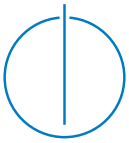
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Information Sharing with Floating Content using Wi-Fi Access Points

Ghulam Nasir



DEPARTMENT OF INFORMATICS

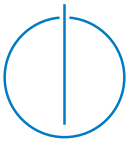
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Information Sharing with Floating Content using Wi-Fi Access Points

Einsatz von Floating Content mit Wi-Fi-Routern zur Verteilung von Inhalten

Author:	Ghulam Nasir
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Prof. Dr.-Ing. Jörg Ott
Submission Date:	November 15, 2018



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, November 15, 2018 Ghulam Nasir

Abstract

Delay Tolerant Networks (DTN) allows asynchronous content sharing between different devices. The basic idea behind DTN is to communicate data/messages from the source to the destination without having complete network connectivity. The Opportunistic Network Environment (also known as ONE) simulator is a tool for evaluating DTNs. The ONE simulator provides us with a number of different movement models and routing algorithms to simulate various conditions. One of the interesting features is the ability to provide Map-based movement which enables us to model any area of the map. The ONE simulator lacks support for *access points* based connectivity, where a node can either be an access point, station adapter or ad-hoc node. There are some universal rules governing the connectivity between these modes such as station adapter can only connect to access point and ad-hoc nodes can only connect to other ad-hoc nodes. The main goal of this thesis is evaluating how content sharing will be impacted by access points as ad-hoc networking is not yet a solution for the real world. For this evaluation, we need to extend the ONE simulator to support access points. Modeling the list of publicly available access points on the map of a city (which is Munich city in our case) and evaluating the effect on the availability of the messages is our desired way for such simulations. The ONE simulator also supports routing of the floating content. Floating content is geo-based content sharing service, solely dependent on mobile devices in the vicinity using principles of opportunistic networking. Floating content ensures that the message is available within a certain area known as the availability/anchor zone. Any node going outside the availability area of the message would drop copies of that message. Since floating content is only available inside the anchor zone, where they can be replicated to other nodes, sometimes there are either not enough nodes or there are a lot of nodes in the anchor zone and the information may die out quicker than anticipated. One way of increasing the availability zone of the message is to snap the messages to the nearest **k** anchor points. The concept is that whenever a message is copied to the access point, the center of that message is set to be the location of the access point, thus effectively changing the center and anchor zone of that copy of the message. Theoretically, this would increase the availability of the message, however, being a DTN there is no such guarantee. The second main task is to implement the snapping to access point features and then run simulations to evaluate the effect of snapping to access points on the life (duration of availability) of the message.

Contents

1 Introduction

The Opportunistic Network Environment (also known as ONE) simulator is a tool for evaluating Delay Tolerant Networks (DTNs). The ONE simulator provides us with a number of different movement models and routing algorithms to simulate various conditions. One of the interesting features is the ability to provide Map-based movement which enables us to model any area of the map. The ONE simulator comes with the map of Helsinki by default. One main feature it lacks is access point based connectivity, which is quite a common feature of connected networks. Implementation of access points in the ONE simulator is necessary for the following two main reasons:

- Access points based networks are very common, while ad-hoc networks are not so common.
- Evaluation of how access points would affect content sharing using DTNs.

One of the main issues of Delay Tolerant Networks(DTNs) is security. Floating content provides a solution for this issue by limiting the availability zone of the messages. However, this security is at the expense of message availability. One way to increase the availability zone is by snapping the message to nearest access points. Snapping to access point is a concept where the access point takes ownership of its copy of the message (thus changing the availability zone). However, all the snapped availability zones would be at least overlapping with the original availability zone. It would be interesting to study the following two scenarios:

- availability of messages vs. number of access points the message is snapped to
- availability of messages vs. new availability (due to snapping to access points)

In short, the main motivation is to study the effect of access points support and snapping to access points on the message life/availability to ascertain these features affect the behavior of Delay Tolerant Networks (DTNs).

1.1 Research Questions/Goals

Several questions/goals were explored in this work to weigh different extensions to the ONE Simulator such as adding access point functionality as well as snapping to access points etc. We have formulated these goals/research questions below :

Q1: Modeling a city and exploring the impact of access points on the performance of geo-based information sharing.
The ONE Simulator currently models Helsinki City and doesn't have any access points feature. The main idea here is to Model Munich City (especially the central part), Implement Access Points and then explore the impact of the access points on the geo-based information sharing performance using ONE Simulator. It also involves mapping publicly available Access Points on the Munich Model City Map.

Q2: How does snapping to access point affect the availability of a floating message in a Delay Tolerant Network (DTN)?
Snapping to Access Point is the concept of increasing the availability of the message beyond its original availability zone. The idea is to change the center of the message when it encounters an access point. However, we do limit the number of access points a message can snap to. After implementation of snapping to access point, we will test how does the snapping to a different number of access points affect the availability of the floating message in a DTN.

2 Literature Review

2.1 Geo-based Content Sharing

In typical networks (such as social networking etc.), the content is shared among users irrespective of their geographic location. There are a number of advantages to this such as:

- High availability
- Reliability
- a large amount of content.

However, there are some disadvantages to such networks such as:

- A connected network is needed for the services to work, so in case the connectivity is lost, normally the content is also gone.
- The system/network is prone to censorship

Geo-based content sharing has been introduced to make the information/content more customized based on the user location such as chatting with people nearby, posting in a local message board etc. There are two types of networks used for Geobased content sharing: *Connected Systems/Networks* and *Peer-to-Peer Systems/Networks*.

2.1.1 Connected Systems/Networks

Connected Network is a connected set of devices where each device is connected to one or more other devices in the network. One common setting is the server-client architecture, which clients connect to servers to access resources. Since server provides resources, there can be different kinds of servers such as file server, web server, application server etc. A very basic example of such architecture is world wide web (www) where users (clients) can access websites, files etc. In such networks, there is a very high probability that the message is delivered to the desired device.

Connected Networks enable the service providers to provide a very reliable service. Social media applications/websites, chatting applications, and e-commerce application etc. use connected networks to provide services to their clients.

2.1.2 Peer-to-Peer Systems/Networks

Peer-to-Peer Networks don't rely on the network infrastructure to relay information, so they can work even when there is no server/central controller or no pre-determined path between the sender and receiver. A key property of such systems is that they do not rely on infrastructure nodes or cloud services to ensure data availability but rather replicate content items within a defined geographic area among mobile nodes in a device-to-device (peer-to-peer) fashion. While this operation does not require infrastructure network access—and thus limits dependencies as well as vulnerability to third-party actions such as traceability or censorship—it comes at the cost of unpredictability: there is no guarantee that content "posted" to a defined geographic area will remain available. We refer to this property as best-effort (probabilistic) content sharing [Ott+17].

Below are a few systems/concepts utilizing this methodology/concept:

- Hovering Information [CSK09]
- Locus [TCK10]
- Ad Loc [CC06]
- Floating Content [Hyy+11]

Hovering Information [CSK09]

Hovering information is a concept characterizing geo-localized self-organizing information responsible to find its own storage on top of a highly dynamic set of mobile devices. This information is attached to a geographical point, called the *anchor location*, and to its vicinity area, called *anchor area*. The main requirement of a single piece of hovering information is to keep itself stored at some specified location (*anchor location*) despite the unreliability of the device on which it is stored. Whenever the mobile device, on which the hovering information is currently stored, leaves the area around the specified storage location, the information has to hop - "hover" - to another device. Hovering information uses mechanisms such as active hopping, replication, and dissemination among mobile nodes. It does not rely on any central server. To achieve this purpose, two main algorithms are used, which are, the *Attractor point* algorithm and the *Broadcast* algorithm. Below is a short description of both of these algorithms:

- **Attractor Algorithm**: The anchor location acts as an attractor for the hovering information attached to it. The purpose of this algorithm is to keep the hovering information alive in its *anchor area* as long as possible.
- **Broadcast Algorithm**: The hovering information broadcasts/replicates itself to all the nodes in the communication range. The purpose of this algorithm is to provide better availability of the information, however, it uses more memory and network resources.

Locus [TCK10]

Locus, a location-based data overlay for DTNs. Locus keeps objects at specific physical locations in the network using whatever devices currently are nearby. Nodes copy objects between themselves to maintain the locality of data. Location utility functions prioritize objects for replication and enable location-based forwarding of data look-ups. As a first-of-its-kind application, Locus is compared against other possible replication policies and shown to achieve query success rates nearly 4 times higher than other approaches. As a mobile device moves through the network, it creates new data periodically. Each piece of data created has a timestamp (also known as creation time), an application specific type indicator and the current location, which becomes the *home location* for the newly created data. The location-based design requires that the devices have a GPS or any other mechanism to capture the current location. The main goal is to keep the data as close as possible to the location it was sensed so that it can be easily found later. To access data in Locus, nodes forward query messages to the physical regions in which the node is interested. Every query message contains a *target location* (location of interest), and the type of data needed. Any intermediate node, that receives the query, can respond if it contains any data with *home location* matching the *target location*. Once matching data is found, a response message containing the data is sent back to the original requesting node.

AD LOC [CC06]

AD LOC is an infrastructure-free, localized, persistent and asynchronous platform for collaboratively annotating the physical environment with *notes*. Below are some characteristics of the notes:

- **Localized**: notes are specific to physical locations.
- **Persistent**: notes remain in the environment.
- **Asynchronous**: notes and published and consumed without the need for both the consumers and publishers to be available at the same time.
- **Collaborative**: anyone can publish or read any note.
- **Infrastructure-free**: no servers or internet connections.

As users with mobile devices pass through locations, they automatically receive notes published there and makes them available to other users. As users leave, their cached notes can be transferred to other devices remaining at the location.

Floating Content [Hyy+11]

Floating content is an ephemeral content sharing service, solely dependent on mobile devices in the vicinity using principles of opportunistic networking. Floating content

has some intrinsic properties which are: 1) information dissemination is geographically limited; 2) the lifetime and spreading of information depend on interested nodes being available; 3) traffic can only be created and caused locally; and 4) content can only be added, but not deleted. Floating Content is discussed in more details *later in the chapter*.

2.1.3 Delay Tolerant Networking (DTN)

Delay-tolerant Networking (DTN) [Fal03] enables communication in sparse mobile ad-hoc networks and other challenged environments where traditional networking fails and new routing and application protocols are required. Past experience with DTN routing and application protocols has shown that their performance is highly dependent on the underlying mobility and node characteristics [KOK09]. The basic idea behind DTN is to communicate data/messages from the source to the destination without having complete network connectivity. The path from the source to the destination is unknown due to the unconnected nature of the network. In such networks, each node store the messages it receives and forwards it to the next node it comes in contact with. The message may or may not reach its final destination.
DTNs are also known *Opportunistic Networks* due to the fact that the nodes connect to each other and transfer message as soon as they get the opportunity (when they come in vicinity of each). Below are a few characteristics of DTNs:

1. **Mobility**: DTN is based on the concept of mobile nodes which enables the flow of information. Unlike traditional networks, node mobility to required to transfer data.

2. **Dynamic**: Since the nodes are mobile, old nodes leave and new nodes connect to the network, making it a very dynamic network.

3. **Limited Resources**: DTNs utilized mobile devices/nodes which have lesser resources than a normal computer. Such resources include processing power, memory and more importantly battery/energy.

2.2 The ONE Simulator

The ONE is an Opportunistic Network Environment simulator [**the-one**] which provides a powerful tool for generating mobility traces, running DTN messaging simulations with different routing protocols, and visualizing both simulations interactively in real-time and results after their completion [Ker].

The ONE simulator was originally developed in Aalto University in the SINDTN [OLK] and CATDTN projects supported by Nokia Research Center (Finland). It is now maintained in by Aalto Univerity in cooperation with the Technical University of Munich.

The ONE Simulator has the following capabilities:

- Message Generation at different intervals.
- Node Movement using different movement models such as Random Walk, Map-based movement etc.
- Routing messages between nodes using different types of routers such as Epidemic Router, First Contact Router etc.
- Graphical User Interface (GUI) for visualizing nodes movement, message creation and message passing in real time.
- Report Generation for different purposes such as Message Delivery Report etc.
- A batch mode where the app can be run without a GUI.

Figure 2.1 illustrates the different components of the ONE (Opportunistic Network Environment) simulator and how they work together. Below is a short explanation of each component.

- **Simulation Engine** is the main component that drives the ONE simulator. In other words, it is the controller of the ONE simulator. It is responsible for handing event generation, routing and report generation, to name a few.
- **Event Generators** are responsible for generating different types of events. One simple example is the generation of a message by a node/host.
- **Movement Models** specify the movement patterns for the hosts. There are a number of movements models that we can use. One example would be the map based movement, where the hosts follow the map (or a specific route on the map).
- **Routers** are responsible for the routing of the messages between nodes. An example would be the DirectDelivery router, which delivers the message only to the final recipient (in other words, it does not use any intermediate node for transferring messages).

- **Results/Reporting** is responsible for storing the different simulation data (such as connectivity between different hosts, the average life of a message etc) on the filesystem. These files are later used either for visualization or for verification/inference of results.

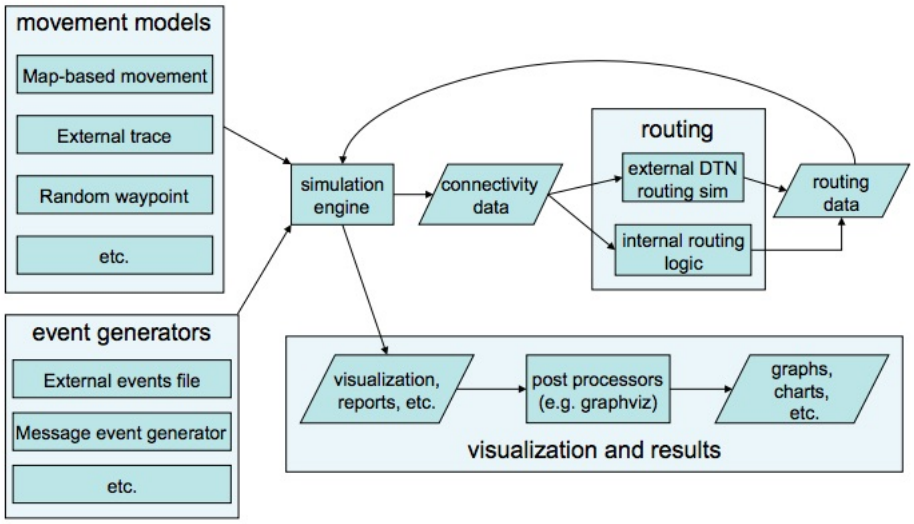


Figure 2.1: Overview of the ONE simulation Environment [KOK09]

2.2.1 Running

Just like Compilation, ONE can be run from both Command Line and Eclipse. However, we will discuss only the command line.

```
./one.sh [-b repetitionCount] [configuration-files]
./one.bat [-b repetitionCount] [configuration-files]
```

-b is an optional parameter. Start the ONE simulator in batch (non-UI) mode. *repetitionCount* specifies the number of times the simulations need to repeat. If not defined, the ONE simulator starts in UI mode.
configuration-files is an optional parameter, where we can specify a number of configuration files separated by space. The simulation parameters are read from these files. Files specified later override values in the earlier config files.

2.2.2 Configuration

One of the amazing features of ONE is that there is no need for changing the code and recompilation if need to change any configuration (such as changing the number of hosts, changing the movement models etc.). All of these can be done in the configuration files which are ordinary text files. By default, the ONE simulator uses **default_settings.txt** configuration file. However, additional configuration files can also be passed.

Configuration files contain key-value pairs. Below is the basic syntax for most of these pairs:

```
Namespace.key = value
```

Both *Namespace* and *key* follow the camelCase and are case sensitive. *value* can be numeric, boolean, strings, comma-separated values, value filling and reference to other files. Numeric values can also use the suffixes *k* for Kilo, *M* for Mega and *G* for Giga.

2.2.3 Host Group

Host Groups represents a collection of hosts with the same configuration such as Movement Model etc. In other words, similar nodes are grouped in a Host Group. Theoretically, each host group must have a different configuration from the other host groups. All the host groups can also share configuration (such as the number of hosts etc.) which are defined under *Group* namespace. However, configuration inside the specific host group takes precedence over the global *Group* namespace.

```
Group.nrofHosts = 20

Group1.groupID = G1
Group1.nrofHosts = 30
Group2.groupID = G2
Group3.groupID = G3
```

In the above listing, *Group* defines the number of hosts (i.e. 20) for all the groups. However, *Group1* overrides this property and sets the number of hosts to 30. So, now we have three groups where *Group1* has 30 hosts while *Group2* and *Group3* have 20 hosts each.

2.2.4 Movement Models

Movement Models dictate the node movements in the simulation. A number of Movement Models come packaged with the ONE simulator such as *Map based Movement*, *Shortest Path Map based Movement*, *Random Waypoint*, *External Movement* and *Stationary Movement*. *Map based Movement* has further types of movements such as *Car Movement* and *Map Route Movement* etc.

```
Group.movementModel = MapbasedMovement
Group1.movementModel = RadomWaypoint
```

In the above listing, we are setting the movement type for all the groups to *Mapbased-Movement*, however, *Group1* is overriding and setting it to *RandomWaypoint*.

