.2 are required to establish a first meeting point and obtain an initial synchronization. In our implementation of slotos, we chose to keep the slot scheduler running while the synchronization module listens for sync messages also outside its assigned time slots.

9.4 Experiments

We have logged the performance of our clock synchronization algorithm while testing our alarming system described in Chapter 10. During the 168 hours of the experiment, a total of 18900 synchronization rounds were performed.

To measure the quality, a node determines its clock offset to its parent whenever it receives a sync message. Throughout the experiment, we measured an average offset of 1.45 time units with a variance of 1.21. This is equivalent to an average offset of 44μs as our hardware clock runs at 32768Hz.

The energy consumption of the clock synchronization is dominated by the energy used to send and receive messages. To get a first approximation on the energy usage, each node logged how long the radio module was enabled for each synchronization round. The cumulated up-time of the nodes is plotted in Figure 9.3. Note that the root node has a much smaller slope than the remaining nodes as it does not receive sync messages. It is also interesting to note that the lines of the non-root nodes diverge the longer the experiment runs. The varying slopes are due to our synchronization algorithm which tries to keep the guard time for receiving the sync messages as short as possible. Nodes that tend to loose packets adapt a larger guard time, slightly increasing their up-time. For instance node 12 (see also Figure 10.4) has the steepest slope. During the experiment, this node changed its parent several times, sometimes even synchronizing to node 7 across the building. The peaks in Figure 9.4 indicate when node 12 was looking for a new parent.

On average, the non-root nodes required an overall duty cycle of 0.06% for the synchronization, node 12 has the highest duty cycle of 0.07%. The temporal progress of the duty cycle is shown in Figure 9.3. The curves show the average (maximum) duty-cycle for the preceding 200 synchronization rounds. Again, the peaks fall together with parent elections.

9.5 Discussion

We sketched the implementation of a clock synchronization algorithm with the slotted programming approach, leaving out quite some details. For instance, the choice of a reliable parent is intrinsically difficult as the quality of a parent may change over time. Our goal was not to provide a synchronization that matches or even exceeds the best existing algorithms, but to demonstrate that it is indeed possible to write a modular clock synchronization that can be replaced by any other synchronization module without consequences for the remaining application. Furthermore, we would like to point out that even a simple implementation as the one described above can be quite energy efficient.

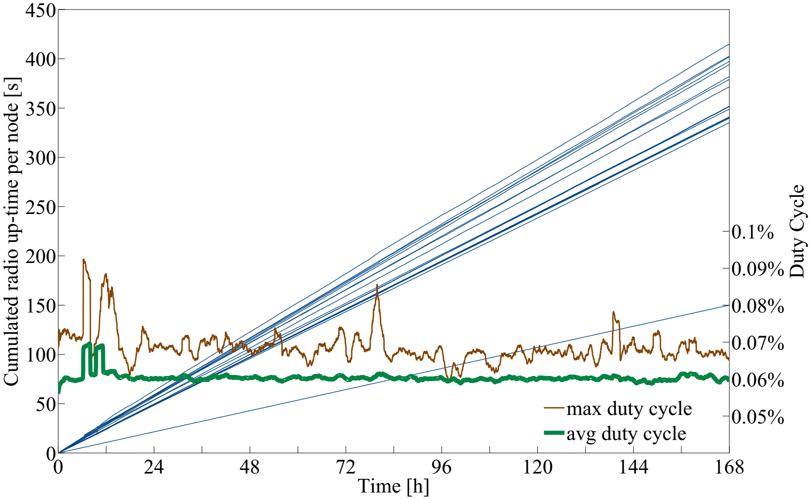


Figure 9.3: The nearly straight lines show the cumulated radio up-time of the 19 nodes using the left scale. The single line at half the rate belongs to the root node which does not need to listen for sync messages. The two horizontal curves indicate the temporal change of the duty cycle using the right scale.

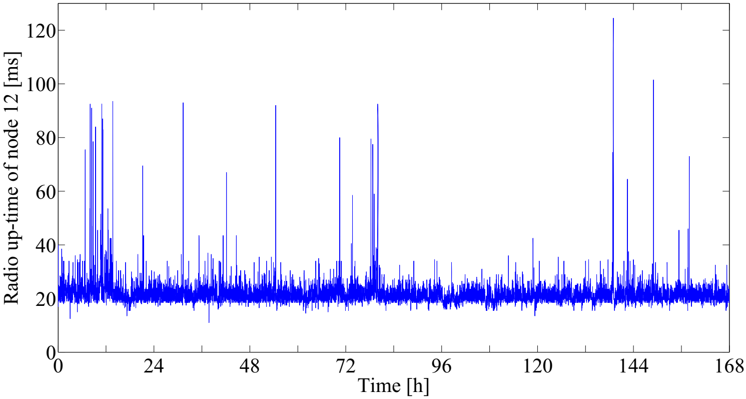


Figure 9.4: Node 12 has the highest duty cycle as it repeatedly loses its synchronization parent. This plot shows the up-time of node 12 for each synchronization round. The peaks occur when node 12 is looking for a new parent.

10

Low-Power Signaling

As a case study for slotted programming, this chapter describes the construction of an alarm system in which any node or any subset of nodes can alarm all other nodes. The reliability of such a system is often guaranteed through the following two phases: In the first phase, the nodes are awaken through an energy efficient wake-up mechanism. Once the nodes are awake, the verification phase kicks in to rule out false alarms, authenticate the originator(s) of the alarm, and perform other security relevant tasks before relaying the alarm to the upper layer. Whereas this second phase may be costly in terms of energy, this is no issue for most scenarios where alarms can be assumed to occur rarely. On the contrary, the wake-up functionality should be as power efficient as possible because it must be constantly available. In addition, the wake-up should provoke as few false alarms as possible in order not to trigger the second phase unnecessarily.

