Geo-process lookup management

Andreas Hägglund

Civilingenjör, Datateknik
2017

Luleå tekniska universitet
Institutionen för system- och rymdteknik
Geo-process Lookup Management

Andreas Hägglund

Luleå University of Technology
Dept. of Computer Science and Electrical Engineering
Div. of Information and Communication Technology

14th November 2016
This thesis presents a method to deploy and lookup applications and devices based on a geographical location. The proposed solution is a combination of two existing technologies, where the first one is a geocode system to encode latitude and longitude coordinates, and the second one is a Distributed Hash Table (DHT) where values are stored and accessed with a <key,value> pair. The purpose of this work is to be able to search a specific location for the closest device that solves the user needs, such as finding an Internet of Things (IoT) device. The thesis covers a method for searching by iterating key-value pairs in the DHT and expanding the area to find the devices further away. The search is performed using two main algorithm implementations LayerExpand and SpiralBoxExpand, to scan the area around where the user started the search. LayerExpand and SpiralBoxExpand are tested and evaluated in comparison to each other. The comparison results are presented in the form of plots where both of the functions are shown together. The function analysis results show how the size of the DHT, the number of users, and size of the search area affects the performance of the searches.
I would like to offer thanks to my supervisors. Johan Kristiansson at Ericsson research in Luleå for support in my work with the project at Ericsson, and also with giving great feedback on this thesis paper improving the paper. I would also like to thank Olov Schelén at Luleå University of Technology for also coming with good feedback.

I would also like to thank Ericsson Research in Luleå for letting me do the work for them and providing a supporting workplace atmosphere. Finally I would like to thank Jessica Hägglund for her help with reading through the paper and coming with feedback on the grammar and language.

Andreas Hägglund
## CONTENTS

**CHAPTER 1 – INTRODUCTION**

1.1 Problem description ........................................... 1
1.2 Geo-process lookup management ............................ 2
1.3 Scope ......................................................... 4
1.4 Use cases ..................................................... 5
  1.4.1 Video conference ........................................ 5
  1.4.2 Smart vehicles ........................................... 5
1.5 Related work ................................................. 6

**CHAPTER 2 – GEO-PROCESS LOOKUP MANAGEMENT**

2.1 Proposed solution ............................................ 9
  2.1.1 Process runtime agent ................................... 11
2.2 Geohash ....................................................... 11
  2.2.1 Encoding and decoding ................................... 12
  2.2.2 Problems with Geohash ................................ 12
2.3 Distributed Hash Table ...................................... 15
  2.3.1 Kademlia .................................................. 15
2.4 Storing GeoHash in Kademlia ............................... 18
2.5 Geo-process lookup algorithm ............................. 19
  2.5.1 StarExpand ............................................... 21
  2.5.2 LayerExpand ............................................. 22
  2.5.3 SpiralBoxExpand ......................................... 23

**CHAPTER 3 – IMPLEMENTATION**

3.1 Kademlia ..................................................... 25
3.2 LayerExpand .................................................. 26
3.3 SpiralBoxExpand ............................................. 28
3.4 Proof-of-concept prototype ............................... 30
  3.4.1 Proof-of-concept implementation ..................... 31

**CHAPTER 4 – EVALUATION**

4.1 Experimental setup .......................................... 33
  4.1.1 Hardware specification ................................ 34
4.2 Result ......................................................... 35
List of Figures

1.1 Geo-process lookup management of software containers. The A symbol in the figure represents process runtime agents available in the P2P network (DHT). ................................................................. 4

2.1 Basic concept. ................................................................. 10

2.2 Even levels. ................................................................. 12

2.3 Z-curve [1]. ................................................................. 12

2.4 Odd levels. ................................................................. 12

2.5 Center location and its neighbors. ................................. 14

2.6 Kademlia routing table. ................................................ 16

2.7 Kademlia look up procedure. ......................................... 16

2.8 The flowchart describes the general idea of how the geo-process lookup is done, by doing lookups at geohash keys in the DHT and expanding the search until a runtime config (RT Config) is found. ...................... 20

2.9 Stars shows positions where the algorithm is looking for neighbors at the lines (outside the cell) illustrate where neighbors is found and the arrows (outsides the cell) where StarExpand starts to miss neighbors. ......... 22

2.10 The figure shows how a search with LayerExpand is done by starting in the center and then taking the first layer (darker) followed by the second layer (brighter). .................................................. 23
2.11 The figure shows how the SpiralBoxExpand this is done by starting in the center and moving north and west on odd distances and south and east on even distances. .......................................................... 24

3.1 Flowchart for LayerExpand shows how the search for a nearby process runtime agent is done by the LayerExpand method. ......................... 27

3.2 Block Diagram for LayerExpand show how the different parts interact with each other when the algorithm is used. ................................. 27

3.3 Flowchart for SpiralBoxExpand illustrates how it searches for a nearby process runtime agent. ............................................................ 29

3.4 Block Diagram for SpiralBoxExpand shows how the interactions are done between the different parts of the program when the algorithm is used. . 29

3.5 Demo screenshot shows the process runtime agent in the area, a deployed container, and where the algorithm have found the closest process runtime agent in Stockholm. ..................................................... 30

4.1 The screenshot illustrates how the 100 nodes is placed and the initial position of the container. The dashed arrow illustrate in what direction the position of the container is moved, as the distance to the node cluster increases. ................................................................. 34

4.2 Spiral vs Layer DHT Calls Comparison. ................................. 35

4.3 Spiral vs Layer Time Comparison 100 nodes. ............................ 36

4.4 LayerGeoCalls the amount of calls to LayerExpand helper functions. . 36

4.5 SpiralBoxExpand vs LayerExpand, time per layer. ...................... 37

5.1 Plot shows increasing NumberOfUsers from 0 up to 1 million with increments of 1000 users, while keeping avgDistance constant at 10 layers and NumberOfNodes at 100000 nodes. .............................................. 42

5.2 Plot shows increasing NumberOfNodes from 0 up to around 1 million with increments of 1024 nodes while keeping AvgDistance constant at 10 layers and NumberOfUsers at 100000 users. .............................. 42
5.3 Plot shows increasing AvgDistance from 1 up to 50 by increments of 1 while keeping both NumberOfUser and NumberOfNodes at 100000 users and nodes respectively.

5.4 The figure illustrates how the distance to the corner of the active search area extends to layers further out. And therefore covering areas that were not covered by the search making it possible to miss closer nodes when a node is found in the corners. The dashed arrow that is projected on the vertical line has the same length as the full arrow $v_1$. The circle shows the area that is at the same distance or closer to the corner that $v_1$ is pinpointing.
List of Tables

2.1 XOR distances. ................................................................. 18
4.1 Hardware specifications. .................................................... 35
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>DHT</td>
<td>Distributed Hash Table</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>PRA</td>
<td>Process Runtime Agent</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
</tbody>
</table>
This thesis aims to define a method for finding IoT devices or application providers that would benefit from running at a particular geographical location. This geographical location would be close to a user or possibly bound to an IoT device at the location. One of the most important aspects of the system was that it should work on a decentralized platform where the user could both contribute to the content of the platform and make use of the services the platform provides. The main reasons for using a decentralized platform is to ensure the system can scale out organically with the users adding their services to their provided hardware and connecting them to the system. The decentralized platform would provide more freedom to the users when they do not have to buy storage or computing power from a provider of a centralized platform. The provider has to extend their server park to keep up with the demand of the users while keeping their machine park on a level where the idle time of the servers is minimized. The primary goals of the thesis was:

1. Based on a decentralized platform

2. Finding and Storing services based on their geographical location

1.1 Problem description

As the number of devices increases, the need to be able to keep track of them arises. The problem lies in how to find the device or services that are close to the user geographically
on a decentralized platform. Finding the devices in turn comes with some sub-problems, one of them being how to store the devices in a data structure and be able to access them based on a particular geographical location. If a device is not found in the initial lookup location, the problem evolves into finding devices in the surrounding area of the specified initial location.

Besides trying to find a solution to the above problems, a performance test and evaluation of the solutions have been performed. The test involves finding out how a wider search area affects the lookup time needed and the load on the infrastructure in terms of traffic to the data structure.