2.2.5 Routing

Routing Module is responsible for handling the transfer (routing) of messages between hosts. ONE simulator one passive and a number of active routers. A passive router is used for interacting with other DTNs or dummy nodes. The active routers implement the well-known routing algorithms for DTN such as *Epidemic*, *Spray and Wait*, *First Contact*, *Direct Delivery*, *Prophet*, *Prophet V2*, *Life*, *Wave*, *MaxProp* and *Floating Content*.

```
Group.router = EpidemicRouter
Group2.router = DirectDeliveryRouter
```

In the above listing, *EpidemicRouter* is used for all the groups except *Group2* which is using the *DirectDeliveryRouter*.

2.2.6 Value Filling

Value Filling is used to dynamically fill the value during run-time. The main purpose is to use the value of a variable during run-time. To use value-filling, we put the key name as value surrounded by

```
MovementModel.warmup = %%Reports.warmup%%
```

In the above listing, we are setting up the warmup for *MovementModel* to be the same as the warmup used for *Reports*.

2.2.7 Run Indexing

There are times when we want to run a number of simulations for different permutations of the configuration such as varying number of hosts, varying size of the message etc. One solution would be to create different configurations and write a script passing the desired configuration for each run. However, it is not a maintainable solution if we have a different number of permutations (changing 3 values each for two different parameters). The ONE (Opportunistic Network Environment) simulator solves this problem by using run indexing. Run indexing allows us to change configuration during run-time by providing the same configuration file; containing a set of semi-colon separated values surrounded by braces ([]) for the desired key(s).

```
Reports.warmup = [0; 100; 200; 300]
```

In the above listing, the first run will use the value 0, next one will use the value 100 and so on. In case the values are less than the total number of simulations, the indexes wrap around (loops back to the start of the array when there is no further item to read).

2.3 Floating Content

Floating content is an ephemeral content sharing service, solely dependent on the mobile devices in the vicinity using principles of opportunistic networking. The net result is a best effort service for floating content in which: 1) information dissemination is geographically limited; 2) the lifetime and spreading of information depend on interested nodes being available; 3) traffic can only be created and caused locally; and 4) content can only be added, but not deleted [Hyy+11].

In Floating Content, a message (a text message, an image, etc.) deemed to be of interest to other people at a certain area is tagged with geographical coordinates of that area. This area is referred to as the anchor zone of the message [Ott+]. Each message has two main properties namely *Center* and *Availability/anchor zone*. For the sake of simplicity, the anchor zone is defined as a circle with a radius. In short, floating messages can be relayed to other nodes inside the anchor zone. Every time a copy of the message is made, the anchor zone properties are copied to make sure that the copied message has the same anchor zone as the original message.

Whenever a message is generated, it is assigned an anchor zone (identified by a radius and the node's current location). The message is transferred to other nodes in the anchor zone. Whenever a node moves out of the anchor zone, it may delete its copy of the message or keep it until first interaction with another node depending on the configuration.

2.3.1 System Operation [Ott+11]

A node generates an information item/message I of size $s(I)$ with a certain lifetime (TTL) and assigns an anchor zone with center C and two radii a and r. The difference between a and r is listed below:

1. **Availability zone (a)**: a defines the availability zone of the item, which is the region/zone within which the item I is still kept around with a limited probability. In other words, this is the region/zone where the item I can exist.
2. **Replication zone (r)**: r defines the replication zone of the item, which is the region/zone within which the node always try to replicate the item to any other node it encounters.

If two nodes A and B meet in the anchor zone (or replication zone to be specific) and A has a copy of M while B does not, then A replicates message M to B. Theoretically, every node in the anchor zone should have a copy of the message. However, practically it is not a possibility due to limited storage of the nodes. Let us explain the whole phenomenon with an example: Assuming we have 2 nodes (A and B). For the sake of simplicity, we assume there is only 1 item/message I generated by A with availability range a and replication zone r. Node B now comes into the availability range a of I, however, nothing

happens (but *I* still exists on *A*). Now *B* enters the replication range *r* or *I*. When *B* comes in contact with *A* inside the replication zone, a copy of *I* is transferred to *B*. Both *A* and *B* keep copy of the item until they are inside *a* (assuming we have unlimited storage).

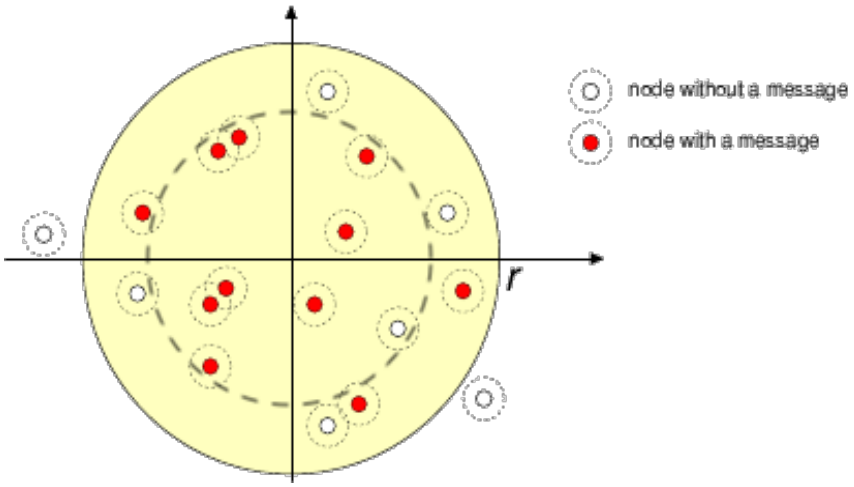


Figure 2.2: Floating-Content and its anchor zone [Ott+]

Figure 2.2 shows two circles, where the outer circle represents the availability zone (*a*) and the inner circle represents the replication zone (*r*) of the item, which is the inner circle.

2.3.2 Important Concepts

The following list summarizes the important concepts:

1. **interval**: The minimum time between the generation of two messages by the same node.

2. **replication range/zone (r)**: The replication range for floating message. It is the range/zone where the message can be replicated (transferred) to other nodes.

3. **availability range/zone (a)**: The availability range of the floating message. It is the range/zone where the message is still kept around with a limited probability.

4. **time to live (TTL)**: The life of the floating message.

5. **size**: The size of the floating message.

6. **replicationPolicy**: Policy which dictates how floating messages are replicated when the node comes in contact with another node.

7. **deletionPolicy**: Policy which dictates when floating messages are deleted.

2.3.3 Deletion Policy

As mentioned in the above list, deletion policy dictates when the floating messages are deleted. We currently have two policies, which are explained below:

1. **Immediate**: When this policy is used, the message is deleted as soon as the node (carrying the message) leaves the anchor zone (availability zone to be specific).
2. **Encounter**: When this policy is used, the message is deleted as soon as the node (carrying the message) comes in contact with the first node outside the anchor zone (availability zone to be specific). The advantage/use case is that the node still carries the message if it leaves the anchor zone for a short period of time (in case it does not encounter another node during that period). It is the default policy in ONE Simulator.

2.3.4 The ONE Simulator and Floating Content

ONE simulator has inherent support for floating content. It handles all the functionality (such as replication of messages, dropping of messages etc.) of the floating message. The following section explains the main components of the ONE simulator used for floating content:

Components

The ONE Simulator has two main components for dealing with floating content which are *FloatingApplication* and *FloatingContentRouter*.

- *FloatingApplication* is responsible for generation of the messages with the floating content parameters (such as anchor, ttl etc.). FloatingApplication can be associated with any Host Group (or all the Host Groups) using the configuration file(s).
- *FloatingContentRouter* is responsible for all the routing related functionalities for the floating messages. It takes care of replication, deletion (and dropping) and prioritization of message. Replication and Deletion is performed as per the Replication and Deletion policies. Different parameters of the FloatingContentRouter can be set in the configuration file(s).

Configuration

As mentioned in the *Configuration of ONE Simulator*, all the configurations need to be provided as configuration files containing key-value pairs. Below are some examples for setting configuration for the floating content.

```
#1.
FloatingContentRouter.replicationPolicy = fifo
FloatingContentRouter.deletionPolicy = immediate
```

```
#2.
floatingApp.type = FloatingApplication
floatingApp.ttl = 300
floatingApp.destination = 0
floatingApp.messageSize = 5k
floatingApp.anchor = 200
#3.
Group.nrofApplications = 1
Group.application1 = floatingApp
Group.router = FloatingContentRouter
```

The above listing shows the settings needed for ONE simulator to correctly handle floating messages.

1. We set different the replication and deletion policies for the floating content router (which is responsible for routing floating messages).
2. In order to handle floating content (message), ONE simulator needs FloatingApplication. Here we define different message properties such as time to live (ttl), message size, size of anchor zone etc.
3. We configure each group to use the floating application and floating content router. In other words, the nodes of each group will generate, send and receive floating messages.

3 Design and Implementation

This chapters explains the design and implementation details. The chapter starts with how to prepare maps for simulations. It then explains the concept and implementation of access points and snapping to access points.

3.1 Maps

This section explains how maps are used, and how to prepare and configure maps for ONE simulator.

3.1.1 Concept

Maps play a very important role in simulating real-world depiction of any place in the ONE simulator. This allows us to simulate the physical location as accurately as possible. The *MapBased Movement* provides us with different types of map-related movements.

The ONE Simulator requires the map in the modified Well-Known Text (WKT) Format. I call it the modified version as the WKT files utilizes points based system instead of coordinates based system. We use the Osm2Wkt tool[May10] for converting Open Street Maps (OSM) files to WKT for the ONE Simulator.

Filtering Maps

The ONE Simulator can accept one or more WKT map files. In case of more than 1 files, all of the files are basically overlayed on the top of each other. One thing to note is that the files need to be fully connected otherwise the ONE simulator won't start and will throw an error.

OpenStreetMap provides with a tool called OSMFilter [Mapc] which enables us to filter the OSM maps. This allows us to filter specific elements such as roads, bus routes etc. To use the filtered maps with ONE simulator, we need to convert each of them to WKT. The current implementation of the ONE simulator uses the following 4 WKT files:

Table 3.1: ONE Simulator: WKT Map files used by default

Map File	**Details**
main_roads.wkt	represents the main roads (primary, secondary, tertiary).
roads.wkt	represents the normal roads and streets.
shops.wkt	represents the shops.
pedestrian_paths.wkt	represents the pedestrian paths.

OSM Maps

The first step in getting the maps is to download the OSM map from the Open Street Map export tool [Mapb]. Figure 3.1 shows the OSM Map for Central Munich.

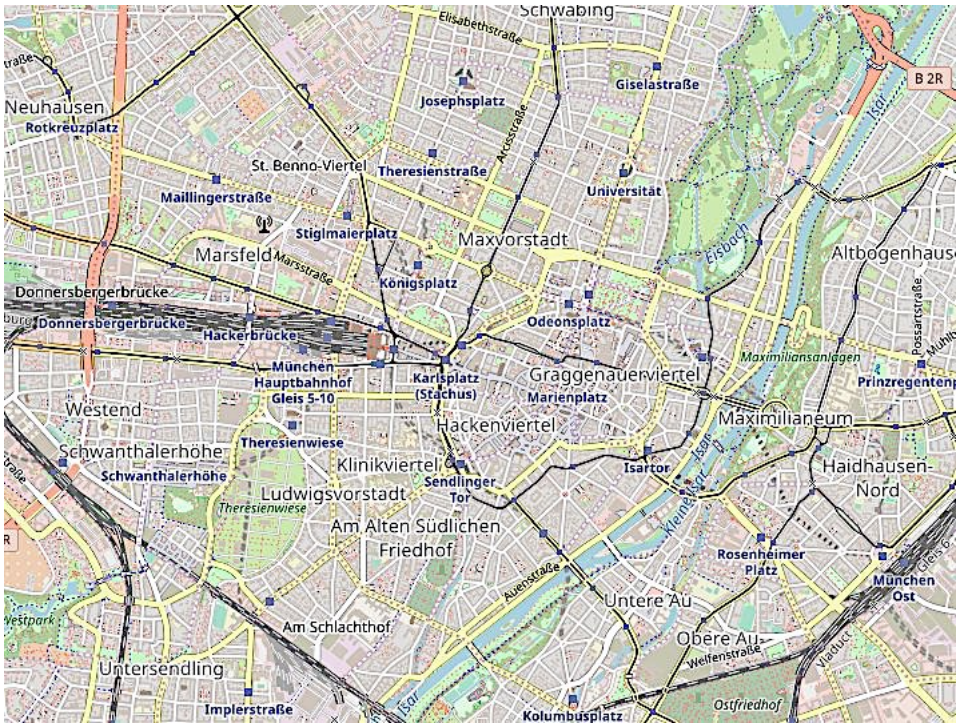


Figure 3.1: Central Munich OSM Map - from OpenStreetMaps [Mapa]

Conversion to WKT

There are a number of tools that can convert an OSM to WKT, however, there are a few reasons we do not use any of those tools:

- These tools just perform the conversion (but do not make sure the converted maps are fully connected).
- These tools do not convert the coordinates to the points-based system (which is used by the ONE simulator). To use any WKT file with the ONE simulator, we need to do proper coordinates conversion.

Figure 3.2 shows the contents of the WKT file. This file is generated using OpenJump [Ope], so they are in the normal coordinates system. In other words, these files are not in the coordinates system used by ONE simulator.

```
POINT (11.5859318 48.1383583)
POINT (11.5763905 48.1367741)
LINESTRING (11.5669265 48.1385126, 11.5670542 48.1384848, 11.5671648 48.1384666,
11.5671242 48.1383751, 11.5670732 48.1382434, 11.5668682 48.1382749, 11.5669134
48.1384592, 11.5669265 48.1385126)
LINESTRING (11.5692615 48.1417543, 11.569205 48.1418468, 11.5691685 48.1419658,
11.5692268 48.142013, 11.5692535 48.1420517, 11.5693459 48.1422348, 11.5693431
48.142284, 11.5693435 48.1423297)
LINESTRING (11.568828 48.1419005, 11.5691311 48.1419929, 11.5692268 48.142013,
11.5692977 48.1420006, 11.5696013 48.1419195, 11.5696696 48.1418847, 11.5697647
48.1418473)
LINESTRING (11.5721983 48.1367036, 11.5723376 48.1366081, 11.5724347 48.136661,
11.5725711 48.1367384, 11.5726917 48.1368081, 11.5725562 48.1369124, 11.5725 48.1368797,
11.5721983 48.1367036)
LINESTRING (11.5637581 48.1384165, 11.5639442 48.1384023, 11.5647922 48.1383376,
11.5649032 48.1383314)
LINESTRING(11.5643483 48.1366282, 11.5641708 48.1366269, 11.5641363 48.1366285,
11.5641466 48.1366981, 11.5641555 48.1367525, 11.5640278 48.1367587, 11.5639643
48.1367618, 11.5637824 48.1367706, 11.5637833 48.136853, 11.5639831 48.1368482,
11.5641354 48.1368384, 11.5643864 48.1368167, 11.5643483 48.1366282)
POLYGON ((11.5639106 48.133894, 11.5637535 48.1338771, 11.5637492 48.1339479, 11.5637481
48.1340577, 11.5637652 48.13413, 11.5638975 48.1341169, 11.5640656 48.1340961,
11.5640161 48.133999, 11.5639036 48.1340153, 11.5639106 48.133894))
```

Figure 3.2: Snapshot of WKT File Contents - Converted using OpenJump [Ope]

Using OSM2WKT tool [May10], we can convert the OSM file to a fully-connected point-based WKT file that can be used with the ONE Simulator. Figure 3.3 shows the contents of such a WKT file.

```
LINESTRING (1248.752 800.047, 1271.99 845.571)
LINESTRING (1103.019 812.246, 1097.124 790.941, 1087.514 793.921, 1084.103 782.868,
1091.984 780.433, 1088.632 769.58, 1099.912 766.089, 1103.456 777.542, 1106.432 776.619,
1107.203 779.121, 1113.118 777.297, 1121.215 806.263, 1103.019 812.246)
LINESTRING (1275.053 852.376, 1281.869 868.343)
LINESTRING (1022.878 526.619, 1014.281 548.336, 1015.027 565.193)
LINESTRING (1021.207 492.76, 1021.517 501.156)
LINESTRING (1477.012 257.739, 1467.313 258.006, 1453.699 239.948, 1459.437 233.565)
LINESTRING (1459.437 233.565, 1465.346 229.34)
LINESTRING (1271.99 845.571, 1275.053 852.376)
LINESTRING (1010.935 458.913, 1013.933 457.967, 1017.567 469.487, 1018.005 470.888,
1015.007 471.833, 1010.935 458.913)
LINESTRING (1026.5 478.227, 1029.343 477.36, 1026.947 469.543, 1025.108 463.538,
1022.265 464.406, 1024.498 471.689, 1026.5 478.227)
LINESTRING (1521.922 462.526, 1531.1 468.965, 1540.879 475.836, 1541.725 476.448,
1542.252 476.815, 1541.822 473.957, 1540.724 474.113, 1531.805 467.886, 1523.243
461.893, 1523.073 460.97, 1520.52 461.503, 1520.98 461.837, 1521.922 462.526)
LINESTRING (431.842 301.683, 431.085 303.573, 432.918 304.296, 433.667 302.406, 431.842
301.683)
LINESTRING (363.027 280.456, 362.426 282.391, 364.303 282.98, 364.912 281.045, 363.027
280.456)
LINESTRING (426.662 293.021, 433.616 295.79, 434.477 293.555, 435.308 291.531, 428.347
288.762, 426.662 293.021)
```

Figure 3.3: Snapshot of WKT File Contents - Converted using OSM2WKT tool [May10]

Figure 3.4 shows a fully connected WKT map of central Munich.