In this chapter, we only consider the wake-up phase and present a novel technique to reliably transmit a wake-up signal. Our idea is the following: Instead of transmitting ordinary messages to announce an alarm, a node sends an unmodulated wave at a constant frequency. With this approach, we solve two problems at one fell swoop. Firstly, several nodes may announce an alarm at the same time without causing interference. This holds because the superposition of several waves of the same frequency is received as a wave of the given frequency. Secondly, we can build tight schedules for the signal propagation. Previously, this was possible only with RSSI sniffs which tend to give many false alarms when used outdoors. Both, the interference resistance and the tight scheduling lay the basis for our energy efficient wake-up phase. We propose a pipelined dissemination of the alarm signal such that the dissemination delay is minimized. For each execution of the pipeline, the nodes need to power the radio module for a total of 14ms in the current implementation. Thus, the overall energy consumption of the wake-up phase depends on the frequency at which these pipelines are scheduled. E.g. having a dissemination every 30 seconds causes a radio duty cycle of only 0.04%, but also delays the delivery of a potential alarm by up to 30 seconds. Clearly, this parameter is application specific.

A special configuration of the radio module is required to generate the desired waves. As the radio also needs to support standard messages, the application must ensure the required radio configuration at any time. This guarantee comes nearly for free with a slotted application where the tasks, and therefore also the conflicting configurations, are temporally decoupled.

10.1 Pipelining

The end-to-end delay of such an alarm can be minimized by pipelining the transmissions of the nodes. But as any node of the network may initiate an alarm, the pipelining would need to provide an any-to-all support. To overcome this complex task, we reduce the problem to the following two pipelinings: any-to-root and root-to-all, where the root is a dedicated node, e.g. a sink node or the node leading the clock synchronization. This results in a 2-phase approach, where the initiating node first signals the root node about the alarm (any-to-root) and the root then signals the alarm to the entire network (root-to-all).

The classic approach to both pipelinings is to build a BFS tree from the root node. When sending messages to transmit the alarm, however, care has to be taken to not cause interference. In the root-to-all phase, the root node first broadcasts its message to all its direct children in the tree. Upon receiving the message, the children should be coordinated to not forward the message at the same instant. The same problem exists in the any-to-root phase, where several children of a parent may wish to signal an alarm, or an alarm is propagated on several branches of the BFS tree. This issue can be solved by either using a global schedule or by applying randomization techniques. But both solutions are not satisfactory as they increase the end-to-end delay. Furthermore, the overhead to maintain a global schedule may be considerable.

10.2 Signaling of Binary States

As we are interested only in binary states (e.g. alarm ON, alarm OFF), a much simpler solution is possible. For instance, the value ON may be encoded by sending a message and the value OFF by being quiet. In the any-to-root case, each parent only needs to detect whether any of its children sends a message, and depending on the outcome, send a message itself or not. To do so, the parent only needs to detect the activity on the radio channel, e.g. through an RSSI sniff, eliminating the interference problem. Therefore,

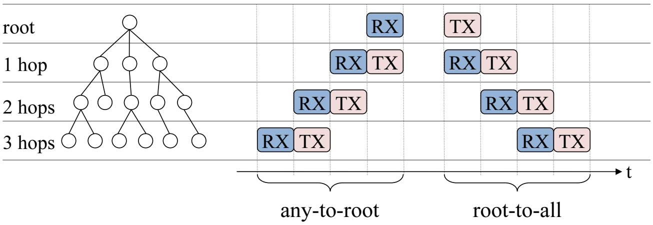


Figure 10.1: Alarms are disseminated in two steps. In the any-to-root step, any subset of nodes can signal the root node. The root in turn then relays the alarm to everybody in the following root-to-all step.

several children may send their message simultaneously without causing any reception problems. The same holds for the root-to-all case where all children of a parent may send their message simultaneously. As a result, we can apply a tight pipelining which schedules the reception and transmission solely based on the hop count to the root node as depicted in Figure 10.1. Such a tight pipelining minimizes the end-to-end delay as well as the energy consumption, as any node only needs to receive and send twice, independent of the number of its children.

In the above example, the proposed encoding requires that the ON state strictly dominates the OFF state. I.e. whenever at least one node signals the dominant ON state, the ON state should be disseminated to the entire network. Only when no node of the entire network requires the dominant state, the subordinate OFF state is applied. This implicit conflict resolution between the two states is a natural choice for many application scenarios such as alarms (the alarm state is dominant), or any other form of wake-up signaling where a single node should be able to signal the root (using only the any-to-root pipeline) or all nodes of the network. Indeed, the proposed signaling approach may be useful in other context than alarms. For instance, it may be used to enable or disable entire software modules, e.g. a debugging or configuration module that needs not to run most of the time.

In the remainder of this section, we demonstrate the advantages of the slotted programming paradigm to implement such a pipelining for alarms. In addition, we propose a more robust alternative to the RSSI sniffing as RSSI measurements tend to give many false positives when used outdoors.

10.3 RSSI vs Waves

Many MAC protocols sample the radio channel prior to sending a message. If activity is detected, the transmission is delayed to avoid a collision with the ongoing transmission. This sampling is called clear channel assessment (CCA) and is often performed by an RSSI module on the radio chip which indicates the received signal strength. If the value is above a given threshold, the channel is assumed to be occupied. We propose to use exactly this tool to circumvent interference problems while signaling an alarm. Indeed, the approach works surprisingly well - as long as all nodes are deployed indoors. But as soon as we placed nodes on the roof of our building, the RSSI module started to intercept foreign signals, leading to roughly 30% false positives, i.e. the RSSI module indicates activity on the channel even though none of the nodes is sending. Whereas this degrades the performance but may still be acceptable for some MAC protocols, the state signaling task requires a far more reliable solution as false alarms or unnecessary wake-ups can be costly. Ideally, an alternative approach still allows for a tight pipelining of the transmissions without causing interference.

In the presented RSSI approach, the transmitters send arbitrary messages as the receiver only looks for activity on the radio channel. To reduce the number of false positives during silent phases, we propose to slightly change the scheme and require all transmitters to send the exactly same message which needs to be decoded by the receiver. Consequently, the probability to decode the expected message even though no node is transmitting drops exponentially. Whereas this method resolves the problem of false positives, it is prone to interference, as all transmitters would need to be highly synchronized to send each bit of the message at exactly the same time. Therefore, we carefully choose the message such that the resulting radio signal is an unmodulated wave of constant frequency. This solves the interference problem at one fell swoop as the mixture of several waves of the same frequency is received as a wave signal of the given frequency. But whereas MIMO and SIMO radios make explicit use of this property to amplify the radio signal in a certain location, we have no influence on the phase at which the distributed transmitters send. As a result, some receivers may get an attenuated signal. Fortunately, the probability for a complete attenuation is extremely low, and our experiments indicate that the number of false negatives is in the range of 0.1% and comparable to the RSSI approach (See Table 10.1). In summary, the proposed approach encodes the dominant state by sending an unmodulated signal of constant frequency, and the subordinate state is encoded by being quiet.