The next section will provide an introduction to the proposed solution of the problems mentioned above.

1.2 Geo-process lookup management

Ericsson Research is currently investigating how to design and implement a scalable and decentralized cloud platform. Such platform would make it possible to manage and easily deploy cloud services, e.g. telecommunication equipment such as radio base stations or new media and communication services.

The work presented in this thesis takes the idea of a decentralized cloud further by making it possible to find, discover and deploy software processes based on geographical location. These process operations are called geo-process lookup management. A process can be a software process running in a software container or a virtual machine, but it can also be a software process running on an end-user device such as a smartphone or an IoT device.

While the initial idea was originally developed to schedule and deploy software containers such as Docker [2] to support a video conferencing system, this technology could have far-reaching application in building a distributed IoT cloud platform. It is likely that all kinds of devices and services in the future will be able to run software containers either virtually or physically. The reason these software containers is going to be important for the future is they enable an easy way of submitting applications on different platforms. Since the software container has an operation system pre-installed that the applications are deployed too. Therefore to allow a platform to be able to run the applications, the platform only needs to be able to run the generic software containers. It is also a risk that today’s cloud technology will fail to scale to support billions of devices, which would motivate the use of Peer-to-Peer (P2P) infrastructure.
A P2P infrastructure let the different nodes in the network to be able to communicate directly with each other but also share their resources to the system. The biggest gain from using a P2P system is that the nodes communicate directly compared to a client-server model where all the communication is routed through a central server, which could lead to scalability issues.

An example of an IoT device that could benefit from the geo-process discovery could, for instance, be a connected car capable of looking up information about pedestrians on a crosswalk. The car could use the geo-process discovery in conjunction with the on-board GPS that can detect that the car is approaching a crosswalk. In this case, a lamppost could run a software container and thus provide information about pedestrians via connected sensors. The lamppost could either run a process locally on a container running on an embedded computer or the process in the lamppost could run in a data center somewhere. As more devices and services become connected, the lines between physical and virtual will become more blurred. At a point where one could view physical objects such as a car or building as a complex system of interconnected processes forming a sphere of geo-process taking place in a meta world. These complex systems can sometimes be referred to as the metaverse [3], layered on top of our real world.

The proposed solutions in the thesis make it possible to specify a location and search a geographical area for process runtime environment. And find more services by performing so-called expand searches in a Distributed Hash Table (DHT), which will be described further in Section 2.3. By using a DHT, it becomes possible to create a distributed and decentralized platforms without central control that are massively scalable. Using a DHT infrastructure makes the system different compared to today’s cloud computing technology typically based on centralized solutions, e.g. centralized cloud orchestrations engines.

To be more precise, an iterative search algorithm is proposed. Along with possible implementations of the algorithm that shows different ways to expand searches around areas to locate a process runtime environment, to deploy a new process upon or find an existing process to use.

Figure 1.1 illustrates how software containers can be deployed to a process runtime environment associated with a geographical area. In the example that Figure 1.1 describes, different process runtime agents (explained in Section 2.1.1) are connected to a P2P network. And when a developer (or software component) deploys a container, the proposed search algorithm can find the geographical closest process runtime environment registered in the P2P network based on a geotag set on the container.
4 Introduction

Figure 1.1: Geo-process lookup management of software containers. The A symbol in the figure represents process runtime agents available in the P2P network (DHT).

1.3 Scope

As the work went forward, the scope was limited and expanded in a different direction, namely geo-process lookup management. The main reason these delimitations were made to the scope was because of lack of time to make a full scheduling service. Geo-process lookup management is much like the location based scheduling that early on started to be one of the scheduling parameters for containers. Geo-process lookup management later evolved into more of a geo lookup for IoT devices shifting the focus from scheduling containers on nearby workers to a tool for finding devices close to a particular location. The evolution in a way it expanded the scope again since the system will both be able to deploy tasks or containers to DHT nodes that got the capability of running them and work as IoT device discovery.
1.4 Use cases

The following two examples of use cases illustrate in more detail the role of the process runtime agent and how it is used to perform a geographical process lookup.

1.4.1 Video conference

One example could be to find video conference equipment (e.g. monitor, projector, drawing, connected drawing boards, etc..) at a particular location to set up a video conference meeting. In this case, the video conference equipment run a software process that is connected to a process runtime agent. As described below, the software process could, for example, connect a video stream to a monitor or computer. The contact would be made by finding the monitor or computer’s process runtime agent in the DHT and acquire the connection information for that monitor or computer.

Typically, when the conference equipment is started, it determines its geographical location (e.g. using an indoor location system) and then locates the closest process runtime agent responsible for the geographical area that includes the determined location. The process runtime agent could run directly on the conference equipment, but it could also run remotely in a data center somewhere. In both cases, the process runtime agent is associated with a particular geographical location.

When a user enters the conference room, it can thus use the proposed geo process lookup scheme to connect to process runtime agent responsible for the conference equipment.

1.4.2 Smart vehicles

Another example could be an intelligent transport system where the invention is used to implement an early-warning collision detection mechanism. When a GPS in a car recognizes that the car is approaching a pedestrian crossing, the system can use the GPS to obtain the coordinates for the crossing. The system could then use the GPS coordinates to do a lookup for a process runtime agent at that location for a pedestrian sensor, thus warn the driver about the existence of potential risks. A system like the previously described could also be combined with automatic security systems in the car making the car slow down to avoid accidents. In this example, the process runtime agent could, for example, be embedded in a lamp post with sensors to detect nearby pedestrians. Note that software process that is running in the lamp post could run remotely in a data
center, but still be associated with a geographical position of the pedestrian or the lamp post.

1.5 Related work

MongoDB [4] has the option to store data with geohashes as keys, which is similar in the sense that it uses geohash as keys to store data with a relationship to a particular location. However, not with a DHT as a data structure and although it can be replicated it does not give the same kind scalability as a DHT provides. And the ways they search an area is by projecting a shape like a rectangle or defined circle with a center and radius and check within that polygon for possible hits. But the proposed solution in this thesis got the world divided into cells. The size of the cells is determined by how many times the cells have been sub-divided which is dependent on the precision, this will be described in Section 2.2.

Elasticsearch [5] is a document storing technology where the users can store data on different indexes and replicated it over shards and nodes to get better robustness. These documents can be searched and utilized in real time. Geohash is one of the ways they let users store and perform queries [6]. Elasticsearch is close to the solution proposed in this thesis, but it is used to store documents. While the proposed solution stores locations to run software on or find devices or software/sensors already running close to a particular location.

P2PR-tree [7] uses a tree structure for storing and searching for objects in spatial P2P environments. It performs efficient pruning by maintaining minimal information about peers far away and more about nearby peers this makes it use less disk space. It is also completely decentralized making it scalable and robust, and it is easy for nodes to join and leave. The P2PR-Tree algorithm does not provide global height balancing. It stores values by dividing the areas into smaller sub-areas, and each parent keep track of their children, which, in this case, are smaller areas that are placed within the parents areas. A child area could both be wholly or partially inside the borders of multiple parent areas this mean that an area can be saved at several parent areas.

GHT: A Geographic Hash Table for Data-Centric Storage [8] provides a method to hash a key k into geographic coordinates. The key together with the corresponding value is stored at a node in the vicinity of the location. The search is routed geographically, and the packets are marked with positions and destinations. Nodes know their position and their neighboring nodes one hop away. Using this local knowledge GPSR can route a packet to any connected destination. It uses the Greedy Forwarding Algorithm [9]
and forwards the packet to a neighbor that is closer to the goal. Greedy forwarding will fail when there’s no neighbor closer than itself. It recovers from these failures with a perimeter mode using the right-hand rule encircling the destination node with the closest nodes. The approach is different compared to the proposed solution in the nodes in the system that can be placed in different geographical position than the position the data correspond too. The search finds the closest nodes in the DHT to the target location, which means that if no nodes are located in the target area the \(\text{key}, \text{value}\) pair is saved in the closest nodes around the position. However, the solution described in this thesis stores the data directly in the DHT with the geohash as a key.
To make it possible to schedule and deploy processes geographically a solution based on a combination of two different technologies is proposed. The first technology is called Geohash [10] [11] [12] which is a geo-encoding system. That makes it possible to represent geographical coordinates (typically a sequence of latitude and longitude) as single value hash codes. The second technology is DHTs, which makes it possible to store geohash data as keys in a key-value database, where the value is connection information to find the service provider handled by the process runtime agent. The most important reason for choosing a DHT as the data structure is for the scalability aspects giving the system the possibility to grow organically when more users add their services to the system. The proposed solution will be further explained in the coming Section.