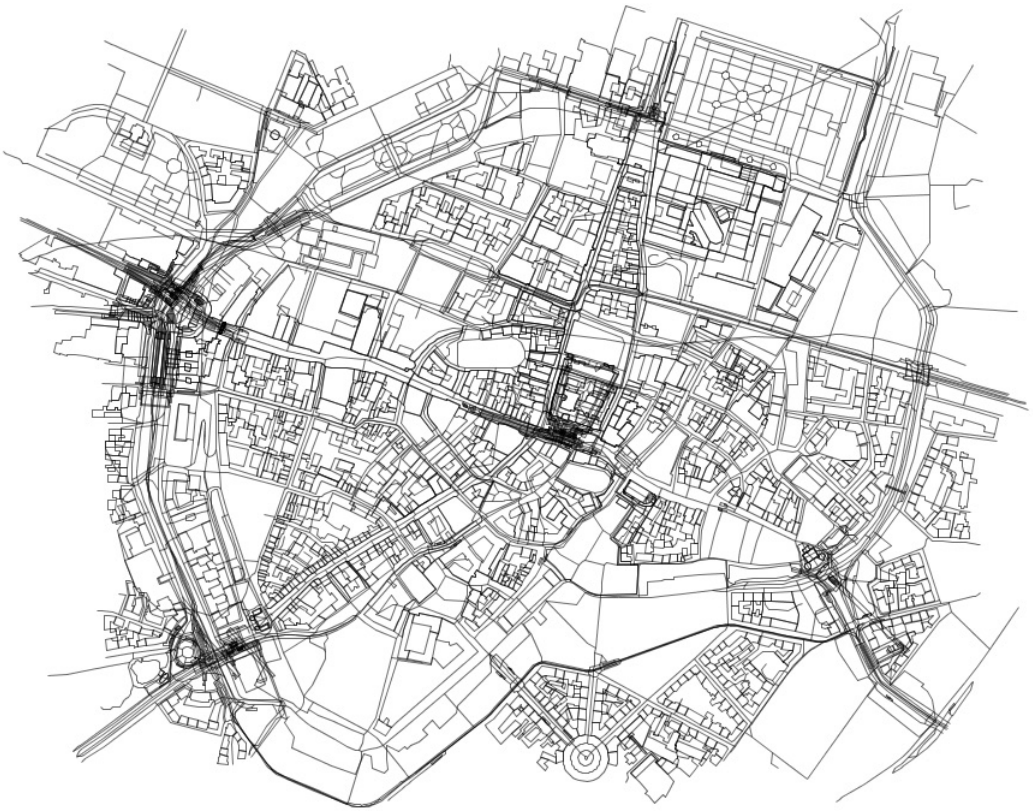


Figure 3.4: Central Munich WKT Map - Created using OSM2WKT tool [May10] and Screenshot using OpenJump [Ope]

We use the above map (Figure 3.4) in our simulations. This map works fine with ONE simulator as it is a fully connected map and it has also been converted to the points-based system used by ONE simulator.

3.1.2 Implementation

This section list and explains all the steps needed to configure and run a new map (downloaded from OpenStreetMap [Mapa]) in ONE simulator.

OSM to WKT

Using OSM2WKT tool [May10], any OSM file can be converted into a fully connected WKT file for use in the ONE simulator. Below are the details of how to use the OSM2WKT tool [May10] to generate a WKT file:

```
java -jar ./osm2wkt.jar mapfile.osm

options:
       -o outputfile - specifying the output file. In case no outputfile
          is mentioned, .wkt is appended to the name of input file.
       -a - open the output file in append mode (adding the wkt output at
          the end of the output file)
       -t deltaX deltaY - translate map by deltaX and deltaY meters
```

Listing 3.1: Using OSM2WKT tool [May10] to generate WKT

The OSM2WKT tool [May10] performs a number of operations on the OSM file. It loads the map from the OSM file. The coordinates are converted to the points based system where:

1. The bounding box of the map is converted into a point based system (width and height in meters).
2. All the coordinates are converted to the point based system.

The completeness of the graph is then checked and all the unconnected edges are deleted so that we have a complete graph. At this stage, the file can be saved to disk as a WKT fie.

Filtering OSM

In order to filter OSM files, OSM Filter tool [Mapc] is needed. Here is the basic syntax to use the OSM Filter tool. [Mapc]. This tool has a number of folders that we can use. Below is the common ones (for our scenario):

1. **highway**: highway represents any route/path such as roads, cycle paths, pedestian paths etc. Some examples of highway include primary, secondary, trunk, residential, service, footway, pedestrian etc.
2. **shop**: shop represents any type of shop.

```
osmfilter inputfile.osm --keep="{condition}" > outputfile.osm
```

Listing 3.2: How to use the OSM Filter tool [Mapc]

The listings below show how to create the four different types of OSM files which can then be converted to WKT files.

```
// Filtering primary, secondary, tertiary (*ary), and trunk roads as main
   roads.
osmfilter mapfile.osm --keep="highway=*ary trunk" > main_roads.osm

// Filtering residential roads, living street (street with special rules),
    service and link roads as the common roads
osmfilter mapfile.osm --keep="highway=residential service *link
   living_street" > roads.osm

// Filtering pedestrian footway living_street as pedestrian paths.
osmfilter map.osm --keep="highway=pedestrian footway living_street" >
   pedestrian_paths.osm

// Filtering all shops as shops.
osmfilter map.osm --keep="shop=*" > shops.osm
```

Listing 3.3: Filtering OSM files

The main issue with using separate osm files is that we cannot use Osm2WKT tool [May10] and it is a difficult task to make sure the resultant map is fully connected. The Osm2WKT tool [May10] can be modified to accept more than 1 OSM files and generate the corresponding WKT files that are fully connected. The main idea is to read all of the OSM files and place them on one graph. Now we start dropping unconnected nodes and paths until we have a fully connected map that can be used by ONE simulator. We are using a single map file in our simulations.

3.1.3 Configuration

In order to utilize the *MapBasedMovement* and the map files we have prepared (using method in the above sections), We need to add a few things to the configuration file.

1. The first step is to set the *movementModel* of the desired group to *MapMapMovement*. This tells the ONE simulator to use map-based movement for these groups.

   ```
   Group.movementModel = MapBasedMovement
   ```

2. Then we need to identify the number of maps files we are providing and provide the relative path to each of the maps files.

   ```
   MapBasedMovement.nrofMapFiles = 1
   MapBasedMovement.mapFile1 = data/maps/map.wkt
   ```

We can also add more than one map files. Below is an example of using more than 1 map files.

```
MapBasedMovement.nrofMapFiles = 4

MapBasedMovement.mapFile1 = data/maps/roads.wkt
MapBasedMovement.mapFile2 = data/maps/main_roads.wkt
MapBasedMovement.mapFile3 = data/maps/pedestrian_paths.wkt
MapBasedMovement.mapFile4 = data/maps/shops.wkt
```

In this section, we have discussed how to prepare map files for the ONE simulator and how to configure the ONE simulator to use those files.

3.2 Access Points

In this section, we will introduce the concept of access points and how they are implemented in the ONE simulator.

3.2.1 Concept

Access Points are devices that connect devices to one another and facilities the flow of information among devices. One of an interesting use case would be to connect the nodes to the internet using these access points. An Access Point can operate in any of the following three modes:

Mode	**Description**
Access Point (AP)	Access Point (AP) provides a bridge for two Station Adapters (SAs) to connect and communicate with each other. Two Access Points cannot talk to each other directly.
Station Adapter (SA)	Station Adapter (SA) is a passive device that can connect to an Access Point and communicate to other Station Adapters using the Access Point (AP). Two Station Adapters cannot talk to each other directly.
Ad-hoc	Ad-hoc Mode is used when we want two devices to connect and communicate directly to each other. Hosts in Ad-hoc mode cannot connect to either an Access Point (AP) or Station Adapter(SA) and vice versa. This is the default mode.
Normal Hosts	Normals hosts are the ones with no access point features. In other words, Normal Hosts behave and perform similar to the *Ad-hoc* hosts.

Table 3.2: ONE Simulator: Modes of Operation for Access Points/Wifi Interface

Figure 3.5 shows the color scheme used by ONE simulator to differentiate between nodes with different modes of operation. It also differentiates between normal and

access point enabled hosts.

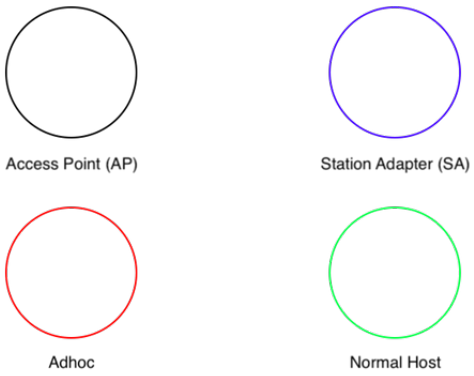


Figure 3.5: Hosts with Access Points (SA, AP, and Ad-hoc) and Normal Host

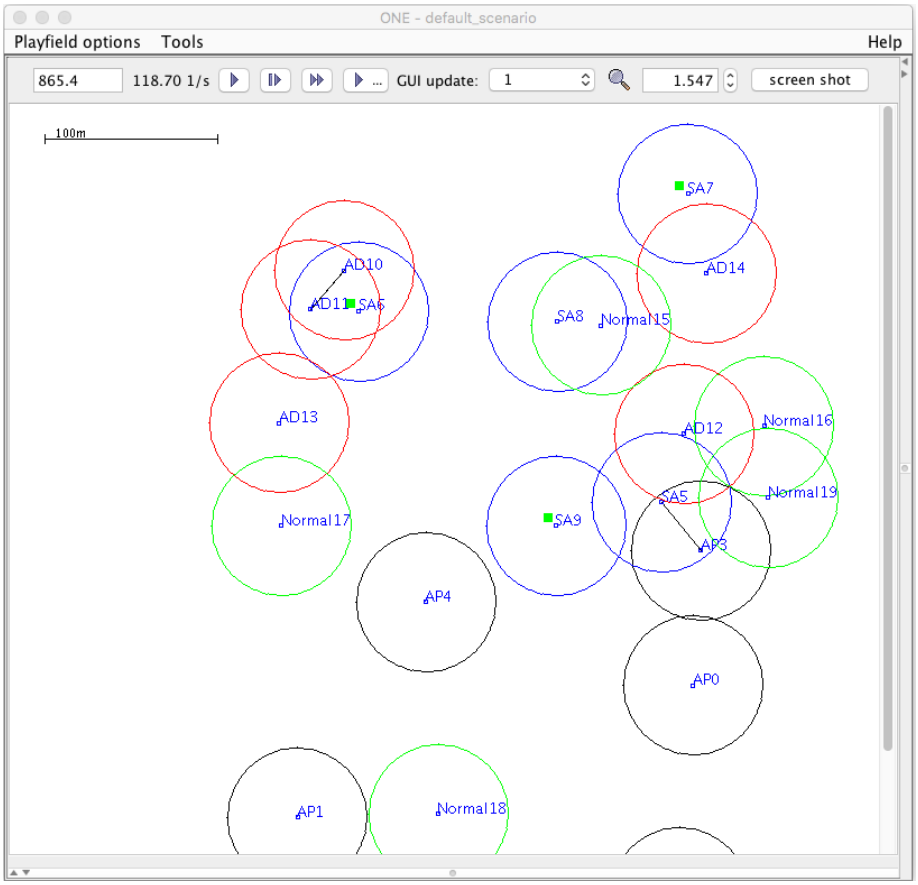


Figure 3.6: ONE Simulator showing all the Hosts (AP enabled and normal) in action

The above figures shows all of the hosts type in action in the ONE Simulator. It shows that station adapters can connect to an access point (*SA5* and *AP3*), ad-hoc nodes can connect to other ad-hoc nodes (*AD10* and *AD11*), ad-hoc nodes cannot connect to access points and vice versa (*AD10*), *AD11* and *SA6*). It also shows that the normal hosts cannot connect to access points enabled hosts (*Normal15* and *SA8*, *Normal16* and *AD12* etc.).

3.2.2 Implementation

Before discussing implementation details, We would like to discuss the pre-requisite for the implementation. The following table lists two classes and what they are used for:

Class	Description
DTNHost	This class represents a DTN Capable host. It contains all the properties and functionalities for the host such as groupID, MessageListeners, MovementListeners etc.
DTNHostWithWifi	This class is a subclass of the *DTNHost* class and represents a host with Access Point/Wifi capability. It inherits everything from the parent class and adds an extra mode (which can either be ADHOC, AP or SA).

Table 3.3: ONE Simulator: Classes related to Hosts/Nodes

DTNHostWithWifi is a very lean host that takes everything from its parent *DTNHost* and adds the mode. The following listing shows the implementation of *DTNHostWithWifi*.

```
public class DTNHostWithWifi extends DTNHost {
        public static int AP = 1;
        public static int SA = 2;
        public static int ADHOC = 3;
        private int mode;

        public DTNHostWithWifi(other parameters, String mode) {
         ...
         this.mode = mode.equalsIgnoreCase("ap") ? AP : mode.
             equalsIgnoreCase("sa") ? SA : ADHOC;
        }

        public int getMode() {
               return mode;
        }
}
```

Listing 3.4: ONE Simulator: DTNHostWithWifi Implementation

In Listing 3.5, *DtnHostWithWifi(..)* initializes the mode with the passed value. It also sets the default mode of operation to ad-hoc (in case an empty or wrong mode is specified). *getMode* returns the mode of operation.

There is another component that needs to be implemented i.e. an interface to the new *DTNHostWithWifi*. We call it *WifiInterface*. The following table discusses the two main classes.

Table 3.4: ONE Simulator: WifiInterface & its parent interface

Class	**Description**
SimpleBroadcastInterface	*SimpleBroadcastInterface* is a Network Interface that is responsible for connectivity among hosts. It allows one-to-one transmission with a constant bit-rate.
WifiInterface	*WifiInterface* is responsible for connectivity between two different *DTNHostWithWifi* objects. It is also responsible that the connection rules (SA can connect only to AP, Ad-hoc hosts can connect to each other only) are upheld.

WifiInterface takes almost everything from its parent but adds the extra ability to check if two *DTNHostWithWifi* can connect to each other. Another method that it overrides is the ability to return *DTNHostWithWifi* with the *getHost()* method.

Listing 3.5 shows a snapshot of relevant parts of *WifiInterface* implementation. For the sake of simplicity, we have divided the code into numbered sections, each of which is explained below:

1. *connect(..)* is used to connect two hosts with wifi interfaces. It uses *canConnectTo(..)* to determine if the two hosts can connect to each other. In case *canConnect(..)* return true, then the transmission speed of the connection is set, which is the minimum speed of both the interfaces. The hosts are connected to each other using their interfaces.

2. *canConnectTo(..)* determines if two hosts with wifi interface connect to each other. The connectivity is based on the mode of the device. Below are the rules used to determine if two such hosts can connect to each other:
   - Ad-hoc hosts (ADHOC) can connect to other ad-hoc hosts but they cannot connect to either station adapters (SA) or access points (AP).
   - Station adapters (SA) can only connect to access points (AP) and vice versa.

```
public class WifiInterface extends SimpleBroadcastInterface {
    //1
    public void connect(NetworkInterface anotherInterface) {
        if (canConnectTo(anotherInterface)) {
            int conSpeed = anotherInterface.getTransmitSpeed();
            if (conSpeed > this.transmitSpeed) {
                conSpeed = this.transmitSpeed;
            }
           //Connectivity code
             }
     }

    //2
     boolean canConnectTo(NetworkInterface anotherInterface) {
         ...
         if(canConnect) {
             WifiInterface thisInterface = (WifiInterface) this;
             WifiInterface secondInterface = (WifiInterface)
                 anotherInterface;

             int thisInterfaceMode = thisInterface.getHost().getMode();
             int secondInterfaceMode = secondInterface.getHost().getMode
                 ();

             if(thisInterfaceMode == DTNHostWithWifi.ADHOC) {
                 canConnect = thisInterfaceMode == secondInterfaceMode;
             } else {
                 canConnect = (thisInterfaceMode == DTNHostWithWifi.AP
                     && secondInterfaceMode == DTNHostWithWifi.SA) || (
                     thisInterfaceMode == DTNHostWithWifi.SA &&
                     secondInterfaceMode == DTNHostWithWifi.AP);
             }
          }
          return canConnect;
      }
      ...
}
```

Listing 3.5: ONE Simulator: WifiInterface Implementation

Now, we need to combine both of these to make sure that any host having a *WifiInterface* is instantiated as *DTNHostWithWifi* objects and all the other hosts are instantiated as *DTNHost* objects. This needs to be done inside *createHosts()* function of *SimScenario* class. Below are the changes that are made to achieve this purpose:

```
boolean hasWifiInterface = false;

//1.
for (int j=1;j<=nrofInterfaces;j++) {
        ...
        if(iface instanceof WifiInterface) {
                hasWifiInterface = true;
        }
        ...
}
...
//2.
List<String> modes = new ArrayList<String>();
String wifiMode = s.getSetting("mode",null);

if(wifiMode == null) {
        String modesFile = s.getSetting("modesFile",null);
        if(modesFile != null) {
                try {
                        Scanner in = new Scanner(new FileReader(modesFile));
                        while(in.hasNextLine()) {
                                modes.add(in.nextLine());
                        }
                        in.close();
                } catch (FileNotFoundException e) {}
        }
}

// 3.
for (int j=0; j<nrofHosts; j++) {
        ...
        if(!hasWifiInterface) {
                DTNHost host = new DTNHost(this.messageListeners,
                        this.movementListeners, gid, interfaces, comBus,
                        mmProto, mRouterProto);
                hosts.add(host);
        } else {
                String mode = wifiMode != null ? wifiMode : (j < modes.size
```

```
                () ? modes.get(j) : "");

            DTNHostWithWifi host = new DTNHostWithWifi(this.
                messageListeners, this.movementListeners, gid,
                interfaces, comBus, mmProto, mRouterProto, mode: mode);
            hosts.add(host);
        }
}
```

Listing 3.6: ONE Simulator: Changes to SimScenario for Implementation of the AP-capable hosts

The above listing performs a few operations, which are divided into the following three groups:

1. In this section, we iterate through all the interfaces looking for a wifi interface (to determine if the host has access point support).
2. In this section, we read the mode for all the hosts. As mentioned earlier, we have two ways of providing mode. The first one is to provide a mode file (containing mosts for all the hosts) and the second way is to to mention a mode for all the hosts of that node. The modes are read from the mode file (if it exists) otherwise it is read from the mode parameter (which is then applicable to all the nodes of the group).
3. This is the section where we differentiate between normal and access points enabled hosts. This section first checks what type of host to create and in case of an access point enabled host, it sets the mode by fetching it from either the *wifiMode* variable or the *modes* array.