The desired radio wave can be generated easily with our XE1205 radio module as it applies an FSK modulation. In this modulation technique, a binary '0' is encoded by sending a wave at frequency f_0 and a binary '1' is encoded by sending a wave at frequency f_1. For sending and receiving standard messages, the radio runs in a buffered mode utilizing a bit synchronizer and pattern recognition block. When generating or sampling waves, we switch to a continuous mode through which we can send raw data or directly access

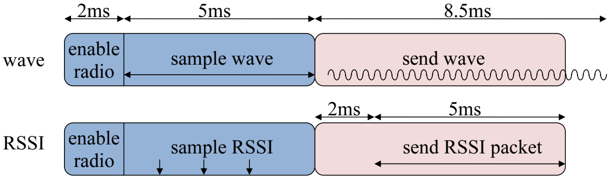


Figure 10.2: When a node is scheduled within the pipeline, it enables its radio, samples the RSSI or the wave, and depending on the outcome, transmits itself an RSSI packet or a wave, respectively.

the received signal. Please refer to Chapter 10.5.1 for a description of the used hardware and more details on its usage. Also note that the generation of these waves may be harder to obtain on packet-based radios or on radios that apply an encoding such as Manchester or NRZI.

10.4 Slotted Signaling

In the remainder of this section we outline the implementation of a test application to compare the RSSI and the wave approach for the signaling task. Similar to the first case study, our focus is on the software architecture, rather than the implementation details.

Signaling Module The task of the signaling module is to run an any-to-root sweep to detect requests to change a state, followed by a root-to-all sweep to disseminate a potential request to the entire network. Both sweeps are implemented in a pipelined fashion as described in Chapter 10.1. As the timings for the RSSI as well as the wave approach are similar, we can use the same pipelining for the comparison, see Figure 10.2. The time to enable the radio is approximately 2ms and the time to send an RSSI message is 4.5ms. The RSSI value is sampled three times in a row, indicating a signal only if all three measurements show high activity on the radio channel. We hoped to reduce the number of false positives with this approach, but the experiments did not show a significant improvement compared to a single RSSI sniff, indicating a high correlation of the RSSI measurements. The sampling time for the wave is set to 5ms yielding 378 samples, and the transmission time for a wave is set to a total of 8.5ms to account for synchronization fluctuations.

The signaling module allocates itself a time slot of 1 second. The first 500ms is reserved for the any-to-root pipeline and the second 500ms for the root-to-all pipeline. As the overlap of the sender and receiver is only 7ms in each pipeline step, the depth of the pipelines may be up to 60 hops (leaving some time gap at the beginning and the end of the time slot). Thus, the current settings theoretically allow for large networks with a diameter up to 120 hops if the root node is chosen in the center.

Whenever `startSlot()` is called, the signaling module retrieves the node's hop-count to the root from the clock synchronization module and determines its start-time in the any-to-root and the root-to-all pipeline. An adaptation of this hop-count is however necessary, as the links used by the synchronization tree may be unidirectional. This adaptation is implemented by a simple online search that is launched whenever the root-to-all sweep does not reflect a request sent on the any-to-root sweep.

Energy Cost of Signaling The energy cost of the signaling is dominated by the energy used by the radio module for sampling and sending. In many application scenarios, a signal will only be disseminated rarely on special events. During the silent periods, a node needs to enable its radio once for the any-to-root and once for the root-to-all pipeline, requiring an up-time of 14ms. In the event of a signaling, a node may enable its radio up to 31ms to participate in the two pipelines. The overall energy cost for the signaling depends on the frequency at which the signaling module is scheduled and how often signals are disseminated.

10.5 Test Application

The integration of the signaling module into an application, in our case a test application to compare the RSSI and the wave signaling, is straight forward. All we need to do is to assign time slots to each module such as to obtain a non-overlapping schedule. In our test application, we utilize the following software modules:

- Clock synchronization module
- RSSI signaling module
- Wave signaling module
- Remote control module
- Feedback composition module

The remote control module is used for collecting the logs of an ongoing experiment and for debugging the application. The module supports multi-hop message forwarding to deliver log messages at a base station such that we can track the progress of an ongoing experiment. In addition, it can also route command messages from the base station to any given node in the network,

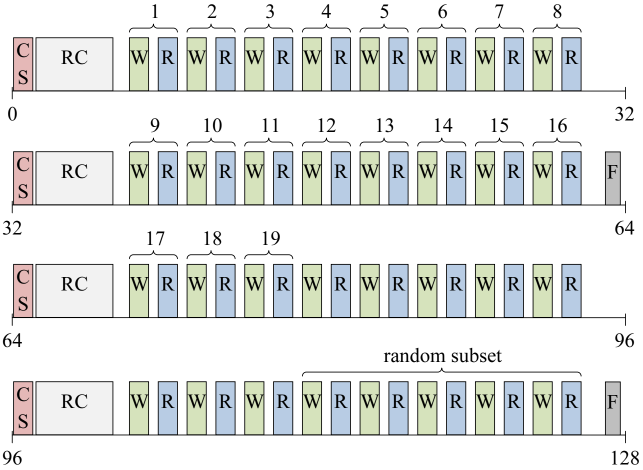


Figure 10.3: The test application to compare the wave signaling and the RSSI signaling allocates a total of 74 time slots within the scheduling window of 128 seconds. The following abbreviations are used for the modules: CS - Clock Synchronization, RC - Remote Control, F - Feedback composition, W - wave signaling, and R - RSSI signaling. The numbers above the time slots indicate the ID of the node which triggers an alarm.