2.1 Proposed solution

The central idea proposed in this thesis is to introduce an agent, a process runtime agent, which represents a process runtime environment. An example of runtime environment can for example be a Docker engine or Mesos [13] [14] [15]. The process runtime agent keeps track of connection information to the underlying process runtime environment and is stored in the DHT with the help of a geohash string. Geohash is a string representation of a geographical position and will be further explained in Section 2.2.
While it is normally only possible to make lookups for specific keys in a DHT and not possible for a search for information. The essence of the proposed solution is to represent geographical locations cells as keys in the DHT. And use that structure to iterate and find process runtime agents in surrounding cells, thus providing a search function albeit limited to geographical location.

Metadata about process runtime agents is stored in the DHT with the key being a geohash and the value a set of metadata of process runtime agents registered at that geolocation. The connection between the geohash and the metadata makes it possible to search for process runtime agents by looking up the geohash in the DHT and use an expand search algorithm to find nearby geographical process runtime agents. The solution is thus distributed and decentralized as well as a scalable system (assuming a large geographical area is not searched), where the DHT can be used for registering, and finding the nearest process runtime agents. See Section 2.5 for more information about the general expand search algorithms and in Section 2.5.1 2.5.2 2.5.3 StarExpand, LayerExpand [16] and SpiralBoxExpand [17] is described respectively.

![Figure 2.1: Basic concept.](image)

Figure 2.1 illustrates the basic idea proposed in the thesis. The process runtime agent can be connected to underlying cloud platforms, e.g. Mesos, Docker, OpenStack [18] to deploy processes or containers. As mentioned before, agents can also be associated with processes running on end-user devices or IoT devices, e.g. a web server running on an IoT device.
2.2. Geohash

Typically, processes (or containers) are tagged with a geohash and deployed to any available process runtime agent found in the DHT. The process runtime agent that is receiving the request and uses the proposed search algorithm to find a more suitable process runtime agent(s), and then dispatch the process/container to that process runtime agent. Alternatively, an external client can be used to search the DHT and then deploy the container directly to the selected process runtime agent found by the client. In the following Section process runtime agents will be described.

2.1.1 Process runtime agent

The process runtime agents is a manager of the system that keeps track of the connection between the different nodes that is stored in the area it is managing. The area it is managing is determined by a geohash code and contain multiple nodes depending how many services or devices have been stored in the location specified. It also serves as the first contact for the users of the system, which means that a user only needs to know of one process runtime agent to be able to use the system. Therefore, when a search is performed at a geographical location, the contacted process runtime agent will forward the query to the rest of the system for the user. In the following Section, it will be explained how the geohash strings are calculated.

2.2 Geohash

Geohash is a geocode system that converts a coordinate pair of latitude(lat) and longitude(long) into a string with a base32 character map. This string will have different lengths depending on what precision is wanted for the hash. A geohash string with the length of 9 characters will give an accuracy of around 2 meters while a precision of 1 is \( \frac{1}{32} \) th of the area of the earth. So each character added to the hash make divides the rectangle denoted by the prefix into 32 smaller rectangles.

For example, by using the geohash.org web page a user can insert the hash string in the URL geohash.org/u7xtr giving us a marker at the coordinates (65.6, 22.1), which basically points at Luleå as a town at that precision. While adding another four characters to the geohash Luleå University of Technology will be pinpointed with geohash.org/u7xtr9pzr at the coordinates (65.6171, 22.1374). Geohash can be used as a way to uniquely identify a certain area, and this is one of the reasons geohash is an attractive option for geo scheduling. In regards of the proposed solution, this would be the way of finding process runtime agents in the same area where a process is to be deployed.
2.2.1 Encoding and decoding

The encoding scheme used got a slightly different order depending on if it is an odd or even level, where the level is the length of the hash string. Figure 2.4 shows how odd levels are ordered and Figure 2.2 is for the even levels. These tables are made using something called a Z-curve, which is described in Figure 2.3.

![Even levels](image1)

![Z-curve](image2)

![Odd levels](image3)

The decoding works as follows. The geohash code ezs42 got the bit representation: 01101 11111 11000 00100 00010 taking the even bits as the longitude code and odd bits as latitude code. To get the decimal representation of the latitude and longitude from this bit string a divide and conquer algorithm is used. For example, when decoding the latitude value it can range between -90 and 90, and since the latitude bit string is 10111 10010 01, the first bit will be inspected first. In this case, it is 1 which means that the value searched for is in the range 0-90. Now the second bit will be examined, as the second bit is 0 it will reduce the range to 0-45. When this has been done for each bit in the string, it will give a more and more precise position with 1’s selects the higher half of the range while 0’s selects the lower.

2.2.2 Problems with Geohash

An issue with geohash is around the prime meridian [19] or the equator where the prefixes of the cell rectangles on different sides of these split lines will be entirely different. The problem is illustrated in Figure 2.4 where the center line between G and U would be the prime meridian. The jumps between sequencing cells on the world map comes from that geohash uses a Z-curve pattern to encode the geohash zones. Possibly causing a problem if someone want to use geohash to perform a closest neighbor type of task, for example.
assigning residences in a town to their nearest post office. Since trying to find the closest process runtime agents to the preferred location of the system user is a closest neighbor type of problem, the issue discussed in this paragraph is, therefore, something that needs a solution.

There is also the same problem around the 180-degree meridian and the function breaks at the poles. The latitude-longitude system is non-linear which makes the cell rectangles less and less square for example at the equator they are close to square. With a difference between the sides of the rectangle of only 0.67% while at latitude 30.0, in level with the North African countries it is at 14.89%. As we move further north to latitude 60 which is about the level of Oslo and Stockholm the difference is 99.67% reaching infinity at the poles. These issues all stem from the fact that it is difficult to map coordinates on a sphere onto a two-dimensional coordinate space.

There is the existing solution for finding surrounding geohash cells, but they only check for the cells in the absolute proximity that is the eight surrounding the initial geohash cell.

**The problem of neighboring prefixes not being close could possibly be solved by:**

1. It could be possible to move the coordinate system to start from the south pole and along the split line in the Pacific Ocean outside Japan. The \(<180, -180>\) split using a geohash that is between 0 and 180 for latitude and 0 and 360 for the longitude direction.

2. Decode hash and check if it is close to the equator or prime meridian. And if that is the case slightly modify the coordinates to the other side of the equator or prime meridian and rehash them again and do a search here as well.

**Implications of mentioned problems**

When evaluating Option 1, it seems this approach will not be possible as it uses z-curve to assign the values to the cell blocks. And something like the Z-curve has to be used to assign the characters to blocks relatively close to each other for the most part. Another option is the Hilbert curve [20] (u-shaped) could be used as well but it, unfortunately, got the same problem as z-curve. The problem is therefore not solved by changing the ranges from having negative values because the problem stems from the fact that the cell IDs are assigned with the Z-curve. It should also be mentioned that this option would just shift the problem to a location where it would matter less for practical purposes but not solve the underlying problem.
The solutions that have been chosen resemble Option 2. The proposed solution finds the eight neighbors in each direction so N, NE, E, SE, S, SW, W, and NW. The chosen solution also works along the meridians and equator since the solution involves decoding the hash and looking on the cell blocks in the different directions and then encode them again saving the result from each direction. Figure 2.5 gives a good idea how this works. The fact that the distance between the lat-long coordinates directly is not a problem merely implementing a great circle distance formula.

