Group creation in a collaborative P2P channel allocation protocol

Identifying connected groups of access points

Hans Jørgen Furre Nygårdshaug

Thesis submitted for the degree of
Master in Informatikk: programmering og
nettverk
60 credits

Institutt for informatikk
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017

Group creation in a collaborative P2P channel allocation protocol

Identifying connected groups of access points

Hans Jørgen Furre Nygårdshaug

Contents

Chapter 1

Disposition

1.1 Introduction part

- Introduction to the wifi interference problem

- Taking a step back and looking at other attempts on solving the problem

- Begin presenting Torleiv and Magnus work, the idea, and maybe the p2p protocol.

1.2 Main thesis part

1.2.1 Group allocation algorithm

- The problem.

- Possible solutions? References.

- Complexity, out of scope for thesis. Assume problem is solved.

1.2.2 The algorithm itself

- Elaborating on the problem, introducing the first algorithm suggestion.

- Explain simulation data creation with stochastic uniform distribution.

- Show how the group creation algorith was created, design decisions (iterations etc).

- Results with visualizations through the visualization tool.

- Evaluate results and consider improvements. How will this work in the wild?

- Introduce SSB data, the data format and why it is relevant. How is the tool made.

- Same procedure with result visualization and result evaluation. Do we still need improvements?

- Introducing Wigle as data source. Show results on map?

1.3 Concluding part

- Have we created a meaningful algorithm that can be implemented in hardware?

Chapter 2

Introduction

2.1 Channel allocation

To deal with the problem of channel allocation we will think of an AP as a vertex in a graph. When an AP scans its radio it can hear the strength of all nearby wireless networks measured in dBm (decibel milliwatts). This decibel value will be the value of the edge between one AP to another. With a graph expressing the wireless network topology, the problem of optimally distributing channels between APs boils down to a graph coloring problem. The number of colors in the color problem, represents the number of non-overlapping channels in 802.11. Exactly how an algorithm can be designed to optimally distribute channels within the interfering topology is out of the scope of this thesis. However we can define some invariants that has to be true for such an algorithm to work:

1. All APs has to run the same algorithm

2. All APs must run the algorithm on the same connected group

3. Because of the complexity of the problem the algorthm must solve, the number of APs in the connected group can not be too big

Point 1 is trivial to solve or mitigate, as only APs running the algorithm will actively participate in the channel selection. A simple way to make sure that the same algorithm is used, is by having a software version that is consistently checked with the other APs in the connected group.

Point 2 and point 3 is will be the main focus of the rest of the master thesis, as these are not so easily solved.

We can define a wireless topology graph as a set of wireless APs that are grouped together and share information about their neighbours and interference levels. This set is what will now on be referred to as a *connected group.* All members of the connected group will be considered when running the channel assignment algorithm. For the connected group to have an actual impact on the quality of a network connection, it has to consist of nodes that normally disturbs each other substantially.

An ideal example of a connected group is an apartment builiding. The channel allocation protocol lets APs share information about who-disturbs-who the most in the building. Then each AP can run the channel allocation algorithm.

Because they run it on the same graph, every AP will find the same optimal channel distribution throughout the building, and then switch to the correct channel.

Even though an apartment building is most likely an optimal delimination of a connected group, in reality creating such a group is a bigger challenge. As the whole channel allocation protocol is based on decentralized peer-to-peer technology, and no centralized server with access to demographical and geographical divisions exists, the protocol will have to discover suitable connected groups on its own. Moreover, when the group is created the protocol will have to replicate data so that all participants of the group has all the data required to perform channel allocation. It will also need a way to make sure that the image of the current group is consistent within all APs in the connected group.

Chapter 3

Connected groups

To enable collaborative channel allocation, it is important for every AP to know which APs it is collaborating with. One way to share information about who collaborates with who, is to let the access points group together. Information relevant for channel allocation can then be shared freely within the group, between APs.

We will proceed by looking at some of the requirements for a group creation algorithm. It should work decentralized in a distributed fashion. Hence, not only does the APs have to be imposed group membership, but they also have to be able to create and definine meaningful groups on their own. Later we will propose an algorithm to create groups, and then evaluate computed groups based on the algorithm.

