UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

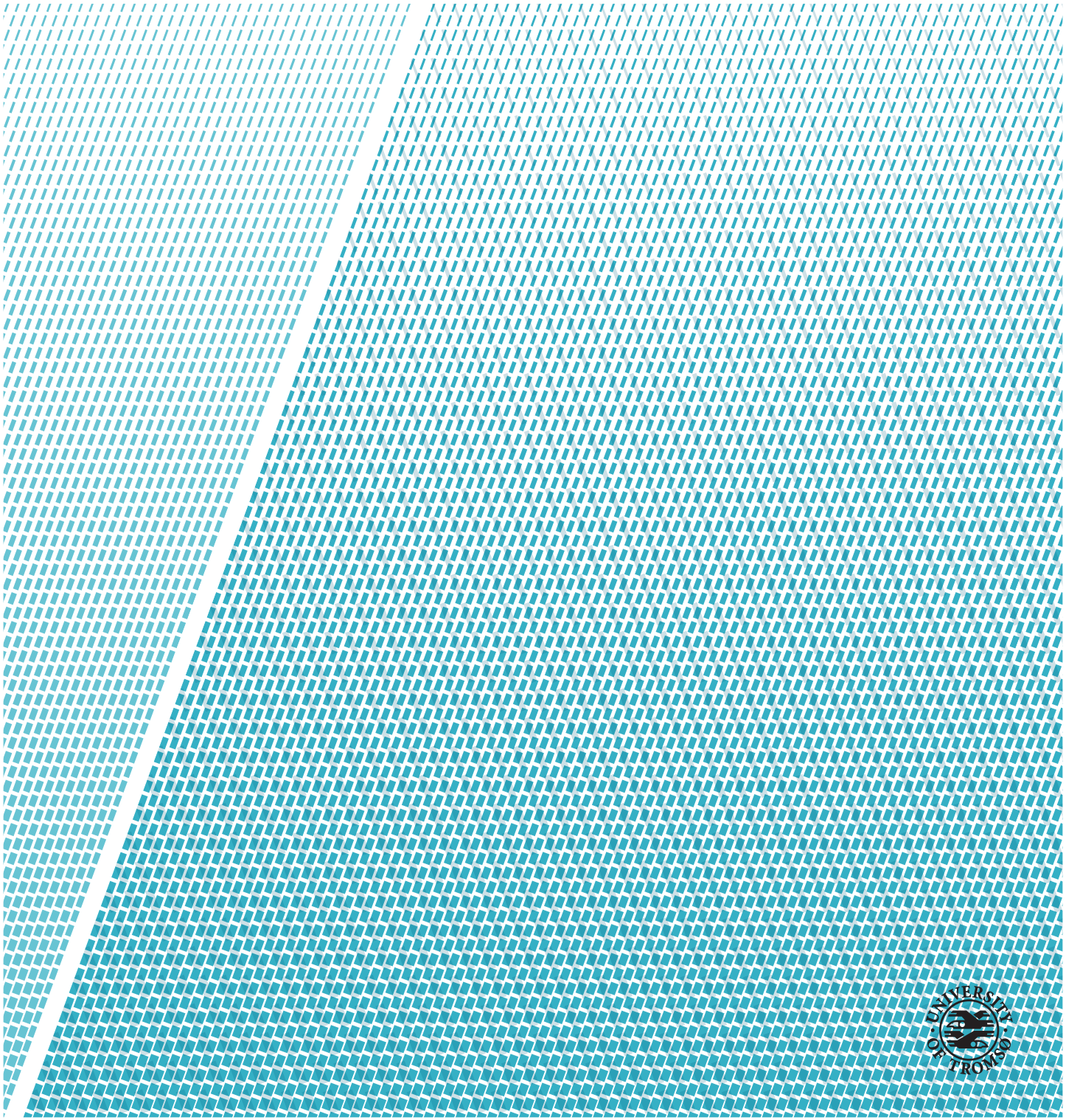
DMNI

Dynamic Mobile Network Infrastructure

—

Simon Kristoffer Nilsen Fagerli

Master thesis in INF-3981 - June 2018



“Sleep when you find upcoming work”
–Emil Jønsson

“Start early, fail early”
–Otto Anshus

Abstract

Each winter the Climate-Ecological Observatory for Arctic Tundra (COAT) project deploys a range of small devices to measure and monitor the climate changes that occur in the Arctic regions in an attempt to gain better understanding of how the changes are affecting the Arctic tundra ecosystems. The deployed devices are often limited in terms of energy and connectivity range. Due to this, researchers face the issue of not being able to efficiently extract data from the devices placed on the Arctic tundra - this is often a manual and tedious task as researchers have to themselves collect data from the devices.

This dissertation describes and implements a simulation of detached, interconnected sub-networks consisting of energy efficient Observation Units (OUs) placed on the Arctic tundra. A mobile data gathering device, a Mobile Ubiquitous LAN Extension (MULE), moving between the sub-networks creates a dynamically, temporary on-demand network which the detached networks may utilize to store and forward data reliably back to persistent storage. Dynamic Mobile Network Infrastructure (DMNI) presents a three layered architecture which forms the basis of the thesis - the application layer consisting of backend services, the network layer consisting of MULEs and the data layer with the isolated partitioned ad hoc networks of interconnected OUs.

By utilizing data MULEs, we show through simulation and experiments that we can mitigate the limitation that systems placed in remote areas may face - permanent partitioning and complete disconnection from backend systems. By using a mesh-like structure in the sub-networks, we show that a MULE only require a single connection to an OU part of the network to accumulate all data - actively reducing the time, power and complexity to collect data. Simulation and experiments show that we can reduce the package-loss ratio to below 5%, even as low as 3.01%, by using a MULE to OU ratio of 30%. It also shows that the system has a low CPU and memory footprint on a real device, only using 2.2% total device CPU and 1.3% total device RAM.

DMNI provides a solid first step towards a more refined MULE based system for data accumulation from remote, partitioned ad hoc networks of interconnected OUs in the Arctic.

Acknowledgements

First and foremost, I would like to say thank you to my main advisor, Professor Otto Anshus and my co-advisor, Professor John Markus Bjørndalen for all the ideas, guidance and general input throughout the period of making this dissertation. Your advice and knowledge has been greatly appreciated.

Secondly, I'd like to say thank you to all the IT and administrative staff at the Department of Computer Science at UiT for all assistance and guidance.

I'd like to give my sincerest gratitude to all my fellow co-students for providing a memorable time with lots of laughter and joy.

Further, I'd like to say thank you to my family and girlfriend for being beside me and supporting me and all my decisions. I would never have been able to do this without all of you by my side.

Lastly, a special mention and thank you goes to *Masterinos* for being an amazing group of friends. You guys have really been there when things have been tough but we have all backed up each other and it has gotten us to this point. Again, thank you guys.

Contents

List of Figures

List of Tables

My list of definitions

List of Abbreviations

b bit

COAT Climate-Ecological Observatory for Arctic Tundra

DAO Distributed Arctic Observation

DMNI Dynamic Mobile Network Infrastructure

DTR Data Transfer Rate

GAF Geographic Adaptive Fidelity

HTTP Hypertext Transfer Protocol

IOT Internet Of Things

JSON JavaScript Object Notation

LAN Local Area Network

MB Megabyte

MPDG Minimum-Path Data-Gathering

ms millisecond

MULE Mobile Ubiquitous LAN Extension

OU Observation Unit

PBS Partitioning Based Scheduling

RPi Raspberry Pi 3 Model B

RR Round-Robin

S second

TGP Two General's Problem

UAV Unmanned Aerial Vehicle

UiT University of Tromsø

UMI Unique Message Identifier

WSN Wireless Senor Network

1 Introduction

With global warming being all over the media for the past decade, it has become ever more important to monitor and preserve the globe which we inhabit. Although there has been a rise in the amount of awareness, there is unfortunately one area which has been given little attention compared to the rest of earth - the Arctic tundra.

Researchers predict that the Arctic tundra will be one of the areas mostly affected by the climate changes we are facing today and will most definitely face in the future[26, 19].

It is with this in mind that the Climate-Ecological Observatory for Arctic Tundra (COAT)[1] project[26] was initiated in 2010 by five Fram Centre[2] institutions. It is the goal of COAT to become world-leading within the field of "*Adaptive long-term research in the face of climate change*"[18]. The project aims to create an observation system to be used for monitoring and documentation of the impacts which climate changes has on the Arctic tundra ecosystems.

In 2018, the Distributed Arctic Observation (DAO)[3] project out of the University of Tromsø (UiT) was funded by the Norwegian Research Council (NRC)[3] as a direct effort to solve some of the technical challenges and issues which

1. COAT: `http://www.coat.no/`
2. Framsenteret: `http://www.framsenteret.no/english`
3. NRC: `https://www.forskningsradet.no/en/Home_page/1177315753906`

such an observation system may experience.
The proposed observation system from DAO consists of small energy efficient, wireless devices (Observation Units (OUs)) placed on the Arctic tundra to provide researchers with the data they require.

Due to the limitations that one is faced with by placing OUs remotely in the Arctic regions, such as a limited radio range for communication, there has to be a method of extracting the data from the OUs in an efficient manner.

1.1 Problem definition

This dissertation focuses on the extraction of data from ad hoc networks of interconnected OUs via a Mobile Ubiquitous LAN Extension (MULE) in an attempt to avoid permanent, or too long, partitioning of the networks. The MULEs are considered data sinks and are used as a method of storing and forwarding data from the ad hoc networks of OUs to a backend system with permanent, non-volatile storage.

The goal of the dissertation is to gain a better understanding of how one can improve the DAO system by utilizing a MULE based data-collection system. In addition, it is important to map out the limitations and pitfalls one can experience by utilizing such a system.

This thesis presents the architecture, design and implementation of a MULE based data-collection system for the Arctic tundra. Due to the early stage which Dynamic Mobile Network Infrastructure (DMNI) and DAO is at, a vast network of OUs are not deployed on the Arctic tundra. Instead, the thesis implements a simulator of the Arctic tundra to be utilized for experimental and evaluation purposes.

1.2 Technical idea

The idea of DMNI is that OUs are placed on the Arctic tundra, forming sub-networks of interconnected devices. A MULE, or multiple MULEs, are then to physically move around until a sub-network of OUs is reached. Once a sub-network of OUs is encountered, the MULE can accumulate data from the entire sub-network by being in contact with one or more OUs of said network.

OUs that are connected in a sub-network will use those connections to forward data on behalf of other members in the sub-network. This way data can then

be passed to the MULE.

1.3 Contributions

This thesis makes the following contributions:

- A description of the utilization of MULEs in an Arctic environment
- A implementation of a simulator and visualizer which can be utilized to simulate the Arctic environment and monitor the interaction between MULEs and OUs in a virtual grid
- A prototype for accumulation of data from ad hoc networks of interconnected OUs utilizing a MULE
- An evaluation of the system
- Insights on future work and potential improvements which can be used to improve the system and potentially allow for a prototype of a MULE based data accumulation system

1.4 Limitations

This dissertation focuses mainly on the interaction between ad hoc networks of interconnected OUs and MULEs and does not implement the backend section which would be required in a fully functional system. MULEs act as data sinks but what happens with the data once it has reached the MULE is not part of this dissertation. The same applies for the data itself as it is irrelevant what type of data is sent and this dissertation simply creates dummy data which is sent between devices.

2

Related work

Over the past decade, the field of Internet Of Things (IOT) has grown significantly and it has allowed researchers to utilize small energy efficient devices to monitor remote areas. With this comes the complicated task of retrieving data from remote and isolated data collecting devices. Most of the time it is the responsibility of the researchers themselves to manually collect data, which has proven to be a tiresome and ineffective method.
There are some systems and methods which has prioritized to reduce the amount of manual labour which researchers has to go through to collect, but most of these are based in milder climates than the Arctic tundra.

2.1 Mobile Ubiquitous LAN Extensions - MULEs

The term *MULE*, or Mobile Ubiquitous LAN Extension, was first coined in the paper "*Data mules: Modeling and analysis of a three-tier architecture for spare sensor networks*"[30] in 2003 with the goal to collect sensor data in sparse Wireless Senor Networks (WSNs). The core of the paper was to retain the advantages one can gain by deploying remote, static base-stations in a sparse WSN yet keeping it cost effective and energy effective enough to still be feasible. This was achieved by utilizing only short-range radios to communicate with the data Mobile Ubiquitous LAN Extensions (MULEs) and it was suggested that Ultra-Wideband (UWD) radio technology would be an ideal candidate for such communication.

It was however stated that the approach would only be suitable for delay-tolerant application due to the increased latency experienced by having mobile data collectors.

Several other projects has experimented with mobile units to provide communication paths, such as '*Ad Hoc Relay Wireless Networks over Moving Vehicles on Highways*'[9] - in which the movement of vehicles on a highway is utilized to relay data. Dynamic Mobile Network Infrastructure (DMNI) does this to an extend, but in contrast to [9], Observation Units (OUs) are static and does not move but it is the responsibility of te MULE to relay the data from the networks to backend systems.

A project that is rather similar to DMNI is ZebraNet[21] from 2002, in which they experimented with mobile data carriers connected to zebras.
The principle which ZebraNet relies on is the movement patterns of the zebras that actively carry data from access-point to access-point and data duplication to ensure that data is successfully transmitted from the zebras to access-points. Fundamentally, ZebraNet is a peer-to-peer system; gossiping data between nodes(zebras). Further, ZebraNet's protocol evaluation shows that a 100% success rate for base station data transfer is possible using mobile agents and a peer-to-peer system. Not only did this result in lower energy consumption due to only using short-range radios which reached 6km instead of 11km, but it also allowed redundancy in data. In contrast to ZebraNet, DMNI utilizes the structure of the ad hoc network to more efficiently extract data. In addition, MULEs in DMNI attempts to wait for data to accumulate from ad hoc networks whilst ZebraNet has no guarantee that zebras will wait long enough for data to move between the nodes.

Further work involving mobile data-collectors involve "*Exploiting Mobility for Energy Efficient Data Collection in Wireless Sensor Networks*"[20] by Sushant Jain et al. from 2006 where they addressed the issue of energy efficient data collection in a sensor network. Much like DMNI, they utilized mobile "agents" to carry data from secluded areas to access-points where data is dropped off, but using mobile agents solely as an alternative to ad hoc networks. The sole purpose of the mobile agents was to collect data and reduce the energy consumption of the system - DMNI integrated the mobile agents as "part" of the sub-networks and thus can be used for multiple other tasks. As an example, MULEs can be placed in areas to bridge gaps between sub-networks to for example allow for data to flow to a sink where the nodes closest to the sink has gone offline.

Definition 1. Sink: Data-access point which has a connection to persistent, non-volatile storage

The paper "*Data Gathering by Mobile Mules in a Spatially Separated Wireless Sensor Network*"(2009)[35] by Fang-Jing Wu et al. explores the possibility of utilizing mobile MULEs for collecting data from spatially separated WSN and focuses on optimizing the routing of the mobile MULE to reduce the latency and energy consumption in the sub-networks. The paper formulates the '*MPDG, or Minimum Path Data-Gathering Problem*' which is a generalization of the Euclidean Traveling Salesman problem[28], in which the optimization goal is to minimize the message drops. It is proven through simulation that a convex-hull algorithm[2] is more useful when there is a large number of sub-networks. DMNI does not focus on the pathing of the MULEs, but instead on the interaction between MULEs and ad hoc networks. This also applies to papers such as '*On Best Drone Tour Plans for Data Collection in Wireless Sensor Network*'[10] where they consider the problem is to find the *Best Drone Tour Plan*.

> *"To find (1) a sequence of C collecting points $c_i \in P$ where the drone will hover for collecting data and (2) C disjoint sets of nodes $S_i \subseteq N$, where all nodes in S_i will transmit their data to the drone hovering at the collecting point c_i and $\cup_{k=1}^{C} S_k = N$ so that the total time needed to fly from p_0 over the collecting points and return to p_0 (T_{trip}) plus the time to collect all the data from the WSN ($T_{collecting}$), which we denote as T_{BDTP}, is minimized."*[10]

2.2 Ad hoc networks

Ad hoc networks, or WSNs are essential for DMNI and are utilized together with the MULEs to together form a better fit solution for the Arctic tundra. For WSNs placed on the Arctic tundra, energy conservation is of great importance.

The Power and Reliability Aware Protocol (*PORAP*) introduces work aimed at lowering the energy consumption in WSNs[23]. This is done by identifying scenarios where a single hop communication scheme between multiple OUs and a sink is feasible and offers "*..benefits with respect to power preservation..*"[23]. Unlike DMNI however, PORAP can only be used in applications where sources are located within the range of static sinks. DMNI on the other hand could possibly use some of the energy preservation techniques found in PORAP as in essence the MULEs in DMNI are mobile sinks.

In contrast to PORAP, *HEED* introduces a distributed, energy efficient clustering approach for ad hoc networks[36]. HEED preserves energy within the cluster of connected devices by "rotating" on the Cluster Head (CH) probabilistically depending on their residual energy level. A CH can be considered a "leader"

within the cluster and can be utilizes for different tasks such as deciding on what devices get to join the sub-network and what devices does not.

A similar approach could be used for DMNI if CHs were of importance, but since DMNI's MULE data-accumulation routine does not rely on CHs, this would not be very efficient. An alternative solution to DMNI however is to have CHs as part of the ad hoc networks and the MULE only collecting data from the CHs. If this was the case, a solution such as HEED could possibly lower the energy consumption considerably by balancing out the energy consumption in the sub-networks.

3

Architecture

The core concept of Dynamic Mobile Network Infrastructure (DMNI) is to exploit highly mobile Local Area Network (LAN) extensions, either in the form of Unmanned Aerial Vehicles (UAVs) or otherwise, to connect to remote, isolated sub-networks and act as a gateway for Observation Units (OUs) to utilize.

There are three separate layers which together form the basis of DMNI;

- The application layer consists of backend services, such as the Distributed Arctic Observation (DAO) store - This layer is **not** part of the implementation of this thesis and is strictly theoretical
- The network layer consists of mobile, temporary on-demand network extensions - Mobile Ubiquitous LAN Extensions (MULEs) such as UAVs
- The data layer consists of the isolated partitioned ad hoc networks of interconnected OUs

The implementation of DMNI simulates the three layers using a simulator which simulates the entire environment that the devices are part of. Section Section 3.1 goes into further detail how the simulation server is merged into the architecture of DMNI whilst Figure 3.1 shows the overall architecture of

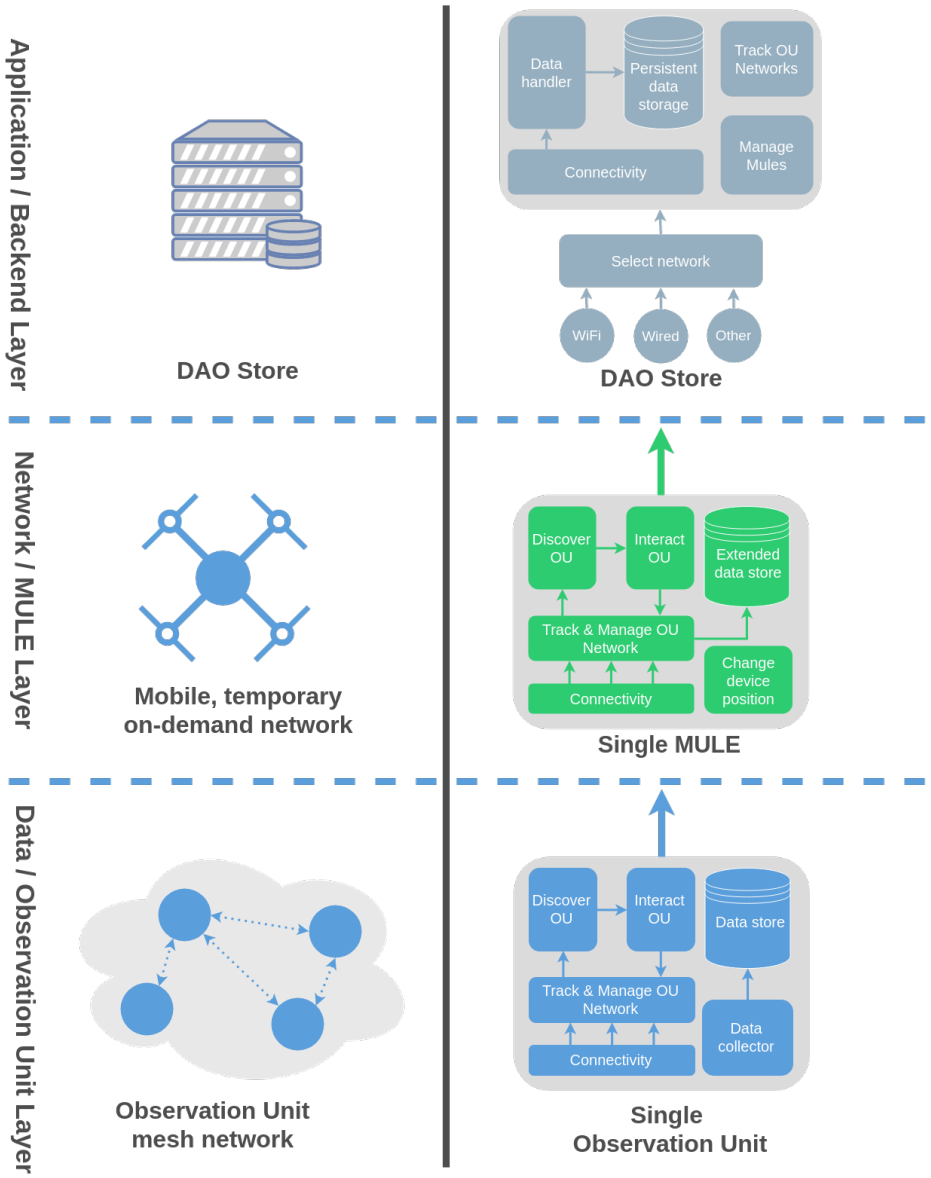


Figure 3.1: DMNI architecture

DMNI.