The problem with the cells having a different shape depending on the distance from the equator does not matter much since the cell is not entirely square anywhere. The reason it is not a big problem is because the geohash system will be used to pinpoint an area where connection information to devices can be stored. If a search is made to find a software container in Luleå using a geohash precision of 4 which correspond to roughly 20 x 20 km. With, for example, five services located in Luleå they will all be stored under the same process runtime agent managing that geohash. And if a container is to be deployed in Luleå the process runtime agent would handle the request. These services can be found by geohasing the location where the container prefers to be deployed and looking in the distributed hash table (DHT). If there are any process runtime agents in that area denoted by the first four chars, they will be stored in the DHT with the geohash as a key. If there are no process runtime agents or not a sufficient number of them to find a good match, the search can be expanded to the neighboring geohash cells and in turn look these up in the DHT. The following section describes DHT in more detail.
2.3 Distributed Hash Table

A distributed hash table, is a decentralized type of hash table, which stores <key, value> pairs. The stored value can be accessed in \( O(\log(n)) \) time by directly looking up the key in the hash table. In the DHT, the responsibility of the mapping is distributed among the network nodes, which give DHTs excellent scalability and robustness characteristics. The DHT used in the thesis is Kademlia, which uses the bitwise exclusive (XOR) to define the distance between the nodes IDs in the DHT, Kademlia is described further in the next section.

2.3.1 Kademlia

This Kademlia description is heavily influenced by the Kademlia paper [21] and summarizes the most important parts. Both the Figures 2.6 and 2.7 are taken from this paper.

To locate a node or nodes close to the target ID, Kademlia uses the same routing algorithm from the start of the lookup until the node(s) is found. Nodes are leaves in a binary tree, where the position of the nodes is defined by the unique prefix of the node ID. The lookup for a node is done in \( \log(n) \) steps, where each step finds IDs that are closer to the ID that was initially searched. Figure 2.6 shows a tree with Kademlia IDs where the black node got the prefix 0011, and the circled areas denotes sub-trees that do not contain the prefix 0011. Every node in the Kademlia network knows of at least one node in each of their sub-trees this makes it possible to guarantee that algorithm can locate nodes by their ID when searching the tree.

Figure 2.7 shows how the node from above 0011 is looking for the node 1110 by querying the closest node to the target it knows off. And by getting information from the current node, about closer nodes in deeper sub-trees that is closer to the target and is more likely to know where the target node is located and finally finding the goal at Step 4.

The Kademlia lookup algorithm builds on being able to define how close two nodes are to each other by using bitwise exclusive or (XOR) between the IDs. The standard ID in a Kademlia network is a random 160-bit identifier, but they can also be constructed to let the ID contain some information itself. Kademlia defines the distance between the two nodes using XOR as an integer where \( d(x, y) \) denotes the distance between 2 IDs \( x \) and \( y \) the distance is: \( d(x, y) = x \oplus y \). The XOR got the triangle property, so that the distance between \( d(x, y) + d(y, z) \geq d(x, z) \). This is proven by the following facts \( d(x, y) \oplus d(y, z) = d(x, z) \) and \( a + b \geq a \oplus b \), where \( a, b \geq 0 \). Unlike Chord[22] which got
a directional circle metric topology, the XOR topology is symmetric so \( d(x, y) = d(y, x) \). Which means in a full tree the distance between two nodes is the height of the smallest common sub-tree, but when the tree is not fully populated the closest leaf is the one with
the longest common prefix with the ID searched.

The nodes in the network keep information about other nodes to be able to route messages. For each $0 \leq i \leq 160$, the nodes maintain a list of triples containing $\langle$IP address, UDP port, Node ID$\rangle$ for nodes at a distance between $2^i$ and $2^{i+1}$ from themselves called k-buckets.

- When $i$ got a low value (few nodes in network): The buckets will most likely be empty since there are few nodes in the network to populate the buckets.

- When $i$ got a high value: The k-buckets can grow up to size $k$ where $k$ is the replication factor.

The replication factor is chosen, so it would be unlikely for a full bucket to fail within the same hour. When a node receives a message, it updates the senders NodeID in the bucket. The k-bucket system gives a resistance to certain denial of service (DoS) attacks since an attacker cannot flood the network with new nodes to flush the routing state of the nodes. And this cannot happen when Kademlia only inserts new nodes when old nodes leave and before removing any nodes they will be pinged to check if they are still alive. If they are alive, they will be moved to the end of the list, so if all nodes in the list responds to the query no nodes will be evicted from the list and the new node will not be added to that k-bucket.

Kademlia has four main Remote Procedure Calls (RPCs):

- **PING**: Probes a node to see if it is online.

- **STORE**: Makes a node store a $\langle$key, value$\rangle$ pair.

- **FIND_NODE**: Takes 160-bit ID, returns the $k$ nodes closest to the target ID that it knows about in the form of $\langle$IP address, UDP port, NodeID$\rangle$ triples. The triples can come from one or many k-bucket. It will always return $k$ items or all items if it does not have a total of $k$ items.

- **FIND_VALUE**: Behaves like FIND_NODE but if the node got a STORED value said value is returned instead.

When a node receives an RPCs, it has to echo a 160-bit random RPC ID, which makes address forgery attacks towards the system harder. However, the most important procedure is the node lookup that finds the $k$ closest nodes to a given node ID, which makes Kademlia be able to find faster the node that was searched after.
Kademlia republishes the key-value pairs once per hour to make sure the pairs is up to
date in the system. To avoid flooding the system with a lot of messages the one-hour
timer is reset when a STORE RPC is received, which means that only one node needs
to republish a particular key-value pair.

### 2.4 Storing GeoHash in Kademlia

An initial idea how to do the geographical lookup was to directly combine geohash and
Kademlia’s ability to find nodes that are close to each other. The essence of the solution
would be to save the process runtime agents by in the DHT with the key being their
geohash string generated from their geographic latitude and longitude coordinates.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>XOR 0</th>
<th>XOR 1</th>
<th>XOR 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>10</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>11</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>12</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>13</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>14</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>15</td>
<td>14</td>
<td>7</td>
</tr>
</tbody>
</table>

Initially, this looked like it could be a working strategy when looking at for example Ids
close to 0x0 in the range 0x0 to 0xF, the XOR metric would return the neighbors in the
correct order with the distance to 0x0 being 0 and 0x1 would be 1 and so on. However,
when looking for other values, say 0x2, the distance before the value would get inverted
and for each time the bit $xx1x$ occurs the previous two values would get flipped from
their correct order. The flip does not give that much change on lower bits when only a few positions are flipped a short distance. But the more significant bits that are looked at, the worse the problem get. For example, when looking at 8, this would make the distance after 8 be in correct order but the positions before 8 would be totally flipped, and it would look like 1 is closer to 8 than 7. This is shown in Table 2.1. The above problems made it clear another solution was needed, which is going to be described in the following section.

2.5 Geo-process lookup algorithm

In this section, a step-by-step algorithm description along with the flowchart shown in Figure 2.8 illustrates a flowchart of the proposed algorithm. The algorithm starts by looking up the geohash cell of the process that is going be found or deployed, and check if there are any process runtime agents stored under that geohash cell key. If no values are found, the search will be extended to the surrounding geohash cells, which in turn will be searched in the DHT. This procedure is then repeated until a process runtime agent is found.

Pre requisites: Each process runtime agent (A) publish its geo metadata in a DHT, i.e. $dht.set(geo\text{data}^A, \text{config})$, where the $\text{config}$ contains information how to connect (e.g. IP address) to that process runtime agent. The key, $geo\text{data}^A$, represents the geocell which A belongs to. The geocell key can be calculated by pruning the geohash to a particular length corresponding to the desired precision specified by the system. The exact geohash location is saved in the $\text{config}$ value to be able to do secondary selections.

Let $P_{\text{Pos}}$ be the geohash position of a particular process, P, and $A_{\text{Pos}}$ the geohash cell of a particular process runtime agent, A, then D is the distance between $P_{\text{Pos}}$ and $A_{\text{Pos}}$.

Return: Return the closest process runtime agent, A, closest to P.

1. Look up $P$ in DHT based on the preferred geohash location, i.e. $A = dht.get(P_{\text{Pos}})$, where $P_{\text{Pos}}$ is the calculated geohash cell.
2. If $A$ is valid (i.e. found in DHT), return $A$.
3. Select the first cell layer$^1$, $layer=1$
4. Use expand search to find an array of geohash blocks, $B$ in the layer.
   (a) For each block, $i$, in $B$, look for possible $A$s, i.e. $A = dht.get(B[i])$.