3.1 Definition

A connected group is a set of access points that are within close geographical range. The group should consist of APs that interfere with each other when their channels are overlapping, so that overlap can be avoided with a channel allocation algorithm run within the group. Not all APs in the connected group will necessarily be able to hear each other on the radio directly, but all nodes should be able to hear each other through a transitive relation (neighbour of a neighbour, etc.).

3.2 Requirements and assumptions

We are going to be looking at how APs can organize themselves without a central coordinator, but it should be noted that the creation of groups could potentially be easier if a centralized entity had the full picture of APs and their radio readings. It could then combine geographical information along with AP readings to create consistent groups. We will consider the two options against each other later on, but proceeding ahead with P2P group creation as the main concern. Essentially this means that an AP has no other knowledge of the world than the other APs it can hear on its radio. For the sake of creating and evaluationg a group creation algorithm, we will assume a number of things that will have to be in solved at a later point:

1. All APs visible on any AP's radio also runs the group creation algorithm.

2. If an AP can see another AP on its radio, it also knows how to directly contact the AP (e.g. via TCP).

3. All APs in a group are completely synchronous, and always have an equal image of the state of the group.

3.3 Proposing an algorithm

With the definition of a connected group in place, and the assumptions and requirements accounted for, we can begin proposing an algorithm.

Henceforth we will for the sake of simplicity be referring to APs that are running the group algorithm as *nodes*.

Each node begins by identifying itself as a member of a group that only contains itself. Let us call it group a. The node shares all of its radio readings with the rest of the members in group a. Group a looks at all the radio readings of every member, and picks the node with the highest dBm value to contact. Of course in the beginning there is only one node in group a, so this node's radio readings alone will decide which group to merge with.

The neighbour node that has the highest dBm value is in group b, and this is the most crucial node to collaborate with. Hence the group a merges its own group together with the group of neighbour b. This happens in the following way: the members of the two groups exchange information about all their member nodes and their radio readings. The data is now identical for all the members of both groups, and they can make identical choices, hence they are now in the same group.

If the number of members n exceeds a predefined threshold $memberThresh$ after a merge, the group should begin kicking out members from the group, starting with the node that influences the rest of the group the least.

To prevent the group from oscillating between kicking out members and rejoining members, once it reaches its maximum size it should be locked for further merges.

3.4 Simulation data

3.4.1 Requirements for data creation

Before implementing and testing the group creation algorithm, we need to gather usable data to perform testing on. The data consists of the topology size, and where on the topology a node is located. There are two ways we can get this knowledge.

The first way is creating our own data, and either assume the location of nodes, or randomly distribute them on a topology.

The second way is getting real world data from a data source, where the topology would represent a section of a map, and the nodes would be APs, most likely located in the households of the map. Either way, the represenation of the topology should be the same.

The testing data should be a set of nodes that has two coordinates x and y on a two-dimensional grid. Henceforth we will call the grid the node topology,

or just the topology. When the nodes are placed on the grid, we can compute which nodes it can hear on the radio, and add these to the nodes neighbour list. This list contains the names of all the nodes it can hear, and how loud it is heard measured in $-dBi$. Additionally the following parameters should be variable depending on each test scenario:

- Topology size, variable width of x- and y-axis.

- Number of nodes

- Minimum distance between nodes (in meters)

- Minimum loudness measured in $-dBi$ for a node to account another node as a neighbour

3.4.2 Program design

The topology generation program consists of two main functionalitites.

The first functionality is being be able to create a topology and generate nodes which are uniformly and randomly positioned on the network topology. The size of the topology, the number of nodes and the minimum distance between nodes are properties that can be given as input arguments to the program.

The second functionality is computing which nodes can hear each other. We are assuming all nodes are transmitting with equal strength, and that the environment is flat and obstacle free. All the neighbouring nodes that can be heard by a node, is added to its list of neighbours, and it stores the $-dBi$ value so it later can be shared with the group.