Definition 2. Partitioned network: Decomposition of a network into several sub-networks

3.1 DMNI Simulator

As seen in Figure 3.1, DMNI is built on a three tiered architecture where there are both mobile and static devices interacting over a network. To achieve an environment which resembles the Arctic tundra with devices placed remotely and secluded from one another in partitioned networks, DMNI also includes a simulator which can be used to evaluate and experiment with the DMNI system.

The simulator contains three main sections:

- Network abstraction
- Device management abstraction
- Environmental simulation abstraction

The simulator creates an artificial environment that the devices in the simulation are not aware of and therefore goes between the layers as seen in Figure 3.2.

3.1.1 Network abstraction

By going between the layers as explained previously, all traffic going from and to devices in the simulation will go through the network abstraction. This means that all traffic may be altered or re-routed as seen fit by the simulator.

3.1.2 Device management abstraction

In order for a device to join the simulation, it is essential that the device registers in the simulator's device management abstraction. Once a device wishes to join the simulation, it will send a message to the simulator's device management abstraction which locally stores information about the device's status in the simulator. This information can be updated at a later point if the device wishes to for example leave the simulation.

3.1.3 Environmental simulation abstraction

In a real-life scenario OUs are placed on the Arctic tundra. DMNI simulates this via the simulator's environmental simulation abstraction. All devices that register in the device management abstraction are assigned a location in a grid-

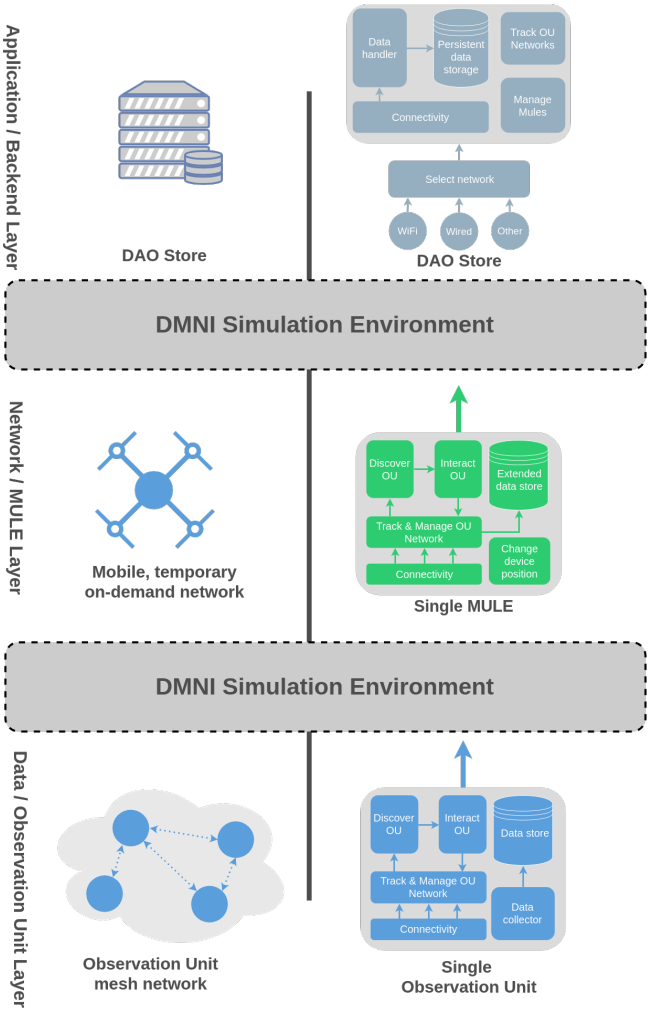


Figure 3.2: DMNI simulator integration

map structure so that all interactions between devices through the network abstraction depends on whether or not they are close enough to each other in the environmental simulation abstraction. It is the responsibility of the environmental simulation abstraction to determine whether or not two devices are close enough to each other in the simulated grid-map - the distance metric may change depending on the simulation configuration.

3.2 Application / Backend Layer

The application layer is responsible for handling all incoming data in a appropriate manner:

- Ensuring that data is not corrupt - all data is tagged with Unique Message Identifiers (UMIs) and data fingerprints and can therefore be checked for corruption at the backend

- Removing duplicate data - the same data may be sent more than once and it is the responsibility of the backend layer to ensure only a single copy of the data is stored

- Making the appropriate log entries before appending the data to persistent storage - data packages contain metadata about the origin of the data and may be analysed at the backend

As previously mentioned, the backend layer has not been implemented in this thesis.

Definition 3. Data fingerprint: Maps arbitrarily large data to a short string, the fingerprint, that uniquely identifies the original data

3.3 Network / MULE Layer

The network layer, also called the MULE layer, is the middle layer in DMNI's architecture. The term *'MULE'* was first coined in 2003 as a way to describe mobile entities that are utilized to collect data from devices that are for some reason unable to get a network connection[30].
The network layer is responsible for reliable transportation of data from the data to the application layer - this is done using mobile data collectors or 'MULEs'. The focus of DMNI is on MULEs and thus MULEs are the only part of the network layer. In a real-life scenario there could potentially be other methods of communication such as static sinks, but for DMNI in particular, MULEs are the only form for of communication between the application and data layer.

3.3.1 Discovery of observation units

Each MULE is equipped with a method of discovery which will discover all nearby available nodes, with resources to respond, in a given radius around the MULE. Figure 3.3 shows the communication distance of a MULE. Once a new device has been discovered, the MULE will store metadata about the device locally on itself for future reference when it needs to interact with the device. The data is kept indefinitely on the MULE, but is updated once newer information about the device is received. This may be from a successful ping

from the device or that the device is unresponsive to network requests.

Definition 4. Metadata: Data or information that provides information about other data

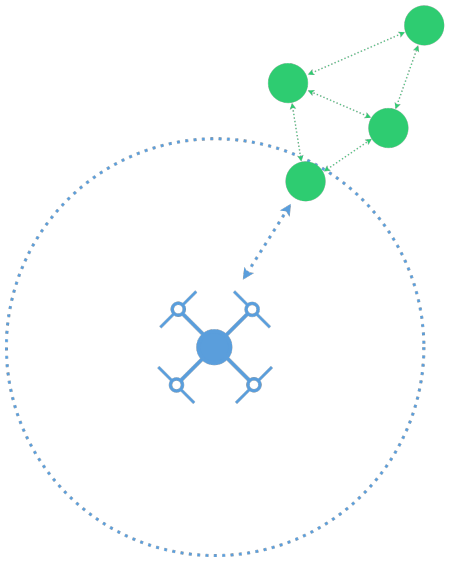


Figure 3.3: Mule discovery

3.3.2 Join network

A MULE may join a sub-network as soon as it has discovered a device which is part of said network. This is done by sharing metadata about itself with the device - the MULE may then be utilized as any other device which is part of the sub-network.
It is important to note that the MULE may change location due to the very nature of a MULE - they are meant to be utilized as mobile network extensions and thus may leave the network after a while. In the event of a MULE leaving a sub-network, this will be handled as any other device leaving the sub-network - it will be discovered that the MULE is unreachable (due to being outside of communication range) and the device will be removed from known, responsive devices within range.

3.3.3 Device interaction

A MULE may interact with OUs for various reasons, such as collection of data, or simply bridging gaps between multiple sub-networks. The use case depends

on the application which utilizes the MULE as what exactly the MULE may be used for may be adjusted over time. However, the primary objective for the MULE is to act as a temporary, on-demand network that OUs may utilize to store and forward data to non-volatile, persistent storage.

3.3.3.1 Temporary network - store & forward

Once a MULE comes in contact with an OU, the MULE will announce its arrival by sharing metadata about itself with the OU - thus joining the ad hoc network which the OU is part of. In addition to this, the MULE will initialize a data store and forward procedure in the ad hoc network which will accumulate the data from all devices in the network and save it to the extended data buffer which the MULE has onboard. The MULE may therefore be considered a mobile data sink. The data may then be forwarded and saved to persistent storage at a later point once the MULE comes close enough to the application layer. In the event that a MULE dies, the data will be lost as DMNI prioritizes power-efficiency and performance over the guarantee of data reaching persistent storage. It follows a '*best-effort*' delivery scheme - see Subsection 7.5.1 for further discussion on this topic.

Definition 5. Ad hoc network: Decentralised, wireless-network consisting of nodes which cooperate in data routing by forwarding data on behalf of other nodes in the network[34]

3.3.4 Network preservation

A MULE will locally store metadata about devices which it has discovered and update said data once a device goes out of range. The updated data will be information about the last known status of the device.

Everytime the MULE attempts to discover new devices within range, it will check the status of the already known devices to update their state. If it discovers that a devices is down or unavailable, the local metadata in the MULE will be updated accordingly.

3.3.5 MULE movement

In order for a MULE to effectively accumulate data from different sub-networks, the MULE has a path which it will follow. The path depends on the predefined route set by the routing algorithm discussed in Section 7.3.

Once a MULE discovers a nearby device and initiates a data accumulation from the device's sub-network, the MULE will halt its location changes for a specified timeout. Once the timeout has run out, the MULE will continue along the path specified. Section 7.9 goes into detail on the issues that arise due to this timeout.

3.4 Data / Observation Unit Layer

The bottom of the three-layer architecture is the data / OU layer. This layer is responsible for collecting data and will create ad hoc networks of cooperative OUs which can be utilized to easier manage data accumulations from the network layer. Each device which is part of the data layer will have methods of collecting data and store it locally until a data-accumulation takes place.

Devices can operate independently, but seek to connect and maintain connections to other nearby DMNI devices to increase the likelihood of successfully passing data to persistent, non-volatile storage.

3.4.1 Discovery abstraction

Each OU is also equipped with a method of discovery. It works exactly in the same way as the MULE discovery and can be seen as in figure 3.3. Once another device has been discovered nearby, metadata about the device, such as address and location, is stored locally on the device for future reference. This data is updated accordingly when for example it is discovered that the device is unresponsive.

3.4.2 Connectivity

At times, an OU may need to send data to another device in the network - be that either another OU or a MULE. This can be because it needs to forward data on behalf of other devices in the network, but also instances where the device can sense that it has for example low battery and will shut off within a short span of time - then data may be forwarded to another device in the system to ensure that data is not lost.

A locally stored dataset about the devices that are within range is preserved and utilized for lookups when data is to be sent to another device. In addition to initializing data transmissions, a device may have to forward data in the sub-network on behalf of other devices. This is handled just as a regular data

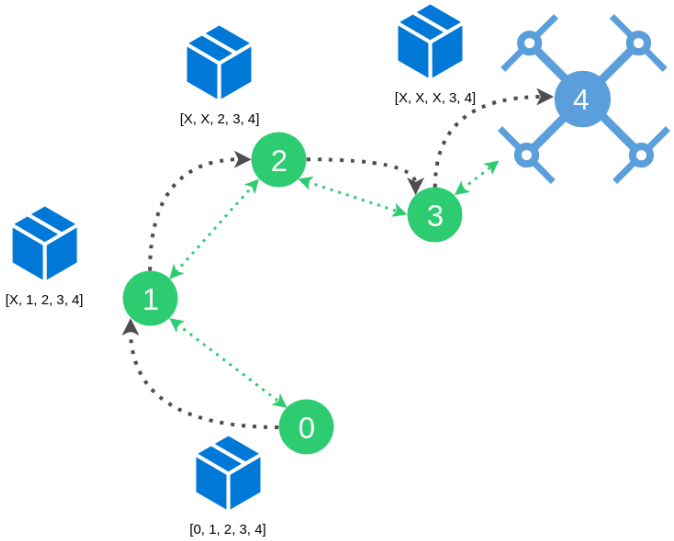


Figure 3.4: Package forwarding

transmission, with the exception that the address for the package is extracted from the '*path*' field within the package before being sent to the next step in the path. Figure 3.4 gives an example of how this may look.

3.4.3 Mesh preservation

In order for a device to maintain the information about the network, it primarily relies on responses from network requests - this is because the only way a device can "observe" the network around it is to listen to the replies from network requests. A device will send out "pings", a small package with no data, to all the devices which has already been discovered close by.

Given that in DMNI there is no defined duty-cycle per device, a device that sends out a ping can assume that the receiving device is unresponsive or dead if no response is received from the original ping. See Section 7.8 for a more in-depth discussion on the topic of duty-cycle.

Definition 6. Duty-cycle: Portion of period in which a system is active

4

Design

Dynamic Mobile Network Infrastructure (DMNI)'s aim is to bridge the gap between isolated, remote ad hoc networks of interconnected Observation Units (OUs) and persistent storage. The goal is to equip Mobile Ubiquitous LAN Extensions (MULEs) with methods of discovering and communicating with these ad hoc networks in an efficient and effective manner to void permanent partitioning of the networks. Mainly, DMNI is split into three components:

1. DMNI simulator server & visualizer
2. DMNI OUs
3. DMNI MULEs

4.1 Simulator & visualizer

The simulator is built with primarily four components as such:

1. HTTP Server & Client
2. Storage
3. Visualizer updater

4. Websockets hub

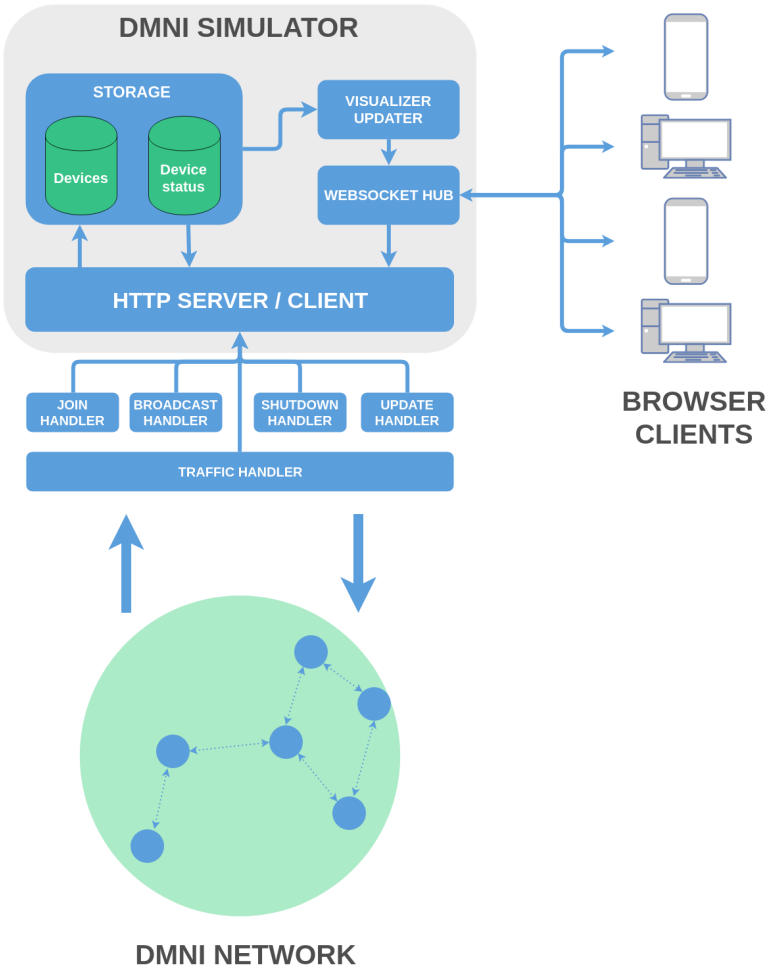


Figure 4.1: Simulation Design

Figure 4.1 gives an illustration of the design of the DMNI simulator.

4.1.1 HTTP Server

The main component of the simulator is the Hypertext Transfer Protocol (HTTP) server and its five HTTP handlers. The five handlers are as follows: Join handler, broadcast handler, leave handler, status handler and traffic handler.

4.1.1.1 Join handler

The join handler is used when a device is first initialized and wants to take part in a simulation - the device will send a request to the simulator's join handler and thus informing the simulator about its presence. This allows the simulator to store data such as network address, location etc about the device in its data-store.

4.1.2 Broadcast handler

As part of the environmental simulation, the broadcast handler takes requests from devices in the simulation and returns a list of devices which are located in given proximity around the device that sent the original request. The proximity varies on the simulation setup.

4.1.2.1 Leave handler

When a device wants to leave the simulation, it will send a request to the simulator's leave handler and this will remove the device from the active devices in the simulation.

4.1.2.2 Status handler

A device may send updates regarding its status to the simulator if need be. The most common kind of update is a location update which MULEs will send to the simulator.

4.1.2.3 Traffic handler

The most crucial handler which the simulator has is the traffic handler. The traffic handler handles all network traffic between devices currently in the simulation and may be used to alter the data which is sent between devices.

A delay relative to the size of network packages and network bandwidth is added to all traffic which goes through the traffic handler, but the traffic handler could potentially add several other features such as chance of data-loss and corruption - see future work in Chapter 9.

4.2 Mules & observation units

4.2.1 Broadcasting

Both OUs and MULEs are equipped with a broadcasting method. The broadcast will send out a "ping" that contains some metadata about the device, such as address and location, and await a reply from any sources. Given that the ping has a limited range, that implicitly means that if a reply is received, that device is within range of communication.
If a reply is received, the reply will contain more metadata about the device which sent the reply and this data is stored locally on the device for future use. Figure 3.3 gives an example of the possible broadcast distance which a MULE may have, although this may vary depending on the simulator setup.

4.2.2 Joining sub-networks

Given that MULEs are able to be utilized as a part of a sub-network, both the MULEs and OUs requires a method of joining a sub-network. The device will join a network by simply discovering other devices which are part of the sub-network.

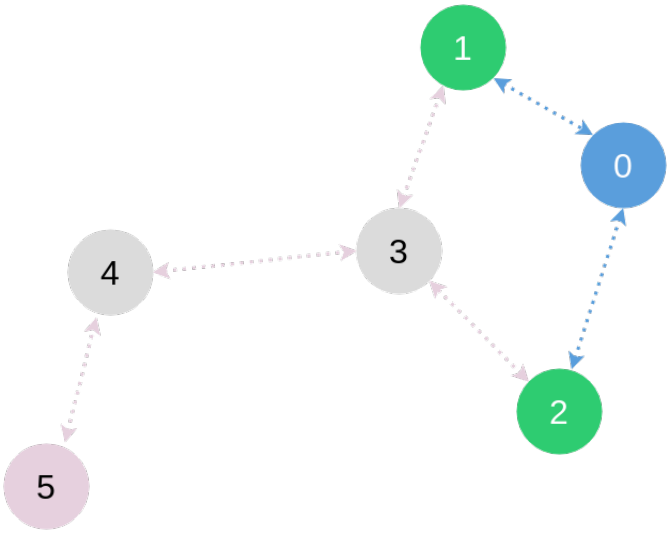


Figure 4.2: Observation unit only aware of direct neighbours

The sub-networks can be partitioned in a way resulting in devices not knowing about everyone. Instead everyone knows simply about their direct neighbours which they are able to contact. Figure 4.2 shows how device(0) only knows about its direct neighbours device(1) & device(2), yet it forms a bigger network with device(3), device(4) & device(5). All devices can be reached in the system

through multi-hop communication, but prior knowledge of the destination is required. In most cases this sort of information is achieved through data-path mapping as seen in subsubsection 4.2.4.5.