$^1layer$: a full encircle of cells around the cell where the search started, each layer adding another encircling around the initial cell.
Figure 2.8: The flowchart describes the general idea of how the geo-process lookup is done, by doing lookups at geohash keys in the DHT and expanding the search until a runtime config (RT Config) is found.

i. If \( A \) is valid, then store \( A \) in a selection array, \( S \).
(b) In \( S \) find and return the closest process runtime agent where the distance, \( D \), between in \( P_{Pos} \) and \( S[i] \) is shortest, alternatively return the whole selection array for secondary filtering.

5. If no \( A \) is found, increase cell layer (layer++), and go to Step 4 to search in an increased geographical area.

Typically, process runtime agents are unlikely to be found directly (Step 1), which means that the search for a process runtime agents needs to be extended, i.e. the cell layer parameter needs to be increased, and Step 4 in the aforementioned algorithm needs to be repeated. This procedure is called expand the search and allows a geographical area to be scanned for \( A \). Possible \( A \) candidates are stored in a selection array.

The \( A \) references in the selection array, \( S \), can then be compared to find the process runtime agent closest to process geo metadata (Step 4b), alternatively, an ordered list of the multiple process runtime agents found in the selection array can be returned. The
ordered list could be useful for performing a secondary selection. For example, finding the process runtime agent that is least loaded (e.g. have least CPU load), or other kinds of attributes.

Note that the proposed solution works both on a virtual and physical infrastructure. It searches for references to process runtime agents running in a data center, or references to process runtime agents running on a physical infrastructure or end-users/IoT devices.

2.5.1 StarExpand

StarExpand was the first naïve attempt at an implementation of the Geo-process lookup algorithm. And the basic idea around it was to make it as simple to implement it as possible using the already existing function Neighbors [23], which returns the eight surrounding cells to a center cell. What this solution does in contrary to the normal Neighbors function it instead moves further away from the center in all eight directions. The way it expands the search is in the shape of an 8-pointed star and for each step it uses the Neighbors function to find the surrounding cells. The way this is implemented means that a map of strings to booleans have to be maintained to make sure that geohash values are not added to the list of geohashes to be searched more than once. In addition to that limitation, which means that the algorithm will examine the same cells multiple times. It will also have to maintain the map between strings and booleans which have to use extra memory to keep track of what cells that already have been visited.

When StarExpand starts to expand past layer 4, it begins to miss parts of the area, the Figure 2.9 shows the case when it fails. Therefore, the location of the stars show where a failed search is performed and it will find the new position denoted by the lines outside the 9x9 grid square and the arrows show where it will start missing spots on the outer layer. When going further out it will begin to lose more and more cells between the corner and the center, so it can only expand four layers outside the center giving a total of 81 geohashes including the center. The returned values are not ordered because it first checks everything in a particular direction e.g. (all values from east to west) and then takes the next direction which means the list will be scrambled. Given that a user wants to find the ten closest process runtime agents to a position. The entire list needs to be checked, and the distance to the NodeIds determined and sorted depending on this distance before the ten nearest can be found.
Figure 2.9: Stars shows positions where the algorithm is looking for neighbors at the lines (outside the cell) illustrate where neighbors is found and the arrows (outside the cell) where StarExpand starts to miss neighbors.

### 2.5.2 LayerExpand

LayerExpand is a quite natural approach to search an area and the easiest way to explain how it works would be that it is like rings on the water around a dropped stone. It starts in the center and for each step it will recursively check the next layer on the outskirts of the area as illustrated by the picture Figure 2.10. LayerExpand returns the values in an ordered matter where the first value is the initial geohash that was provided followed by the values at index 1-8 (darker inner square on Figure 2.10) being the first layer and 9-24 (brighter square on Figure 2.10) being the second layer and so on. This makes implementing a FindN function quite easy. A FindN function would be looking if the geocells contains any process runtime agents and saving them in a list while doing the LayerExpand and when n values are found it terminate the search. One of the downsides using this solution is that FindN is implemented by using the Neighbors() function that will return all the surrounding cells for each cell in the current layer. When using the Neighbors() function in this manner means that a list of already visited cells have to be maintained to avoid visiting the same cell twice.

It should be mentioned that some process runtime agents, which is closer, can be missed cause when n process runtime agents are found the method will terminate and return the list of process runtime agents. But the process runtime agents missed will not be closer than around 20km (geohash precision 4) compared to the other process runtime agents found in that layer. The reasoning around having this functionality was to be
able to return instantly when n process runtime agents have been found would minimize the lookup calls to the DHT. This behavior could be modified to always return all the values in the layer that hit n values, which would mean that all the closest values would always be returned.

2.5.3 SpiralBoxExpand

SpiralBoxExpand is a search method, which involves checking in an expanding spiral pattern around the center. This implementation is a counter-clockwise spiral and the picture Figure 2.11 shows that when it moves an odd distance it walks North followed by West and on even distances it walks south and east, forming a counter-clockwise outgoing spiral.

The SpiralBoxExpand method gives a reduced amount of calls to the DHT lookup function to find n process runtime agents since as soon as it finds n process runtime agents the function will terminate. However, it has the same problem (which was pointed out in the LayerExpand description) where it can exit before a full layer\(^2\) have been searched. Letting some process runtime agents that could be up to 20km closer, which is due to the same reasoning that was given for the LayerExpand function.

\(^2\)full layer: an encircling of the cell where the search started.
Figure 2.11: The figure shows how the SpiralBoxExpand this is done by starting in the center and moving north and west on odd distances and south and east on even distances.

Compared to the LayerExpand method, SpiralBoxExpand does not require a list of already visited cells to be maintained because it will never visit the same geoash cell twice. The fact that the search is done in an outgoing spiral pattern means that it will not need to keep track of which cells it has already visited, making it allocate less memory runtime compared to LayerExpand, but it also prevents unnecessarily geohash fetches that would have been discarded because they already have been searched like in the other solution. As in LayerExpand, SpiralBoxExpand can also be modified to ensure always giving the closest solutions by making sure that full layers always will be searched.
This chapter describes the implementations of the LayerExpand and SpiralBoxExpand and how the functions interact with the DHT to store and search for process runtime agents. The implementations are described with the help of some code examples, flowcharts, and block diagrams. It also covers the proof-of-concept Google Maps demo that can be used to do test queries for process runtime agents in a web browser.

3.1 Kademlia

Ericsson has developed a DHT called Bitverse based on Kademlia. Bitverse provides the following functions: Store-, Merge- and FindObject to store and search for objects in the DHT. Store and Merge are mostly used to set the DHT up before it is used, but the FindObject is called for each geohash cell to check if something is saved at that locations. When a lookup is performed on the DHT, FindObject will return the object if there is one stored while if nothing is stored on that position nothing will be returned, and the search will continue. When FindObject is called Kademlia checks the DHT if the object exists this will take up to $O(\log(n))$ time.
3.2 LayerExpand

LayerExpand uses one of the existing function \textit{Neighbors()} that was ported from a python geohash library. A downside of using the Neighbors() functions for the expanding of the search is that a map of strings and booleans have to be maintained in order to make sure that the same cell is not visited more than once during the search, this is due to the fact that the Neighbors() method checks all the 8 surrounding cells and it is called for each cell in every layer.

Figure 3.1 shows how LayerExpand searches for a process runtime agent. The method works by using the \textit{Neighbors()} function to decode and find the surrounding position and decode them back into geohashes, which are then stored in a list. In the following step, the geohashes in the list is checked in the DHT. By maintaining a list of what is going to be the next layer that is to be searched, the function can recursively call itself when the current layer has been thoroughly explored. Figure 3.2 is a box diagram that show how the interaction between the different parts of the system. Listing 3.1 is a pseudo code block that show how the search is done while maintaining the nextLayer list.

\begin{boxedverbatim}
Listing 3.1: LayerFindN function

func LayerFindN(hashcode string, goal int, nodeIds [] string, currentLayer [] string, checked map[string]bool, node *dht.Kademlia) (error, ret [] string) {
    for j := range currentLayer {
        tmp = Neighbors(currentLayer[j])
        for i := range tmp {
            if !checked[tmp[i]] {
                GeoHashCellValues(tmp[i], node)
                checked[tmp[i]] = true
                nextLayer = append(nextLayer, tmp[i])
            }
        }
    }
    if goal met {return}
    _, layer := LayerFindN(hashcode, goal, nodeIds, nextLayer, checked, node)
    return layer
}
\end{boxedverbatim}
3.2. **LayerExpand**

**Figure 3.1:** Flowchart for LayerExpand shows how the search for a nearby process runtime agent is done by the LayerExpand method.