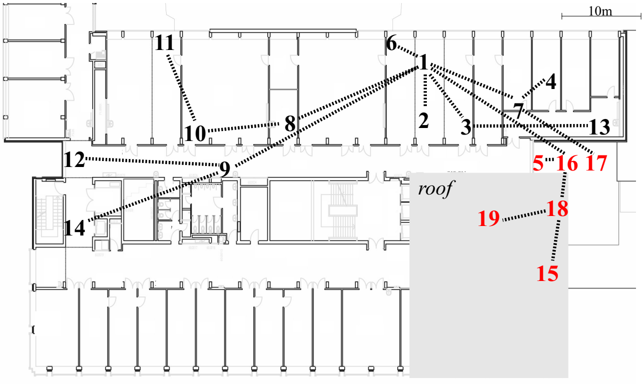


Figure 10.4: The nodes 5, 15, 16, 17, 18, and 19 are placed outside the building and are powered by battery. Whereas the nodes 15, 18, and 19 are on the roof, the nodes 5, 16, and 17 are placed on the facade along the four floors to the roof to establish a radio connection with the nodes on the roof. Node 1 is configured to be the root for the clock synchronization and the signaling. The dashed lines show an instance of the dynamic clock synchronization tree.

which was convenient to debug the application in its initial phase. The feedback composition module collects and preprocesses the local data from the experiment and feeds the remote control module with the log messages. Both of these modules support our experiments, but they do not influence the synchronization and signaling tasks in any way as they are temporally decoupled. Therefore, they could be removed from the application without consequences for the other modules.

The test application uses an allocation window of 128 seconds within which it schedules its modules, see Figure 10.3. The clock synchronization module is scheduled every 32 seconds, followed by the remote control module that gathers the log messages. Every 64 seconds, the feedback composition module prepares the log messages which are sent in the forthcoming remote control time slot. The RSSI signaling module and the wave signaling module are assigned a total of 64 time slots within the remaining allocation window to test as many signaling rounds as possible. The two approaches are tested in an alternating fashion such that they undergo similar external influences.

10.5.1 Deployment and Experiment Setup

We performed our experiments with the TinyNode 584 from Shockfish SA [87]. This sensor node features an MSP430 CPU with 10kB RAM and 48kB program space and works with the Xemics XE1205 radio which applies an FSK modulation. For receiving data packets, the radio is configured in a buffered mode with a built-in bit synchronizer and a pattern detector. As the bit synchronizer requires at least some bit transitions every now and then to operate correctly, we cannot use the buffered mode for detecting alarms, as they consist of a wave of a given frequency. We therefore use the continuous mode of the radio which skips the bit synchronization and pattern recognition and outputs the raw bit sequence on a data pin. Our application samples this stream during 5ms to obtain the described sampling. Whereas we could transmit waves in the buffered mode by sending a message with every bit set to 1 (or 0), this requires quite some modification on the radio stack. In continuous mode, the transmission of a wave is straight forward, as the radio sends the pattern that is applied to a given pin. Keeping the pin at 1 (or 0) results in the desired wave.

For our experiment, we have deployed a total of 19 nodes, 6 of them are outside the building, see Figure 10.4. Node 1 acts as base station and as root for the clock synchronization. The goal of the experiment is to test the reliability of the proposed wave signaling and compare it to the RSSI approach. In particular, we are interested in the following two errors: A “false negative” signaling happens when a node fails to detect a signal sent by a neighboring node. This type of error is especially undesirable in alarm systems where the system should immediately react to a new situation. A “false positive” signaling occurs when a node detects a signal even though no node is transmitting. This fault is often called a “false alarm”.

To test for these errors, the following scenario is run through in each window of the slot scheduler, Figure 10.3 depicts the setup.

- Exactly one node triggers an alarm in the first 38 time slots. Each node is assigned a time slot pair (consisting of a time slot for the wave signaling and one for the RSSI signaling) during which it triggers an alarm. This is done by sending a wave or an message, respectively, at the assigned time within the any-to-root pipeline.
- None of the nodes triggers an alarm during the following 16 time slots.
- A randomly chosen subset of the nodes triggers an alarm in the remaining 5 time slot pairs. This last test case is used to explicitly examine the decoding resistance when several nodes signal synchronously.

For each of these time slots, we decide whether the signaling was successful or not. As the signaling module described in Chapter 10.4 performs an any-

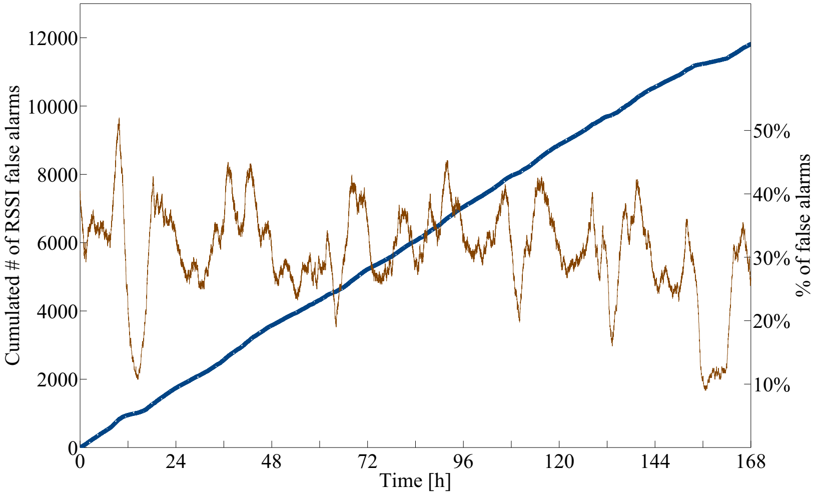


Figure 10.5: The solid line shows the cumulated number of false positives when using RSSI sniffs (left scale). The bumpy curve depicts the temporal variability of the error probability (using the right scale).

to-root sweep followed by a root-to-all sweep, all nodes are supposed to have the same state at the end of the signaling time slot. Therefore, if one or several nodes fail to detect the signal in the root-to-all sweep, we count the time slot as false negative. Similarly, if one or several nodes detect a signal in one of the 16 silent time slots, we count the time slot as false positive. Please note that we test the overall signaling procedure rather than each individual signal decoding as the latter is prone to report subsequent errors.