4.2.3 Maintenance of network information

All devices, both MULE and OU, are themselves responsible for maintaining a list of known devices and their status. This is done through broadcasting and pinging both new devices, but also known devices within range, in the network. If a device is unresponsive, it will be removed from the list of active devices.

4.2.4 Data transmission

Due to the fact that OUs are following a cycle in which they are collecting data periodically and they have limited storage space, it is essential that they are able to deliver the collected data to a stable, long-term, non-volatile storage to avoid loss of data in the network.
Only when data has been successfully transferred to a MULE can the data be deleted locally from the OU.

Since there is a trade-off between reliability of service and energy-efficiency, data is deleted locally from the device once it has been sent to the MULE - this is discussed further in Section 7.12. Data at the long-time storage may then at a later point be processed and evaluated by researchers.
There are essentially three main functions which are served by both OUs and MULEs:

1. Forwarding - section 4.2.4.1
2. Data flooding - section 4.2.4.2
3. Data accumulation - section 4.2.4.3 - this is split up in two phases
    - Data-path mapping
    - Data accumulation

4.2.4.1 Forwarding

Forwarding in DMNI's ad hoc networks is essential to facilitate for cooperative networks and to ensure that data-communication is effective and efficient.
All devices (OUs and MULEs) which are part of a sub-network, can at times be part of a chain of devices which a package has to go through to reach a destination. If a package is required to be sent to a specific destination, metadata to the package is added about the specific path which the package should traverse. Figure 3.4 gives an example of forwarding a package.

The package should go from device(0) to MULE(4), in which the path itself is found previously through a data-path mapping phase which a MULE may have initialized.
Device(0) will pass the data to device(1), where device(1) will send an acknowledgement back to device(0) that the package was received. Device(1) will then go through the metadata that contains the path and locate itself and the position in the path. If device(1)'s address in the path is the last in the line, it means that device(1) is the destination and no further forwarding is required. If not, the next step in the path is extracted from the path and used as address to send to.These steps are repeated until the destination is reached.

An issue that arises with this approach is the question when devices can delete data locally with the knowledge that the message has been stored safely in persistent storage. This comes down to the fact that networks may not be reliable enough to ensure that an acknowledgement is sent and received as expected. This is discussed further in Subsection 7.12.1.

4.2.4.2 Flooding

Another method of data transmission is flooding - this is the action of forwarding a message to all direct, known neighbours. Figure 4.3 gives an example of this.

In the figure we see that the origin of the message is set to node(0) and it wishes to spread the package to all the devices in the sub-network. Before the package is sent, the package is labeled with a Unique Message Identifier (UMI) to identify the message (See Section 5.6 for implementation details). The package is then sent to node(1), which will send it to all of its neighbours as well. Node(1) will also in this case send it back to node(0), but since node(0) can identify the message using the UMI, it can see that it already received the package and thus it does not need to process it again, nor forward it. This is not shown in the figure to keep it easier to understand. The same process is repeated for all nodes that receive the flooded package until node(7) that

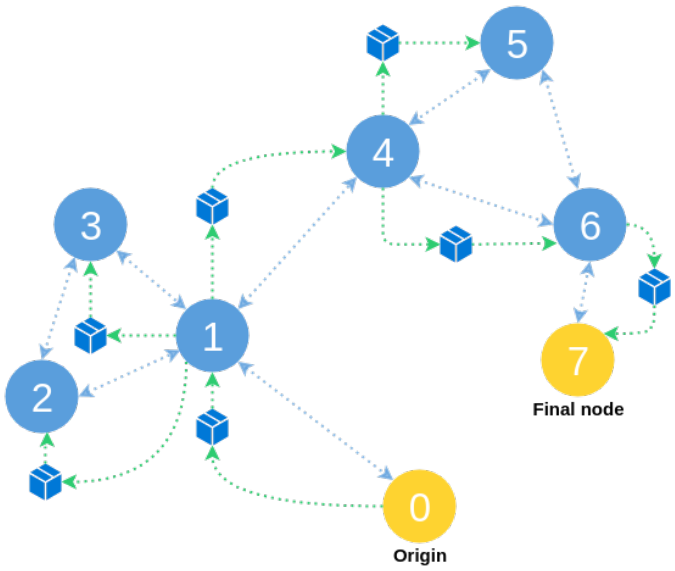


Figure 4.3: Example flooding in DMNI Network where package reaches all nodes in sub-network.

attempts to send it back to node(6) which refuses the package since it was already received.

4.2.4.3 Data accumulation

The basis of DMNI is for MULEs to act as temporary, on-demand networks for sub-networks to dump data to whenever necessary. This is achieved through the data-accumulation method which both MULEs and static data sinks may be equipped with.

The method is split into two steps: data-path mapping(4.2.4.5) and data accumulation(4.2.4.8).

4.2.4.4 Data accumulation initialization

A data accumulation is initialized by a MULE or sink interacting with a device which is part of a sub-network (Or not, see section Section 7.1). If the MULE has the objective to accumulate data from the entire sub-network, the MULE will send a request to the OU telling it to initialize a complete sub-network data accumulation.
The MULE will then await data for a given timeout(See section 7.9), as its uncertain how much time exactly the accumulation may take. Once the timeout

is up, the MULE will continue on its path.

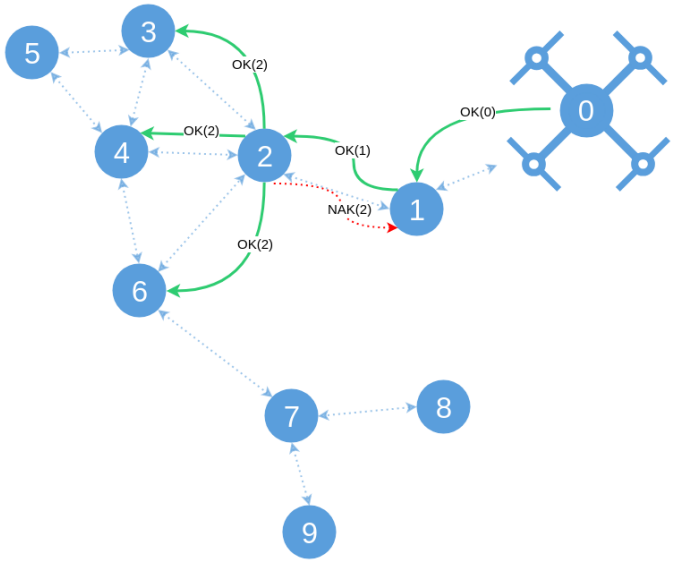


Figure 4.4: First step of DMNI data-path mapping

4.2.4.5 Data-path mapping

The goal of a data-path mapping is to map out paths to the source (MULE or otherwise) which initialized the accumulation in such a manner that every single node in the sub-network is included in at least one of the paths.
Since the sub-networks devices are only aware of their closest neighbours (discussed previously in Subsection 4.2.2), devices are unable to directly map out the shortest path to a destination, especially if the network is of significant size.

Data-path mapping takes advantage of both forwarding and flooding - the data-path mapping package sent from the MULE will be flooded to all the neighbours of the OU and packages will be forwarded back to the MULE when accumulating data.

In Figure 4.4, MULE(0) initializes a data-path mapping by sending a data-path mapping request to OU(1) with metadata about itself and a UMI for future use. As long as OU(1) have not seen the UMI before and it is not on a data-accumulation timeout for this specific MULE(see subsubsection 4.2.4.6), it will continue with the data-path mapping by flooding the instruction to all of its neighbours. This step is repeated for OU(2) which has not seen this unique data-path mapping identifier before, so it floods the message forward to all its neighbours. Here it also floods back to OU(1), but the message is simply

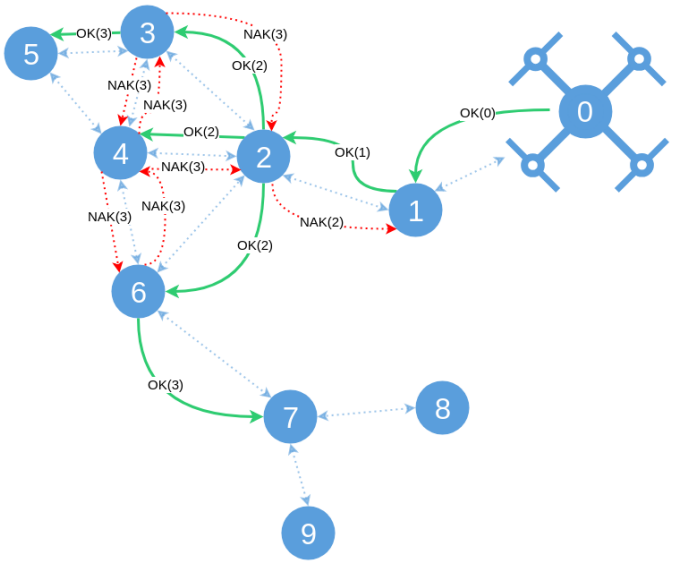


Figure 4.5: Second step of DMNI data-path mapping

dropped as the UMI has been seen before.

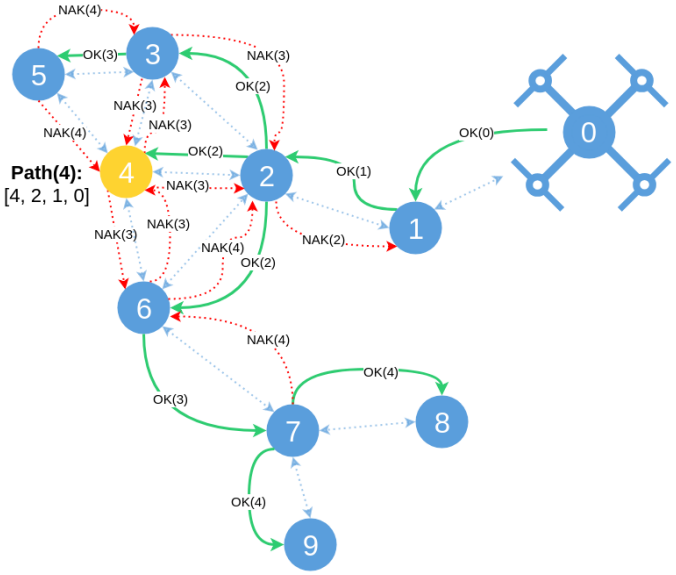


Figure 4.6: Third step of data-path mapping with first accumulation initializer defined

In Figure 4.5, the message is being spread further by OU 3, 4 & 6. It can also be seen that the same message is being sent to sources which has already received it and thus are declined for further flooding as their path is already set.

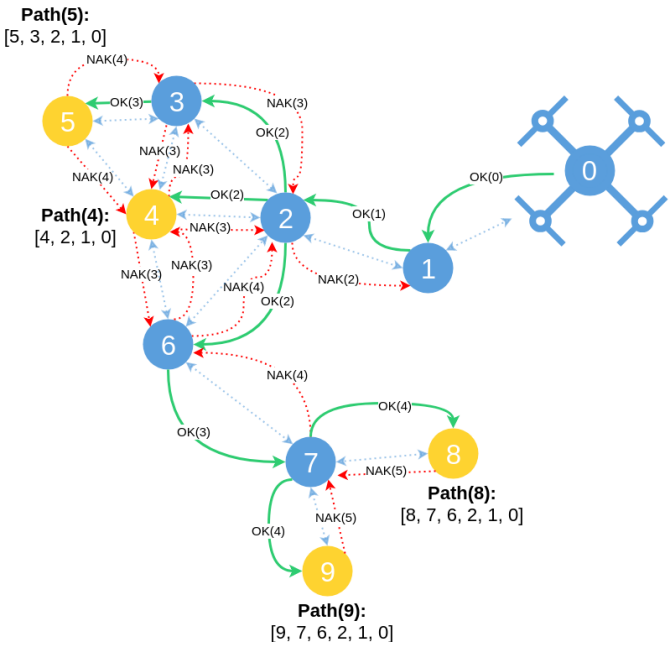


Figure 4.7: Finalizing data-path mapping of sub-network

4.2.4.6 Data-accumulation timeout table

To avoid the issue of MULEs attempting to accumulate from several devices which are part of the same sub-network at the same time, every device in DMNI is equipped with a data-accumulation timeout table.

Once a device receives a data-accumulation from a MULE, it will map the address of that specific MULE to the current time. Whenever a device receives a data-accumulation request, it will check the data-accumulation timeout table and see how long it has been since the last data accumulation from that specific MULE. If there is still an active timeout in place for the specific MULE, the device simply rejects the data-accumulation request.

4.2.4.7 Accumulation initializer

In Figure 4.6, the first "end-destination" has been reached in OU(4) because it has attempted to send the package to all of its neighbours but all requests has been declined as all the neighbours has already received the same message (identified with the unique UMI). OU(4) is thus considered a **accumulator initializer** and will initialize an accumulation on the path $4 \longrightarrow 2 \longrightarrow 1 \longrightarrow 0$. Read more about accumulation in subsubsection 4.2.4.8.

Finally, Figure 4.7 is the very end of the data-path mapping in which OU 5, 9 & 8 is set as accumulation initializers due to the fact that all their flooding has been declined by all neighbours. All three will initialize an accumulation on their set path and thus the entire sub-network is covered in one of the four paths as follows:

- **4** → 2 → 1 → **0**
- **5** → 3 → 2 → 1 → **0**
- **8** → 7 → 6 → 2 → 1 → **0**
- **9** → 7 → 6 → 2 → 1 → **0**

Subsubsection 4.2.4.8 goes into further detail on the accumulation of the data in the system.

Definition 7. Accumulation initializer: Device which initializes a data accumulation due to being at the very edge of a sub-network of devices

4.2.4.8 Data accumulation

Once a device in the network has detected that it is indeed a data-accumulation initializer (as explained in subsubsection 4.2.4.7), the device can initialize a data-accumulation using the specified accumulation path previously found in the data-path mapping. As seen in Figure 4.7, OU 4, 5, 8 & 9 may be defined as accumulation initializers.

In Figure 4.8, all initializers will append their own data to the package before forwarding the data to the next in their path. Note that the initializers does not necessarily start at the same time - in this example OU(8 & 9) will start at a later point as they have gone through more steps than for example OU(4).

In Figure 4.9, OU(7) received the accumulation package first from OU(8) and thus it appends its own data to the package. It does not append its data to the package received from OU(9), but instead forwards it on the path described in the metadata of the package. The same applies for OU(2) which receives a package from OU(4) and OU(3) which received a package from OU(5) - both append their data to the package before forwarding it.

Further, Figure 4.10 shows how the package from OU(2) has been received in OU(1) and from there sent to the MULE. The package from OU(5) has simply been forwarded as both OU(2 & 1) has already appended their own data to

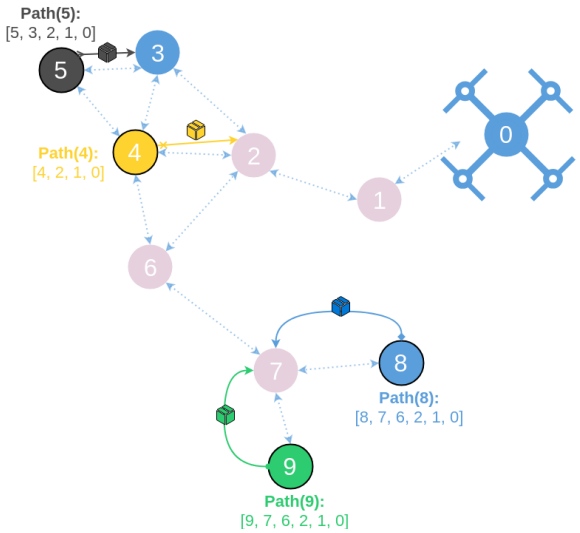


Figure 4.8: Example accumulation of data in sub-network

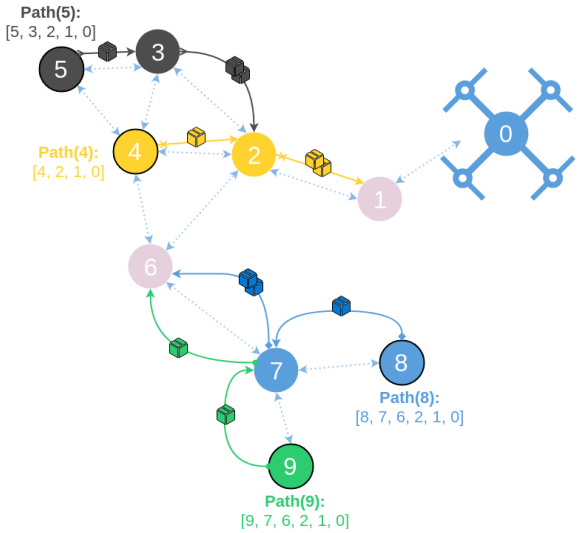


Figure 4.9: Race of which accumulation initializer will send their accumulated data first.

this accumulation. OU(6) has appended its data to the package received from OU(7) which is originally from OU(8) and forwards the package originally from OU(9). These packages are then forwarded all the way to the MULE.

Once the timeout for the MULE ends, it will continue on its path.

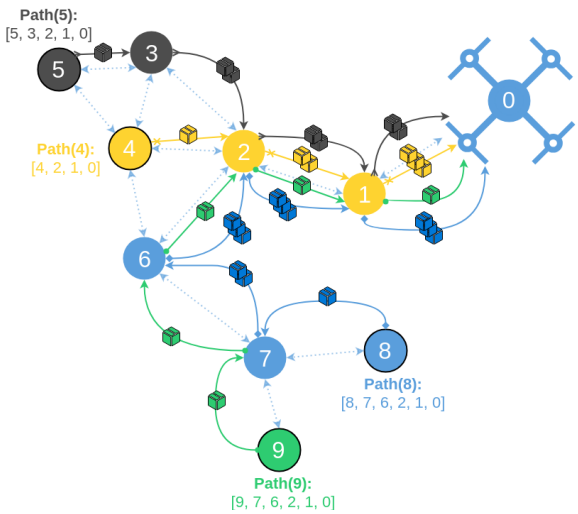


Figure 4.10: Data accumulation from all sub-network reaches mule

5

Implementation

The Dynamic Mobile Network Infrastructure (DMNI) mesh network simulation was implemented in golang version 1.10[1] on a Lenevo ThinkCentre MT-M 10FL-S1S800 with an Intel(R) Core(TM) i5-6400T CPU @ 2.20GHz - 16GB DDR3 RAM. Linux, Ubuntu 17.10 64-bit.

The visualization server served using golang and utilized the *'net'*[2] package for all network related elements such as Hypertext Transfer Protocol (HTTP) server and HTTP client.

DMNI is split up in four separate processes/programs that each can be executed individually:

- Mobile Ubiquitous LAN Extension (MULE) process
- Observation Unit (OU) process
- Simulator process
  - Visualization process - started alongside the simulator process
- DMNI process - used to initialize a complete simulation with simulator

1. Golang: https://golang.org/dl/
2. Net package: https://golang.org/pkg/net/

process, a defined amount of MULE processes and a defined amount of OU processes - this can all be defined in the ‘config.json‘ file as explained under Section 5.1.

All processes utilize a ‘common‘ library implemented for networking and utilities purposes that can be used the same way by several different processes such as logging.

All dependencies (with the exception of golang) will automatically be fetched by utilizing the ‘Makefile‘ provided with the implementation. Appendix A gives more information on this.