**Figure 3.2:** Block Diagram for LayerExpand show how the different parts interact with each other when the algorithm is used.
3.3 SpiralBoxExpand

Figure 12 shows how the searching for a process runtime agent is done with SpiralBoxExpand. The method will start by decoding the geohash into latitude, longitude, latLength and lonLength where latLength and lonLength are the lengths of the sides of the rectangle. The algorithm then traverses in the current direction by increasing or decreasing the latitude and longitude values; these modified values is encoded back into geohash and stored in a list. In the next step, the geohashes in the list is looked up in the DHT. If a node is found at one of the locations, the process runtime agent is returned. Otherwise, the last value in the list will be sent back to the decode phase, and the search continues. Figure 13 is a box diagram showing how the SpiralBoxExpand inside the Geo-process lookup management package interact with the other part of the system like the DHT lookups in Kademlia. Listing 3.2 is a pseudo code block of SpiralBoxExpand and spiralBoxJump, which is a helper function that does the directional walk.

```
func SpiralBoxFindN(hashcode string, goal int, node *dht.Kademlia) (err error, ret []string) {
    for i := 1; true; i++ {
        if i%2 == 1 {  //Check north followed by west
            newHash, list = spiralBoxJump(newHash, "N", i)
            newHash, list = spiralBoxJump(newHash, "W", i)
        } else {  //Check south followed by east
            newHash, list = spiralBoxJump(newHash, "S", i)
            newHash, list = spiralBoxJump(newHash, "E", i)
        }
        for j := range list {
            _, nodeIds := GeoHashCellValues(list[j], node)
            if goal met {return}
        }
    }
    return
}
// Simplyfied for only one direction
func spiralBoxJump(hashcode, direction string, jumpLen int) (newHash string, ret []string) {
    lat, lon, latLength, lonLength := decode_c2i(hashcode)
    if direction == "X" {  //Check a direction for jumpLen step
        for i := 1; i <= jumpLen; i++ {
            newHash = encode_i2c(lat-i, lon, latLength, lonLength)
        }
    }
    return
}
```
3.3. **SpiralBoxExpand**

![Flowchart for SpiralBoxExpand](image1)

**Figure 3.3:** Flowchart for SpiralBoxExpand illustrates how it searches for a nearby process runtime agent.

![Block Diagram for SpiralBoxExpand](image2)

**Figure 3.4:** Block Diagram for SpiralBoxExpand shows how the interactions are done between the different parts of the program when the algorithm is used.
3.4 Proof-of-concept prototype

Figure 3.5 illustrates a web interface to visualize how the process runtime agents are placed on an area. An area in which it is possible to look for services (for example somewhere to run a software container). The map will show the process runtime agent that was closest to where the container was positioned on the map; this will be illustrated by standard markers showing the different process runtime agents on the map. A container icon is used for where the container is located and a circle that got the radius of the distance between the container location and the closest process runtime agent illustrating the searched area.

![Demo screenshot](image)

*Figure 3.5: Demo screenshot shows the process runtime agent in the area, a deployed container, and where the algorithm have found the closest process runtime agent in Stockholm.*

A circle is used instead of a square, which is a better representation of the searched areas because the implementation is done using Google Maps. And since the Google Maps application programming interface (API) [24] for circles are defined by a center
position and a radius while the squares are defined by the south west and north east corner. Calculating the coordinates of the corners is very clunky when the geohash cells will have a different size depending both on the non-linearity of the longitude/latitude system and cells becoming less square further from the equator. The geometry of the rectangles are also changed depending on if the geohash precision is odd or even due to how the 8x4 current precision rectangle is oriented (lying or standing) in one of them they are closer to a square and the other they are closer to a 2:1 rectangle.

### 3.4.1 Proof-of-concept implementation

The visualization map is made using Google Maps 3.0 API with HTML and Javascript. The communication with the Golang backend is done using JavaScript Object Notation (JSON) over WebSockets. The JSON Object is built in the backend by using the `Marshal()` function that is included in the encoding/json package. The `Marshal()` function takes a defined struct type as an argument, and it will return that struct in JSON format. The JSON which is sent consist of a list of process runtime agents, the closest process runtime agent, the target location, (which in Figure 3.5 is visualized as a container) and the distance between the target location and the nearest process runtime agent (which is used to draw the circle showing the rough area that have been searched).
The Proof-of-concept prototype was helpful to make sure that the closest process runtime agent was found. Making sure the closest was found something that was harder at first but with the help of the visualization on the web page, it became easier. With the initial search location being placed on the map along with the center of the search and a circle being based on the center of the search and the distances to the chosen process runtime agent. The map made it possible to confirm that the closest process runtime agent was found but if any other process runtime agents would be located within the area of the circle, which means the closest process runtime agent was not found.

The experiment that has been conducted, aimed to test and measure how fast the algorithm can find a process runtime agent. The metrics examined was the number of DHT calls vs. distance in kilometers, time in seconds vs. distance in kilometers, and time in seconds vs. number of layers from the center that have been searched.

### 4.1 Experimental setup

The testing was mainly done by running tests on a setup with a pre-generated set of nodes in a DHT containing 100 nodes simulated on a single machine, which hardware is specified in Section 4.1.1. These process runtime agents were generated and placed on the globe by incrementing both the longitude and latitude values making a big square of nodes close to the equator. The test was done by first placing a container on one of the edge nodes and after that got moved further and further away from the initial location.
Figure 4.1: The screenshot illustrates how the 100 nodes is placed and the initial position of the container. The dashed arrow illustrate in what direction the position of the container is moved, as the distance to the node cluster increases.

This setup made it possible to measure the amount of DHT calls, time, and helper function calls. The collected data is then used to make various plots that are used in this paper. The plots was a good way to see a difference in performance between the algorithms.

### 4.1.1 Hardware specification

The hardware specification of the computer used to run the experiment is described in Table 4.1.
4.2 Result

In this section, the results of the experiment described in Section 4.1.

Figure 4.2: Spiral vs Layer DHT Calls Comparison.

Figure 4.2 shows the comparison of the performance of the two methods in terms of the number of calls made to the DHT to find a process runtime agent at a growing distance. The graph in Figure 4.2 was not updated to a 100 nodes version because it only measure the amount of calls to the DHT and will not differ depending on how many nodes is in the DHT.

Figure 4.3 shows the comparison of the performance of the two methods in terms of the time it takes to find a process runtime agent at a growing distance. The plot is based on data collected from a DHT containing 100 nodes; this gives a better representation when using timed tests because it will get the time effects from the messaging inside the DHT.
Figure 4.3: Spiral vs Layer Time Comparison 100 nodes.

Figure 4.4: LayerGeoCalls the amount of calls to LayerExpand helper functions.

Figure 4.4 shows a plot of the number of calls to the internal geo functions, which translates the geohash and finds surrounding geohash cells that is made when the LayerExpand works. If we compare the number of calls made in this plot to the amount of DHT calls that was made by LayerExpand in the plot from Figure 4.2, it can be noted that the
amount of calls for the same distance is about eight times as high in the LayerGeoCalls plot. Indicating the extra work in the background that LayerExpand perform to find a process runtime agent.

Figure 4.5: SpiralBoxExpand vs LayerExpand, time per layer.

Figure 4.5 illustrates how long it takes the algorithm to search a specific amount of layers around the target location. If the geohash precision is set to be around one square meter, it will take the algorithm around 17-18 seconds to search a 50x50m area or 2500 DHT lookups. However, if the precision is 5 square meters it would take roughly 1 second and 100 DHT calls to cover the same area.