10.5.2 Results

We have run the above experiment for 168 hours. During this time, the schedule of Figure 10.3 was repeated 4725 times, generating 113400 rounds in which at least one node triggered an alarm and 37800 silent rounds without alarm. Table 10.1 summarizes the overall outcome of the experiment. On average, the RSSI had false alarms in 31% of the silent rounds, compared to only 0.8% of the wave approach. The probability for a dropped alarm (false negative) is comparable for both the wave and RSSI approach.

Figure 10.5 depicts the cumulated number of RSSI false alarms, and the horizontal curve shows how the error probability changes over time. Whereas RSSI suffers from false alarms in 31% on average, the curve shows that the error probability is sometimes as low as 10% for an extended period of time.

Table 10.1: Performance of the wave and the RSSI signaling.

	Wave signaling	RSSI signaling	Number of tests
False positives	304 (0.8%)	11815 (31%)	37800
False negatives	94 (0.08%)	132 (0.11%)	113400

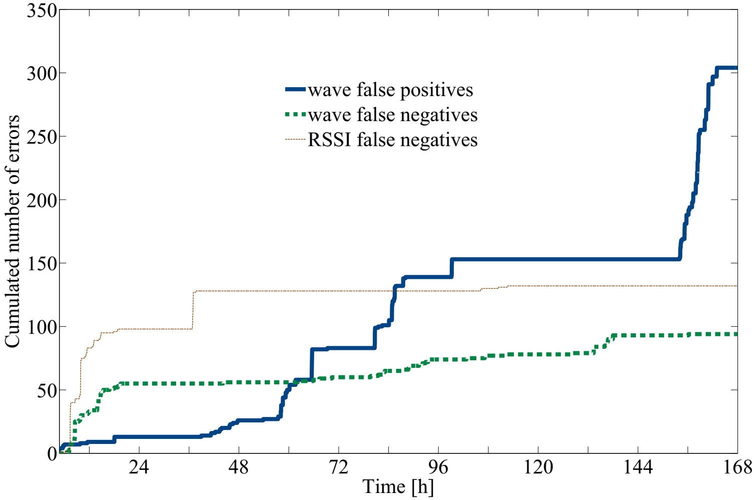


Figure 10.6: Cumulated number of errors without the RSSI false positives. Please note that errors tend to happen in bursts.

This fact can also be observed on the cumulative curve, where the slope flattens temporarily during the early afternoon of the first and last day of the experiment. (Please note that all plots start at midnight.) We belive that this improvement could be related to the weather, as only the first and last day of the experiment was sunny.

In contrast to the RSSI approach, the wave signaling has much less false alarms. Figure 10.6 shows the temporal developing of the false positives and false negatives of the wave signaling as well as the false negatives of the RSSI approach. Please note that most errors come in bursts, indicating a temporal correlation between successive samplings of the same kind. So far, we have no explanation for these errors.

When decoding a wave transmission, a node samples the radio channel during 5ms resulting in 378 samplings. As our radio module applies an

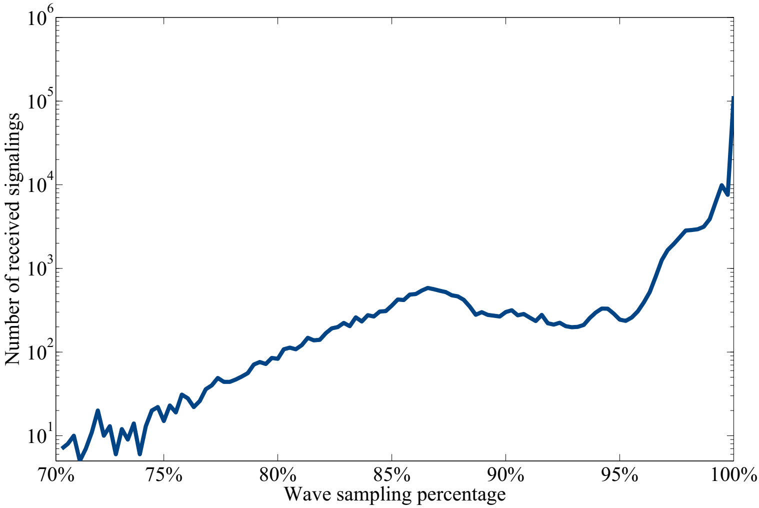


Figure 10.7: For each decoded wave, the nodes remembered the percentage of correct bits. This plot shows that a large majority of the alarms has all or nearly all bits correctly, and that the probability for errors decreases exponentially.

FSK decoding, each sample indicates which of the two frequencies is more likely. If no wave is being transmitted, the radio module decodes white noise indicating either frequency with a probability of 50%. However, if a neighboring node transmits a wave at frequency f_x, the radio module reports all samples to be at frequency f_x. The node counts the number of samples with the expected frequency and indicates an alarm only if this number is above a given threshold. In our implementation, at least 70% of the samples must indicate the expected frequency.

Figure 10.7 shows the number of received alarms as a function of the decoding percentage. For over 63% of the decoded alarms, all samples indicated the expected frequency. In 99.9% of the detected alarms, at least 75% of the samples indicated the expected frequency. Thus, setting 70% as threshold seems reasonable.

11

Conclusion

In the second part of this thesis, we proposed a novel signaling technique to build energy efficient alarm systems. Our wave based approach is resilient to interference, allows for a tight pipeline scheduling without unnecessary hop-by-hop delay, is energy efficient at a duty cycle of less than 0.1%, and causes few false alarms even when used outdoors. We believe that our signaling scheme is a valuable alternative to traditional message based approaches which need to handle interference. For instance, low power listening schemes assume that messages are transmitted only sporadically such that message collisions are rare. In an alarming scenario, however, a possibly large subset of nodes may detect an abnormal state and trigger an alarm at the same instant. While message-based approaches are burdened with the overhead to avoid interference, the proposed wave signaling is specifically designed to support simultaneous alarms, allowing for a better energy efficiency.

The implementation of such an alarm system is rather involved. Firstly, the transmission and reception of waves requires a special configuration of the radio module which does not support data messages. Thus, the application must ensure a suitable configuration at all times. Secondly, the tight pipelining for the alarm dissemination requires each node to enable its radio exactly at the assigned time. If this operation is delayed by another task running on the node, a potential alarm may not be detected. With the slotted programming approach, both issues are solved with the temporal separation of the modules. In addition, the modularity of the software components allowed us to setup a rather complex application to compare the RSSI and the wave signaling with little effort.