3.2.3 Configuration

The following listing shows how to configure *DTNHost* and *DTNHostWithWifi*. It also shows how to configure *mode* for *DTNHostWithWifi* using the two different methods.

```
//1.
wifiInterface.type = WifiInterface
wifiInterface.transmitSpeed = 50k
wifiInterface.transmitRange = 40

btInterface.type = SimpleBroadcastInterface
btInterface.transmitSpeed = 250k
btInterface.transmitRange = 10
```

```
//2.
Scenario.nrofHostGroups = 5

Group.nrofInterfaces = 1
Group.interface1 = btInterface

//3.
Group1.groupID = Normal

//4.
Group2.groupID = SA
Group2.interface1 = wifiInterface
Group2.mode = sa

Group3.groupID = AP
Group3.interface1 = wifiInterface
Group3.mode = ap

Group4.groupID = ADHOC
Group4.interface1 = wifiInterface
Group4.mode = ad-hoc

//5.
Group5.groupID = D
Group5.interface1 = wifiInterface
Group5.modesFile = data/modes.txt
```

Below is a summary of the above listing:

1. We define two interfaces with transmission range and speed. One of the interfaces is a *WifiInterface*.
2. We are defining the general settings for the simulation and all the groups. We are setting the total number of groups to 5, the number of interfaces for each group to 1 and setting *btInterface* as the default group of each group.
3. We are setting the ID for Group1 which is using the default interface.
4. We are defining three groups to use *wifiInterface*. We are also setting the mode for each group (all the hosts of that group share the same mode)
5. We are defining one more group to use *wifiInterface*. However, here we are setting the modes using a text file. A typical mode contains one mode per line.

```
ap
sa
ap
sa
sa
sa
```

Listing 3.7: Contents of mode file

The above listing shows the contents of the mode file. The mode for first and third host is set to access point (ap) while the mode for second, fourth, fifth and sixth host is set to station adapter (sa). The modes are assigned in a circular way. In the above case, the seventh host will have the mode access point (ap) and the eigthth one will be station adapter(sa).

3.3 Snapping to Access Points

In this section, we define a new concept "snapping to access points".

3.3.1 Concept

Section 2.3 discusses that a message is tagged with geographical coordinates and an availability/radius. *Snapping to Access Point* is the concept of increasing the availability of the message beyond its original availability zone. In the configuration, we can define how many Access Points should the message snap to at maximum. Theoretically, the number of snapped Access Points should directly influence the TTL (Time to live) of the message. Below is a step-wise explanation of the process:

1. The host Station Adapter (*SA1*) generates a message *M* with a center of the anchor zone *C* (which is the location of *SA1* at that time), availability *a* and number of maximum access points to snap to (*k*). Let's assume k to be 3 *k = 3*.

2. *SA1* comes in contact with an Access Point (*AP1*), a connection is made and a copy of the message *M* is transferred (*M'*).

3. If *k*-value for the original message *M* is greater than 0:
   a) *k*-value for the original message (*M*) is decreased by 1.
   b) *k*-value for the copied message (*M'*) is set to zero.
   c) The center of the copied message (*M'*) is changed to the center of the Access Point (*AP1*).

4. The above process continues for the original message *M* until *k*=0.

Since the ID of the message *M* and all of the copies is the same, they are treated as the same message; which allows us to extend the availability of the message. There are a few important points to note:

1. Copies of the message are not snapped to any access point, Only the original message is snapped.
2. If a message is snapped to 1 access point which drops it due to any reason (apart from TTL expiry), whenever that message (the original or even the copy which knows about this access point) is copied again, it is snapped to that access point again.
3. The availability zones of the copies might or might not intersect each other, however, the availability zone of the original message and each copy would have an overlap.

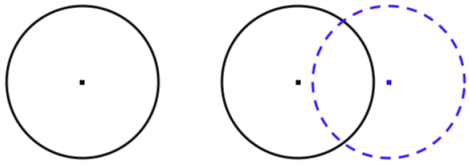


Figure 3.7: Availability zones of Message with k=0 (not snapped yet) to the left and k=1 (snapped to 1 Access Point) to the right

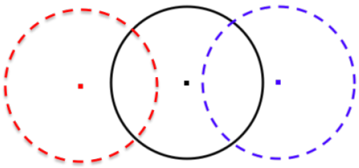


Figure 3.8: Availability zones of Message with k=2 (snapped to 2 Access Points)

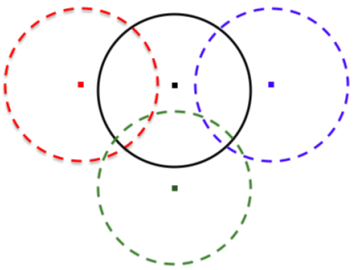


Figure 3.9: Availability zones of Message with k=3 (snapped to 3 Access Points)

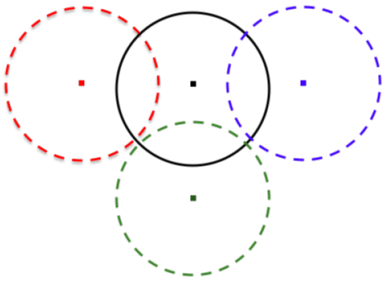


Figure 3.10: Special Case: Message with k=3 where there is no overlap between availability zones of the copies

In the above figure, the solid circle shows the original anchor zone while the dotted ones show the added anchor zones (due to snapping to access points).

3.3.2 Implementation

We will explain the implementation of this feature step by step below:

- The very first point is to attach an *originalAnchor* to the message (which is the point where the message was originally generated).

```
m.addProperty("originalAnchor", m.getProperty("anchor"));
```

- The settings for the number of Access Points to snap to (*kNextAPs*) is fetched inside *FloatingContentRouter*.

```
if (fcSettings.contains(FC_NUM_OF_APS))
        kNextAPs = fcSettings.getInt(FC_NUM_OF_APS);
```

- When message transfer is complete (*messageTransferred* method in *FloatingContentRouter*), we need to check f the message need to be snapped to any Access Point (by checking *kNextAPs* property of the message).:
  - If the host does not have a wifi interface, the control is returned back to the caller.

```
if(!(this.getHost() instanceof DTNHostWithWifi)
        return m;
```

  - If the receiving node is not an access point, the control is returned back to the caller.

```
DTNHostWithWifi receivingNode = (DTNHostWithWifi) this.getHost();
if(receivingNode.getMode() != DTNHostWithWifi.AP)
        return m;
```

– *kNextAP* for the copied message is set to 0.

```
m.updateProperty(FC_NUM_OF_APS, 0);
```

– The *anchor* of the copy is set to the location of the access point if the message had been already anchored to this access point and the message still can be anchored to access points (*kNextAPs>0*). Decrease *kNextAPs* only if the message was not previously anchored to this access point.

```
if ((int) msg.getProperty(FC_NUM_OF_APS) > 0 || (
    anchorAccessPoints != null && anchorAccessPoints.contains(
    receivingNode.toString()))) {
        m.updateProperty(FC_ANCHOR, receivingNode.getLocation().
            clone());

        if((int) msg.getProperty(FC_NUM_OF_APS) > 0)
                msg.updateProperty(FC_NUM_OF_APS, (int) msg.
                    getProperty(FC_NUM_OF_APS) - 1);
}
```

– The access point ID is added to the list of APS (*anchorAPs*) property of the message. This helps us in re-snapping the message to the access point if the access points get a new copy of the same message.

```
anchorAccessPoints.add(receivingNode.toString());
msg.updateProperty(FC_ANCHOR_APS, anchorAccessPoints);
m.updateProperty(FC_ANCHOR_APS, anchorAccessPoints);
```

– The access point location is added to the list of APS (*anchorAPsLocation*) property of the message. This is used to make sure that the new availability zones are also accounted for before trying to delete a message.

```
anchorAccessPointsLocations.add(receivingNode.getLocation().clone
    ());
msg.updateProperty(FC_ANCHOR_APS_LOC, anchorAccessPointsLocations
    );
m.updateProperty(FC_ANCHOR_APS_LOC, anchorAccessPointsLocations);
```

Changes to Message Deletion Algorithm

Here we are making sure that the message is deleted only after it is outside of the new availability zones (in case a message is snapped to access point, we need to check that the message is outside all the availability zones for that message).

```
for (Message m : m_set) {
        ArrayList<Coord> locations;
```

```
        try {
                locations = new ArrayList<Coord>((HashSet<Coord>)m.
                    getProperty(FloatingContentRouter.FC_ANCHOR_APS_LOC));
        } catch(NullPointerException ignored) {
                locations = new ArrayList<>();
        }

        locations.add(0, (Coord)m.getProperty(FC_ANCHOR));
        locations.add(1,(Coord)m.getProperty(FC_ANCHOR_ORIGINAL));

        int counter = 0;
        while(counter<locations.size()) {
                distance_curr = loc.distance(locations.get(counter));
                if ((deletion_check(distance_curr, (Double) m.getProperty(
                    FC_R), (Double) m.getProperty(FC_A)) != 1))
                            break;
                counter ++;
        }
        if(counter == locations.size() && !d_list.contains(m.getId()))
                d_list.add(m.getId());
}
```

3.3.3 Configuration

```
FloatingContentRouter.kNextAPs = 1
```

In the above listing, we are setting the number of access points a message can snap to 1. It means that each message can snap to 1 access points at maximum.

3.4 Access Points Data

3.4.1 Data Collection

We are using the access points database from RadioCells [Rad]. The database has a number of tables, however, we are only concerned with *wifi_zone*. We are only using *latitude* and *longitude* data.

3.4.2 Data Processing

We follow the following steps to process the data for the ONE Simulator:

1. The data from access points database (which is a SQLite database) [Rad] is exported to a text file.

2. The data is processed by removing the double quotes and separating latitude and longitude by a comma (,).
3. To make sure that the coordinates are converted to the correct scaling, we add the boundary coordinates to the file. Below are the boundary coordinates:
   - minimumLatitude,0
   - maximumLatitude,0
   - 0,minimumLongitude
   - 0,maximumLongitude
4. The processed file is then passed to a Coordinate Converter which returns a WKT file with the transformed coordinates.
5. The WKT file can then be used for setting up the access points on the map using *StationaryMapBasedMovement*.

```
POINT (1062.64139413 199.02659474)
POINT (376.98851001 206.11914023)
POINT (992.97309405 214.68331232)
POINT (520.67853701 221.98222492)
POINT (520.9567434 228.63351036)
POINT (557.31248005 238.69873078)
```

Figure 3.11: Transformed Access Points Data

3.4.3 Implementation

A majority of the implementation is taken from Osm2Wkt [May10]. The implementation is pretty straight-forward. At first, we open the file and read the content into an array of Landmarks. *Landmark* is a class that stores the original coordinate (represented by *latitude* and *longitude*) as well as the transformed coordinates (represented by x and y). The following code shows how the coordinates are stored as landmarks.

```
Scanner in = new Scanner(new FileReader(filePath));

while(in.hasNextLine()) {
        String text = in.nextLine().trim();
        if(text.length() > 0) {
                String[] coordinates = text.split(",");
                Landmark landmark = new Landmark(Double.parseDouble(
                    coordinates[0]), Double.parseDouble(coordinates[1]));
                landmarks.add(landmark);
        }
```

```
}
in.close();
```

The coordinates are then transformed using *transformCoordinates* [May10]. Finally, the coordinates are written in a WKT file which can be used by the ONE Simulator.

To read the WKT file in ONE simulator, we have implemented *StationaryMapBasedMovement* and *MapNodeExtended* classes. The following table explains the functionality of each of these classes.

Class	**Details**
MapNodeExtended	Inherits from the *MapNode* class. It reads the WKT file and apply transformations (mirror and translation) to the location points.
StationaryMapBasedMovement	Inherits from *MapBasedMovement* class. Gets the list of transformed locations (from the WKT file) by passing it to *MapNodeExtended*.

Table 3.5: ONE Simulator: Classes for WKT based Node positioning

Below code snippet shows the main functionality from *StationaryMapBasedMovement*.

```
public StationaryMapBasedMovement(Settings settings) {
        super(settings);

        String fileName = settings.getSetting(ROUTE_FILE_S);
        allNodes = MapNodeExtended.readNodes(fileName, getMap());
}

@Override
public Coord getInitialLocation() {
        counter = (counter + 1) % allNodes.size();
        this.lastMapNode = allNodes.get(counter);
        return lastMapNode.getLocation();
}
```

Listing 3.8: Main features of *StationaryMapBasedMovement* class

Below is the main code snippet from *MapNodeExtended* class.

```
WKTReader reader = new WKTReader();
List<Coord> coords;
File routeFile = null;

//1.
double xOffset = map.getOffset().getX();
double yOffset = map.getOffset().getY();

//2.
List<MapNode> nodes = new ArrayList<MapNode>();
try {
        routeFile = new File(fileName);
        coords = reader.readPoints(routeFile);
}
catch (IOException ioe) {
        throw new SettingsError("Couldn't read MapRoute-data file " +
                        fileName +  " (cause: " + ioe.getMessage() + ")");
}

//3.
for (Coord c : coords) {
        if(map.isMirrored())
                c.mirror();

        c.translate(xOffset, yOffset);

        MapNode node = map.getNodeByCoord(c);
        if (node == null) {
                node = new MapNode(c);
        }
        nodes.add(new MapNodeExtended(node));
}

return nodes;
```

Listing 3.9: Getting List of all MapNodes (taken from *MapNodeExtended*)

The following listing explains the main features of the above code:

1. Normally, the map will have an offset of 0 on both the x and y axes. However, we can configure a different offset for the map. In this step, we fetch the offset for the

map. This is necessary as we need to apply the same offset to all the nodes.

2. All the points from the map files are read and stored as a list of coordinates.

3. Here we check if the map has been mirrored (flipped on the X coordinate) because we need to apply the same transformation to each node. The node is then translated (as per offset of the map) and then a node is created which can be used in the simulation.

3.4.4 Configuration

```
Group1.movementModel = StationaryMapBasedMovement
Group1.routeFile = data/path_to_wkt_file/file_name.wkt
```

The above listing shows how we can configure ONE simulator to use a WKT map file (containing access points). First, we need to set the *movementModel* of the group to *StationaryMapBasedMovement*. Then, we specify the relative path to the map file using *routeFile*. The *routeFile* contains a list of points corresponding to locations of the coordinate. The ONE simulator reads the file and then generate a host for each of the location entry in the route file. It provides us with a lot of convenience when we are dealing with a large number of access points. Below is an example of the *routeFile*.

```
POINT (639.05096629 151.42031916)
POINT (1124.6343505 156.31790994)
POINT (1229.05174141 160.25170123)
POINT (291.6431266 165.00433641)
POINT (330.2305165 169.25611572)
POINT (385.48305451 171.98452587)
POINT (329.33378635 175.55010239)
POINT (1063.92493432 178.36989042)
POINT (285.92316817 181.90934024)
POINT (381.15773888 185.35371429)
POINT (600.20507564 188.42838956)
POINT (1232.16833224 191.79426032)
POINT (350.99008131 194.8568775)
```

Listing 3.10: Conents of routeFile

4 Simulations

Simulation is the representation of the way a system or process functions. Simulations can be used for various purposes. We are using simulations for the following two main purposes:

1. To check the validity of our implementations. For example, We have implemented the Access Point capability. An Access Point can have three modes and there are certain rules of connectivity. We are using simulations to verify that these rules of connectivity are correctly implemented.

2. Inference of results from the output of simulations. For example, building on the example of Access Points, we are using simulations to explore the impact of access points on the performance of geo-based information sharing.

To make things easier, we have divided the simulations into a number of scenarios. Each scenario is used for either validation of the implementation or inference of results as mentioned above. One thing to keep in mind is that since we have to run a lot of simulations, all the simulations are run in batch (non-GUI) mode and the result is written to different reports. All the reports are stored on disk as text files.