The resulting program, written in Python 3[1], contains an importable *topology class*. This way, for further testing we can use different data sources to get the positions of nodes, and only let the topology class compute the list of neighbours.

The interference levels between APs is calculated by iterating through all the nodes. For each node N we record its x and y position, and then start a second iteration through the nodes. For each node n in the second iteration we calculate the distance d in meters between N and n using Euclidean distance. Knowing the the distance between the nodes, we can use the formula for free space path loss [2] to compute the $-dBI$ values.

For the sake of cross compatibility with other applications, the result of the computation represented in json. A topology consists of as many JSON node-objects as there are nodes.

3.4.3 Data output and visual representation

The following illustrates the node structure, and is an example of how a node with two neighbours will look:

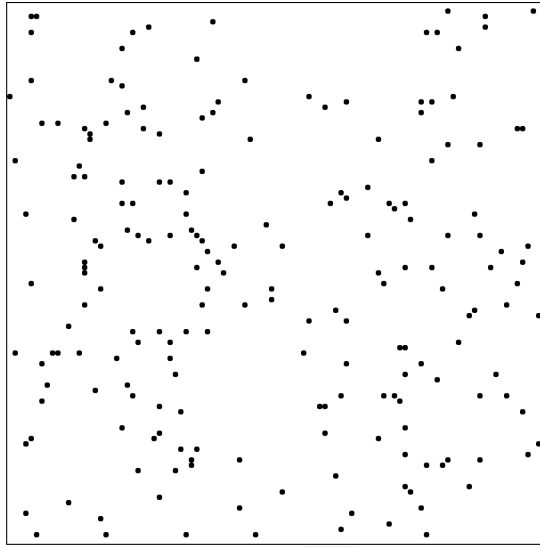Figure 3.1: Generated topology with random, uniform distribution

```
 1          1: {
 2                  "posX": 100,
 3                  "posY": 100,
 4                  "ssid": "NODE1",
 5                  "neighbourCount": 2,
 6                  "neighbours": {
 7                          0: {
 8                                  "ssid": NODE2,
 9                                  "dbi": -77.23
10                          },
11                          1: {
12                                  "ssid": NODE3,
13                                  "dbi": -79.52
14                          }
15                  }
16          }
```

We can run the program in the following way

```
./GenerateTopology.py -n 200 -w 100 -h 100 --space 10 --dbi 85
```

Which instructs the program to create a topology with 200 nodes. The topology should be 100 by 100 meters large, and there should be at least 10 meters between each node. The *dbi* parameter makes sure that only nodes which can be heard with a $-dBi$-value of -85 or larger should be considered neighbours.

The output is a 3.8 MB large file containing the resulting topology-data in JSON.

By writing a simple JavaScript browser application meant to interpret the output, we can visually represent the nodes on a grid, and the result will look like what can be seen in figure 3.1

9

3.5   Algorithm implementation

The algorithm was implemented in Python 3 [1], designed to run on the output from the data generation program.

For this computation, we are assuming that all members of a group are precicely synchronized and the group members are consistent across all nodes. With that assumption, we can be sure all nodes will make the same decisions.

A group object can then be responsible for keeping track of current members, represented as node objects. The group object will also look through the radio readings of all its member nodes to find new member candidates, handle merges, locking the group and kicking out extraneous members, all according to the algorithm description in section 3.3.

To make it easier to evaluate the results, the computation is done sequentially and in iterations. This is a simple way to make sure the same groups are created for the same topology every time, and it will make the evaluation process easier.

The group creation program outputs a JSON-file containing the groups and their members for every iteration. The first iteration being every node having a group for their own, and the last iteration is the resulting groups.

3.6   Results

3.6.1   Uniformely distributed nodes

We will look at how groups were created in different topology scenarios. All topologies presented in this section was created by the topology generation program, but with different input parameters. Groups are distinguished by node color, where nodes of the same color represents members of the same group.

Scenario 1

Computed with 200 nodes with a maximum of 128 members in each group.