Overall, we feel that the slotted programming approach greatly supported our work and we believe the slotted programming paradigm to be a valuable tool for developing sensor network applications in general. But the advantages are not only in the facilitated software development. Just as important are the modularity and temporal separation of the different tasks which allow for a predictable execution of the application. This predictability is also a basic requirement to build systems for which certain properties can be guaranteed. In fact, the development of provably correct software is very hard in general. The core difficulty lies in the fact that it is hard to compare the application (written in a programming language) and the specification (written in a descriptive language). The slotted programming paradigm cannot solve this problem, but it shortens the gap between the two worlds by reducing the complexity of the software implementation, rendering the comparison easier. Most importantly, the slotted programming achieves this by temporally decoupling independent tasks such that the different components can be checked separately. In addition, the temporal decoupling renders unnecessary many asynchronous code blocks as timely execution is ensured implicitly by the slotted execution model. For instance, the application to evaluate the alarming system of Chapter 10 requires no asynchronous code block except for the system events. Such synchronous code is much easier to analyze as there are much fewer execution patterns to be considered and the simplified code itself reduces the risk of bugs.

Wireless sensor networks will gain more attention in the near future as engineers in various fields become aware of the novel technology and its capabilities. But as wireless ad hoc networking is still in its infancy it needs to go a long path to unfold its full potential. Towards this goal, the technology first needs to gain acceptance by proving reliability and practicability. Therefore, we not only need to build appropriate hardware, but also provide suitable tools to efficiently develop the corresponding software. We believe that it is important to facilitate the software development process for wireless sensor networks such that programming is not only easier, but also more bug-resistant. Within this context, the slotted programming paradigm is only a small piece of a huge puzzle through which we hope to contribute to a more efficient development of energy efficient software for sensor networks.

Bibliography

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support for Multimodal Networks of In-Situ Sensors. In *WSNA*, 2003.

[2] I. Abraham, D. Dolev, and D. Malkhi. LLS: a Locality Aware Location Service for Mobile Ad Hoc Networks. In *DIAL-M-POMC*, 2004.

[3] I. Abraham, C. Gavoille, A. Goldberg, and D. Malkhi. Routing in Networks with Low Doubling Dimension. In *ICDCS*, 2006.

[4] J. Aspnes, D. K. Goldenberg, and Y. R. Yang. On the Computational Complexity of Sensor Network Localization. In *ALGOSENSORS*, 2004.

[5] C. Avin and G. Ercal. On the Cover Time and Mixing Time of Random Geometric Graphs. *Theoretical Computer Science*, 380(1-2):2–22, 2007.

[6] C. Avin and B. Krishnamachari. The Power of Choice in Random Walks: An Empirical Study. In *MSWiM*, 2006.

[7] B. Awerbuch and D. Peleg. Sparse Partitions (Extended Abstract). In *FOCS*, 1990.

[8] B. Awerbuch and D. Peleg. Online Tracking of Mobile Users. *Journal of the ACM*, 42(5):1021–1058, 1995.

[9] G. Barnes and U. Feige. Short Random Walks on Graphs. In *STOC*, 1993.

[10] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward. A Distance Routing Effect Algorithm for Mobility (DREAM). In *MobiCom*, 1998.

[11] L. Blazevic, J.-Y. Le Boudec, and S. Giordano. A Location-Based Routing Method for Mobile Ad Hoc Networks. *IEEE Trans. Mob. Comput.*, 4(2):97–110, 2005.

[12] P. Bose and P. Morin. An Improved Algorithm for Subdivision Traversal without Extra Storage. In *ISAAC*, 2000.

[13] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing With Guaranteed Delivery in Ad Hoc Wireless Networks. In *DIAL-M*, 1999.

[14] H. Breu and D. G. Kirkpatrick. Unit Disk Graph Recognition is NP-hard. *Computational Geometry, Theory and Applications*, 9(1-2):3–24, 1998.

[15] N. Brouwers, P. Corke, and K. Langendoen. A Java Compatible Virtual Machine for Wireless Sensor Nodes. In *SenSys*, 2008.

[16] J. Bruck, J. Gao, and A. A. Jiang. Localization and Routing in Sensor Networks by Local Angle Information. In *MobiHoc*, 2005.

[17] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks. In *SenSys*, 2006.

[18] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In *IPSN*, April 2007.

[19] H. T.-H. Chan, A. Gupta, B. M. Maggs, and S. Zhou. On Hierarchical Routing in Doubling Metrics. In *SODA*, 2005.

[20] A. K. Chandra, P. Raghavan, W. L. Ruzzo, and R. Smolensky. The Electrical Resistance of a Graph Captures its Commute and Cover Times. In *STOC*, 1989.

[21] J. Chen, A. Jiang, I. A. Kanj, G. Xia, and F. Zhang. Separability and Topology Control of Quasi Unit Disk Graphs. In *INFOCOM*, 2007.

[22] M. B. Chen, C. Gotsman, and C. Wormser. Distributed Computation of Virtual Coordinates. In *SCG*, 2007.

[23] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient Memory Safety for TinyOS. In *SenSys*, 2007.

[24] S. M. Das, H. Pucha, and Y. C. Hu. Performance Comparison of Scalable Location Services for Geographic Ad Hoc Routing. In *INFOCOM*, 2005.

[25] S. Datta, I. Stojmenovic, and J. Wu. Internal Node and Shortcut Based Routing with Guaranteed Delivery in Wireless Networks. *ICDCSW*, 5(2):169–178, 2002.

[26] F. de Rango, M. Gerla, B. Zhou, and S. Marano. Geo-LANMAR Routing: Asymptotic Analysis of a Scalable Routing Scheme with Group Motion Support. In *BROADNETS*, 2005.

[27] S. Dobrev, J. Jansson, K. Sadakane, and W.-K. Sung. Finding Short Right-Hand-on-the-Wall Walks in Graphs. In *SIROCCO*, 2005.

[28] A. Dunkels, B. Grönvall, and T. Voigt. Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Emnets*, 2004.

[29] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *SenSys*, 2006.