5.1 Parameter settings

There are several parameter settings that can be set in DMNI, most of which are located in the ‘config.json‘ file, or the ‘constants.go‘ file. The configuation file is set up using the JavaScript Object Notation (JSON) format and configures the following settings for the simulator:

- X & Y virtual grid size of simulation
- Number of OUs (Will start a separate process per OU)
- Number of MULEs (Will start a separate process per MULE)
- IP & Port of simulation server
- Range of communication - how many grids can a "radio" reach in the virtual grid-map
- LogLevel - what kind of logs are visible to the user
    1. TRACE
    2. DEBUG
    3. INFO
    4. WARN
    5. ERROR

6. FATAL

- OU buffer size (Amount of packages)
- OU package collection interval (millisecond (ms)) - how often does an OU collect data
- MULE accumulation timeout (second (s)) - how long does a MULE wait for accumulated data to be received before continuing on its path

In addition to these config settings, each individual program for the MULE and OU can be started with some parameter settings for setting for example the start position in the virtual grid. These settings are described further in Appendix A.

5.2 DMNI simulator

Due to the early stage at which DMNI is at, it is not practically achievable nor reasonable to deploy a vast network of OUs on the Arctic tundra. Therefore, DMNI implements a simulation of a real-life environment with both a cooperative, mesh-network of OUs that interact with each other and collect mock data, in addition to MULEs that interact with the sub-networks to accumulate data.

The simulator is split into two separate processes as such:

- DMNI visualizer
  - Visualizing the simulation to the users
  - Interacting with browser clients
- DMNI simulation environment
  - Network simulation: handles all network traffic between devices
  - Environmental simulation: creates an artificial environment where devices reside

5.2.1 DMNI visualizer

The DMNI visualizer can be seen in Figure 5.1 - the main purpose of the visualizer is to show the user how the network may appear and visualize connections within the network to the user. Appendix C gives a higher resolution image of this in addition to a video of the visualizer whilst running.

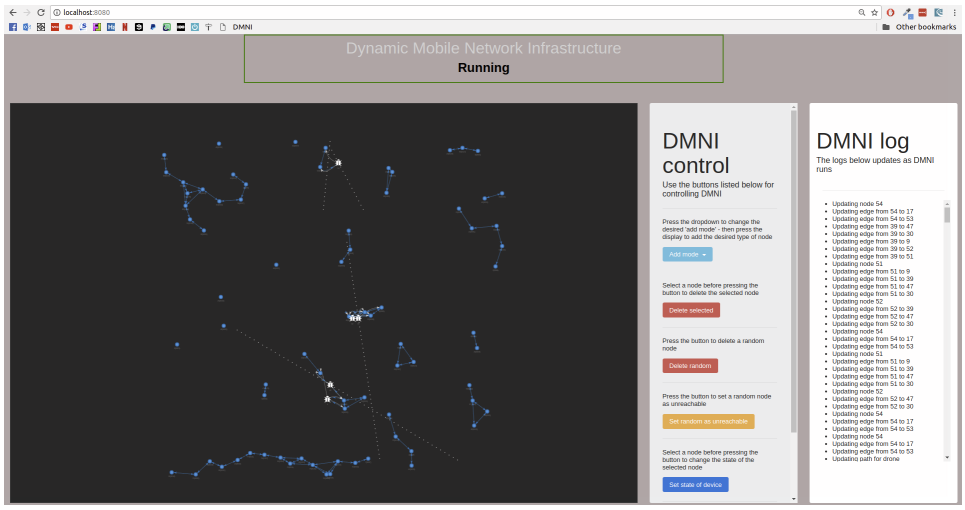


Figure 5.1: DMNI Visualizer

In addition to visualizing the network to the user, the visualizer is able to modify the network in several different fashions such as:

1. Adding OUs or MULEs
2. Removing random or specific OU or MULE
3. Changing state of either a random or specific OU or MULE (Unavailable/Available)

All interaction which the user has with the visualizer is pushed through (Gorilla[6]) web-sockets[3] to the simulator which then reacts to the command sent by the visualizer. In the same way, updates from the visualization server are pushed continuously through web-sockets to all the connected browser clients. Figure 5.2 gives an example how the updates are pushed through a websockets tunnel to and from the client(s) to and from the Gorilla Web-Sockets hub[6].

3. WebSockets: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

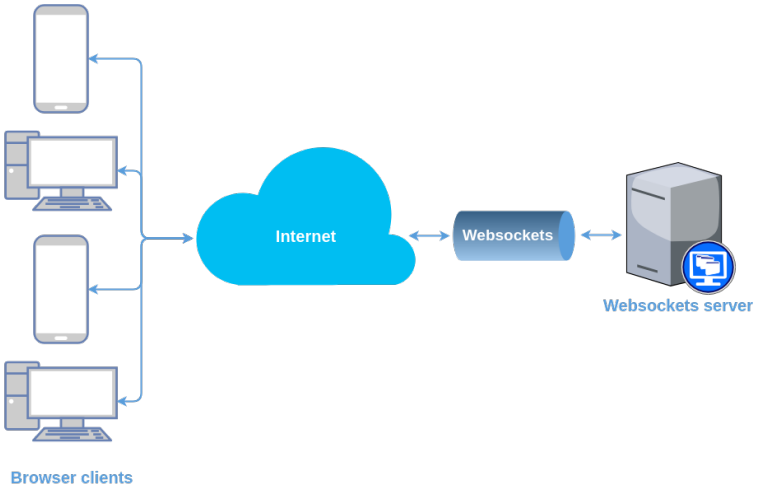


Figure 5.2: DMNI WebSockets implementation

The visualizer utilizes the JavaScript vis library[7] to output a visualization of the network and is updated dynamically through the use of jQuery[4], JavaScript[5] and web-sockets. In addition, the visualizer utilizes the Bootstrap framework[25] for some of the front-end visualization.

In the visualization interface, each OU is represented by a simple coloured circle. Each OU will have their respective relations/connections represented by arrows pointing to other devices in the system - OUs or MULEs. Furthermore, MULEs are represented by a "drone"-like symbol which will relocate following the path represented by white dots. In the same way as OUs, MULE relationships are also represented by arrows stretching to other devices in the network.

All devices in the system will have their status represented by the colour of their icon - if their icon is anything but red, they are available and responsive. If a device has their colour changed to red, this means that the node is either unresponsive, unavailable or simply dead. In addition, the device's "ID" is represented by a white label for easier debugging.

Nodes represented by a green star is what can be considered Cluster Head (CH) - this feature is not complete and was not used for the experiments nor evaluation of DMNI.

4. jQuery: `https://jquery.com/`
5. JavaScript: `https://www.javascript.com/`

5.2.2 Environmental simulation

The environmental simulation may be split into five different handlers as following:

1. Data traffic
2. Join
3. Broadcasting
4. Status
5. Leave

5.2.2.1 Data traffic handler

All traffic which OUs and MULEs create to communicate with one another will always be routed through the simulator's traffic HTTP handler. This way all traffic can be both monitored and altered as pleased by the simulator. All traffic is sent using HTTP to an address such as '*http://localhost:8080/traffic*' and will contain a specific package format (using JSON) as seen below which the simulator can then use to further forward the package to the correct destination. Each data package contains metadata in the '*type*' fields which can be used to identify the form of data which is sent as the '*data*' field is generic and may change depending on the data transmission.

Listing 5.1: Golang data package structure

```
1
2  type DataPackage struct {
3      FromAddress    string       `json:"fromaddress,omitempty"`
4      FromID         int          `json:"fromid,omitempty"`
5      Position       Position     `json:"position,omitempty"`
6      Path           []int        `json:"path,omitempty"`
7      URI            string       `json:"uri,omitempty"`
8      To             string       `json:"to"`
9      Type           string       `json:"type"`
10     Data           interface{}  `json:"data"`
11     Forwarding     bool         `json:"forwarding"`
12     ForwardVia     string       `json:"forwardvia"`
13     MsgIdentifier  string       `json:"msgidentifier"`
14 }
```

Figure 5.3 gives an example of how traffic by both OUs and MULEs are handled. Data going from device(0) to device(1) is forwarded as normal. Due to

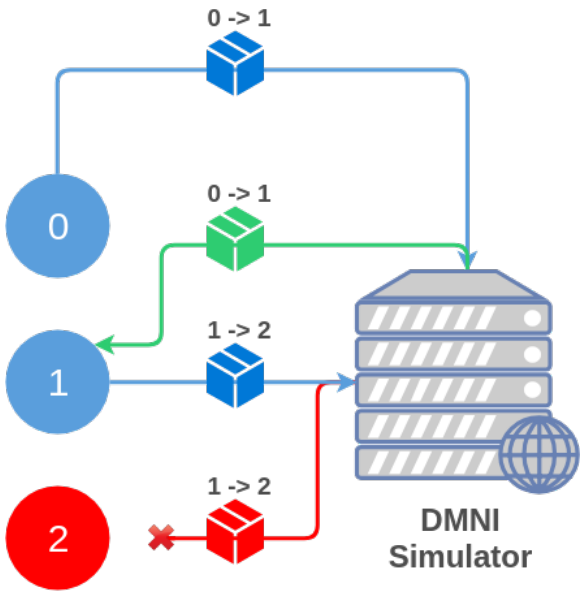


Figure 5.3: DMNI TrafficHandler

device(2) being either down or unavailable, the package from device(1) cannot be forwarded to device(2). The package is dropped by the traffic handler.

In addition to forwarding requests, the traffic handler also adds an artificial delay corresponding to the computed transfer time of a packet with a bandwidth of **3.6817** Mbit/sec.

The number 3.6817 Mbit/sec is used as this is the average bandwidth achieved through testing (See Subsection 6.4.1) and given that bandwidth and latency will always differ depending on different factors, a static number as such was used. Code listing 5.2 shows in detail the delay which is added to each request through the traffic handler.

Listing 5.2: Code of bandwidth simulation

```
1  pkgSize := r.ContentLength
2
3  tc := make(chan bool)
4
5  delay := int64((float64(pkgSize) /
6      float64((common.BandwidthMbitSec))))
7
8  go func(tc chan bool, delay int64) {
9      msDelay := time.Duration(delay)
10     time.Sleep(time.Millisecond * msDelay)
11     tc <- true
12 }(tc, delay)
13 <-tc
```

The delay is calculated by taking the bits (bs) size of the request and dividing by the bandwidth. A go routine is spawned and the main thread will wait until it gets a message in the ‘tc‘ channel. The go-routine sleeps and once its done it sends a message in the ‘tc‘ channel to continue execution in the traffic handler.

5.2.2.2 Join handler

In order for a device to be recognized as either an OU or MULE, the device has to "join" the network. Once a device is started, the process will initialize all its local variables and send a join request to the simulator server once done. The join request will contain metadata about itself, such as network address and port, location and type of device. Once the simulator server receives the request, a new device ID will be picked by the simulator and sent back to the device. In addition, the simulator will send an update to the visualization updater which will update all connected browser clients. When the response with ID is received by the device, the device can commence with the tasks it has to commit.

5.2.2.3 Broadcast handler

In order for a device to effectively "broadcast", meaning find those devices which are close by in a simulated environment, the device will send a broadcast request to the simulator on an address such as ‘*http://localhost:8080/broadcast*‘. Given that broadcast requests may be sent rather often and by multiple devices at the same time, the simulator has to streamline the broadcast routine which it has to go through. In order for the simulator to effectively and timely return the devices which are within range of another device, the simulator keep three separate lists for each device in the simulation as seen in Table 5.1.

Table 5.1: Simulator device broadcast management - broadcast list overview

List name:	**Type of list:**
To examine for range (*EFR*)	List of devices where the range has to be examined
Not within range (*NWR*)	List of devices which are known to not be within range
Known devices within range (*KDIR*)	List of devices which are known to be within range

When the simulation is initialized, all respective device lists in the simulation will have all other devices in their broadcast list and all have empty not within range lists. This way all devices will first check all the distances to all other

devices in the system.

$$\sqrt{(pos1.X - pos2.X)^2 + (pos1.Y - pos2.Y)^2} \tag{5.1}$$

Equation 5.1 shows the calculation which the simulator has to go through in order to see if two devices are within range of one another.

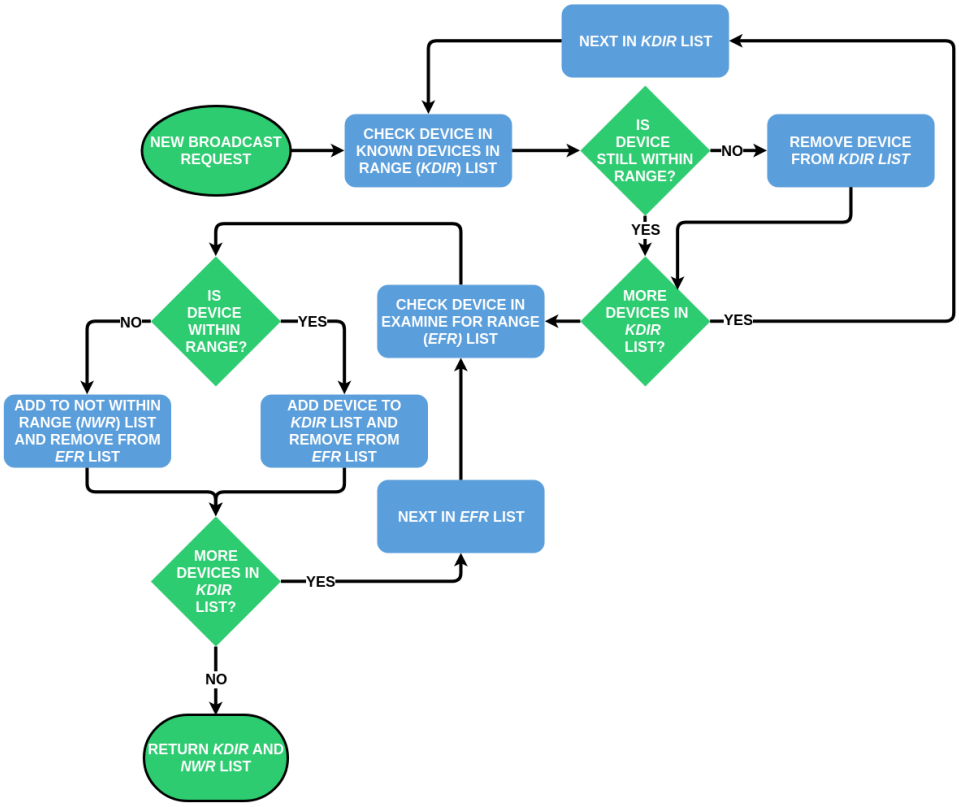


Figure 5.4: Simulation process of locating devices in range

Figure 5.4 shows the procedure which the simulator has to go through when a broadcast request is received. First the simulation will go through all of the already known devices which has been found to be in range - if some are not available nor within range, they are removed from the within range list and added to the list of devices which are no longer within range.

Once finished, the "to examine for range" list is iterated over. All the devices within range are added to the known devices within range list and those not are added to the not within range list. Both the list of devices within range and those not are returned to the device which originally sent the request.

Once a device has been found to either be within range or not, the device will be either appended to the known devices within range or the not within range list - this way the "to examine for range" list will eventually be emptied out. The next time the simulator has to handle a broadcast from the same device, the "to examine for range" list is empty and thus the simulator does not have to spend time examining the range for all devices in the simulation. This increases the effectivity of the simulator.

It is due to the very nature of OUs that it can be done this way - OUs do not move often. However, MULEs do and they may be part of the sub-networks. Therefore, once a device changes its state or relocates, an update is sent to the simulator with this info and the device is appended to the "to examine for range" list of all other devices so the device's distance from all other devices may be recalculated.

5.2.2.4 Status & leave handle

As previously explained, updates of a device's status may be sent to the simulator through the status handle. In the same way, if a device is leaving the simulation, it may send a message to the simulation leave handle so that the shutdown is graceful and data is not sent to a network addresses which are not active.

The simulator will ensure that the update, for example that a device has left the simulation, will be properly updated in both the visualizer and those devices that needs the update. When a device attempts to broadcast, the device that left the simulation will not be returned as within range and thus devices that previously had the device within range will just see that is no longer within range.

5.3 Mule & Observation Unit

Both the MULEs and OUs are built using the 'net.http' go library[6] to create a HTTP server and client. They both serve the following handles:

- Ping handler
- Shutdown handler
- Update handler

6. Http: `https://golang.org/pkg/net/http/`

- Path-mapping handler
- Accumulation handler

In addition to the following handles, the MULE also serves a *'sink'* handler.

5.3.1 Update & shutdown handler

The update and shutdown handler are primarily used for the simulation server to either update or gracefully shut down devices which are part of the simulation environment. If a device receives a request in the shutdown handler, the device will call its shutdown method and thus gracefully shut down and leave the simulation environment.

5.3.2 Ping handler

The MULE and OUs ping handler will receive requests from other devices that are broadcasting. The request will be routed through the simulators broadcast handler (See handler in subsubsection 5.2.2.3) and then forwarded to the receiving device's ping handler. The purpose of the ping handler is for the device to stay updated in regards to which devices are still available and running and new devices that are nearby. This implies that a device may discover other nearby devices via either direct broadcasting or receiving a ping in their ping handler.

Once a ping is received, the device will check if the request type is a 'Data Path Mapping' request - if true, the device will spawn a new go-routine[7] that initializes a data-path mapping routine - see Section 5.4.

Further, the device will check whether or not the sender of the request is already known. If not, the device will add the sender of the request to the list of known devices within range.

5.4 Data-path mapping handler

Given that both MULEs and OUs may take part in data accumulations, the data-path mapping routine applies to both types of devices. Figure 5.5 gives

7. Goroutine: `https://gobyexample.com/goroutines`

an insight in how the procedure of data-path mapping functions.

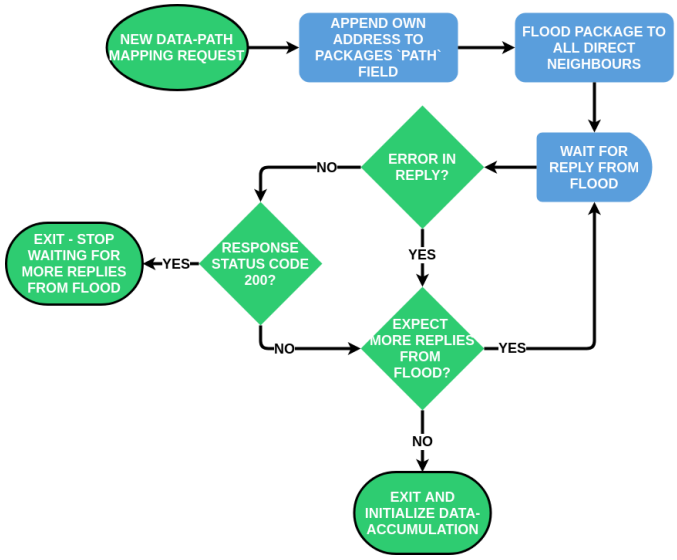


Figure 5.5: Process of data-path mapping

Once a new package is received in the data-path mapping handler, the very first thing the device does is to append its own path to the package's '*path*' field. This way the path which the package has to take is tracked. Further, the package is flooded to all the known direct neighbours of the device. To effectively wait for replies, the implementation utilizes golang's 'channels'[8] and the process will then await the exact number of flooded messages which are sent out.

An issue arises here with the default implementation of golang's 'http.client', as the implementation does not specify request timeouts and thus allowing services to hijack the goroutines[33] - this is resolved by utilizing a custom HTTP client which specifically sets the timeout of requests to 25 seconds. In addition to getting around the issue of the possibility of hijacked goroutines, the timeout also resolves the issue of possible deadlocks in which the routine will wait for-ever for a reply which never comes.

Once a reply is received, error checking is done and if there are no errors the status code is checked. If the statuscode is 200, this means that one of the neighbours that received the data-path mapping request has yet to be included in one of the data-paths. In essence, this means that either that neighbour or another device further out in the network will initialize a data accumulation. If however the response does not contain the statuscode 200, this means that

8. Golang channels: `https://tour.golang.org/concurrency/2`

the other neighbour that received the message have indeed been included in one of the paths already and thus will not forward the data-path mapping. If no more replies are expected then the device will initialize a data accumulation as seen in Section 5.5.

5.5 Data Accumulation

When a request is received in the accumulation handler, the very first thing that the device does is find the index which its own address resides at in the package's '*path*' field. The very first address in the path is the original sender, in other words the MULE. If the device's address is not part of the package's path, an error is raised and the accumulation is exited.