4.1 Concepts

The following concepts are necessary to understand how the simulations work as well as the generated report files.

- **Warmup time**: A real-world system normally takes some time (after it is started) to come to a stable state. In order to simulate this, we are making use of the warmup time. Any events generated during this time (such as Message Generation, Replication etc.) are ignored by the reports.

- **Cooldown time**: Cooldown time refers to the time period we are allowing the system to cool down. In other words, cooldown time ensure that message created before the start of the cooldown time have enough time to live (TTL) before the simulation ends. The cooldown time equal to TTL (time to live) of the messages. All the events generated during this time (such as Message Generation, Replication etc.) except *MessageDeleted* are ignored by the reports. *MessageDeletion* event will not be ignored for those message created before the cooldown time starts. We have achieved the following using cooldown time:

1. Any messages created after the cooldown time starts are not recorded in the report. This gives consistency to the report and makes it more reliable. For example, any message created after the cooldown time starts (1500 minutes timestamp) might still be in the system when the simulation time completes (because their TTL is 300 minutes).
2. It allows the messages created before cooldown time to be safely deleted or dropped. An edge case example would be messages created right before the cooldown starts (at 1500 minutes timestamp). In case these messages are not dropped before the end of the simulation, they would be dropped right before the end of simulation due to the expiry of their time to live (TTL).

Let's explain this with the help of an example: Consider we are running a simulation for 1800 minutes and the TTL (time to live) for messages is 300 minutes. The cooldown time is equal to the TTL (time to live) of messages which would mean that the cooldown will start right after the 1500th-minute mark.

4.2 Scenarios

As mentioned above, We have grouped the simulations in Scenarios. Each of the Scenario is responsible for different components of the simulation. Below is a brief introduction of the scenarios:

1. **Scenario 1.*x***: Broadly speaking, these scenarios are used for validation of our access points logic and exploring the impact of access points on the performance of geo-based information sharing.

2. **Scenario 2**: These scenarios are used to evaluate how snapping to access point affects the availability of a floating message in a Delay Tolerant Network (DTN).

4.2.1 Scenario 1.1: Validation of Access Point based connections

As mentioned in the earlier chapters, one of our goals was to implement Access Points based connectivity in the ONE Simulator. The main idea is to simulate the Access Point that we use in normal networks. It has already been mentioned that an Access Point can have three modes of operations: Access Point (AP), Ad-hoc and Station Adapter (SA). Below are the conditions/rules that are used for connectivity:

1. Station Adapters (SA) can only connect to Access Points (APs). The main idea is that Access Points serve different Station Adapters. A related example would be a mobile device connected to a wifi router. In this example, the mobile device is the Station Adapter (SA) and the wifi router is the Access Point (AP).

2. A device in Ad-hoc mode can only connect to another device in Ad-hoc mode.

In order to make sure that the above conditions/rules hold, We have created a simulation that records the connectivity (Connection UP and DOWN) when the connectivity status between the two nodes change. We are using a special report for this purpose which is basically recording the connectivity events for only Access-Point enabled clients. It is worth mentioning here that the normal clients (non-Access Point enabled) can co-exist with Access Point enabled clients but this report will not record connectivity events of the normal clients. We call this the *ConnectivityWifiONEReport*.

```
public class ConnectivityWifiONEReport extends ConnectivityONEReport {
        public static final String HEADER = "simTime, host1, mode, host2,
            mode, up/down";
        public ConnectivityWifiONEReport() {
                init();
        }
        @Override
        public void init() {
                super.init();
                write(HEADER);
        }
        @Override
        public void hostsConnected(DTNHost h1, DTNHost h2) {
                if(h1 instanceof DTNHostWithWifi || h2 instanceof
                    DTNHostWithWifi) {
                        write(createTimeStamp() + "," + h1.toString() + "," +
                             ((DTNHostWithWifi)h1).getModeName() + "," + h2.
                            toString() + "," + ((DTNHostWithWifi)h2).
                            getModeName() + "," + "UP");
                }
        }
        @Override
        public void hostsDisconnected(DTNHost h1, DTNHost h2) {
                if(h1 instanceof DTNHostWithWifi || h2 instanceof
                    DTNHostWithWifi) {
                        write(createTimeStamp() + "," + h1.toString() + "," +
                             ((DTNHostWithWifi)h1).getModeName() + "," + h2.
                            toString() + "," + ((DTNHostWithWifi)h2).
                            getModeName() + "," + "DOWN");
                }
        }
}
```

Listing 4.1: Code for ConnectivityWifiONEReport

The above listing shows the implementation for *ConnectivityWifiONEReport*. Below is a summarized explanation of the above code:

- The *ConnectivityWifiONEReport* inherits from the *ConnectivityONEReport* (which is responsible for recording the connectivity reports of all the nodes).
- The HEADER is basically the first line written to the report file. It provides context to any user reading the report so that anyone reading the report understand what each value correspond to.
- hostsConnected function writes the connection UP status to the report file only if at least one of the hosts is an Access Point enabled host (also called *DTNHostwithWifi*).
- hostsDisconnected function writes the connection DOWN status to the report file only if at least one of the hosts is an Access Point enabled host (also called *DTNHostwithWifi*).

4.2.2 Scenario 1.2: Modeling Munich City with Access Points

The main purpose of this scenario is to model the Munich city (central part of the city) with as well as without Access Points. The main purpose is to simulate both of these sub-scenarios with similar parameters and then compare the results to study the impact of access points on the performance of geo-based information sharing. Below is an explanation of these sub-scenarios:

1. **Access Points**: In this sub-scenario, Publicly available access points are modeled on the map of Munich city. In order to make the simulations fast enough, we are utilizing just a subset of those Access Points.
2. **No Access Points**: In this sub-scenario, we are not using any access points. In order to have the same configuration for both the scenarios, we are modeling the access points as stationary nodes/hosts on the map.

In order to get comparable results, both of these sub-scenarios are simulated using a set of different parameters. The following list explains each of these parameters in details:

1. **Message Size**: Message Size plays a vital role in the simulations. Since each node can have a limited amount of space, message size indirectly depicts the capacity of these nodes. The smaller the message size, the more messages a node can store without dropping any message, thus a higher storage capacity. We are using the message sizes *1 Kilobyte (1k)*, *50 Kilobyte (50k)* and *1 Megabyte (1M)* for our simulations. We are using *1 Kilobyte(1k)* to give our hosts virtually unlimited storage space and we are using *1 Megabyte (1M)* to give very limited storage space to our hosts.

2. **Message Availability and Replication Zone**: Message Availability Zone defines the area where a message can exist. Message Replication Zone defines the area where a message can be replicated (copied to) other nodes. It has a direct impact on the life of the message (as any message that leaves this zone is dropped). We are using a radius of *100*, *200* and *300* for message availability and replication zones. We are using *300* to simulate message availability over a bigger area and *100* to simulate message availability over a relatively smaller area.

3. **The number of Hosts/Nodes**: The number of hosts/nodes also play a vital role in the simulations. We are using different sets of configuration for this purpose. Below is an explanation:
   - Cars: Since our simulation area mostly consists of the pedestrian zone, we are not simulating a lot of cars. We are using either 30 or 50 hosts as cars.
   - Pedestrians: In case the number of cars is less than 50, the number of pedestrians is 1.5 times the number of cars. Otherwise, the number of pedestrians is 100. Pedestrians are the main component of our simulations as we are using a map with a high percentage of pedestrian zone.
   - Access Points/Stationary Nodes: In case the number of cars is less than 50, we are using 64 access points/stationary nodes. Otherwise, we are using 132 access points/stationary nodes.
   - The main objective is run the simulations with a lower as well as a higher density of hosts. This enable us to study network performance (measured in fraction of TTL) in both of these conditions. We have chosen the above numbers to fulfill this objective.

For both of these sub-scenarios, we are utilizing the *FloatingMessageSummaryReport*. The main purpose of this report is to record the creation, deletion time, time to live, life (the time between the creation of the message to the deletion of the last copy), total copies as well as the number of dropped copies of the message. The messages are dropped in case there is no more space left. One thing to note here is that creation time is the time when the message is first created while deletion time is the time when the last copy of the message is deleted.

```
protected boolean isCooldown(Message m) {
        double cooldownTime = m.getInitTtlInSeconds();
        return (SimScenario.getInstance().getEndTime() - getSimTime()) <=
            cooldownTime ;
}

public static final String HEADER = "msgId, created_on, deleted_on, ttl,
    life, totalCopies, dropped";

public void newMessage(Message m) {
```

```
        if(isWarmup() || isCooldown(m))
                return;

        MessageSummary messageSummary = new MessageSummary(m.
            getInitTtlInSeconds());
    messagesSummary.put(m.getId(), messageSummary);
}

public void messageTransferred(Message m, DTNHost from, DTNHost to,
    boolean firstDelivery) {
        if(isWarmup() || isCooldown(m))
                return;

        MessageSummary messageSummary = messagesSummary.get(m.getId());
        //This makes sure that we only count messages which were created
            before cooldown
        if(messageSummary != null) {
                messageSummary.incrementCount();
                messagesSummary.put(m.getId(), messageSummary);
        }
}

public void messageDeleted(Message m, DTNHost where, boolean dropped) {
        if(isWarmup())
                return;

        MessageSummary messageSummary = messagesSummary.get(m.getId());
        //This makes sure that we only count messages which were created
            before cooldown
        if(messageSummary != null) {
                messageSummary.decrementCount(dropped);
                messagesSummary.put(m.getId(), messageSummary);
        }
}
```

Listing 4.2: Code snippets from FloatingMessageSummaryReport and Report

The above listing shows code snippet from *FloatingMessageSummaryReport* and *Report*. Below is a summarized explanation of the above code:

- isCooldown function returns true if the cooldown period has started otherwise it returns false.
- newMessage function reports when a new message is created. The message

creation event is not recorded in case it is the warm-up time or the cool down time.

- messageTransferred function reports when the message is replicated to another host. In the case of replication, a copy of the message is created and sent to the new host. The transfer event is not recorded in case it is the warm-up time or the cool down time.

- messageDeleted function reports when a copy of the message is deleted/dropped by a host. The transfer event is not recorded in case it is the warm-up time or the cool down time.
  MessageSummary messageSummary = messagesSummary.get(m.getId()); will only return a non-null value if the message was created before the cooldown time otherwise it will return *null*. The deletion event is only recorded if this line returns a non-null response.

4.2.3 Scenario 2: Simulating Snapping to Access Points

The main purpose of this scenario is to simulate the snapping to access point and determine how does it affect the availability of the messages in a Delay Tolerant Network (DTN). We are also using the Publicly available access points data for this scenario.
We are using the same combination of parameters as *Scenario1*.
We are using two different types of reports for this scenario. The first report is *FloatingMessageSnapToAPSummaryReport* while the second is *FloatingMessageAnchorPointsReport*. The following listing explains both fo these reports:

- FloatingMessageSnapToAPSummaryReport: This report records snapping of the message to the access point. It records the creation and deletion time, time to live (TTL), life (the time between the creation of the message to the deletion of the last copy), count of dropped copies and number of message copies snapped to 0,1,2 or 3 access points.

- FloatingMessageAnchorPointsReport: This report also records snapping of the message to the access point but with a different perspective. Instead of recording the number of access points the message is snapped it, it records the original anchor as well as all the snapped anchors for a message.

The following listing shows the relevant code snippet from *FloatingMessageSnapToAPSummaryReport*.

```
public void messageDeleted(Message m, DTNHost where, boolean dropped) {
        if(isWarmup())
                return;

        MessageSummary messageSummary = messagesSummary.get(m.getId());
        if(messageSummary != null) {
```

```
                messageSummary.decrementCount(m);
                messageSummary.updateSnapCount(m, dropped);
                if(dropped)
                        messageSummary.droppedMessagesCount++;
                messagesSummary.put(m.getId(), messageSummary);
        }
}

public void messageTransferredWithNewAnchor(Message m, Message
    replicatedWithNewAnchor, DTNHost from, DTNHost to) {
        if(isWarmup() || isCooldown(m))
                return;

        MessageSummary messageSummary = messagesSummary.get(m.getId());
        if(messageSummary != null)
                messagesSummary.put(m.getId(), messageSummary);
}

@Override
public void done() {
    double sum = 0;
    double count = 0;
    for(String key: messagesSummary.keySet()) {
        count ++;
        MessageSummary messageSummary = messagesSummary.get(key);
        write(key+","+messageSummary.toString());
        sum += (messageSummary.deletionTime - messageSummary.creationTime)
            / 60.0;
    }

    write("Average,,,," + sum / count);
    super.done();
}
```

Listing 4.3: Code snippets from FloatingMessageSnapToAPSummaryReport

The above listing shows code snippet from *FloatingMessageSnapToAPSummaryReport*. Below is a summarized explanation of the above code:

- messageDeleted function has two main tasks. It records deletion of the message and it also records the number of access points the message is snaped to for those messages created before the cooldown time. We are recording the number of access points the message is snapped to here because this is the only place after

which this information does not change for that copy of the message (as the copy has been deleted).

- messageTransferredWithNewAnchor function reports when a copy of the message is snapped to a new access point. The transfer event is not recorded in case it is the warm-up time or the cool down time.
- done This method summarizes all the information and writes it to the report file. It also writes the average life of message (for this simulation) to the file.

The code for *FloatingMessageAnchorPointsReport* is roughly the same, however, the only difference is that *messageTransferredWithNewAnchor* records the anchor point (location) of the *access point* the message is snapped to.

4.3 Results

Before going into the discussion of results, below is a list of the different parameters we are using for our simulations:

1. **Message Size**: We are message size of *1 Kilobyte (1k)*, *50 Kilobyte (50k)* and *1 Megabyte (1M)*.
2. **Message Availability and Replication Zone**: We are using *100*, *200* and *300* meters for both availability and replication zones.
3. **The number of Hosts/Nodes**: We are using either *Low*(126 hosts) or *High* (282 hosts) for the nodes count configuration.
4. **Simulation Runs**: We run the simulations for 100-120 times for each of the parameter combination.

We are using all the 18 combinations of the above three parameters (from 1k-100-L to 1M-300-H). These combinations will be listed in the *Message Size-Availability and Replication Zone-Nodes Count-Number of Hosts*) format.

In each of the following scenarios, the performance of the network (measured in a fraction of Time-To-Live (TTL))is our benchmark for comparison. We will start with benchmarking each parameter individually, then a combination of two parameters and finally all of the parameters together (if needed).

4.3.1 Scenario 1

In this Scenario, we are mainly interested in understanding and evaluating the effect of access points based setup on delay tolerant networks. To achieve this goal, we have run a number of simulations (with different parameters as stated above) which we will analyze below.

Message Size vs The fraction of TTL (time to live)

Message size represents two important features of the system. The first is the message size itself and the second is capacity of the nodes. However, both of these are inverse relationships (the smaller the message size, the higher the capacity of the node and vice versa). *Figure* 4.1 plots the message life vs the fraction of TTL (Time To Live), regardless of the other two parameters i.e. the availability zone and the number of hosts. This figure shows that traditional hosts perform better than the access points enabled hosts when the message is smaller in size (or the nodes have higher capacity). As the message size increases (or the node capacity decreases), the performance of traditional hosts decrease while that of access points enabled hosts remains almost the same until message size of 50k. After the 50k mark, enabled access points starts to perform better than their traditional counterparts. So the access points enabled hosts perform much better when the message size is higher than 50k (or when nodes have lower capacity). In short, the performance of the wifi enabled hosts increase with increase in message size (or decrease in node capacity).

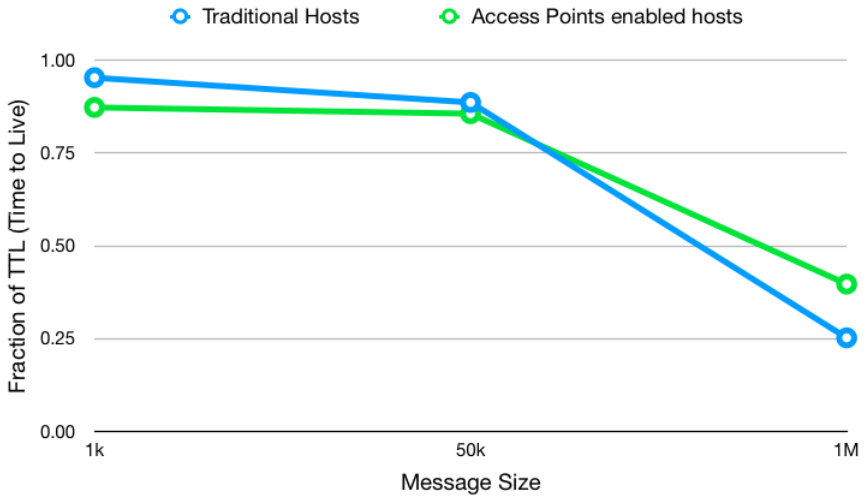


Figure 4.1: Message Size vs The fraction of TTL (time to live) with 90-95% accuracy

Message Availability/Replication Zone vs The fraction of TTL (time to live)

Availability/Replication zones represent the area where a message can exist (and copied to other hosts). *Figure* 4.2 plots the availability zone vs the fraction of TTL (time to Live), regardless of the other two parameters i.e. message size and the number of hosts. This figure shows that traditional hosts perform better than the access points enabled hosts when the availability/replication zone size is smaller. As the availability/replication zone size increases, the performance of access point enabled hosts increases at a good rate while that of the traditional hosts almost stay the same until 200 meters and then

increases by a relatively small rate (w.r.t wifi enabled access points). In short, the performance of wifi enabled hosts increase with increase in the availability zone size.