As can be seen in figure 3.2 the algorithm divides the nodes in two sections. For clarity, a divisive line has been drawn around each group, in case colors are not available. When two major groups merged, the biggest groups surpasssed 128 members and began kicking out members. The excess members formed the black group at the bottom.

Scenario 2

Computed with 200 nodes with a maximum of 10 members in each group.

The result of this computation, seen in figure 3.3, is a little less obvious. The groups are again distinguished by different color, but for clarity we add a gray connecting blob for nodes in the same group. Also blobs connected with a line are in the same group.

It is worthy to take notice that one group is especially scattered around the graph. At first eyesight, it looks like an algorithm deficiency, but the reason is quite simple: when nodes are kicked out of a group during a merge, they will connet to other nodes that belong in a group where n has not yet reached *memberThresh*. When this have happened a couple of times, everyone has
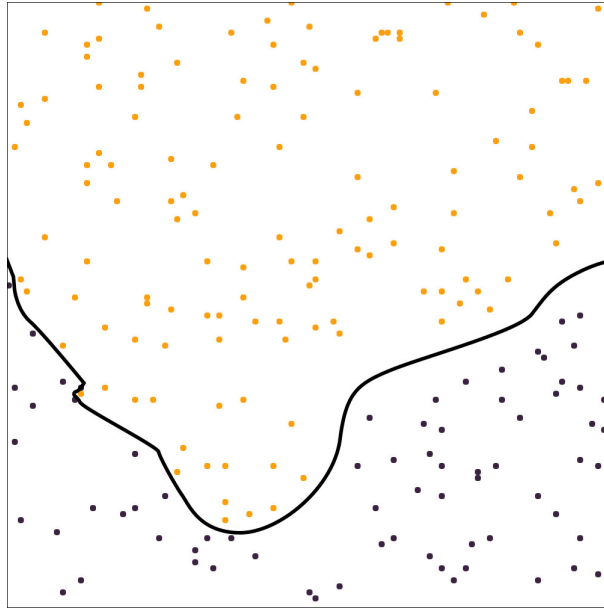
Figure 3.2: 200 nodes, $memberThresh = 128$

found a group except for the remaining few. These are typically straggler nodes or smaller clusters separated from the others. They are not big enough to reach the group $memberThresh$ on their own, so the merge with other nodes that are in unmaxed groups. So even though they have neighbours which influence them more, they can only merge with nodes further away, because that is the only unlocked group that remains.

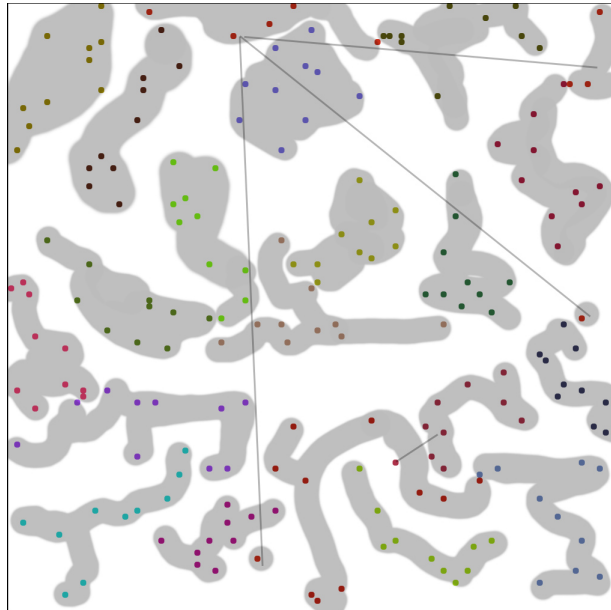### 3.6.2 Wigle as data source

3.7 Evaluating the results

Figure 3.3: 200 nodes, $memberThresh = 10$

Bibliography

[1] Python Software Foundation. *Python 3*. Accessed July, 2017. 2017. URL: http://python.org.

[2] Wikipedia. *Free-space path loss — Wikipedia, The Free Encyclopedia*. Accessed 4th July, 2017. 2017. URL: http://en.wikipedia.org/w/index.php?title=Free-space%5C%20path%5C%20loss&oldid=765763753.