If there are no errors, the device will check if it has already accumulated its data as a device may be part of several data-path mappings. If a device is part of multiple data-path mapping paths, there may be race for which path first accumulates data from the device. This is checked using a hash-map structure in which the original Unique Message Identifier (UMI) is set as key and the the value is a structure containing info whether data has been collected and a timestamp for when.

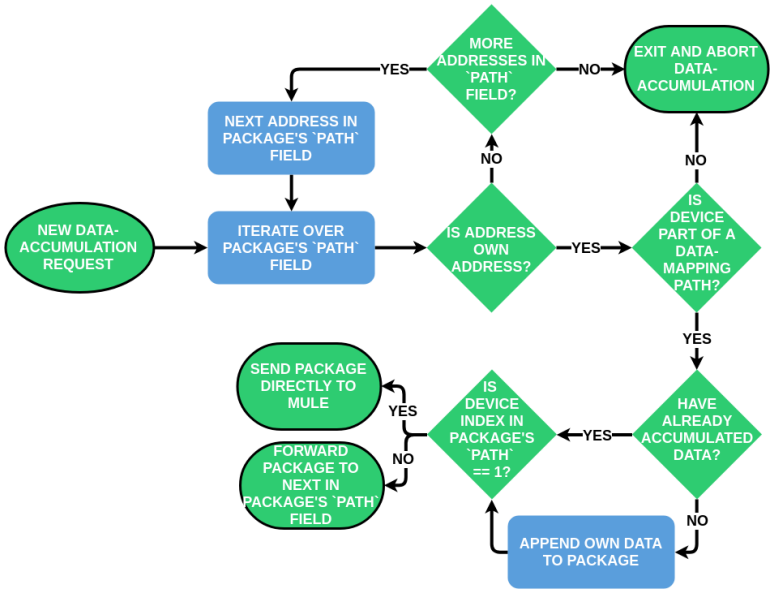


Figure 5.6: Process of data accumulation

If the device has yet to accumulate data, the device will append its collected data to the package. Depending on the index which was earlier extracted from

the package's '*path*' field, the device will either forward the package to the next address in the path or send it directly to the MULE.

5.5.1 Sink handler (MULE only)

The sink handler is unique for MULEs as they would have an extended data-buffer to store data before forwarding it to persistent storage. In the current implementation of DMNI, the extended buffer is only limited by the amount of memory on the device itself.

Once a MULE receives a request in its sink handler, the data will be structured in a way so that the origin of the data may be tracked by analysing the data - this is a step that would be taken at the backend and thus DMNI does not implement this.

As earlier shown in Listing 5.1, each data package contains a generic '*data*' field in which all type of data may be stored. To track the origin of data, the '*data*' field in each package received in the sink handler is simply an array of more data packages - this way each origin can be tracked and within each of the packages' '*data*' field again is the actual collected data. Figure 5.7 illustrates the structure of the accumulated data package.

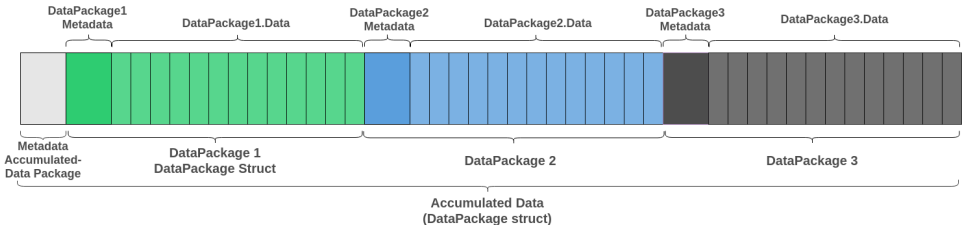


Figure 5.7: Data-accumulation package

5.5.2 Collection of data

The OUs collect data - in the current implementation of DMNI this data is generated by the OUs themselves.

Listing 5.3: Golang data collection

```
1
2 func CollectData() Data {
3     data := Data{
4         Seqnum: time.Now().Unix(),
5         Data:   GenerateRandomBytes(Random(10, 150)),
6     }
7     data.Fingerprint = FingerPrint(data.Data)
```

```
8       data.Size = len(data.Data)
9       return data
10  }
11
12  func GenerateRandomBytes(num int) []byte {
13      token := make([]byte, num)
14      rand.Read(token)
15      return token
16  }
```

Listing 5.3 shows how the data is generated. A 'Data' data structure is generated with a random number of bytes. The fingerprint and size of the data is also collected and returned.

5.6 Unique message identifiers

All messages are labeled with a UMI which can later be used to identify whether or not a message has previously been received. The UMI is created as seen in Listing 5.4.

Listing 5.4: Golang code showing how a Unique Message Identifier is created

```
1
2   func FingerPrint(data []byte) string {
3       hasher := sha1.New()
4       hasher.Write(data)
5       sha := base64.URLEncoding.EncodeToString(hasher.Sum(nil))
6       return sha
7   }
8
9   func GetDataIdentifier(address string) string {
10      t := time.Now().UnixNano()
11      b := make([]byte, 8)
12      binary.LittleEndian.PutUint64(b, uint64(t))
13      c := append(b[:], []byte(address)...)
14      return FingerPrint(c)
15  }
```

6 Evaluation

Given that the Dynamic Mobile Network Infrastructure (DMNI) simulator simulates the Arctic environment, it is possible to answer bigger questions which are related to the system as a whole by utilizing explicitly the DMNI simulator to run simulations, but questions related specifically to Mobile Ubiquitous LAN Extensions (MULEs), such as power consumption, cannot be answered this way.

Therefore, the experiments conducted are split into two sub-groups in which one contains experiments conducted utilizing the DMNI simulator explicitly and the second section contains experiments conducted utilizing a real device in the form of a Raspberry Pi 3 Model B (RPi).

All experiments were designed to determine how well the system would perform on the Arctic tundra and reveal both devices capabilities and down-sides.

6.1 Experimental design

The configuration for each experiment differs and is therefore mentioned specifically in each section. Section 6.2 goes into further detail on the simulator experiment details whilst Section 6.3 explains the details around the RPi experiments.

6.2 Simulator experiments

The experiments conducted explicitly using the DMNI simulator was conducted on a Lenevo ThinkCentre MT-M 10FL-S1S800 with an Intel(R) Core(TM) i5-6400T CPU @ 2.20GHz - 16GB DDR3 RAM. Separate processes were spawned for:

1. Environmental simulation
2. Simulation visualizer
3. MULEs
4. Observation Units (OUs)

The number of OUs and MULEs varies from experiment to experiment and each experiment ran for specific duration and a number of iteration which then was used to calculate an average. Each specific number of MULEs, OUs, duration and interval count is specified in each experiment.

6.2.1 Package drop-rate vs. MULE count

This experiment was designed to evaluate the package drop rate which can be caused by a lack of MULEs to accumulate data on the Arctic tundra. An OU will after a time drop packages due to their data buffer being full. The drop rate can be defined as the relation between the amount of packages which has been successfully accumulated vs. the amount of packages which had to be dropped by the OU due to their data buffer being full. The setup of the experiment was as follows:

- 3000x3000 virtual grid (size of simulated area)
- OUs has a data-buffer with room for 1000 data packages
- OUs are configured with a rounded buffer - once the data buffer is full, the oldest data will be deleted
- Data collection in OUs happen every 300ms
- Packages collected by OU vary in size from 10 to 150 bytes (Bs) (randomized)
- Communication range is set to 150 virtual grids

- No (artificial) delay in retrieving data
- MULE timeout of 10 seconds
- The amount of OUs varies (Specified)
- The amount of MULEs varies (Specified)
- Location of OUs and MULEs in grid randomized
- All experiments ran for 10 minutes and repeated 10 times (10 iterations) - the average collected result is shown

The large virtual grid size was purposely set in order to trigger package drops in the simulation. As the grid size increases, the distance between OUs increase. As the distance increases, the chance of OUs forming sub-networks decreases. Since MULE can collect data from all members of sub-networks by only being in contact with a single device part of said system, it is beneficial that sub-networks do not form in order to trigger package drops.

If the virtual grid was set to for example 1500x1500 in size, the chances of data packages being dropped would be low and thus not very interesting.

By looking at the settings above, the time before an OU starts to drop packages(t) can be calculated by the 'OU data collection time(300ms) * OU data buffer size(1000)', given in Equation 6.1.

$$t = 300ms \times 1000 = 300000ms \tag{6.1}$$

This equals 300 seconds or 5 minutes, meaning that in order for a OU to drop a package it has to not be part of any data accumulations for at least 5 minutes out of the 10 minutes the experiment ran.

The maximum packages a OU can drop during one iteration(n) of one experiment is thus 'Time of one experiment iteration (10min or 600 000ms) - time before OU starts dropping packages (5min or 300 000ms) / time to collect one data package (300ms)', this Equation 6.2.

$$n = (600000 - 300000) \div 300 = 1000 \tag{6.2}$$

The experiment was first ran with 50 OUs, then 75 OUs and lastly with 100 OUs. All experiments was conducted with a different % of MULEs to OUs count as

seen in Figure 6.1. The amount of experiments(ex) would therefore be equal to $ex = 3 \times 7 = 21$.

Each experiment was repeated 10 times, thus 210 total repetitions of 10 minutes was conducted. The total time to run the experiments was therefore $210 \times 10 = 2100$ minutes or 35 hours.

Note that the experiment does not take into account that a OU's data buffer will be filled up with packages from other OUs.

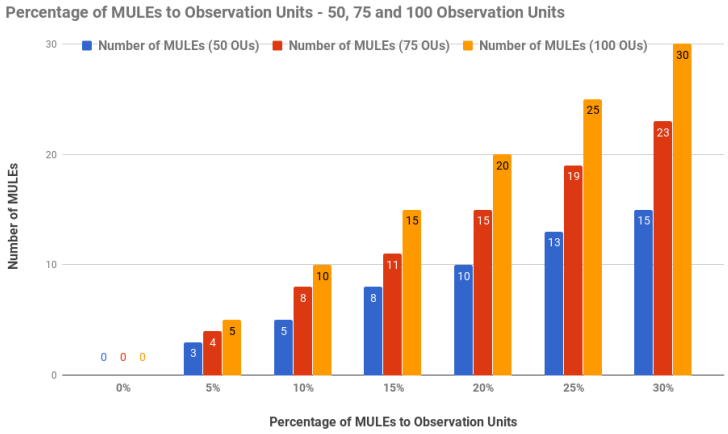


Figure 6.1: Percentage of MULEs at 50, 75 and 100 Observation Units - 3 groups of 7 experiments

6.2.2 Mule timeout

This experiment was aimed at examining whether or not the MULE timeout is of big significance when accumulating data from the ad hoc network. As previously explained in subsubsection 5.2.2.1, the simulator applies a calculated delay relative to the network request size equal to what would be achieved with a bandwidth of 3.6817 Megabit (Mbit)/sec. This can be seen in code listing 5.2.

The setup of the experiment is as follows:

- 1500x1500 virtual grid (size of simulated area)
- Data collection in OUs happen every 300ms
- Packages collected by OUs vary in size from 10 to 150 Bs(randomized)

- Communication range is set to 150 virtual grids
- Simulator bandwidth of 3.6817 Mbit/sec
- MULE timeout varies from 0 - 15 seconds
- OU count of 50
- MULE count set to 15% of OU count, thus 8 MULEs
- Location of OUs and MULEs in grid randomized
- All experiments ran for 10 minutes and repeated 10 times (10 iterations) - the average collected result is shown

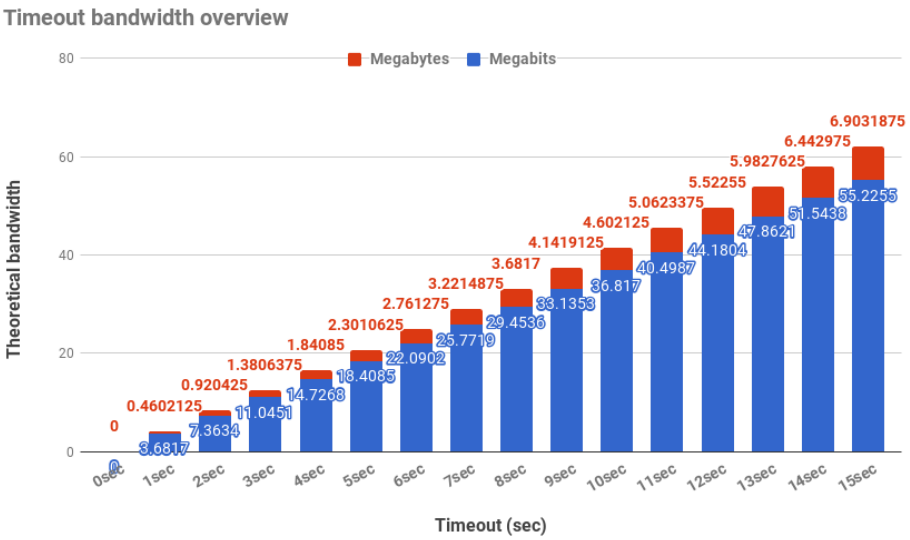


Figure 6.2: Theoretical bandwidth at increasing MULE timeouts

Figure 6.2 gives an overview of the theoretically possible amount of data which can be transmitted in relation to the MULE timeout.

This means that for the maximum amount of data a single OU can collect in a 10 minute span for a single iteration is as follows:

$$b = 150bytes \times (600000ms \div 300) = 300000bytes \tag{6.3}$$

Following, the total amount of data that can be collected in the ad hoc network

is:

$$b = 50 \times 300000 = 15000000bytes = 15MB \tag{6.4}$$

This equals a maximum amount of data collected by all OUs in the ad hoc network to be 15 Megabyte (MB).

Each experiment was repeated 10 times, thus $16 \times 10 = 160$ total repetitions of 10 minutes was conducted. The total time to run the experiments was therefore $160 \times 10 = 1600$ minutes or 26.6 hours.

6.3 Raspberry Pi experiments

The RPi experiments were executed on a single RPi[1] with the following hardware:

- Quad Core 1.2GHz BMC2837 64bit CPU
- 1GB RAM

Alongside the RPi itself ran a simulation server with the following setup experimental setup:

- 1500x1500 virtual grid (size of simulated area)
- 50 OUs in simulation (Randomized position)
- 8 MULEs (Randomized position and pathing/routing)
- 150m communication range
- OUs collect data every 300millisecond (ms)
- MULE timeout set to 10 seconds
- Location of OUs and MULEs in grid randomized
- The experiments was measured over 10 minutes and repeated 10 times (10 iterations)

The RPi ran a single MULE process which communicated with the other

1. Raspberry Pi 3 Model B: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`

devices (processes) spawned on another computer by the simulator via local network.

The network measurements were taken by using the go library 'gopsutil'[31] and it parses out the specific network utilization used by the single running MULE process. The CPU and memory measurements were taken by parsing the '*ps*' command in linux to get the CPU and memory utilization per process and not the device as a whole. All measurements were taken over a time of 10 minutes and was repeated 10 times (10 iterations).

Measurements for the energy consumption was not conducted due to time restrictions and issues whilst setting up the multimeter for the experiment.

6.3.1 Raspberry Pi WiFi Bandwidth

The main purpose of this experiment was to map out what one can expect from a RPi in terms of bandwidth between two devices. Both devices were connected to a router over a 2.4GHz WiFi connection. Two RPis were set up, connected via WiFi to the same network and router, and via the tool '*iperf*'[11], an average bandwidth was measured over an interval of 100 measurements (100 repetitions). The measurements were taken over a distance of approximately 10m and through a wall to the router.

6.3.2 CPU, memory and network utilization

DMNI is to be executed on small, low-power devices that are to function on the Arctic tundra, it is crucial that the system is lightweight in terms of computational power, memory footprint and network utilization. All CPU, memory and network specific measurements were taken whilst running the MULE specific process alongside the simulator and the setup described previously.

6.4 Results

This section presents the results of the experiments conducted - both for the RPi specific experiments and the simulator experiments.

6.4.1 Raspberry Pi bandwidths

Figure 6.3 shows the measured bandwidths between two RPis over a distance of approximately 10m with some minor physical blockage between(two wooden walls). As expected, the bandwidth measurements fluctuates considerably as one would expect between two WiFi points. The standard deviation in the measurements is calculated to be **1.1616** Mbit/sec which is quite considerable.

Unfortunately, it is rather hard to predict the exact bandwidths[24] one can expect to receive when sending over WiFi because a lot of factors has to be taken into account such as:

- Physical distance between sources
- WiFi card and antenna on both sources
- Signal interruption / interference
  - Especially relevant as the measurements were conducted at the Institute of Computer Science (IFI) at University of Tromsø (UiT) where many devices are connected wirelessly. This can cause interference in signals
- Physical blockage

From the measurements, an average of **3.6817** Mbit/sec from 100 measurements was achieved. This value was used as a baseline for the simulated bandwidth delay in the traffic handler for the remainder of the experiments where this was relevant.

6.4.2 Package drop-rate vs. number of MULEs

The purpose of this experiment was to measure the relation between package drop-rate and the number of MULEs in the simulation.

A package is defined as the data structure created by OUs containing data collected by sensors and metadata about the data collected.

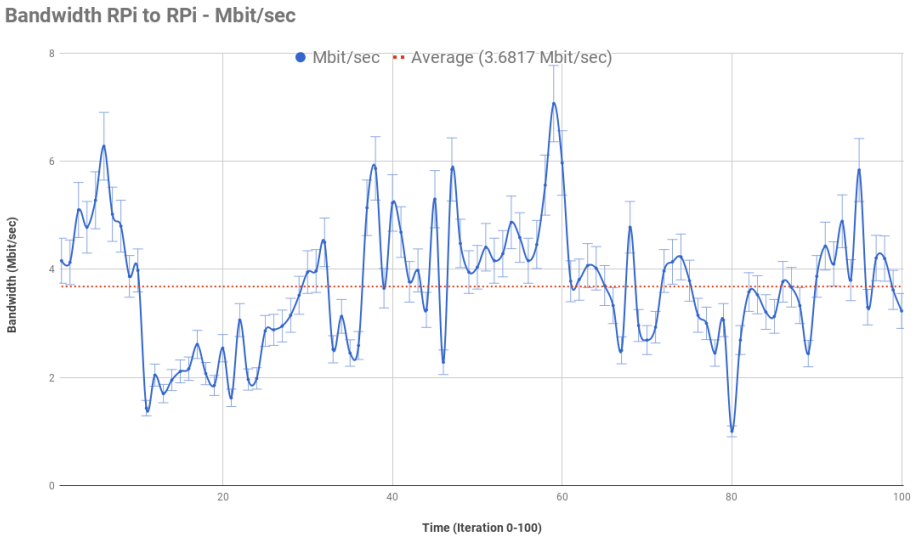


Figure 6.3: WiFi Bandwidth measurements - Raspberry Pi to Raspberry Pi

6.4.2.1 50 Observation Units

The results provided in Figure 6.4 show, in the blue dotted line, the percentage of packages dropped versus the amount of total packages collected in the simulation. The black line with circles represents the total amount of packages that were accumulated successfully to a MULE. Lastly the starred orange line represents the packages that were lost in transmissions, meaning those packages that were sent from OUs but never reached a MULE.

As seen, when there are no MULEs available(0%), all packages are eventually dropped. In the very first MULE percentage, we see that the MULE percentage compared to OU count is set to 5% of 50 OUs, thus 3 MULEs. The package drop-rate is 36.44% - since a OU can maximum drop 1000 packages in one iteration (as previously explained Equation 6.2) in of the experiment and there are 50 OUs, the total amount of packages lost by all OUs (pl) would equal to:

$$pl = (1000 \times 50) \times \frac{36.44}{100} = 18220 \tag{6.5}$$

Table 6.1 shows the number of dropped packages for 50 OUs with the respective MULE count.