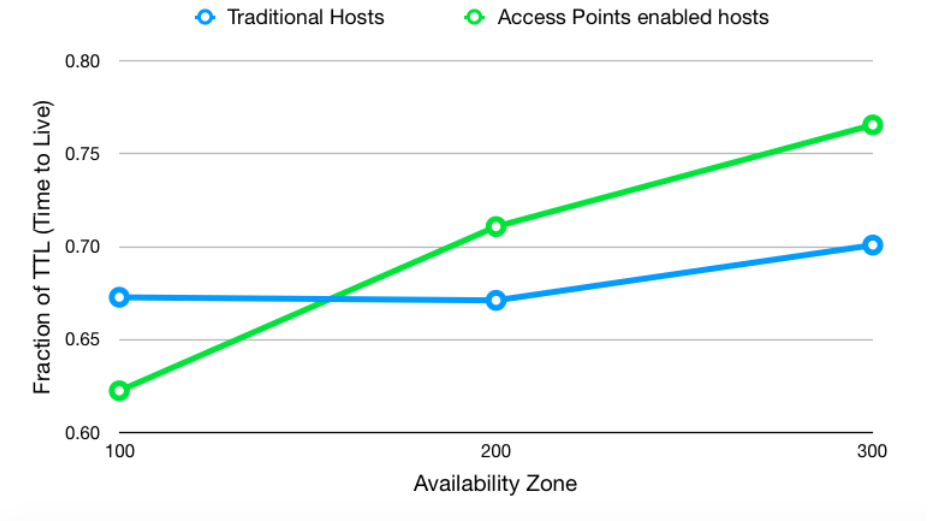


Figure 4.2: Availability Zone vs The fraction of TTL (time to live)

The number of hosts vs The fraction of TTL (time to live)

The number of hosts used is the simulations basically refer to the density of the hosts (which is either *Low* or *High*). *Figure* 4.2 plots the availability zone vs fraction of TTL (time to live), regardless of the other two parameters i.e. message size and the number of hosts.

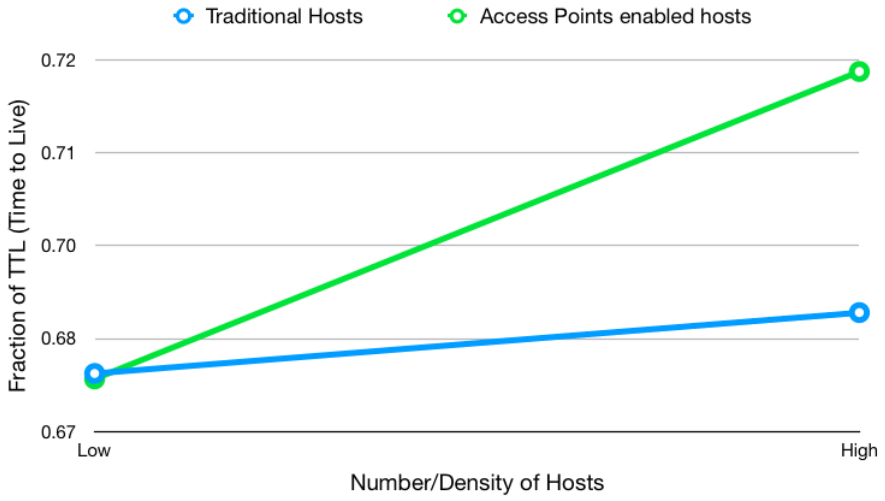


Figure 4.3: Number of hosts vs The fraction of TTL (time to live)

The above figure shows that traditional hosts perform better than the access points enabled hosts when the availability/replication zone size is smaller. As the availability/replication zone size increases, the performance of access point enabled hosts increases at a good rate while that of the traditional hosts increase by a relatively small rate (w.r.t wifi enabled access points). In short, the performance of wifi enabled hosts increase with increase in size of the availability zone size. Increasing the number of hosts indicate an increase in resources that can hold and replicate the message, causing in an increase in the average message life. The performance is much better in access points enabled hosts because the hosts working as access points do not generate any new messages and are just responsible for storing and routing of the message.

Message size and Availability zone size vs The fraction of TTL (time to live)

Figure 4.1 and 4.2 shows that message size and availability zones both have an effect on the performance (measured in a fraction of TTL (time to live)) of the network. In order to figure out which parameter has more influence on the system, we plot them together and then try to figure out which parameter has the main influence on the network performance.

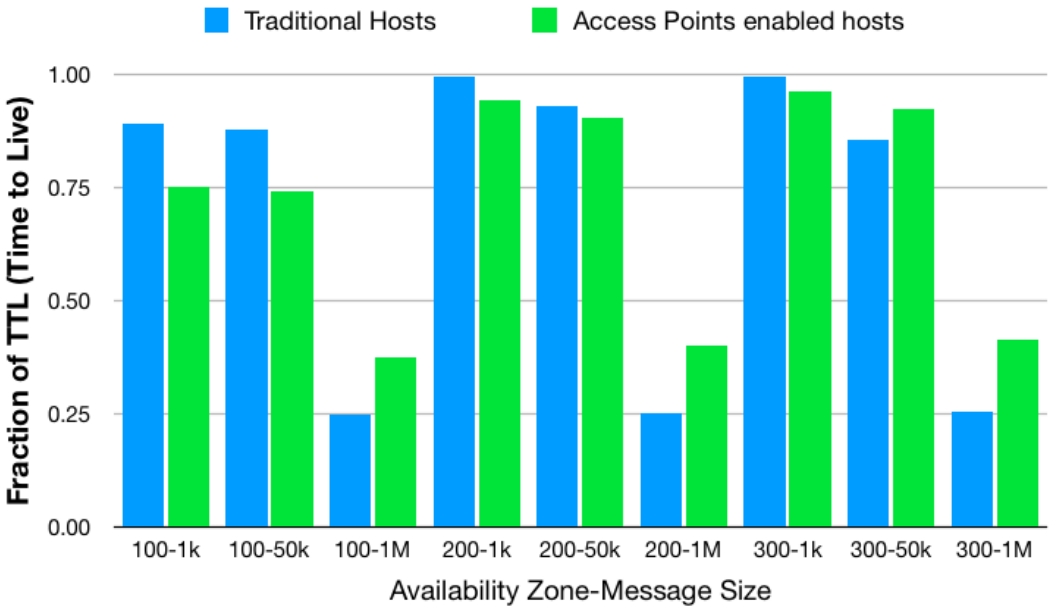


Figure 4.4: Availability Zone-Message Size vs The fraction of TTL (time to live)

Figure 4.4 plots the message size and availability zone vs fraction of TTL (time to live), regardless of the number of hosts. These charts/figures show that message size has the dominant effect on the network performance (measured in a fraction of TTL (time to live)). To come to this conclusion, we divide the graph into three main components w.r.t

the availability zones. By looking into each of the components, there is a significant change in the message availability as the message size changes. Thus, confirming our observation that message size is the main component while the availability zone is a supplementary component. In other words, the performance (fraction of TTL (time to live) depends mainly on message size.
Our conclusion is supported by *Figure* 4.5, where we interchange the two parameters, to have the chart visually grouped by message size instead of the availability zone.

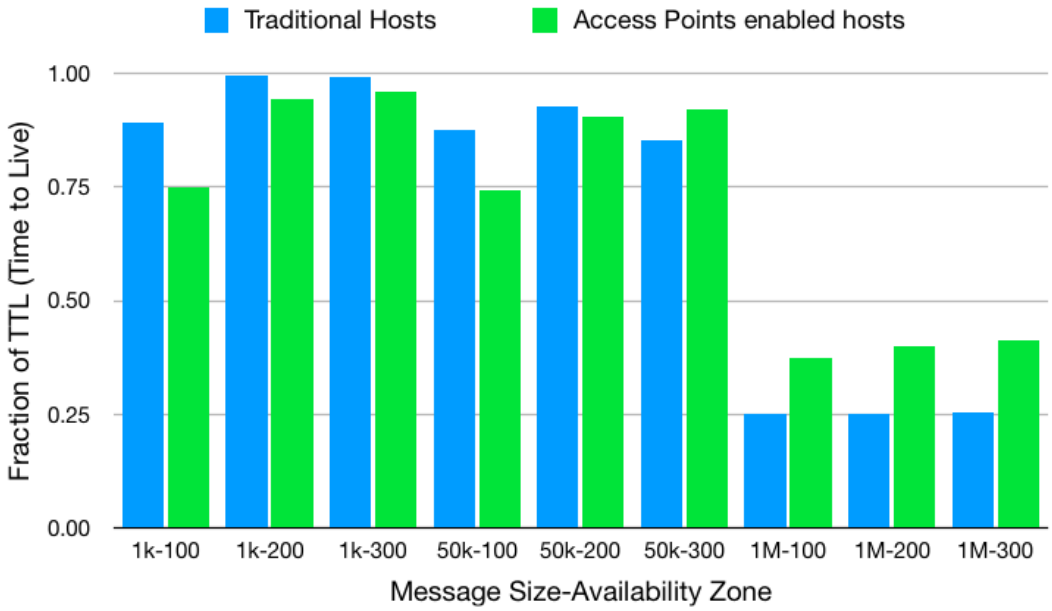


Figure 4.5: Message Size-Availability Zone vs The fraction of TTL (time to live)

4.3.2 Scenario 2

In this scenario, our main goal is to compare the network performance (measured in a fraction of TTL (time to live)) with respect to the number of access points a message can snap to (also known as **k**).

Message Size, Availability Zone, Hosts Count vs The fraction of TTL (time to live)

Let us start by plotting the fraction of TTL (time to live) against all the three parameters (message size, availability zone, and hosts count). *Figure* 4.6 plots all the variables against the fraction of TTL (time to live). This figure shows *snapping to access points* perform better than *no snapping*. Examining the figure closely reveals that the difference in performance is negligible in some cases while it is quite substantial in other cases. *Figure* 4.6 reveals that snapping to access points perform better than no snapping. We now need to understand if increasing the number of access points a message can snap

to (also known as **k**) really increases the fraction of TTL (time to live).

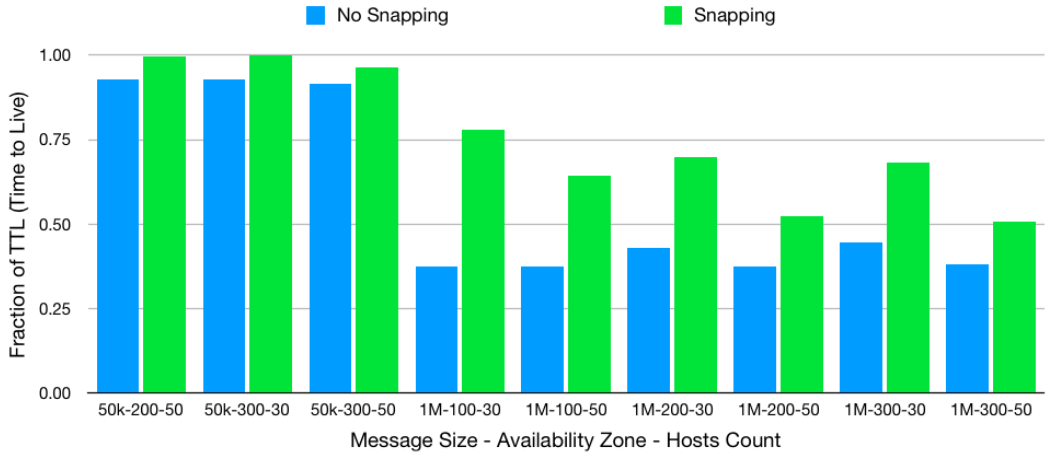


Figure 4.6: Message Size-Availability Zone-Hosts Count vs The fraction of TTL (time to live)

Figure 4.7 plots the fraction of TTL (time to live) w.r.t the number of access points the message is snapped to (**k**). We are plotting for all the k-values we are supporting i.e. 0,1,2,3. In case of smaller message size (or higher node capacity), increasing **k** does not have any effect on the fraction of TTL (time to live). However, in case of large message size (>= 1M), increasing **k** have a positive effect on the fraction of TTL (time to live).

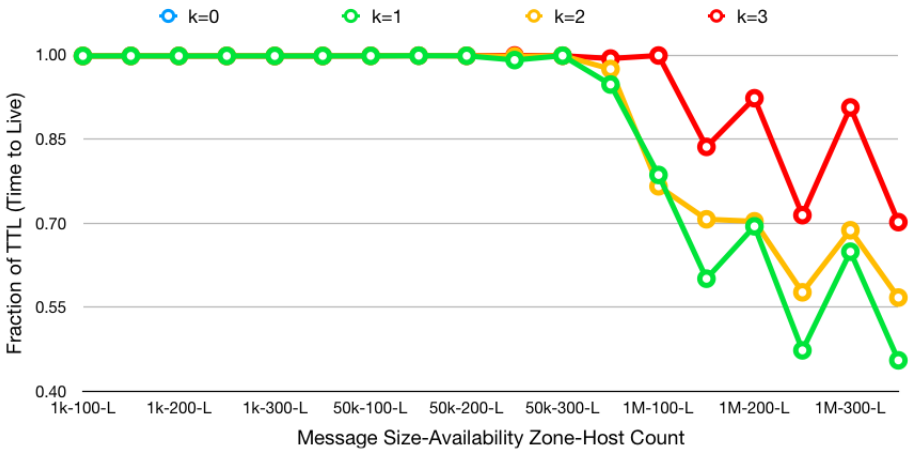


Figure 4.7: k (number of access points a message can snap to) vs The fraction of TTL (time to live)

In order to ascertain if the behavior depicted in *Figure* 4.6 and *Figure* 4.7, we will take each of the individual components and evaluate its effect on the fraction of TTL (time to live) for k = 0,1,2,3.

Message Size vs The fraction of TTL (time to live)

Figure 4.8 plots message size against the fraction of TTL (time to live). This figure shows that the case of no snapping (*k=0*) performs worse than the other cases (*k>0*). It also shows that when the message size is smaller (up to 50k), increasing **k** does not have any effect on the fraction of TTL (time to live). However, as the message size increases, increasing **k** have a positive effect on the fraction of TTL (time to live).

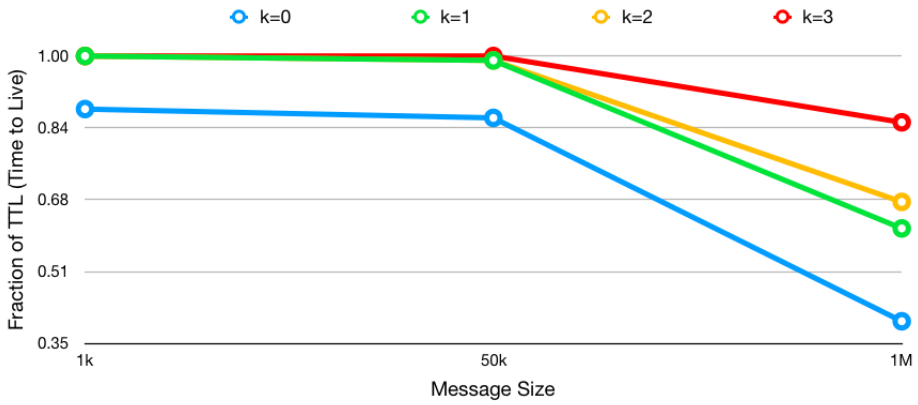


Figure 4.8: Message Size vs The fraction of TTL (time to live)

Availability Zone vs The fraction of TTL (time to live)

Figure 4.9 plots availability size against the fraction of TTL (time to live).

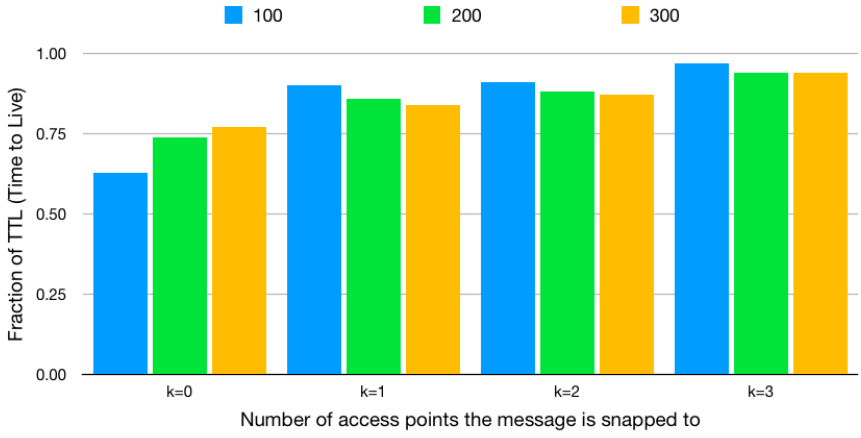


Figure 4.9: Availability Zone vs The fraction of TTL (time to live)

The above figure shows that the case of no snapping (*k=0*) always performs worse than the other cases (*k>0*). It also shows that increasing **k** (when changing the availability zone) always have a positive effect on the fraction of TTL (time to live). However, one

thing to note is that increasing the availability zone does not cause a minor decrease in network performance for access points enabled hosts. The main reason might be: as the availability zone increases, the access points would have to deal with more messages, resulting in more messages being dropped.