It should be noted that this search systems performance is dependent on the distance to the initial search location. Meaning that if a certain area is searched and the density of nodes is either high or low the search will be performed at roughly the same time. The only difference between a low and high-density area would be the copying of metadata for the nodes managed by the process runtime agent. That means that the high-density example would contain information about more nodes within the process runtime agents area. The searches for the geographical area would still be performed in as many calls to the DHT.
In addition to the results of the experiments in the previous section. This section will cover an inspection and analysis of the code and how the functions perform in theory. It also looks at some performance implications on DHTs and geohash precision and look into some of the limitations of the proposed solutions.

5.1 LayerExpand

The time complexity, in this case, is how many geocells or geohashes have to be checked before a process runtime agent is found and is expressed by Equation 5.1.

\[ n = \text{distance from the center} \]
\[ l = \text{currentLayer} \]
\[ l^*8 + 8 \text{ (size of currentLayerSize) } (l=0, \text{currentLayerSize} = 8) \quad (l=1, \text{currentLayerSize} = 16) \]

\[ \sum_{l=0}^{n} 8l + 8 = 8 + 16 + 24 + \ldots + 8n + 8 = 4l^2 + 12l \rightarrow O(n^2) \quad (5.1) \]

As Equation 5.1 shows the time complexity for LayerExpand is \( O(n^2) \) this is because the summation needs to be done \( n \) times to reach \( 8n + 8 \). The values in the equation come from inspecting the code of the function in Listing 3.1 and a list of time consumption of
the different parts is given below. But the Layer function also have to maintain a list that keeps track of what geocells have been visited, this memory consumption is \( O(n^2) \). The memory consumption looks a lot worse than it is because it only stores each node once and since it is a square, and \( n \) is just the distance from the center to the target and since an equal amount of \( n \) cells is searched around the middle until the target is found the memory used will be \( 2n \times 2n \) since it also checks \( n \) steps away from the target.

Neighbors/Expand: \( O(1) \) only check a constant number of blocks 8 or 9
LayerFindN: \( O(j) \) looping through the currentLayer
LayerFindN: \( O(8) \) 2nd for / if statement checking all neighbours
GeoHashCellValues: worst case lookup \( O(\log(n)) \) in DHT for the specific geohash.

### 5.2 SpiralBoxExpand

The time complexity for SpiralBoxExpand comes from how many geocells or geohashes around the starting location that will have to be visited to find a process runtime agent.

\( n \) is denoted by the distance to the found process runtime agent.
\( i \) is the number of half rotations around the starting location.
\( j \) is the length of the current side that is being checked.

This is expressed in Equation 5.2 which shows that the time complexity of the method is \( O(n^2) \) for the number of lookups that is done towards the DHT in order to find a process runtime agent at the distance \( n \), \( i \) can directly be transformed to \( n \) in \( \text{bigO} \) notation because \( i \) is only \( 2n \).

\[
2 \sum_{j=1}^{i} 2j = 2(2 + 4 + 6 + \ldots + 2i) = 4i^2 \rightarrow O(n^2) \quad (5.2)
\]

When analyzing the code itself in listing 3.2 the time complexities of the different parts of the function is as follows.

encode_i2c / decode_c2i: \( O(1) \) only loop for constant amounts.
SpiralBoxFindN: main loop \( O(i) \) while true until found.
SpiralBoxFindN: tmp loop \( O(2j) \) spiralBoxJump looked on \( i \) positions but is called twice.
spiralBoxJump: side length \( O(i) \) where \( i \) is length of the sides.
GeoHashCellValues: worst case lookup \( O(\log(n)) \) in DHT for the specific geohash.
5.3. Simulation

This section presents a simulation study of the resource utilization the different parameters would have on the performance of the system. The simulation has been done in the form of a formula that calculates the maximum data traffic across the system given a certain set of parameter values; the formula is shown by Equation 5.3. This formula is used to generate a few plots to show the change in data traffic when increasing the different variables. The variables in the equation correspond to:

- **NumberOfUsers**: The number of users that is using the system at a given point in time.
- **NumberOfNodes**: The number of nodes that is part of the DHT
- **AvgDistance**: The average distance in layers the algorithm have to search before a process runtime agent is found.
- **PacketSize**: The size of the packet that is sent within the DHT when doing the key lookup. This parameter is set to 1KB for all the tests.

\[
\text{DataTraffic} = \text{NumberOfUsers} \cdot \log_2(\text{NumberOfNodes}) \cdot \text{AvgDistance}^2 \cdot \text{PacketSize} \quad (5.3)
\]

The plots in Figures 5.1, 5.2 and 5.3 show how the different parameters affect the data traffic of the network as the parameters gets increased. Figure 5.1 shows that increasing the number of users will increase the data traffic in a linear manner. By inspecting Figure 5.2 it can be seen that increasing the number of nodes will result in a logarithmic increase in data traffic, which comes from the number of messages that needs to be sent. Figure 5.3 shows that the avgDistance grows at a rate of \(2n^2\), this is not really surprising given Equation 5.3 since avgDistance is squared. The \(2n^2\) growth is caused by the fact that the algorithm got no notion about in what order to search, so it has to search in all direction from the initial position. The plots give a good idea of how increasing the different parameters will affect the load on the network.
Figure 5.1: Plot shows increasing NumberOfUsers from 0 up to 1 million with increments of 1000 users, while keeping avgDistance constant at 10 layers and NumberOfNodes at 100000 nodes.

Figure 5.2: Plot shows increasing NumberOfNodes from 0 up to around 1 million with increments of 1024 nodes while keeping AvgDistance constant at 10 layers and NumberOfUsers at 100000 users.
Increasing the number of nodes is not a big issue because of the logarithmic behavior of Figure 5.2, which means that the number of nodes can be increased significantly, for example, going from 1 thousand to 1 million nodes will only increase the number of lookups from 10 to 20. In Figure 5.2 the number of nodes is incrementing by 1024 nodes per step compared to 1000 in Figure 5.1. The reasoning behind picking 1024 for this value was to make sure the simulation results consisted of even Integers and since $\log_2(1024) = 10$, 1024 was chosen. The linear behavior of Figure 5.1 will increase the system load depending on the number of users, which means the scaling should be manageable. The big issue from this approach can be seen in Figure 5.3 the $2n^2$ growth of this parameter is the most problematic. If a geohash precision of one square meter and the average layer distance is 50, it will cause a big load when all the 100000 users search at the same time. It should be mentioned that there might not be very likely that so many users search for items that many layers away at the same time and the number of likely layers that need to be searched for a certain area can be modified by changing the precision to cover a bigger area with fewer layers.
5.4 Spiral and layer comparison

This is a comparison of SpiralBoxExpand and the LayerExpand functions in how they perform when it comes to how much load they put on the DHT over different distances. And how they do their internal lookups to find the geocell blocks.

When inspecting the spiral vs. layer DHT calls comparison graph in Figure 4.2, it can be seen that both solutions follow each other quite close with a slightly higher amount of calls for the Spiralbox function. However when the test is run with a higher amount of nodes in the distance vs. time comparison for 100 nodes graph shown in Figure 4.3 SpiralBoxExpand performs better.

One downside of the layer function that is not present in the spiral function can be seen in the LayerGeoCalls graph in Figure 4.4. LayerGeoCalls illustrates how many times a geocell block is looked up during the search. The reason there is no graph for SpiralBoxExpand is because the function only does as many geocalls as it does DHT lookups. The extra geo calls that were mentioned in the explanation of LayerExpand and the fact that it needs to keep track of what geocells that have been visited already, which will give a slight increase in memory consumption. The memory consumption is a factor that might be worth taking into account when choosing which function to use. Considering how close they lay in time consumption either one could be a good option however SpiralBoxExpand has the advantage of not having to keep track of visited positions during runtime.

It should also be noted that up to around 300 km distance or 0.5 seconds lookup time the difference between the two options is below 50ms and this difference shrinks as the distances do. With this in mind and given that the DHT is fairly well populated with process runtime agents at different geographic positions, the lookup time would be very close between LayerExpand and SpiralBoxExpand.