Not surprisingly, the number of packages dropped goes down with the amount of MULEs that are in the simulation. What is rather interesting is to see that

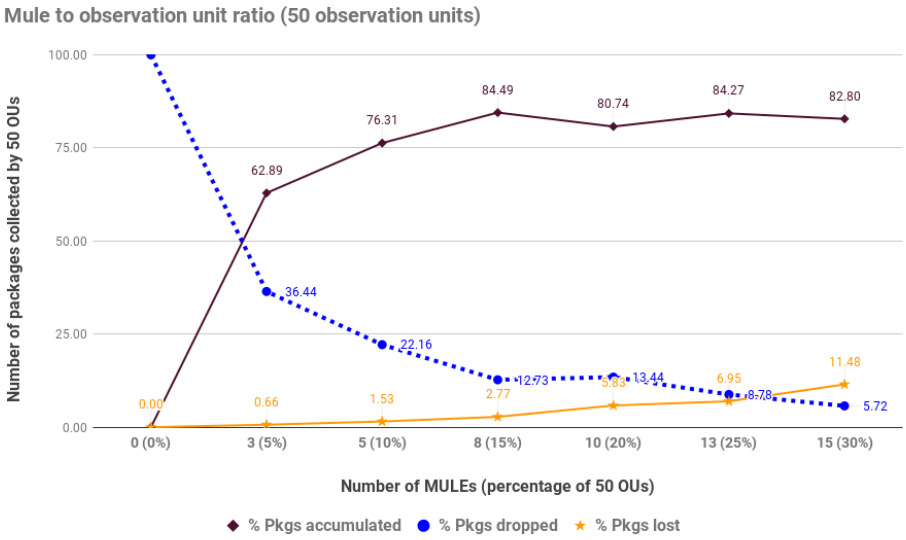


Figure 6.4: Mule to observation unit ratio - 50 observation units

Table 6.1: Package-drops - 50 observation units

Percentage mules	Package drop %	Dropped packages
5	36.44	18 220
10	22.15	11 075
15	12.73	6 365
20	13.43	6 715
25	8.77	4 385
30	5.72	2 860

the amount of accumulated packages stabilizes to an extend at around 82-84% accumulated packages compared to total packages collected by the 50 OUs. In addition to this, when utilizing 13 MULEs (20% of 50 OUs), the amount of packages that are lost exceeds the amount of packages that are dropped by the OUs. From this we can see that the network related issues (due to lost packages = packages sent to MULE - packages accumulated by MULE) exceeds the amount of packages dropped by OUs. It is expected that the reason behind this is that as there are more MULEs in the simulation, especially since they have a static set timeout of 10 second (s), the amount of network issues will steadily increase due to for example a MULE simply not waiting long enough for all packages to accumulate from a sub-network.

In addition, a thing to note is that the amount of packages dropped actually went up when using 20% MULEs compared to 15% of MULEs. This is most likely an anomaly as we can see from the other experiments with 75 and 100

OUs that this is not case. One can suspect that it might be that during one of the experiment iterations, the simulator process crashed and thus did not collect any data for that iteration, meaning the average calculation would be incorrect compared to the other iterations.

6.4.2.2 75 Observation Units

Figure 6.5 gives rather similar results compared to what was achieved in subsubsection 6.4.2.1. It does however show a generally lower percentage of packages dropped and this comes down to the increased amount of OUs in the system.

By increasing the density of the OUs, the chance of the OUs being close to each other is higher, thus the chance of collecting data from them is higher as a sub-network of OUs covers a bigger area than a single OU. As the area they cover gets bigger, the chance of a MULE discovering the sub-network increases.

Figure 7.1 gives an illustration of this - the blue network consists of a single OU whilst the green is a sub-network of interconnected OUs. A MULE needs to come in contact with either the blue or the green and thus one can see that it is easier to come in contact with the green than the blue, even though the green does overlap.

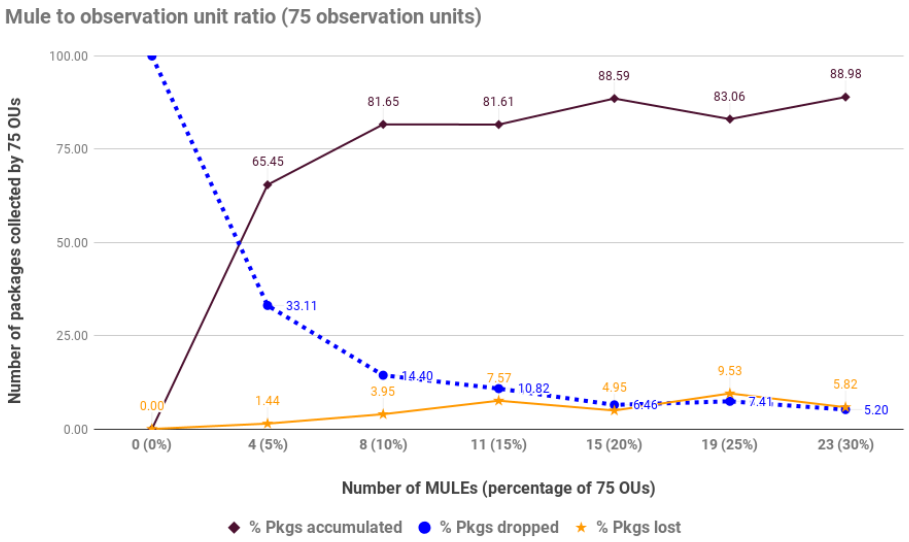


Figure 6.5: Mule to observation unit ratio - 75 observation units

The package drop-rate when using 5% MULEs is 33.11% - since a OU can maximum drop 1000 packages(As previously explained Equation 6.2) in one iteration and there are 75 OUs, the amount of packages lost(pl) for all OUs would equal to:

$$pl = (1000 \times 75) \times \frac{33.11}{100} = 24832 \tag{6.6}$$

Table 6.2 shows the number of packages dropped in a system with 75 OUs.

Table 6.2: Package-drops - 75 observation units

Percentage mules	Package drop %	Dropped packages
5	33.11	24 832
10	14.39	10 792
15	10.81	8 107
20	6.45	4 837
25	7.40	5 550
30	5.19	3 892

An interesting thing to see in Table 6.2 compared to Table 6.1 is that the total amount of packages dropped in the two experiments are somewhat comparable. This is even though the amount of OUs has been increased by 50% from 50 OUs to 75 OUs.

The results here gives an indication that the amount of OUs in the same sized virtual grid has an impact on the amount of packages dropped - this is further discussed in subsubsection 6.4.2.3.

6.4.2.3 100 Observation Units

In the experiment with 100 OUs the trend is similar to what we have seen with 50 and 75 OUs. We can see that the package-loss ratio decreases as the amount of OUs increases.

The package loss ratio keeps declining compared to the total amount of packages accumulated with 100 OUs.

Figure 6.6 shows how the trend of declining package loss continues with a starting package-loss(pl) of 21.92% for 5% of MULEs.

$$pl = (1000 \times 100) \times \frac{21.92}{100} = 21920 \tag{6.7}$$

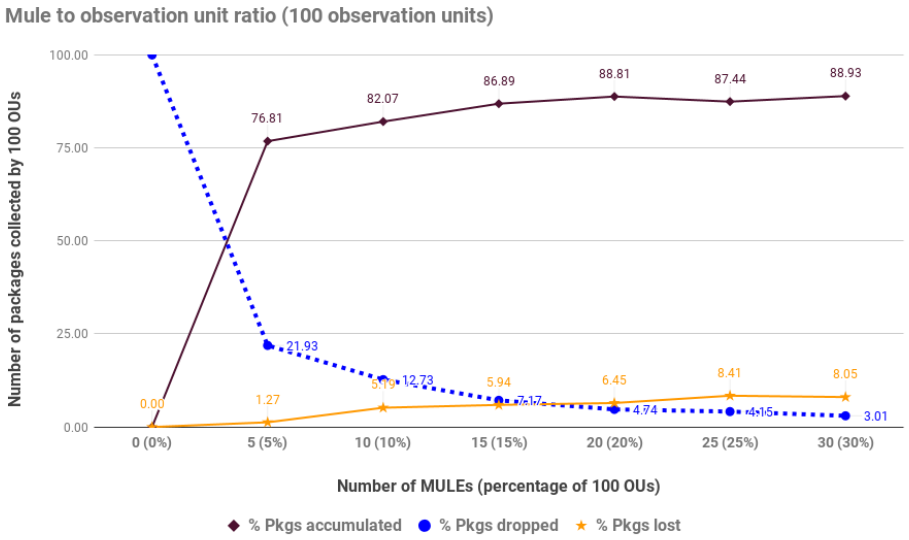


Figure 6.6: Mule to observation unit ratio - 100 observation units

Table 6.3 gives an overview of the packages dropped with 100 OUs.

Table 6.3: Package-drops - 100 observation units

Percentage mules	Package drop %	Dropped packages
5	21.92	21 920
10	12.73	12 730
15	7.17	7 170
20	4.73	4 730
25	4.15	4 150
30	3.01	3 010

The trend is that with an increasing amount in density of OUs, the amount of package-loss within the ad hoc network is decreased.

Figure 6.7 shows a comparison between the three experiments where the blue bar represents the experiment with 50 OUs, the red represents the experiment with 75 OUs and the orange represents the experiment with 100 OUs. In addition to plotting the amount of package drops, the figure shows the trend line which each experiment has.

From looking at the trend lines, we can see that the experiments with the highest amount of OUs has a steeper decline compared to the experiments with a lower number of OUs.

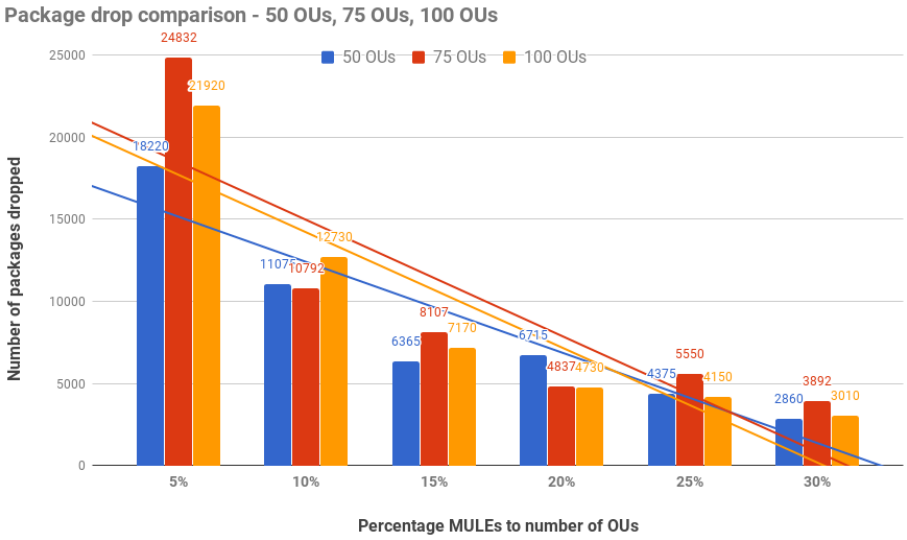


Figure 6.7: Comparison of packages dropped - 50, 75 and 100 Observation Units

We can from this deduce that the number of OUs has a direct impact on the amount of packages that are dropped. This is because as the number of OUs increases, the density of OUs also increases. With an increases density of OUs, the chance of OUs forming sub-network is greater. As shown in figure 7.1, sub-networks enabled OUs to "reach" further and thus the chance of a MULE accumulating data from the sub-network increases.

From the experiment done, we can see that the package-loss rate can be reduced to below 5%, even as low as 3.01%, when using a MULE to OU ratio of 30%.

6.4.3 Mule timeout

The goal of this experiment was to document the impact which the MULE timeout has on the amount of packages which are dropped by OUs and accumulated by MULEs. The experiment was conducted 10 times (10 iterations) for a time period of 10 minutes (to obtain an average value) and with a different timeout for the MULEs ranging from 0 to 15 seconds.

Figure 6.8 shows the results from the experiment where the MULE timeout increased from 0 to 15 seconds. The figure shows that the number of packages sent to the MULE (but not necessarily received) decreases as the timeout for MULE increases. This is because of the MULE physically moving less as more

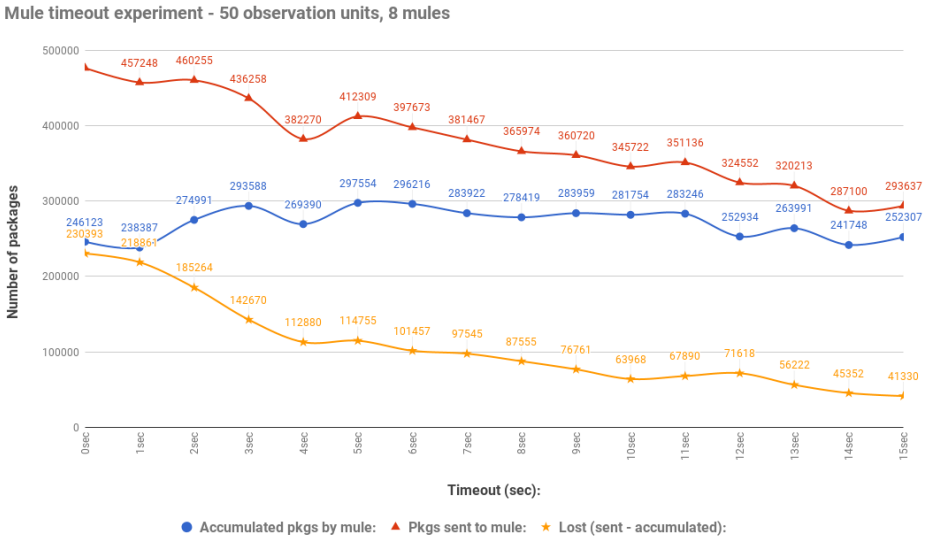


Figure 6.8: Timeout experiment results - 50 observation units, 8 mules

time is spent waiting for data accumulating from OUs. Less movement for the MULE means that the MULE will also initialize less data-accumulations from sub-networks.

An interesting part however is that the number of packages which are accumulated by the MULEs does not change too much when the timeout for the MULE is above 3 seconds. This is because as the MULE timeout increases in seconds, the MULEs will be able to wait for more packages even though not as many packages are originally sent by the OUs. This is also seen in the yellow starred line which shows the number of packages which are lost in network transmission between the OUs and MULEs due to the MULEs going out of range of the sending OUs.

Figure 6.9 shows how the percentage of packages received and accumulated by the MULE increases as the timeout increases. We see that the percentage of packages lost due to MULEs going out of range can be as low as 14.08% when using a 15 seconds timeout for the MULEs.

It can therefore be concluded that as the MULE timeout increases, the number of packages lost decreases.

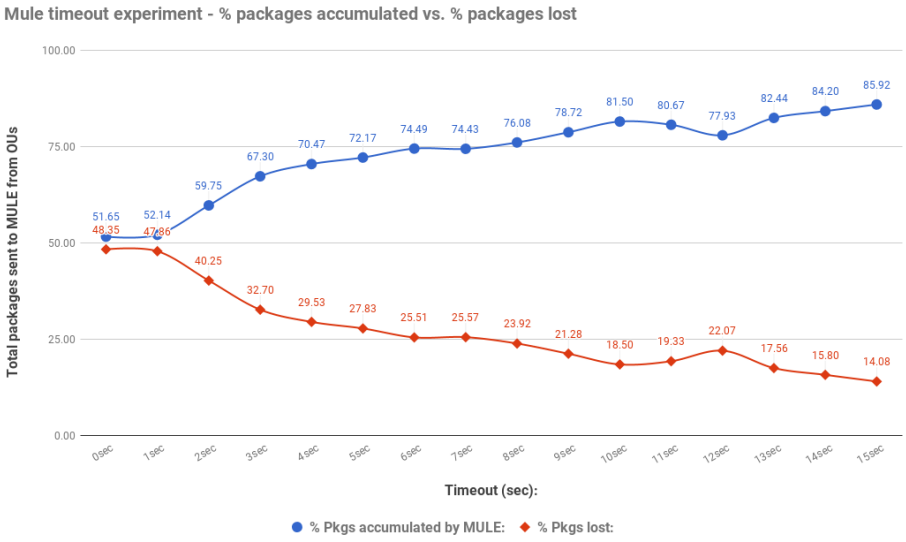


Figure 6.9: Timeout experiment results - percentage of packages accumulated vs. percentage of packages lost due to MULE going out of range

6.4.4 Raspberry Pi CPU and memory utilization utilization

Figure 6.10 shows the results measured from the single RPi over a duration of 10 minutes which was repeated 10 times (10 iterations). It has to be pointed out that the measured CPU and memory footprint is for the single MULE process and not the device as a whole.

From the figure we can see that both the CPU and memory footprint for the single MULE process is very low and stable. The memory footprint averages at around 2.2% of total CPU whilst the memory (RAM) utilization is at around 1.3% of total RAM. 1.3% is of 1GB RAM (which the RPi has) is 13MB.

Figure 6.11 shows the total system CPU and memory usage with the results from only running a single MULE also plotted in dark blue and black. The total system memory usage is seen spiking but by comparing it to the memory footprint of the single MULE process, one can deduce that it is another non-DMNI process causing this.

Further it can be concluded that running just a single MULE process on a RPi does not take up an excessive amount of resources. This comes down to the hardware in the RPi which has a quad core CPU and 1GB of memory. In addition, this proves to show that another device, the Raspberry Pi Zero, could possibly turn out to suit DMNI better as it is not require as much energy as the

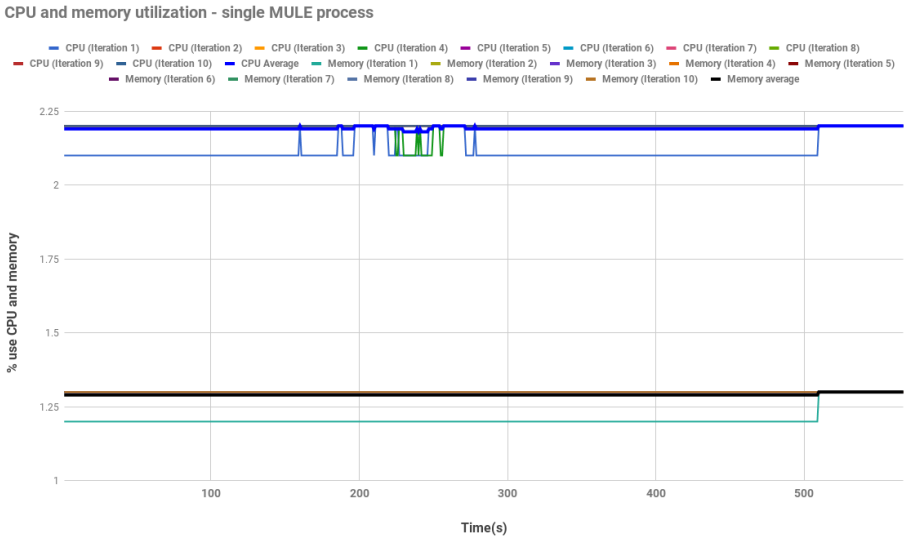


Figure 6.10: CPU and memory utilization - single MULE process

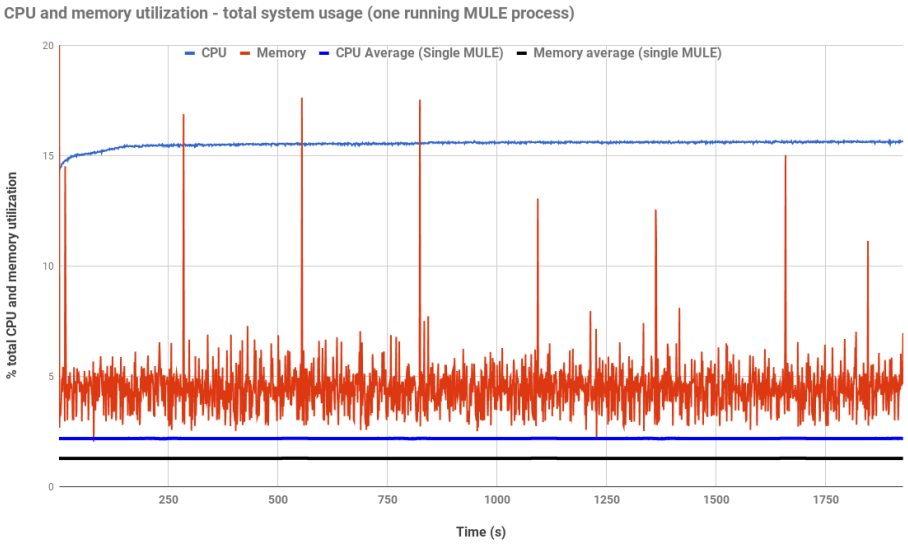


Figure 6.11: CPU and memory utilization - total system utilization

RPi does.