[30] S. Durocher, D. Kirkpatrick, and L. Narayanan. On Routing with Guaranteed Delivery in Three-Dimensional Ad Hoc Wireless Networks. In *ICDCN*, 2008.

[31] M. Dyer, J. Beutel, and L. Thiele. S-XTC: A Signal-Strength Based Topology Control Algorithm for Seonsor Networks. In *ISADS*, 2007.

[32] A. El-Hoiydi and J.-D. Decotignie. WiseMAC: An Ultra Low Power MAC Protocol for the Downlink of Infrastructure Wireless Sensor Networks. In *Computers and Communications*, 2004.

[33] D. Eppstein and M. T. Goodrich. Succinct Greedy Graph Drawing in the Hyperbolic Plane. In *GD*, 2008.

[34] J. Eriksson, M. Faloutsos, and S. Krishnamurthy. Scalable Ad Hoc Routing: The Case for Dynamic Addressing. In *INFOCOM*, 2004.

[35] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *RTSS*, 2005.

[36] Q. Fang, J. Gao, and L. J. Guibas. Locating and Bypassing Routing Holes in Sensor Networks. In *INFOCOM*, 2004.

[37] Q. Fang, J. Gao, L. J. Guibas, V. Silva, and L. Zhang. GLIDER: Gradient Landmark-Based Distributed Routing for Sensor Networks. In *INFOCOM*, 2005.

[38] U. Feige. A Tight Upper Bound on the Cover Time for Random Walks on Graphs. *Random Struct. Algorithms*, 6(1):51–54, 1995.

[39] R. Flury. Sinalgo - Simulator for Network Algorithms. `http://sinalgo.sourceforge.net`, May 2009.

[40] R. Flury, S. Pemmaraju, and R. Wattenhofer. Greedy Routing with Bounded Stretch. In *INFOCOM*, 2009.

[41] R. Fonseca, R. Fonseca, S. Ratnasamy, S. Ratnasamy, D. Culler, D. Culler, S. Shenker, S. Shenker, I. Stoica, and I. Stoica. Beacon Vector Routing: Scalable Point-To-Point Routing in Wireless Sensornets. In *NSDI*, 2005.

[42] C. Gavoille and M. Gengler. Space-Efficiency for Routing Schemes of Stretch Factor Three. *Journal of Parallel and Distributed Computing*, 61(5):679–687, 2001.

[43] C. Gotsman and Y. Koren. Distributed Graph Layout for Sensor Networks. In *GD*, 2004.

[44] A. Grilo, M. Macedo, and M. Nunes. *An Energy-Efficient Low-Latency Multi-sink MAC Protocol for Alarm-Driven Wireless Sensor Networks*. LNCS, 2007.

[45] M. Grossglauser and M. Vetterli. Locating Mobile Nodes With EASE: Learning Efficient Routes From Encounter Histories Alone. *IEEE/ACM Transactions on Networking*, 14(3):457–469, 2006.

[46] L. Gu and J. A. Stankovic. t-kernel: Providing Reliable OS Support to Wireless Sensor Networks. In *SenSys*, 2006.

[47] A. Gupta, A. Kumar, and R. Rastogi. Traveling with a Pez Dispenser (or, Routing Issues in MPLS). *SIAM Journal on Computing*, 34(2):453–474, 2004.

[48] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *MobiSys*, 2005.

[49] T. Harbaum. NanoVM. `http://www.harbaum.org/till/nanovm`, March 2009.

[50] T. He, C. Huang, B. M. Blum, J. A. Stankovic, and T. Abdelzaher. Range-Free Localization Schemes for Large Scale Sensor Networks. In *MobiCom*, 2003.

[51] J. W. Hui and D. E. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *SenSys*, 2008.

[52] R. Kleinberg. Geographic Routing Using Hyperbolic Space. In *INFOCOM*, 2007.

[53] G. Konjevod, A. W. Richa, and D. Xia. Optimal-Stretch Name-Independent Compact Routing in Doubling Metrics. In *PODC*, 2006.

[54] L. Kou, G. Markowsky, and L. Berman. A Fast Algorithm for Steiner Trees. *Acta Informatica*, 15:141–145, 1981.

[55] A. Kroeller, D. Pfisterer, C. Buschmann, S. P. Fekete, and S. Fischer. Shawn: A New Approach to Simulating Wireless Sensor Networks. In *DASD*, 2005.

[56] A. Kröller, S. P. Fekete, D. Pfisterer, and S. Fischer. Deterministic Boundary Recognition and Topology Extraction for Large Sensor Networks. In *SODA*, 2006.

[57] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer. Fast Deterministic Distributed Maximal Independent Set Computation on Growth-Bounded Graphs. In *DISC*, September 2005.

[58] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What Cannot Be Computed Locally! In *PODC*, July 2004.

[59] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric Ad-Hoc Routing: of Theory and Practice. In *PODC*, 2003.

[60] F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically Optimal Geometric Mobile Ad Hoc Routing. In *DIAL-M*, 2002.

[61] J.-Y. Le Boudec and M. Vojnovic. Perfect Simulations and Stationarity of a Class of Mobility Models. In *INFOCOM*, 2005.

[62] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *SenSys*, 2003.

[63] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A Scalable Location Service for Geographic Ad Hoc Routing. In *MobiCom*, 2000.

[64] M. Li, W.-C. Lee, and A. Sivasubramaniam. Efficient Peer-to-Peer Information Sharing over Mobile Ad Hoc Networks. In *MobEA*, 2004.

[65] N. Linial, E. London, and Y. Rabinovich. The Geometry of Graphs and Some of its Algorithmic Applications. *Combinatorica*, 15(2):215–245, 1995.

[66] R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Computing*, 36(2):177–189, 1979.

[67] K. Lorincz, B. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource Aware Programming in the Pixie OS. In *SenSys*, 2008.

[68] L. Lovász. Random Walks on Graphs: A Survey. *Combinatorics*, 2:353–398, 1996.

[69] M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.

[70] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The Flooding Time Synchronization Protocol. In *SenSys*, 2004.

[71] W. P. McCartney and N. Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *SenSys*, 2006.

[72] A. Moitra and T. Leighton. Some Results on Greedy Embeddings in Metric Spaces. In *FOCS*, 2008.