Number of Hosts vs The fraction of TTL (time to live)

Figure 4.10 plots number of hosts against the fraction of TTL (time to live). This figure shows that the case of no snapping (*k=0*) always performs worse than the other cases (*k>0*). It also shows that the increasing **k** (when changing the number of hosts) always have a positive effect on the fraction of TTL (time to live). In other words, higher values of **k** almost always correspond to a better fraction of TTL (time to live) than lower values.

Figure 4.10: Number of Hosts vs The fraction of TTL (time to live)

This marks the completion of our simulations and the analysis of their results. Using the analysis in this chapter, We would infer the results and discuss them in the next chapter.

5 Conclusions

In this chapter, we will discuss what we have infered from the simulations and their results. This will enable us to understand and summarize the behavior of access points as well as snapping to access point in ONE simulator. We will list the research questions below and then answer them as per the outcome of the simulation).

Q1: Modeling a city and exploring the impact of access points on the performance of geo-based information sharing.

Access points have an overall positive effect on the network. Below is a break-down with respect to three main parameters (message size, availability zone and, the number of hosts):

1. Information sharing works better with the increasing size of the message (or decreasing capacity of the nodes). In other words, as the message size increases, the average message life also increases.

2. Increasing the availability zone has a positive effect on information availability. In other words, as the availability zone size increases, so does the average message life.

3. Increasing the number of hosts also have a positive effect on information availability.

Q2: How does snapping to access point affect the availability of a floating message in a Delay Tolerant Network (DTN)?
Snapping to Access Points always has a positive impact on the average message life which means increased network performance. The number of access points a message can be snapped to is denoted by **k**. We have also noted the following:

1. Increasing *k* has no effect when the message size is small (less than or equal to 50k), however, as the message size increases, increasing **k** has a positive effect. The average message life is always better for higher values of **k** when compared to lower values.

2. Increasing **k** always has a positive effect regardless of the availability zone size. The average message life is always better for higher values of **k** than lower values.

3. Increasing **k** always has a positive effect regardless of the number of hosts. The average message life is always better for higher values of **k** when compared to lower values.

5.1 Future Work

We can add a number of improvements to the features we have implemented. The following paragraph explains some of those improvements:

The current behavior is that in case a host has a copy of the message, it will not accept new copy of the message. The disadvantage is that both the hosts might not have the updated information for the snapped access points. A feature would be to replicate this information (information about the access points the message is snapped to) between hosts even when a copy of the same message exists on that host.

Another feature would be to have specific nodes (preferably access points) responsible for holding the access points snapping information of all messages. These nodes will read such information from all the hosts and will send them this information for the message it holds.

Both of the above features would greatly improve the performance of the network.

6 Appendix

This appendix explains all the configurations that we have used in our simulations. In other words, we explain how you can reproduce our simulations by using the details in this chapter. Before going ahead, let me explain the three main components of our configurations:

- Static Configurations
- Configuration Generation Scripts
- Simulation Runners

The following sections would explain each of the above components in details.

6.1 Static Configurations

Static Configurations are constant (non-changeable) configurations. In other words, these are the configuration that is the same for all simulations. We are using two simulation files for such a configuration. One is known as *default_settings.txt* and the other is *floating_app.txt*. Both of these files will be explained in details in their own sub-section.

6.1.1 Default Settings

The file (default_settings.txt) contains all the general settings. In our case, it only contains the *Map file to be used* and some optimization settings. Below is a snapshot of this file.

```
# 1.
MapBasedMovement.nrofMapFiles = 1
MapBasedMovement.mapFile1 = data/maps/central_munich/map.wkt

# 2.
Optimization.cellSizeMult = 5
Optimization.randomizeUpdateOrder = true
```

Listing 6.1: default_settings.txt

The above listing is explain in details below:

1. Here we are specifying that we are using 1 map file for simulation, name of the file is map.wkt and it is located at data/maps/central_munich/.
2. Here are setting some optimizations. These optimizations affect the speed of the simulation. In our case, we are just using the default optimization settings.

The following subsection contains the second static configuration file we will be using.

6.1.2 Floating Application Settings

The file (floating_app.txt) contains all the general floating content related settings such as *message size and ttl* etc. Below is a snapshot of this file.

```
# 1.
floatingApp.type = FloatingApplication
floatingApp.seed = [1; 2; 3; 4; 5; 6; 7; 8; 9 ]
floatingApp.destination = 0

# 2.
floatingApp.startTime = 0
floatingApp.interval = 1800

# 3.
floatingApp.messageSize = 50k
floatingApp.ttl = 300

# 4.
floatingApp.mode = variable

# 5.
floatingApp.anchor = 0,0,500,500
floatingApp.anchorMax = 200,200,500,500

# 6.
floatingApp.anchorGranularity = 0,0,100,100

# 7.
FloatingContentRouter.replicationPolicy = none
FloatingContentRouter.replicationAlgorithm = none
FloatingContentRouter.deletionAlgorithm = none
FloatingContentRouter.seed = [1; 2; 3; 4; 5; 6; 7; 8; 9 ]
```

Listing 6.2: floating_app.txt

The above listing is explain in details below:

1. Here we are defining the type of *floatingApp* to be *Floating Application*. We are also defining a global destination for all the messages (0 means the destination is unreachable). We are also defining run-indexed seed (seed is used for random number generation).
2. Here we are defining the start time and interval for floating message generation. The first message is generated at *startTime* and the next message is uniformly chosen from [startTime + 0.75*interval ; startTime + 1.25 * interval]].
3. Here we are setting the size of the message (50 kilobytes) and time-to-live (ttl) of the message (to 300 minutes).
4. We are setting the anchor mode to variable, which means that the anchor zone is created w.r.t the current location of the message generating node.
5. Here we are specifying the *anchor (x,y,r,a)* and max *anchor (xmax,ymax,rmax,amax)* of the floating message. In case of variable mode (which is the one we are using), the anchor specifies the lower bound and max anchor specifies the upper bound of the anchor zone. In other words, *(x,y) and (xmax,ymax)* define the anchor zone. The unit for *r [r,rmax] and [a,amax]* shows the interval from which a and amax are selected. We are keeping the $a = r = amax = rmax$. The values for a and r are in meters.
6. Here we are setting the anchor granularity *(xg,yg,rg,ag)*. *xg and yg* defines the step-size at which content maybe floated, while *rg and ag* defines the granularity at which the respective ranges are chose.
7. Here we are setting the replication and deletion policies as well as seed for the floating content router.

The next section explains the scripts we are using for generating the dynamic configuration files.

6.2 Configuration Generation Scripts

This section contains the configuration generation scripts as well as some sample generated configuration files. We assume that these scripts are placed in a subfolder (one level inside the ONE simulator). In other words, going one folder back from this folder should take us to the main folder of ONE simulator (which is the folder containing *one.sh* or *one.bat* file).

6.2.1 Scenario 1

Here we will explain the configuration generation scripts for the first scenario. The script is annotated inline using comments (starting with # symbol). Below is the script we are using for generating the configuration.

```
#!/bin/bash
# filename: scenario1-gen.sh
# List of command line arguments to pass to this script
  # ${1} Scenario Name
  # ${2} Interface Type
  # ${3} Router
  # ${4} Nr of hosts
  # ${5} Simulation Type (it can either be adhoc or ap-sa but it will only
      work if ${2} is WifiInterface)
  # ${6} Message Size
  # ${7} Anchor Size

# 1.
outputfilepath="settings/scenario_${1}_${6}_${7}_${4}.txt"

# 2.
echo "Scenario.name = simulation_%%Scenario.runCount%%" >> $outputfilepath

# 3.
echo "Scenario.simulateConnections = true" >> $outputfilepath

# 4.
echo "Scenario.updateInterval = 0.1" >> $outputfilepath

# 5.
echo "Scenario.runCount = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]" >>
    $outputfilepath

# 6.
echo "Scenario.endTime = 43200" >> $outputfilepath

printf "\n" >> $outputfilepath

# 7.
echo "Scenario.nrofHostGroups = 3" >> $outputfilepath

printf "\n" >> $outputfilepath

# 8.
echo "MovementModel.rngSeed = 1" >> $outputfilepath

# 9.
```

```
echo "MovementModel.worldSize = 1940,1515" >> $outputfilepath

# 10.
echo "MovementModel.warmup = 1000" >> $outputfilepath

printf "\n" >> $outputfilepath

# 11.
echo "interface.type = ${2}" >> $outputfilepath

# 12.
echo "interface.transmitSpeed = 50k" >> $outputfilepath

# 13.
echo "interface.transmitRange = 40" >> $outputfilepath

printf "\n" >> $outputfilepath

# 14.
if [[ ${3} = "FloatingContentRouter" ]]; then
        echo "Group.nrofApplications = 1" >> $outputfilepath
        echo "Group.application1 = floatingApp" >> $outputfilepath
else
        echo "Group.nrofApplications = 0" >> $outputfilepath
fi

# Common settings for all the groups
# 15.
echo "Group.movementModel = MapBasedMovement" >> $outputfilepath

# 16.
echo "Group.router = ${3}" >> $outputfilepath

# 17.
echo "Group.bufferSize = 5M" >> $outputfilepath

# 18.
echo "Group.waitTime = 0, 120" >> $outputfilepath

# 19.
echo "Group.speed = 0.05, 0.15" >> $outputfilepath
```

```
# 20.
echo "Group.msgTtl = 300" >> $outputfilepath

# 21.
echo "Group.nrofHosts = ${4}" >> $outputfilepath

printf "\n" >> $outputfilepath

# 22.
echo "Group.nrofInterfaces = 1" >> $outputfilepath
echo "Group.interface1 = interface" >> $outputfilepath

printf "\n" >> $outputfilepath

# 23.
if [[ ${2} = "WifiInterface" ]]; then
        if [[ ${5} = "adhoc" ]]; then
                echo "Group1.groupID = AD" >> $outputfilepath
                echo "Group1.mode = adhoc" >> $outputfilepath
        else
                echo "Group1.groupID = AP" >> $outputfilepath
                echo "Group1.mode = ap" >> $outputfilepath
                echo "Group1.nrofApplications = 0" >> $outputfilepath
        fi
else
        echo "Group1.groupID = P" >> $outputfilepath
fi

# 24.
echo "Group1.movementModel = StationaryMapBasedMovement" >>
    $outputfilepath

# 25.
if (( ${4} >= 50 )); then
echo "Group1.routeFile = data/maps/central_munich/APs/every_200th_ap.wkt"
    >> $outputfilepath
echo "Group1.nrofHosts = 132" >> $outputfilepath
else
echo "Group1.routeFile = data/maps/central_munich/APs/every_400th_ap.wkt"
    >> $outputfilepath
echo "Group1.nrofHosts = 66" >> $outputfilepath
fi
```

```
echo "Group2.groupID = C" >> $outputfilepath

# 26.
if [[ ${2} = "WifiInterface" ]]; then
        if [[ ${5} = "adhoc" ]]; then
                echo "Group2.mode = adhoc" >> $outputfilepath
        else
                echo "Group2.mode = sa" >> $outputfilepath
        fi
fi

# 27.
echo "Group2.speed = 2.7, 13.9" >> $outputfilepath

# 28.
echo "Group2.okMaps = 1" >> $outputfilepath

echo "Group3.groupID = P" >> $outputfilepath

# 29.
if [[ ${2} = "WifiInterface" ]]; then
                if [[ ${5} = "adhoc" ]]; then
                        echo "Group3.mode = adhoc" >> $outputfilepath
                else
                        echo "Group3.mode = sa" >> $outputfilepath
                fi
fi

# 30.
if (( ${4} >= 50 )); then
echo "Group3.nrofHosts = 100" >> $outputfilepath
else
echo "Group3.nrofoHosts = $((${4} + ((${4}/2))))" >> $outputfilepath
fi
printf "\n" >> $outputfilepath

# 31.
prefix=""
if [[ ${3} = "FloatingContentRouter" ]]; then
        prefix="fcr_"
else
```

```
        prefix="er_"
fi

if [[ ${2} = "WifiInterface" ]]; then
        prefix="${prefix}wifi_"
        if [[ ${5} = "adhoc" ]]; then
                prefix="${prefix}adhoc"
        else
                prefix="${prefix}apsa"
        fi
else
        prefix="${prefix}nonwifi"
fi

printf "\n" >> $outputfilepath

# 32.
echo "floatingApp.anchor = 485,379,${7},${7}" >> $outputfilepath
echo "floatingApp.anchorMax = 1455,1136,${7},${7}" >> $outputfilepath
echo "floatingApp.anchorGranularity = 0,0,100,100" >> $outputfilepath

# 33.
echo "floatingApp.messageSize = ${6}" >> $outputfilepath
printf "\n" >> $outputfilepath

# 34.
echo "Report.reportDir=simulations/reports/Scenario1/${6}-${7}-${4}/${
    prefix}/"$(date "+%Y-%m-%d %H%M%S") >> $outputfilepath
reports=( "MessageReport" "MessageStatsReport" "DistanceDelayReport" "
    DeliveredMessagesReport" "MessageDeliveryReport")
if [[ ${3} = "FloatingContentRouter" ]]; then
        reports=( "FloatingMessageSummaryReport" "ConnectivityWifiONEReport
            ")
fi

# 35.
echo "Report.nrofReports=${#reports[@]}" >> $outputfilepath

# 36.
echo "Report.warmup=1000" >> $outputfilepath
echo "Report.granularity=1000" >> $outputfilepath
```

```
# 37.
printf "\n" >> $outputfilepath
counter=1
while [ $counter -le ${#reports[@]} ]
do
        echo "Report.report$counter=${reports[$(($counter-1))]}" >>
            $outputfilepath
        counter=$(( $counter + 1 ))
done

# 38.
cd ..

# 39.
./one.sh -b 10 floating_app.txt simulations/$outputfilepath
```

Listing 6.3: Scenario 1: Configuration Generation Script (scenario1_gen.sh)

Below is a detailed explanation of the above script.

1. We are setting the output file name for the configuration. The file is named as *scenario_ScenarioName_MessageSize_AnchorSize_HostsCount.txt*. We need to make sure that the settings folder is already created.

2. We are the setting name of the scenario. Here, We are using value-filling to get the scenario name number from *Scenario.runCount* using value filling.

3. We are making sure that the connections are simulated in this scenario.

4. We are setting the update interval (which is basically the speed) of the simulation.

5. We are setting an arbitrary variable (runCount) used for naming the scenario.

6. We are setting the scenario's end time (total duration) to 43200 seconds (or 12 hours).

7. We define the total number of node/hosts groups for the scenario.

8. We are setting the seed for movement model. This seed is used for random number generation. A number of parameters of movement models are generated using random numbers. Using the same seed provides us with the same random numbers.

9. We are setting the size of the movement model (width, height). The Munich city map we are using is 1940 meters wide and 1515 meters long.

10. We are setting the warm-up time for the movement models. This is the time when no interaction happens between the hosts.
11. We are setting the communication interface type to be used in this simulation. It is passed using the second command line argument.
12. We are setting the transmission speed for the communication interface.
13. We are setting the transmission range (in meters) for the interface.
14. We are setting the application to *floatingApp* for floating content router only. For other routers, we are setting the number of applications to 0.
15. We are setting the movement model for all the groups.
16. We are setting the router for all the groups. It is passed using the third command line argument.
17. We are setting the storage capacity (buffer size) of the nodes.
18. We are setting the wait time in seconds (minimum, maximum) for the nodes. It is the amount of time the hosts wait after reaching the destination.
19. We are setting the speed in m/s(minimum, maximum) for nodes of all groups. We are defining the walking speed to be the default speed.
20. We are setting the message's time-to-live(TTL) to 300 minutes (5 hours).
21. We are setting the number of hosts per group. It is passed using the fourth command line argument.
22. We are setting the interfaces for all the groups.
23. We are setting the mode of the first group to either ad-hoc or access point depending on the fifth command line parameter. In the case of access points, we are not attaching *floatingApp* as access points are not supposed to generate messages.
24. We are setting the movement model for Group 1.
25. We are setting the number of hosts for Group 1. In case the number of hosts is greater than or equal 50, we are using 132 hosts. Otherwise, We are using 66 hosts.
26. We are setting the correct mode for hosts of Group 2 based on the fifth command line arguments.
27. Group2 is used to simulate cars, so we are setting higher speeds(2.7 - 13.9 m/s = 10-50 km/h).
28. We are making sure that cars can only drive on roads.

29. We are setting the correct mode for hosts of Group 3 based on the fifth command line argument.
30. In case the number of hosts is greater than or equal to 50, we are using 100 as the total hosts otherwise we are using lower number.
31. We are calculating the prefix that determines the subfolder for reports.
32. We are setting the anchor properties. These settings override the settings from floating_app.txt.
33. We are setting the message size. This setting overrides the settings from floating_app.txt.
34. We are setting the reports directory as well as the reports that we are interested in.
35. We are setting the number of reports to generate.
36. We are setting the warm-up time for reports to 1000 seconds. It means that nothing will be reported until 1000 seconds.
37. We are providing the names of (classes of) the report files we are interested in.
38. Going one directory back to the directory of *one.sh*.
39. Starting ONE simulator in batch mode with 10 consecutive runs.