Experiments were not conducted on a Raspberry Pi Zero due to network related issues. This was because the Raspberry Pi Zero was connected to the Eduroam network at UiT which blocked access to the simulator.

6.4.5 Raspberry Pi Network utilization

Figure 6.12 shows the amount of file descriptors used by a single MULE process over a matter of 10 minutes and repeated 10 times (10 iterations). It has to be noted that the amount of open file descriptors due to having open files (such as log files etc) is also included in these numbers.

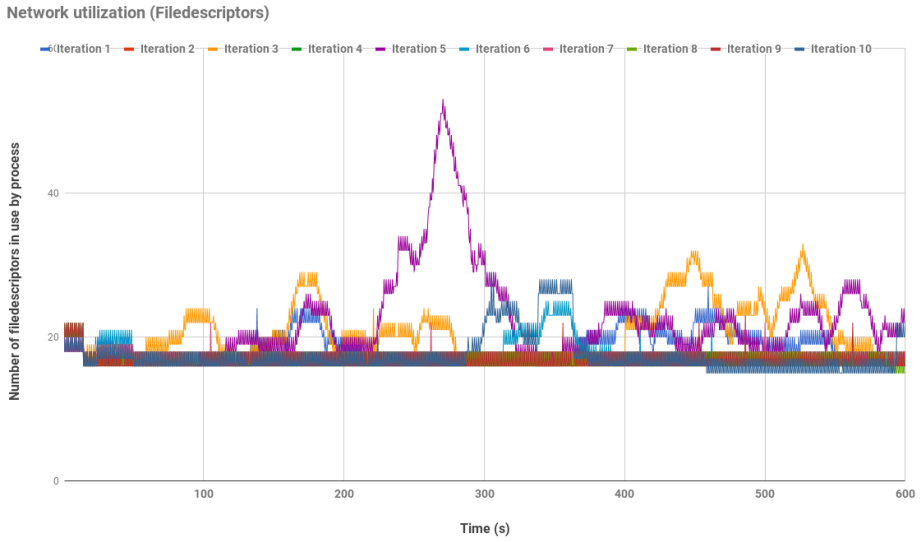


Figure 6.12: Network utilization - 10 iterations running for 10 minutes

The figure shows how for each experiment conducted, the network utilization for the RPi follows a wave like pattern which corresponds to the amount of file descriptors which the RPi is utilizing. A higher number of file descriptors in use means more open connections, thus more network traffic than if the number of file descriptors was low.

The wave pattern is expected as the number of file descriptors in use will vary depending on how many open connections to other devices there are. Since the routing for MULEs in the simulation is randomized, the spikes seen in the figure will also be randomized depending on where the MULE was located in the simulation grid.

One interesting thing however is that during iteration 5, after approximately 250 seconds, or about 4 minutes, the amount of open connections spiked up to approximately 60 open connections. This number is rather high and seems to be an anomaly in the measurements.

A possible reason for this may be due to the MULE opening a number of

connections to a sub-network of OUs, but relocating before they are able to close the connection. Since each network connection has a timeout (of 25 seconds), the MULE could possibly have located to another sub-network before the previous connections were closed. This could potentially lead to a high number of open connections.

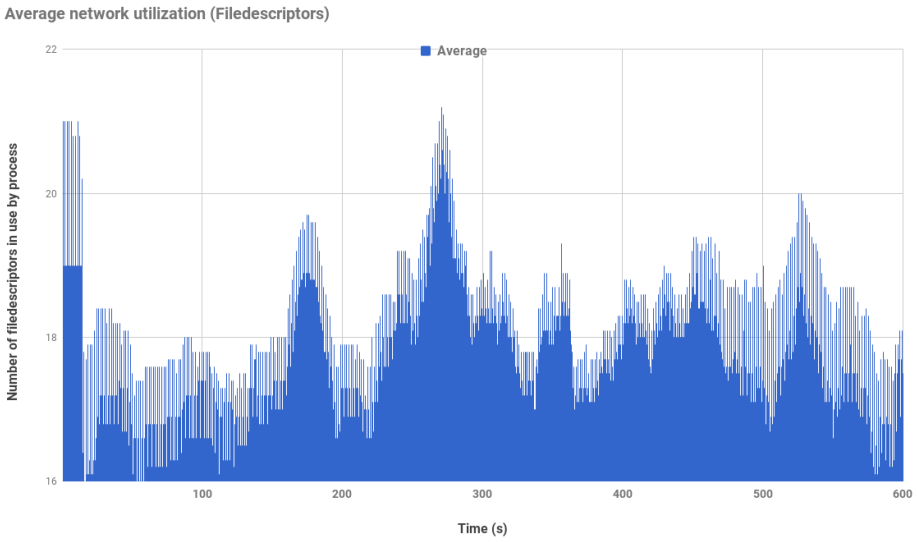


Figure 6.13: Network utilization average

Figure 6.13 shows the average number of open connections for the 10 times the experiment was conducted. Here we can see that the average connections is anywhere from 16 to 22 open connections which is reasonable. Again, the simulator has a small grid size of 1500x1500, which would cause a lot of open connections as the MULE would encounter OUs rather often.

Figure 6.14 shows the average number of open connections per iteration with the relevant standard deviation. If compared to Figure 6.12, it can be seen that for iteration 5, the standard deviation is quite high and this is due to the anomaly which happened during the 5th iteration of the experiment.

Further the figure shows that an average of around 16-17 open connections. This comes down to that MULEs will have open connections on hold whilst data is being accumulated. This comes in addition to having open connections towards the simulator server and open log files.

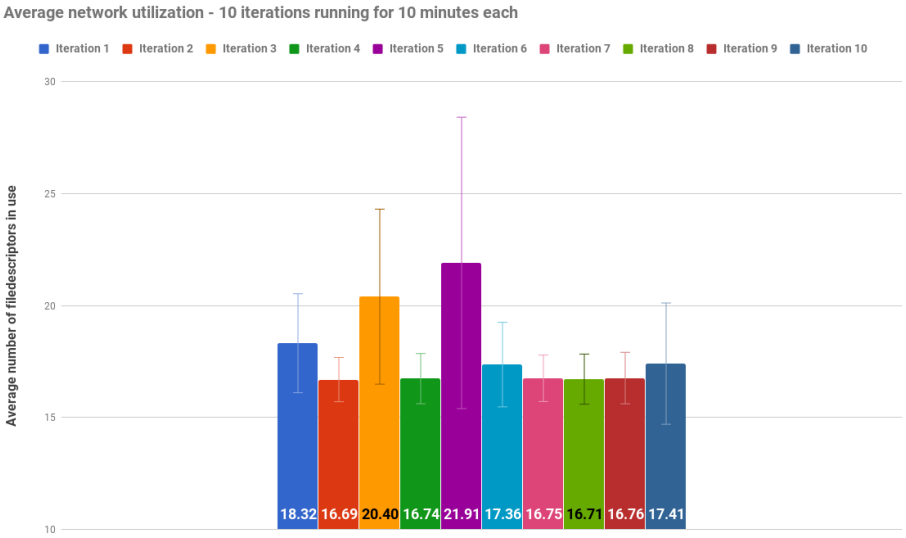


Figure 6.14: Network utilization per iteration average

7 Discussion

7.1 Data accumulation from single devices

Data accumulation from single devices are handled the same as accumulations from devices which are part of sub-networks. If a device is separated, meaning not part of a sub-network, its effective communication range is limited. This is because the range which a Mobile Ubiquitous LAN Extension (MULE) has to be within to accumulate data from a device is directly related to the size of the sub-network which the device is part of. Figure 7.1 shows this where the reach of the blue node is limited to what its own communication device can provide, the light blue circle. On the other hand, all the devices marked in green can effectively reach out to the area marked in light green as they are connected together in a sub-network.

Seeing how a device does not need direct contact, but may utilize multi-hop routing to reach a MULE, this implicitly implies that a device's effective range is directly related to the reach of the sub-network which it is a member of. Section 7.6 goes into some detail on the impact which multi-hop may have on the time a MULE will have to wait for accumulated data to be received.

The result of this is that devices which are not part of any sub-network or are part of very small networks will have limited reach and will in some situations have to wait longer before being able to have data passed to persistent storage.

A possible way to solve this is for devices which are either alone or part of

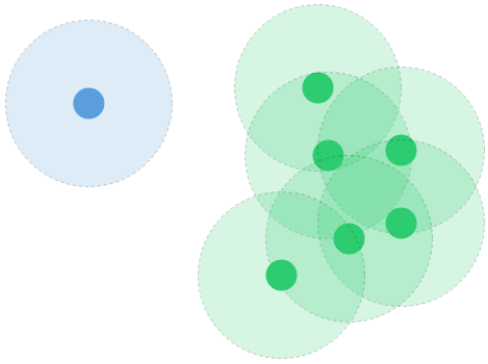


Figure 7.1: Single Observation Unit range vs. sub-network of interconnected Observation Units

a small sub-network to utilize a long-range antenna for communicating with MULEs. This would mean a trade of energy for communication range which could prove to be reasonable. Chapter 9 goes into some further ideas to solve this.

7.2 Network OU size vs. latency & energy consumption

Although bigger Observation Unit (OU) sub-networks may result in a greater chance for an OU to have its data passed back to persistent storage, the overall network energy consumption and latency will increase as sub-networks grow in size.

The size of an OU sub-network is determined by the area which the sub-network effectively covers, as described in Figure 7.1.

7.2.1 Overall energy consumption

An issue that arises when utilizing multi-hop routing between devices in a Wireless Senor Network (WSN) is that certain areas may become '*hot-spots*'[32, 5] in the sense that there is an energy imbalance among the devices in the network, especially close to static sinks as seen in Figure 7.2. This is since some routes are predestined to be chosen more often when there are static sinks - however due to the very nature of MULEs being mobile, this issue is limited and comes down the very movement pattern that the MULE uses. If a MULE follows the same path repeatedly, the MULE based solution will also suffer

from energy imbalance. If for example the MULEs follows a randomized route, different routes within the sub-networks will be used to reach the MULE. See Section 7.3 for further discussion.

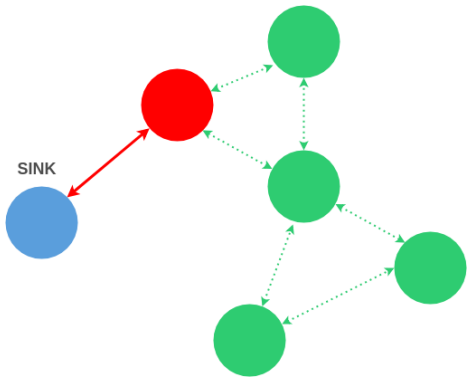


Figure 7.2: Static sink - energy imbalance close to sink

There are proposed solutions for the power-consumption issue and power imbalance, such as using a more intelligent transmission power control policy which requires nodes further away from sinks to use longer transmission routes to reach the sink instead of the shortest path possible[27].

7.3 MULE routing

In the given implementation of Dynamic Mobile Network Infrastructure (DMNI), the path which the MULEs follows in the virtual grid is randomized - it does not go from a specific point in the grid to another specific point in the grid. It was decided to not focus on MULE routing as this is a whole other topic in itself.

There are several different solutions for solving this however - although most depend on the question whether or not there is any information available for the MULE about the networks before selecting a route. Are the MULEs aware of the location for OUs or are they simply navigating blindly without any prior knowledge.

Given that a system such as DMNI may be utilized in several different fashions, a randomized routing seemed most appropriate. Several other routing techniques may be utilized, but as mentioned, this depends on the use case of the specific system.

7.3.1 Solutions where prior knowledge about network exists

In the first scenario, the goal is to solve the Minimum-Path Data-Gathering (MPDG) problem - find a path which minimizes the path whilst still reaching all sub-networks in a graph.

> "*Given a graph G = (V, E), v_s , and G_s , the MPDG problem is to find a gathering path P, starting from v_s , connecting one landing port in each subnetwork, and returning to an ending node in G_s such that the path length |P | is minimized.*"[35]

'*Data Gathering by Mobile Mules in a Spatially Separated Wireless Sensor Network*'[35] proposes two heuristics algorithms for the MPDG problem - a greedy algorithm (7.3.1.1) and a convex hull-based algorithm (7.3.1.2).

7.3.1.1 Greedy Algorithm

The greedy algorithm is based on that the MULE will choose the next node by determining that the node is within an unvisited sub-network and the node is the closest to the current location of the MULE. This is repeated until all sub-networks has been visited.

7.3.1.2 Convex Hull-Based Algorithm

The second solution proposed in [35] is based on a convex-hull concept which was first proposed in [16].

> "*This algorithm is designed based on selecting a delegation node in each subnetwork and constructing a convex hull from these delegations for continuous polishment.*"[35]

The essence of this proposed solution is to for each sub-network calculate a delegation node which is closest to the center-of-gravity - these nodes are set as landing-nodes. Then a convex hull of the system is calculated and it is attempted to keep the cost of paths to a minimum.

It is proven in [35], through simulation, that the algorithms perform rather close to optimal solutions in most practical cases.

7.3.1.3 Partitioning Based Scheduling Algorithm

The Partitioning Based Scheduling (PBS) algorithm, bases its solution on the fact that in many systems knowledge of data aggregation rate in sensors is known[12]. This may also be true for a sensor network based on the Arctic tundra.

In PBS, each node n_a is associated with a buffer overflow time o_a - the basic idea behind PBS is that a path needs to be created so that two consecutive visits to n_a are at the most o_a apart. This is done by calculating the MULE route in two steps - a *partitioning* phase and a *scheduling* phase. The partitioning phase groups nodes with close buffer overflow times and closely located. The scheduling phase within the groups themselves are calculated and then concatenated to create a complete route through the sub-networks.

A similar system could possibly be created to also include sub-networks of devices - not only single nodes. It would however require knowledge of the overflow time for the OUs.

7.3.2 Mule routing without any prior knowledge

In the event that no prior knowledge is known about the network which the MULE will operate in, there are some questions that has to be answered to best choose the routing path for the MULEs:

- Are nodes in the network static or do they move?
- Can information about the network gradually be collected and used?
- Is there a fixed number of devices in the system or may this change?

In the event that devices in the network may change their position, a randomized routing algorithm may be the best fit. This comes down to that regardless if a complex algorithm for pathing is used, the nodes can move. If not randomized, one can build up a map (split into sections like a grid) where the MULE has been and follow a '*Round-Robin (RR)*'[4] sort of algorithm in which each section of the map will be appended to an array. Each array element (map area) will then be visited in sequence and repeated once the end of the array is met. Much like RR, areas would be visited without any priority and in a circular order.

In the event that information can be gradually built up, a MULE can enter a scanning routine in which it "scans" a grid and builds up the information it has

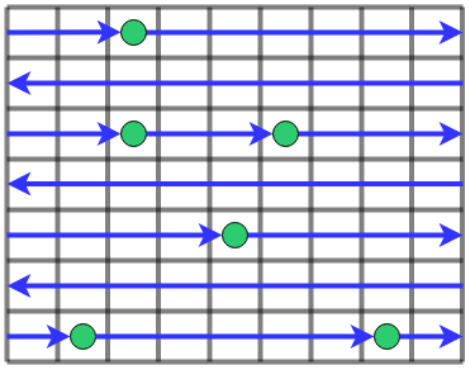


Figure 7.3: Grid scan example

about the devices in the area. Figure 7.3 gives an illustration of this in which the device moves through each row and each column of the given grid.

7.4 Network topology

In the current implementation of DMNI, the network topology design is that there is no limitation nor control over which devices join which network. A node will connect with all nearby DMNI devices and thus implicitly join a sub-network which consists of all the devices that are linked together in said network.

Unfortunately, this limits the design in terms of scalability and reliability since the sub-networks can (in theory) grow indefinitely. Subsection 7.4.2 goes into some detail about how this can be resolved by limiting the number of forwards a request can have.

7.4.1 Implosion and overlap when accumulating data

Given that DMNI utilizes flooding for various reasons such as data-path mapping, there is a possibility of implosion and overlapping of requests[13, 1]. Implosion and overlapping occur when devices receive multiple copies of the same message as a result of using flooding. When flooding, the devices does not take into account who has already receives this message and who has not. Chapter 9 goes into some suggested ways to solve this.

7.4.2 Limited number of request forwards

Since there is no limit on the length a request may reach in a sub-network of devices, it may cause throttling of the network if the network is of significant size. This is because, in theory, the sub-networks can grow to a size which is not feasible. Since some requests are time sensitive, such as data-accumulations, a large system would only slow the system and cause devices part of the sub-network to have their energy drained as a large number of requests could potentially flow through the sub-network.

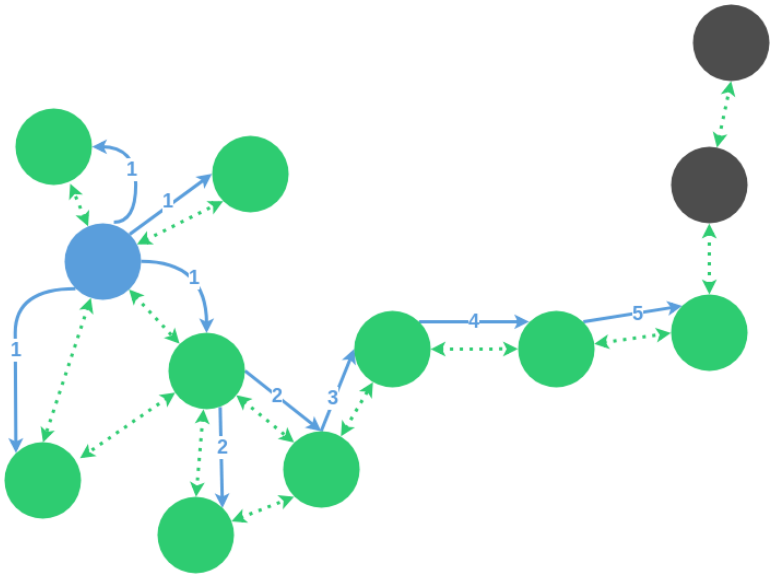


Figure 7.4: Limited request range example

One way this could potentially be resolved is by limiting the amount of hops a request can take in a sub-network before the request will no longer be forwarded.

Figure 7.4 gives an example of how a limited range for requests can function. In the example the blue node initiates a flooding request with may only reach 5 steps in the sub-network - once it has reached the 5 steps the flooding stops. This means the two devices marked in black would not be affected by this request nor would they even have knowledge that such a request took place. The green circles are the devices which the request reached. This way sub-network could grow in size without it affecting the entire sub-network and all devices part of it.

7.4.3 Knowledge of entire sub-networks vs. direct neighbours

As previously explained, DMNI devices are only aware of their direct neighbours - this is by design. Since the number of devices in a DMNI network may be in the order of tens, hundreds or even thousands, it crucial that the networks can scale without it affecting the performance of each device too much. Devices have limited computational power and memory and it would not be feasible nor scalable to constantly have to update the information about the entire sub-network if it is of significant size.

It is possible for a device to have knowledge of the entire sub-network which it is part of by gossiping data between nodes in the network. It does however raise a scalability and consistency issue once the network grows in size.

If the device only has knowledge of its direct neighbours, the amount of data which the device has to keep track of and update is considerably lower than if the device had to keep information about the entire network. In addition, data about the direct neighbours is fetched directly from the neighbours themselves and does not have to be routed (gossiped) through a number of nodes to reach the destination.

7.5 Accumulation of data

Accumulation of data in DMNI is of big importance and it is therefore crucial to do this in an effective manner. To increase the likelihood of successfully accumulating data from all devices in a sub-network, DMNI bases its accumulation routine on these outlines:

1. Every device in a sub-network may be utilized to accumulate data from

2. DMNI should function both with and without prior knowledge of the sub-networks

3. The chance of duplicated data should be minimal - OUs will only send data once

4. DMNI prioritizes performance over consistency - performance and power-efficiency is prioritized over the chance of data-loss

7.5.1 Chance of data-loss

With these points in mind, it is clear that one drawback that DMNI has to deal with is the chance of data-loss due to the fact that once data has been accumulated from a device, the device will not await for any confirmation or acknowledgement that the data has been successfully transferred to persistent storage. This results in devices possibly deleting data which has yet to either reach a MULE or be duplicated on another device in the network.