5.5 Distributed Hash Table

Distributed hash tables got an average time complexity of $O(\log(n))$ for search, insertion, and deletion while the worst case is $O(n)$ for the same operations. For these solutions, the average performance was used, and the DHT lookups will work at $O(\log(n))$. The DHT average performance is the working condition as long as all the process runtime agents are not all located extremely closely together. Alternatively that the geohash precision, determined by the length of the geohash string, is poorly chosen. For example,
the tests that were run was using a precision of 4 which gives squares size around 400 km² while increasing the precision gives smaller of around areas while decreasing gives a larger area. So to make it work well a good precision level have to be chosen cause if the precision is too high, the amount of lookups will increase significantly. For example, increasing the precision from 4 to 5, results in a cell size change of 20 x 20 km to 2.4 x 2.4 km. That would greatly increase the number of geohash cells the search algorithm have to visit to find the same process runtime agent. While if choosing a precision that is too low, for example, 1 would mean that the "U" geohash cell, being \( \frac{1}{32} \)th of the entire world. The area would cover the better part of Europe which in turn will have a high chance of approaching the worst case of \( O(n) \). The poor performance would be caused by that a lot of nodes being served by the same process runtime agent, and all the nodes will have to be looped through to find the closest nodes.

5.6 Geohash precision

The current solution is based on geohash using a base32 system that makes each step in changing precision quite big, for example, precision 3 is ±78 km, 4 is ±20 km and 5 is ±2 km. It might be possible to modify the functions to support another base instead so, for example, base2 or base4 instead of base32. If this works, it will provide the possibility to customize the size of the cells more depending on the geographical distribution of the process runtime agents. If it is known that the process runtime agents are going to be scattered over a large area choosing a lower precision would be preferable. But if they are located in a specific area a higher precision would be better, so all process runtime agents are not placed under the same key.

5.7 Limitations

This section mentions some of the limitations and problems that could be an issue when using the system for storing and searching for devices using this approach. These issues might have to be considered for further work when especially the verticality problem mentioned in Section 5.7.2, can cause a lot of devices to share the same location.
5.7.1 Corners and distances

One of the problems with using geo-process lookup management method of finding the closest nodes is the fact that, for example, the case of perfect square cell blocks the distance to the very corner of the area is going to be \( d = \sqrt{x^2 + x^2} \) from the center. Making the corner point to be 41.4\% further away from the center compared to the edge of a cell completely perpendicular to the center cell. For example, \( \sqrt{5^2 + 5^2} = 7.07 \), which means that if layer=5 the farthest corner of the corner cell will be at the same distance from a perpendicular edge of layer 7, this problem is illustrated in Figure 5.4.

\[ \text{Figure 5.4: The figure illustrates how the distance to the corner of the active search area extends to layers further out. And therefore covering areas that were not covered by the search making it possible to miss closer nodes when a node is found in the corners. The dashed arrow that is projected on the vertical line has the same length as the full arrow } v_1. \text{ The circle shows the area that is at the same distance or closer to the corner that } v_1 \text{ is pinpointing.} \]

The mentioned error is quite significant but could be circumvented by extending the search for 40\%, which would reduce the efficiency. What this means, in reality, is that if the search is not extended further after the first hit, there is a chance of missing process runtime agents that could be located up to 40\% closer compared to the one that was found. The example is when looking at perfect squares, if instead a 2:1 rectangle is inspected the error will be even bigger for lower layer values but as the number of layers increases the error will approach 40\% for this case as well.
5.7. Limitations

5.7.2 2D solution for a 3D world and its performance implications

The approach used is as mentioned in Section 2.2 is based on a 2d-grid, which makes living or offices in skyscrapers an issue when there could be a lot of devices place on the same geohash position. Especially in the case of multi-level apartment buildings where the layout for each level is similar and the same kind of IoT devices is likely to be placed on the same spot (e.g. kitchen white wares like refrigerators or microwave ovens) this could make it hard to differentiate between your own devices and your neighbours living above and below. This particular problem with the users who owns IoT devices could probably be solved by having the users have some synchronization process with their own devices. But the problem persist when searching for something with multiple levels if the user is on the ground level and finds something on the top the distance might be shorter, but the device will be irrelevant to that user, which could lead to false positives. Another possible solution for the multi-level problem could be Content-Addressable Network (CAN) [25], which is a DHT that have support for multi-dimensioned coordinate spaces. CAN have not been tested in the thesis when it involves changing the DHT infrastructure for the system and it was presented as an option late in the process and is therefore mentioned as an option to put into future work.
The aim of the thesis was to find a way to be able to deploy services and devices on a distributed platform based on geographical location, which is something that has been accomplished with the help of the geo-process lookup management system. Where a user can store, the services depending on what area the service is located. The area searches can both work as a primary search parameter, but would most likely be at its best when used in conjunction with other search parameters. Where the geographical search serves as an initial filtering process before other requirements such as required CPU, memory or some other parameter is taken into consideration.

While the proposed solution geo-process lookup management finds the closest process runtime agent in a particular area that have been searched, it does have some downsides, some of these downsides were described in Section 5.7. As illustrated by Figure 5.4, the corner cells in each layer are further out compared to the cells directly perpendicular to the cell where the search started. Therefore, if finding the closest process runtime agent is the goal, additional layers, after the first one was found, will have to be performed. Another issue is the case with multi-level apartment buildings where multiple IoT devices of a similar type could be placed under the same geohash key. This is caused by the fact that the apartments have the same layout with for example the kitchen being on the same spot when seeing from a top down 2d-view.

When looking at the results from the test cases, it can be seen that as the distance from the initial location increases the time and number of calls to the DHT increases at a rate of $2n^2$. The problem with the $2n^2$ growth with increasing distances is further investigated in the simulation section where the data usage scales very poorly with increasing distances from the target location, given this, the geohash precision should be chosen carefully to
make sure that the number searched layers does not go too high.

It should also be noted that depending on the geohash precision the system could be used for different kinds of tasks. With a high precision that grants around one square meter area, it could be used for IoT device discovery in an apartment. While a lower precision of maybe a few square kilometer areas could be used to deploy something like a video conference server in a location that would give the highest performance benefits for all the participants.

Increasing the number of nodes in the DHT network only increases the data usage. It should be possible to be able to use the same DHT network for multiple geohash precisions e.g. some process runtime agents is stored with a precision corresponding to one square meter while others to bigger areas. If this works, it could also make it possible to develop a system where devices could be stored in multiple precision layers. If a search for a bigger area has too many hits on a certain geohash, the precision could be increased, to reduce the number of search result down to a more manageable size. Multiple geohash lengths have not been tested however and is left as further work as it was outside the scope of the thesis.

6.1 Possible applications

As mentioned in Section 1.4, the proposed solution has many applications. For example, it can be used to deploy a video conference server or other time-critical real-time systems (running in containers) geographically close to the end-users, assuming there is a relationship between IP delay and geographical distance. Another use case is Augmented Reality (AR), where an application needs to be responsive and give information about object a user is interested in. The AR service could be done by letting the building or object that the user wants to get AR information about have the information stored in their information service stored in the DHT. The service could be found by the phone or AR device, by looking for AR services within a certain radius around the users to give extra information about the surroundings. A service like this might even make the user notice something they otherwise would have missed.

Additionally, there could be interesting IoT use cases. For example, in an intelligent transport system, a car could ask a crosswalk IoT device for possible hazards, e.g. ask if there are any pedestrians that might have been missed. Providing the driver with an advance warning of possible dangerous situations and give the driver the chance to take appropriate actions to the situation.
6.2 Future work

As mentioned earlier in this chapter, it could be useful to be able to store process runtime agents in multiple precision layers within the DHT. And research on when it would be appropriate to make additional searches within a certain geohash to limit the set of results.

In addition, as mentioned in Section 5.6 it could be useful to make a geohash system using a different base, instead of base32. A system like this would provide a bigger variety in the precisions that could be used, since using a lower base would only increase the key string length. Kademlia uses a 160-bit key ID. Therefore, a geohash on binary base could give good maximum precision while having a lower difference between one precision and the next one without running into problems with the key length.

The verticality problem that was discussed in Section 5.7.2 and more specifically the possibility of using CAN to solve the verticality problem is also something that is left for further work.