The next sub-section explains the configuration generation script for the second scenario.

6.2.2 Scenario 2

Below is the script we are using for generating the configuration for scenario 2.

```
#!/bin/bash
# file name: scenario2_gen.sh
# List of command line arguments to pass to this script
  # ${1} Scenario Name
  # ${2} kNextAPs
  # ${3} Hosts
  # ${4} Message Size
  # ${5} anchor

# 1.
outputfilepath="settings/scenario2/scenario_${1}_${2}_${4}_${5}_${3}.txt"

# 2.
echo "Scenario.name = simulation_%%Scenario.runCount%%" >> $outputfilepath
```

```
# 3.
echo "Scenario.simulateConnections = true" >> $outputfilepath

# 4.
echo "Scenario.updateInterval = 0.1" >> $outputfilepath

# 5.
echo "Scenario.runCount = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]" >>
    $outputfilepath

# 6.
echo "Scenario.endTime = 43200" >> $outputfilepath

printf "\n" >> $outputfilepath

# 7.
echo "Scenario.nrofHostGroups = 3" >> $outputfilepath

printf "\n" >> $outputfilepath

# 8.
echo "MovementModel.rngSeed = 1" >> $outputfilepath


# 9.
echo "MovementModel.worldSize = 1940,1515" >> $outputfilepath

# 10.
echo "MovementModel.warmup = 1000" >> $outputfilepath

printf "\n" >> $outputfilepath

# 11.
echo "interface.type = WifiInterface" >> $outputfilepath

# 12.
echo "interface.transmitSpeed = 50k" >> $outputfilepath

# 13.
echo "interface.transmitRange = 40" >> $outputfilepath

printf "\n" >> $outputfilepath
```

```
# 14.
echo "Group.nrofApplications = 1" >> $outputfilepath
echo "Group.application1 = floatingApp" >> $outputfilepath

# 15.
echo "Group.movementModel = MapBasedMovement" >> $outputfilepath

# 16.
echo "Group.router = FloatingContentRouter" >> $outputfilepath

# 17.
echo "Group.bufferSize = 5M" >> $outputfilepath

# 18.
echo "Group.waitTime = 0, 120" >> $outputfilepath

# 19.
echo "Group.speed = 0.05, 0.15" >> $outputfilepath

# 20.
echo "Group.msgTtl = 300" >> $outputfilepath

# 21.
echo "Group.nrofHosts = ${3}" >> $outputfilepath

printf "\n" >> $outputfilepath

# 22.
echo "Group.nrofInterfaces = 1" >> $outputfilepath
echo "Group.interface1 = interface" >> $outputfilepath

printf "\n" >> $outputfilepath

# 23.
echo "Group1.groupID = AP" >> $outputfilepath
echo "Group1.mode = ap" >> $outputfilepath

# 24.
echo "Group1.movementModel = StationaryMapBasedMovement" >>
    $outputfilepath
```

```
# 25.
if (( ${3} >= 50 )); then
echo "Group1.routeFile = data/maps/central_munich/APs/every_200th_ap.wkt"
    >> $outputfilepath
echo "Group1.nrofHosts = 132" >> $outputfilepath
else
echo "Group1.routeFile = data/maps/central_munich/APs/every_400th_ap.wkt"
    >> $outputfilepath
echo "Group1.nrofHosts = 66" >> $outputfilepath
fi

# 26.
echo "Group1.nrofApplications = 0" >> $outputfilepath

printf "\n" >> $outputfilepath

# 27.
echo "Group2.groupID = C" >> $outputfilepath
echo "Group2.mode = sa" >> $outputfilepath

echo "Group3.groupID = P" >> $outputfilepath
echo "Group3.mode = sa" >> $outputfilepath

#28.
echo "Group2.okMaps = 1" >> $outputfilepath

# 29.
echo "Group2.speed = 2.7, 13.9" >> $outputfilepath

printf "\n" >> $outputfilepath

# 30.
if (( ${3} >= 50 )); then
echo "Group3.nrofHosts = 100" >> $outputfilepath
else
echo "Group3.nrofoHosts = $((${3} + ((${3}/2))))" >> $outputfilepath
fi

printf "\n" >> $outputfilepath

# 31.
echo "FloatingContentRouter.kNextAPs = ${2}" >> $outputfilepath
```

```
printf "\n" >> $outputfilepath

printf "\n" >> $outputfilepath

# 32.
echo "floatingApp.anchor = 485,379,${5},${5}" >> $outputfilepath
echo "floatingApp.anchorMax = 1455,1136,${5},${5}" >> $outputfilepath
echo "floatingApp.anchorGranularity = 0,0,100,100" >> $outputfilepath

# 33.
echo "floatingApp.messageSize = ${4}" >> $outputfilepath
printf "\n" >> $outputfilepath

# 34.
echo "Report.reportDir=simulations/reports/Scenario2/${4}-${5}-${3}/k_$
    {2}/"$(date "+%Y-%m-%d %H%M%S") >> $outputfilepath
reports=("FloatingMessageSnapToAPSummaryReport" "
    FloatingMessageAnchorPointsReport")

# 35.
echo "Report.nrofReports=${#reports[@]}" >> $outputfilepath

# 36.
echo "Report.warmup=1000" >> $outputfilepath
echo "Report.granularity=1000" >> $outputfilepath

printf "\n" >> $outputfilepath

counter=1

#37.
while [ $counter -le ${#reports[@]} ]
do
        echo "Report.report$counter=${reports[$(($counter-1))]}" >>
            $outputfilepath
        counter=$(( $counter + 1 ))
done

# 38.
cd ..
```

```
# 39.
./one.sh -b 10 floating_app.txt simulations/$outputfilepath
```

Listing 6.4: Scenario 2: Configuration Generation Script (scenario2_gen.sh)

1. We are setting the output file name for the configuration. The file is named as *scenario_ScenarioName_kNextAPs_AnchorSize_MessageSize_HostsCount.txt*. We need to make sure that the settings folder is already created.
2. We are setting name of the scenario. Here, We are using value-filling to get the scenario name number from *Scenario.runCount* using value filling.
3. We are making sure that the connections are simulated in this scenario.
4. We are setting the update interval (which is basically the speed) of the simulation.
5. We are setting an arbitrary variable (runCount) used for naming the scenario.
6. We are setting the scenario's end time (total duration) to 43200 seconds (or 12 hours).
7. We define the total number of node/hosts groups for the scenario.
8. We are setting the seed for movement model. This seed is used for random number generation. A number of parameters of movement models are generated using random numbers. Using the same seed provides us with the same random numbers.
9. We are setting the size of the movement model (width, height). The Munich city map we are using is 1940 meters wide and 1515 meters long.
10. We are setting the warm-up time for the movement models. This is the time when no interaction happens between the hosts.
11. We are setting the communication interface type to be used in this simulation. Since this simulation is related to access points, we always use WifiInterface.
12. We are setting the transmission speed for the communication interface.
13. We are setting the transmission range (in meters) for the interface.
14. We are setting the application to *floatingApp* for floating content router only. For other routers, we are setting the number of applications to 0.
15. We are setting the movement model for all the groups.
16. We are setting the router for all the groups. We are making sure all the groups use *FloatingContentRouter*.

17. We are setting the storage capacity (buffer size) of the nodes.
18. We are setting the wait time in seconds (minimum, maximum) for the nodes. It is the amount of time the hosts wait after reaching the destination.
19. We are setting the speed in m/s(minimum, maximum) for nodes of all groups. We are defining the walking speed to be the default speed.
20. We are setting the message's time-to-live(TTL) to 300 minutes (5 hours).
21. We are setting the number of hosts per group. It is passed using the third command line argument.
22. We are setting the interfaces for all the groups.
23. We are setting the first group's hosts as access points.
24. We are setting the Movement model for Group 1.
25. We are setting the number of hosts for Group 1. In case the number of hosts is greater than or equal 50, we are using 132 hosts. Otherwise, We are using 66 hosts.
26. We are setting the number of applications for access points (group 1) to 0 as access points don't generate messages.
27. We are setting Group2 and Group 3 hosts to be station adapters (SA).
28. We are making sure that cars can only drive on roads.
29. Group2 is used to simulate cars, so we are setting higher speeds(2.7 - 13.9 m/s = 10-50 km/h).
30. In case the number of hosts is greater than or equal to 50, we are using 100 as the total hosts otherwise we are using the the lower number.
31. We are setting the number of access points a host can snap to. It is passed as the second command line argument.
32. We are setting the anchor properties. These settings override the settings from floating_app.txt.
33. We are setting the message size. This setting overrides the settings from floating_app.txt.
34. We are setting the reports directory as well as the reports that we are interested in.
35. We are setting the number of reports to generate.
36. We are setting the warm-up time for reports to 1000 seconds. It means that nothing will be reported until 1000 seconds.

37. We are providing the names of (classes of) the report files we are interested in.
38. Going one directory back to the directory of *one.sh*.
39. Starting ONE simulator in batch mode with 10 consecutive runs.

6.3 Simulation Runners

This section combines the static configuration files as well as the configuration generation scripts to actually run the simulations for different combinations of message size, number of hosts and anchor zone size.

6.3.1 Scenario 1

Below are the two annotated script files we are using for this scenario.

```
#!/bin/bash
#file name: scenario1.sh

# Making sure all the parameters are provided
if [ -z "${1}" ] || [ -z "${2}" ] || [ -z "${3}" ]; then
 echo "Missing arguments. Please use the following syntax:"
 echo "./scenario1.sh messageSize availabilityZone hostCount"
else

# Running scenario1-gen.sh (the configuration generator) for the three
    different scenarios (non access points enabled hosts, adhoc hosts and
    access-points enabled hosts). We are using tmux so that each task
    starts in its own detachable process.

tmux new -d -s fcr_wifi_apsa_${1}_${2}_${3} ./scenario1-gen.sh
    fcr_wifi_apsa WifiInterface FloatingContentRouter ${3} apsa ${1} ${2}
sleep 1
tmux new -d -s fcr_wifi_adhoc_${1}_${2}_${3} ./scenario1-gen.sh
    fcr_wifi_adhoc WifiInterface FloatingContentRouter ${3} adhoc ${1} ${2}
sleep 1
tmux new -d -s fcr_non_wifi_${1}_${2}_${3} ./scenario1-gen.sh fcr_non_wifi
     SimpleBroadcastInterface FloatingContentRouter ${3} apsa ${1} ${2}
fi
```

Listing 6.5: Scenario 1: Intermediate script (scenario1.sh)

Now, we have the configuration generator (*scenario1-gen.sh*) and the intermediate script (*scenario1.sh*) in place, we can use the final script to start simulation for all of our desired

combinations. Below is the code for the final script (*scenario1-runner.sh*).

```
#!/bin/bash
#file name: scenario1-runner.sh

./scenario1.sh 1k 100 30
./scenario1.sh 1k 100 50
./scenario1.sh 1k 200 30
./scenario1.sh 1k 200 50
./scenario1.sh 1k 300 30
./scenario1.sh 1k 300 50

./scenario1.sh 50k 100 30
./scenario1.sh 50k 100 50
./scenario1.sh 50k 200 30
./scenario1.sh 50k 200 50
./scenario1.sh 50k 300 30
./scenario1.sh 50k 300 50

./scenario1.sh 1M 100 30
./scenario1.sh 1M 100 50
./scenario1.sh 1M 200 30
./scenario1.sh 1M 200 50
./scenario1.sh 1M 300 30
./scenario1.sh 1M 300 50
```

Listing 6.6: Scenario 1: Final script (scenario1-runner.sh)

We have all the scripts already in place now, we can easily use the following command to start the simulation:

```
./scenario1-runner.sh
```

6.3.2 Scenario 2

Scenario 2 has a similar setup as Scenario 1, however, the contents of the runner scripts are different. Below are the two annotated script files that we are using for this scenario.

```
#!/bin/bash
#file name: scenario2.sh

# Making sure all the parameters are provided
if [ -z "${1}" ] || [ -z "${2}" ] || [ -z "${3}" ]; then
 echo "Missing arguments. Please use the following syntax:"
```

```
 echo "./scenario2.sh messageSize availabilityZone hostCount"
else

# Running scenario2-gen.sh (the configuration generator) for the four
    different values snapping to accessing point(k=0,1,2,3).

tmux new -d -s fcr_k_0_${1}_${2}_${3} ./scenario2-gen.sh k 0 ${3} ${1} ${2}

sleep 1
tmux new -d -s fcr_k_1_${1}_${2}_${3} ./scenario2-gen.sh k 1 ${3} ${1} ${2}

sleep 1
tmux new -d -s fcr_k_2_${1}_${2}_${3} ./scenario2-gen.sh k 2 ${3} ${1} ${2}

sleep 1
tmux new -d -s fcr_k_3_${1}_${2}_${3} ./scenario2-gen.sh k 3 ${3} ${1} ${2}


fi
```

Listing 6.7: Scenario 2: Intermediate script (scenario2.sh)

Now, we have the configuration generator (*scenario1-gen.sh*) and the intermediate script (*scenario1.sh*) in place, we can use the final script to start simulation for all of our desired combinations. Below is the code for the final script ((scenario1-runner.sh)).

```
#!/bin/bash
#file name: scenario2-runner.sh

./scenario2.sh 1k 100 30

./scenario2.sh 1k 100 50
./scenario2.sh 1k 200 30
./scenario2.sh 1k 200 50
./scenario2.sh 1k 300 30
./scenario2.sh 1k 300 50

./scenario2.sh 50k 100 30
./scenario2.sh 50k 100 50
./scenario2.sh 50k 200 30
./scenario2.sh 50k 200 50
./scenario2.sh 50k 300 30
./scenario2.sh 50k 300 50
```

```
./scenario2.sh 1M 100 30
./scenario2.sh 1M 100 50
./scenario2.sh 1M 200 30
./scenario2.sh 1M 200 50
./scenario2.sh 1M 300 30
./scenario2.sh 1M 300 50
```

Listing 6.8: Scenario 2: Final script (scenario2-runner.sh)

We have all the scripts already in place now, we can easily use the following command to start the simulation:

```
./scenario2-runner.sh
```

This marks the completion of all the steps needed to reproduce our simulation scenarios.

List of Figures

List of Tables

Bibliography

[CC06] D. J. Corbett and D. Cutting. "AD LOC : Location-based Infrastructure-free Annotation." In: 2006.

[CSK09] A. A. V. Castro, G. D. M. Serugendo, and D. Konstantas. "Hovering Information – Self-Organizing Information that Finds its Own Storage." In: *Autonomic Communication*. Ed. by A. V. Vasilakos, M. Parashar, S. Karnouskos, and W. Pedrycz. Boston, MA: Springer US, 2009, pp. 111–145. ISBN: 978-0-387-09753-4. DOI: 10.1007/978-0-387-09753-4_5.

[Fal03] K. Fall. "A delay-tolerant network architecture for challenged internets." In: (2003), pp. 27–34.

[Hyy+11] E. Hyytiä, J. Virtamo, P. Lassila, J. Kangasharju, and J. Ott. "When does content float? Characterizing availability of anchored information in opportunistic content sharing." In: (May 2011), pp. 3137–3145.

[Ker] A. Keränen. *The Opportunistic Network Environment simulator README*. URL: https://github.com/akeranen/the-one/wiki/README.

[KOK09] A. Keränen, J. Ott, and T. Kärkkäinen. "The ONE Simulator for DTN Protocol Evaluation." In: *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. Rome, Italy: ICST, 2009. ISBN: 978-963-9799-45-5.

[Mapa] O. S. Maps. *Open Street Maps*. URL: https://www.openstreetmap.org/.

[Mapb] O. S. Maps. *Open Street Maps Export*. URL: https://www.openstreetmap.org/export.

[Mapc] O. S. Maps. *OSM Filter*. URL: https://wiki.openstreetmap.org/wiki/Osmfilter.

[May10] C. P. Mayer. *osm2wkt - OpenStreetMap to WKT Conversion*. http://www.chrismc.de/osm2wkt. 2010.

[OLK] J. Ott, C. Luo, and A. Keränen. *Security Infrastructure for DTN*. URL: https://www.netlab.tkk.fi/tutkimus/sindtn/.

[Ope] OpenJump. *OpenJump*. URL: http://www.openjump.org.

[Ott+] J. Ott, J. Kangasharju, J. Virtamo, P. Lassila, and E. Hyytiä. *Floating Content*. URL: http://www.floating-content.net/.

[Ott+11] J. Ott, E. Hyytiä, P. E. Lassila, T. Vaegs, and J. Kangasharju. "Floating content: Information sharing in urban areas." In: *2011 IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2011), pp. 136–146.

[Ott+17] J. Ott, L. Kärkkäinen, E. A. Walelgne, A. Keränen, E. Hyytiä, and J. Kangasharjui. "On the sensitivity of geo-based content sharing to location errors." In: (Feb. 2017).

[Rad] RadioCells.Org. *Wifi and Cell Data - Germany*. URL: `https://cdn.radiocells.org/de.sqlite`.

[TCK10] N. Thompson, R. Crepaldi, and R. Kravets. "Locus: A Location-based Data Overlay for Disruption-tolerant Networks." In: *Proceedings of the 5th ACM Workshop on Challenged Networks*. CHANTS '10. Chicago, Illinois, USA: ACM, 2010, pp. 47–54. ISBN: 978-1-4503-0139-8. DOI: 10.1145/1859934.1859945.