[73] D. Moore, J. Leonard, D. Rus, and S. Teller. Robust Distributed Network Localization With Noisy Range Measurements. In *SenSys*, 2004.

[74] ns-2. `http://nsnam.isi.edu/nsnam/index.php`, May 2009.

[75] C. H. Papadimitriou and D. Ratajczak. On a Conjecture Related to Geometric Routing. *Theoretical Computer Science*, 344(1):3–14, 2005.

[76] D. Peleg and E. Upfal. A Trade-Off between Space and Efficiency for Routing Tables. *J. ACM*, 36(3):510–530, 1989.

[77] S. V. Pemmaraju and I. A. Pirwani. Good Quality Virtual Realization of Unit Ball Graphs. In *ESA*, 2007.

[78] M. Penrose. The Longest Edge of the Random Minimal Spanning Tree. *The Annals of Applied Probability*, 7(2):340–361, 1997.

[79] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *SenSys*, 2004.

[80] A. Rao, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic Routing Without Location Information. In *MobiCom*, 2003.

[81] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: a Geographic Hash Table for Data-Centric Storage. In *WSNA*, 2002.

[82] A. L. Rosenberg and L. S. Heath. *Graph Separators, with Applications.* Springer: Frontiers in Computer Science, 2000.

[83] P. Santi. *Topology Control in Wireless Ad Hoc and Sensor Networks.* ACM Computing Surveys, 2005.

[84] S. Schmid and R. Wattenhofer. Algorithmic Models for Sensor Networks. In *WPDRTS*, April 2006.

[85] Sentilla. Sentilla Perk. `http://sentilla.com/perk`, May 2009.

[86] Y. Shang and W. Ruml. Improved MDS-based localization. In *INFOCOM*, 2004.

[87] Shockfish SA. TinyNode. `http://www.tinynode.com`, November 2008.

[88] A. Slivkins. Distance Estimation and Object Location via Rings of Neighbors. In *PODC*, 2005.

[89] A. Smith, H. Balakrishnan, M. Goraczko, and N. B. Priyantha. Tracking Moving Devices with the Cricket Location System. In *MobiSys*, 2004.

[90] Y. Sun, O. Gurewitz, and D. B. Johnson. RI-MAC: a Receiver-Initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks. In *SenSys*, 2008.

[91] K. Talwar. Bypassing the Embedding: Algorithms for Low Dimensional Metrics. In *STOC*, 2004.

[92] TinyOS Alliance. TinyOS. `http://www.tinyos.net`, May 2009.

[93] A. C. Viana, M. D. de Amorim, S. Fdida, Y. Viniotis, and J. F. de Rezende. Easily-Managed and Topology-Independent Location Service for Self-Organizing Networks. In *MobiHoc*, 2005.

[94] P.-J. Wan, K. M. Alzoubi, and O. Frieder. Distributed Construction of Connected Dominating Set in Wireless Ad Hoc Hetworks. *Mobile Networks and Applications*, 9(2):141–149, 2004.

[95] P.-J. Wan, G. Calinescu, and C.-W. Yi. Minimum-Power Multicast Routing in Static Ad Hoc Wireless Networks. *IEEE/ACM Transactions on Networking*, 12(3):507–514, 2004.

[96] M. Wattenhofer, R. Wattenhofer, and P. Widmayer. Geometric Routing Without Geometry. In *SIROCCO*, 2005.

[97] R. Wattenhofer and A. Zollinger. XTC: A Practical Topology Control Algorithm for Ad-Hoc Networks. In *WMAN*, April 2004.

[98] S.-C. M. Woo and S. Singh. Scalable Routing Protocol for Ad Hoc Networks. *Wireless Networks*, 7(5):513–529, 2001.

[99] Y. Xue, B. Li, and K. Nahrstedt. A Scalable Location Management Scheme in Mobile Ad-Hoc Networks. In *LCN*, 2001.

[100] W. Ye, F. Silva, and J. Heidemann. Ultra-Low Duty Cycle MAC with Scheduled Channel Polling. In *SenSys*, 2006.

[101] Y. Yu, G.-H. Lu, and Z.-L. Zhang. Enhancing Location Service Scalability with HIGH-GRADE. In *MASS*, 2004.

Curriculum Vitae

September 10, 1980	Born in Stans, Switzerland
1987–2000	Primary and secondary schools in Oberdorf, NW, Switzerland and high school at the Kollegium St. Fidelis in Stans, Switzerland
2000–2005	Studies in Computer Science, EPFL, Switzerland
2002/2003	Exchange year at the Carnegie Mellon University in Pittsburgh, PA, USA
April 2005	MSc in Computer Science, EPFL, Switzerland
2005–2009	PhD student, research and teaching assistant at the Distributed Computing Group of Prof. Dr. Roger Wattenhofer, ETH Zurich, Switzerland
September 2009	PhD degree, Distributed Computing Group, ETH Zurich, Switzerland Advisor: Prof. Dr. Roger Wattenhofer Co-examiners: Prof. Dr. Sándor P. Fekete, TU Braunschweig, Germany Prof. Dr. Leonidas J. Guibas, Stanford University, USA

Publications

The following list enumerates the publications I co-authored during my PhD at ETH Zurich.

1. Slotted Programming for Sensor Networks. Roland Flury and Roger Wattenhofer. *Under submission.*

2. Greedy Routing with Bounded Stretch. Roland Flury, Sriram Pemmaraju, and Roger Wattenhofer. In *28th Annual IEEE Conference on Computer Communications (INFOCOM), Rio de Janeiro, Brazil* April 2009.

3. Randomized 3D Geographic Routing. Roland Flury and Roger Wattenhofer. In *27th Annual IEEE Conference on Computer Communications (INFOCOM), Phoenix, USA*, April 2008.

4. Routing, Anycast, and Multicast for Mesh and Sensor Networks. Roland Flury and Roger Wattenhofer. In *26th Annual IEEE Conference on Computer Communications (INFOCOM), Anchorage, Alaska, USA*, May 2007.

5. MLS: An Efficient Location Service for Mobile Ad Hoc Networks. Roland Flury and Roger Wattenhofer. In *7th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC), Florence, Italy*, May 2006.