7.6 When to stop waiting for accumulated data

In DMNI once a MULE initiates a data accumulation, the MULE will wait for a static amount of time (default 10 seconds) before continuing on its path. In the case for the DMNI simulation, this is sufficient for data-accumulation from a large amount of nodes. However in a real life scenario, the amount of time a MULE would have to wait for a data would vary from time to time. The reasons for this may be due to[22]:

- Radio fluctuations
- Temporary node failures
- Imbalanced/unknown network size

In addition, the MULEs need to account for multi-hop latency which will occur within the sub-network. Multi-hop latency is the time which is used to forward data via multi-hop routing in a sub-network of OUs. As Subsection 6.4.3 shows, the multi-hop latency is hard to determine when the MULE is unaware of the amount of OUs which it is accumulating data from. In the event that the system limits the number of forwards a request can take, as discussed in Subsection 7.4.2, the MULE can better predict how long it has to wait for accumulated data to arrive in its sink handler. This is further discussed in Chapter 9.

7.7 Simulated bandwidth

In DMNI, there is a simulated bandwidth delay which the simulator will add to all traffic which goes over the traffic handler - the implementation details for this is found in code listing 5.2. The delay is calculated depending on the size of the request, but the bandwidth itself is static and does not change.

It is known that such a static solution is not sufficient in providing simulation accuracy compared to how it is in real life. The bandwidth between devices placed on the Arctic tundra will vary for numerous reasons, some of which are mentioned in Subsection 6.4.1.

It does give some insight into the challenges which the OUs will face on the Arctic tundra and thus is acceptable for such an early stage of prototyping. Implementing a more complex system was not done because of time constraints but would be an interesting area to investigate in the future.

7.8 Device duty cycle

An issue that has not been addressed in DMNI is duty cycle of devices. Nonetheless, it is a requirement for devices which are places on the Arctic tundra to follow some sort of duty cycle in which it sleeps in inactive periods and then is awaken to take sensor recordings or to finish other tasks. Duty cycle can help reduce the overall energy consumption of the devices and since listening consumes as much energy as actually receiving[8], it is crucial that a device has the option to save energy when not in active use.

There has been work done on duty cycle, some of which are not mainly oriented towards a MULE based solution but could potentially be used in combination with other works.

One solution as an example is Geographic Adaptive Fidelity (GAF)[17] in which the devices that are closely located cooperates in saving energy. Only necessary nodes participate in transmissions whilst other enter sleep mode. A similar system to GAF is Span[8], where a power-saving technique for multi-hop ad hoc WSN was developed. Span builds on the fact that when a region for a shared channel WSN has a sufficient density of nodes, only a small number of them needs to be used at any time to forward traffic for active connections.

There is a potential for extending these sort of systems to sub-networks in DMNI where nodes that are closely located can take turns in powering down and saving energy.

An interesting issue that arises due to this is the tradeoff between device duty cycle and time in which a device will be able to communicate with a MULE - this is further discussed in Subsection 7.8.1.

7.8.1 Device duty-cycle vs. mule communication time

In a system which utilizes duty-cycles for OUs where MULEs are used for data collection, there are several impacts which has to be considered. *'Exploiting Mobility for Energy Efficient Data Collection in Wireless Sensor Networks'*[20] goes into further detail about these issues and lists the following points which impacts the sensor duty-cycle:

1. *"A MULE may not be discovered at all because the sensor was asleep during the time the MULE was in communica- tion range of sensor..."*

2. *"The amount of data that can be transferred (K) in one contact may decrease if the MULE is discovered in the middle of the duration it is in the communication range of the sensor..."*

Point two is the reason why DMNI utilizes timeout for the MULE - it will halt its location change if a data-accumulation has begun. This effectively increases the time the MULE will remain within communication distance.

By using timeouts instead of completely relying on the Data Transfer Rate (DTR) to successfully transferring the required data to the MULE, the MULE can fine-tune the time it waits for accumulated data to be received in its sink handler. This way the amount of data that can be sent from OUs can be increased and the MULE can accumulate data from multiple OUs at once.

7.9 Mule timeout

In the model proposed in DMNI, devices and MULEs can only communicate with one another if they are within a certain range of r meters in the virtual grid. Since the system relies on multi-hop communication to accumulate data, there is no telling how long a MULE will have to wait for all data to be accumulated and to arrive in the sink handler. This is because in the current implementation of DMNI, there is no limitation of number of devices in a sub-network - a solution for this was previously discussed in Subsection 7.4.1.

The time (t) a MULE has to wait relies on the following criterias[1]

- Amount of data collected, bits (B)

1. Ignoring multi-hop communication delay

- Bandwidth between each device, Mbit/sec (m)
  - Converted to bits/sec in Equation 7.1 (1 Mbit = 1 000 000 bits)
- Number of devices collected from (n)

$$t = \frac{B}{m \times 1000000} \times n \tag{7.1}$$

In addition to these points, there is some latency in the data-path mapping algorithm introduced in DMNI - although minimal, it has to be accounted for as it will indeed affect the time a MULE has to wait. Not only that, but the transfer time between the devices themselves will vary from where in the accumulation path the device is located.

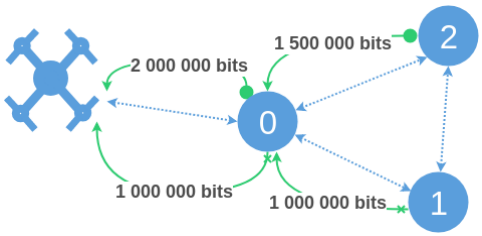


Figure 7.5: Mule accumulation delay example

As an example, Figure 7.5, shows how the accumulation from three OUs will occur:

1. First, a data-path mapping routine is carried through - the time this takes depends on size of the sub-network and if the "tree" is balanced or not - it takes t_1 time.
2. OU(2) sends 1 500 000 bits to OU(0) - with the average transfer bandwidth of 3.6817Mbit/sec, this will take 407ms.
3. OU(1) sends 1 000 000 bits to OU(0) - again with an average transfer bandwidth of 3.6817Mbit/sec, this takes 272ms
4. In this example, OU(0) receives the data from OU(2) first and following the data-accumulation algorithm in DMNI, it appends its own data (500 000 bits), equaling 2 000 000 bits. The data is then sent to the MULE. The time equals 543ms
5. Lastly, OU(0) receives from OU(1), but since it has already added and

sent its own data, it simply forwards the data to the MULE, again with a time of 272ms

The minimum communication time for the MULE in this case is thus:

$$t = t_1 + 407ms + 272ms + 534ms + 272ms = t_1 + 1485ms \tag{7.2}$$

If one follows the equation 7.1, the result is thus:

$$t = (3000000 \div (3.6817 \times 100000)) \times 2 = t_1 + 1629ms \tag{7.3}$$

As seen, the results are not equal and it can thus be deducted that the equation in 7.1 is incorrect due to multi-hop communication not being accounted for. In addition, some OU's will not append their own data to the request (As seen in step 5 above) but just forward the request.

Section Subsection 6.4.3 goes into some detail about the achieved results when testing different timeouts in DMNI.

7.10 Limited mule resources

In the current implementation of DMNI, it is not taken into account that Unmanned Aerial Vehicles (UAVs) have a limited power-supply and range. As previously discussed in Section 2.1, there has been work done which focuses mainly on the energy-efficiency of MULEs, more specifically UAVs. A more complete prototype would require this to be investigated further.

7.11 Multi-hop routing vs. single-hop routing

One of the decisions that were taken for DMNI is to utilize multi-hop routing within the ad hoc network. Some of the reasons behind this decision is as follows:

1. Multi-hop routing allows for a broader reach within the ad hoc network: instead of having to directly visit a OU, a MULE simply needs contact with another OU which is in contact with the specific OU

2. Since MULEs can move in the ad hoc network, the classic power depletion close to the sink commonly seen in multi-hop ad hoc networks can be avoided: if a static sink is placed in a multi-hop environment, the OUs

close to the sink will quickly run out of energy - this can be avoided with MULEs by flying to another device in the sub-network the next time it flies by

3. Multi-hop communication allows the OUs to utilize their low-range radio instead of long-range radio to reach far away sinks as they can use the other devices in the network to reach the sink.

7.12 Reliability vs. power-efficiency

As previously discussed in Subsection 7.5.1, DMNI data accumulations come with the risk of data-loss. It is a deliberate decision in the design of DMNI not to send acknowledgements on data once it has been accumulated in order to save energy.

In addition to this, the complexity of adding acknowledgements on data transmissions is higher than one would expect. This boils down to the problem which is known as "*Two General's Problem (TGP)*"[15], also known as the *'Coordinated Attack Problem'*. See section 7.12.1 for a further discussion on this.

7.12.1 Two General's Problem

The *'TGP'* is an experiment meant to show the challenges one has to face by attempting to communicate and coordinate over an unreliable link/network. The basic problem lies in that no matter how many times you send an acknowledgement to a message, be that the original message or an acknowledgement to an acknowledgement, you cannot be sure whether or not the receiver has actually received the acknowledgement.

This problem is especially relevant for ad hoc networks where data-links may fail at any given time. Therefore DMNI chose to simply accept the fact that data may be lost and it thus provides a *'best-effort'* delivery scheme.

7.13 Energy conservation by using data-accumulation timeout tables

Previous work show through experimental measurements that in general, data transmissions are expensive compared to data processing due to power aware-

ness within the field of radio subsystems remain rather unexplored[29].

Therefore, DMNI implements a timeout for OUs which is active for a given amount of time after every data-accumulation. Not only does this avoid the problem with MULE contacting several OUs in the same sub-network, but in addition it preserves energy by simply not allowing data-accumulations to take place too often.

The exact timeout for this has to be further experimentet with and is discussed in Chapter 9.

8

Conclusion

The goal of this Master's thesis was to design and implement a Mobile Ubiquitous LAN Extension (MULE) based data-accumulation system for the Arctic tundra. A prototype, namely Dynamic Mobile Network Infrastructure (DMNI), was designed and implemented together with a simulator in this dissertation. The thesis also includes documentation and experiments conducted for the system.

DMNI is an attempt at utilizing a MULE based data-accumulation system which utilizes the mobility of mobile agents, MULEs, to initiate and collect data from Observation Units (OUs) placed on the Arctic tundra. By allowing the OUs to discover and communicate with each other, the OUs are able to form ad hoc networks - sub-networks of interconnected OUs. Within these networks, OUs are able to send network requests through a series of other OU's, effectively using multi-hop communication to send requests further than just their direct neighbours.

The MULEs exploit the mesh-network structure found in the ad hoc networks as by having minimum one node part of said network enables the MULE to accumulate data from the entire sub-network. The sub-network uses flooding and forwarding of packages to accumulate packages from all OU's in the sub-network.

We show through experiments and simulation that a MULE based system can reduce the chance for permanent partitioning in an ad hoc network of

interconnected OUs. Through experiments conducted on an actual device, a Raspberry Pi 3 Model B (RPi), we show that the system has little impact on the overall system resource utilization. Experiments show that the system, on average, utilizes as low as 2.2% CPU and 1.3% memory. This indicates that the system could potentially run on other types of devices as well.

9

Future work

At the given point in time there still remains a fair amount of work to be done in relation to Dynamic Mobile Network Infrastructure (DMNI) and simulating the Arctic tundra. The first step to improving DMNI is to improve the simulator to make it more realistic and to add more options to it, such as being able to add network corruption and package loss. In addition, the network bandwidth delay has to be fine-tuned to make it more realistic in relation to the exact type of data one can expect the Observation Unit (OU) to collect.

Further, the Mobile Ubiquitous LAN Extension (MULE) has to be prototyped using a an actual Unmanned Aerial Vehicle (UAV) - possibly a drone such as a quadcopter. Some experiments has been conducted at University of Tromsø (UiT) where results were quite promising. The experiment can be read about in Appendix B.

Not only does a real prototype has to be used, but the *type* of device connected to the MULE has to be thoroughly tested. Testing of CPU, memory and network has been done on a Raspberry Pi 3 Model B (RPi) and it is believed that the power-demand that such a device requires is simply too great. A possibility is to utilize a Raspberry Pi Zero which according to sources utilizes around half of the energy that a RPi does[14].

In order for the MULE to be able to collect data from the networks, the networks themselves has to be limited in size. From the experiments conducted in this dissertation, it is clear that the timeout which MULEs has to wait will grow

depending on the size of the sub-networks. In theory, since there is no limitation on number of OUs in a sub-network, a MULE could potentially be collecting data from the entire network of OUs by simply being connected to a single OU and this does not scale. Therefore it is clear that an improvement for the system would be to put a limit on the size of the ad hoc networks that could be created. An alternative to this is to limit the number of forwards a request within a sub-network can go through - this was previously discussed in Subsection 7.4.2.

Further work has to be done to keep the energy consumption of the ad hoc networks to a minimum. As an example, the data-accumulation timeout which the OUs has could be further improved. This could be done by experimenting with the exact timeout they should have for data-accumulations to ensure that they do not reject data-accumulations when they should in fact accept it.
As an alternative, devices in the ad hoc network could reject data-accumulation requests if they have less than a given amount of data collected.

To further improve the accumulation of the system, one could implement a rendezvous-based-based solution. If the OUs are aware of the path for the MULE, they could accumulate data to the OU which is closest to that path before the MULE arrives. This way the time the MULE waits for accumulated data to be receives can be decreased. It would however require that the OU know what path the MULE will take - this could potentially be solved with the MULE utilizing a long-range radio to announce its path to OUs. It would mean that the MULE would utilize more energy but could potentially make up for that by reducing the data-accumulation timeout.

Bibliography

[1] I. F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, Aug 2002.

[2] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216 – 219, 1979.

[3] O. Anshus. Distributed arctic observatory (dao): A cyber-physical system for ubiquitous data and services covering the arctic tundra. `https://www.forskningsradet.no/prognett-iktpluss/Nyheter/NOK_200_million_for_13_research_projects_on_Ubiquitous_data_and_services/1254032932215?lang=no`, 2018. Norwegian Research Council (NRC) - Project no: 270672.

[4] R. Arpaci-Dussea and A. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[5] Muhammad Ayaz, Azween Abdullah, and Low Tang Jung. Dynamic cluster based routing for underwater wireless sensor networks. 2009.

[6] G. Burd and J. Bauch. Gorilla websocket. `https://github.com/gorilla/websocket`, 2018.

[7] Almende B.V. vis.js, 2010.

[8] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *Wirel. Netw.*, 8(5):481–494, September 2002.

[9] Zong Da Chen, H.T. Kung, and Dario Vlah. Ad hoc relay wireless networks over moving vehicles on highways. In *Proceedings of the 2Nd ACM International Symposium on Mobile Ad Hoc Networking &Amp; Computing*, MobiHoc '01, pages 247–250, New York, NY, USA, 2001. ACM.

[10] Rone Ilídio da Silva and Mario A. Nascimento. On best drone tour plans for data collection in wireless sensor network. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 703–708, New York, NY, USA, 2016. ACM.

[11] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. iperf - the ultimate speed test tool for tcp, udp and sctp. `https://iperf.fr/`, 2018.

[12] Eylem Ekici, Yaoyao Gu, and Doruk Bozdag. Mobility-based communication in wireless sensor networks. *IEEE Communications Magazine*, 44(7):56–62, 2006.

[13] Luis Javier García Villalba, Ana Lucila Sandoval Orozco, Alicia Triviño Cabrera, and Cláudia Jacy Barenco Abbas. Routing protocols in wireless sensor networks. *Sensors*, 9(11):8399–8421, 2009.

[14] J. Geerling. Power consumption benchmarks. `https://www.pidramble.com/wiki/benchmarks/power-consumption`, 2018.

[15] Piotr J. Gmytrasiewicz and Edmund H. Durfee. Decision-theoretic recursive modeling and the coordinated attack problem. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 88–95, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[16] Bruce Golden, Lawrence Bodin, T Doyle, and W Stewart Jr. Approximate traveling salesman algorithms. *Operations research*, 28(3-part-ii):694–711, 1980.

[17] Jitender Grover, Shikha Sharma, and Mohit Sharma. A study of geographic adaptive fidelity routing protocol in wireless sensor network. 16:2278–661, 10 2014.

[18] R. A. Ims, J. Jepsen, A. Stien, Å. Ø. Pedersen, E. Soininen, J. E. Knutsen, and S. T. Pedersen. Climate-ecological observatory for arctic tundra (coat). `https://coat.no/`, 2010.

[19] R. A. Ims, J. Jepsen, A. Stien, and N. Yoccoz. Science plan for coat: Climate-ecological observatory for actic tundra, 2013.

[20] Sushant Jain, Rahul C. Shah, Waylon Brunette, Gaetano Borriello, and Sumit Roy. Exploiting mobility for energy efficient data collection in wireless sensor networks. *Mobile Networks and Applications*, 11(3):327–339, Jun 2006.

[21] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. *SIGARCH Comput. Archit. News*, 30(5):96–107, October 2002.

[22] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, Ltd, 2006.

[23] I. Khemapech, I. Duncan, and A. Miller. Energy preservation in environmental monitoring wsn. In *2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 312–319, June 2010.

[24] William Lehr and Lee W McKnight. Wireless internet access: 3g vs. wifi? *Telecommunications Policy*, 27(5):351 – 370, 2003. Compeitition in Wireless: Spectrum, Service and Technology Wars.

[25] M. Otto and J. Thornton Twitter Inc. Bootstrap. `http://getbootstrap.com/`, 2018.

[26] Å. Ø. Pedersen, A. Stien, E. Soininen, and R. A. Ims. Climate- ecological observatory for arctic tundra - status 2016. `https://issuu.com/framcentre/docs/framforum-2016-issu`, March 2016.

[27] M. Perillo, Z. Cheng, and W. Heinzelman. An analysis of strategies for mitigating the sensor network hot spot problem. In *The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pages 474–478, July 2005.

[28] Samuel Raff. Routing and scheduling of vehicles and crews: The state of the art. *Computers & Operations Research*, 10(2):63 – 211, 1983. Routing and Scheduling of Vehicles and Crews. The State of the Art.

[29] V. Raghunathan, C. Schurgers, Sung Park, and M. B. Srivastava. Energy-aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, Mar 2002.

[30] Rahul C. Shah, Sumit Roy, Sushant Jain, and Waylon Brunette. Data mules: modeling and analysis of a three-tier architecture for sparse sensor networks. *Ad Hoc Networks*, 1(2):215 – 233, 2003. Sensor Network Protocols and Applications.

[31] W. Shirou. gopsutil. `https://github.com/shirou/gopsutil`, 2018.

[32] Shio Kumar Singh, MP Singh, Dharmendra K Singh, et al. Routing protocols in wireless sensor networks–a survey. *International journal of computer science & engineering survey (IJCSES) Vol*, 1(63-83):29–31, 2010.

[33] N. Smith. Don't use go's default http client (in production). `https://medium.com/@nate510/don-t-use-go-s-default-http-client-4804cb19f779`, 2016.

[34] C.K. Toh. *Wireless Atm and Ad-Hoc Networks: Protocols and Architectures*. Kluwer Academic Publisherb Group, 1997.

[35] F. J. Wu, C. F. Huang, and Y. C. Tseng. Data gathering by mobile mules in a spatially separated wireless sensor network. In *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 293–298, May 2009.

[36] O. Younis and S. Fahmy. Heed: a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Transactions on Mobile Computing*, 3(4):366–379, Oct 2004.

Appendices

A

Use of DMNI

DMNI requires that golang is installed. This can be installed via the following link: `https://golang.org/dl/`. Version 1.10.2 was used - there is not a guarantee that older nor newer versions of golang will function as expected.

A.1 Setup

The source code of DMNI has to be placed in golang's GOPATH - this defaults to '$HOME/go/'. The following folder structure has to be used in order for DMNI to function as expected:

'$HOME/go/src/github.com/fagerli93/dmni'

A.2 Dependencies

There are several dependencies required to run DMNI. By utilizing the Makefile provided in the project, all dependencies with the exception of golang can be installed by using the command:

Listing A.1: Make command for installing dependencies and compiling DMNI executeables

```
1  $ make clean && make all
```

All dependencies can also be installed manually via the the command seen in Listing A.2.

Listing A.2: Terminal commands for installing dependencies manually

```
1  $ go get github.com/shirou/gopsutil
2  $ go get github.com/gorilla/websocket
3  $ go get github.com/mitchellh/mapstructure
4  $ go get golang.org/x/sys/unix
```

A.3 Compiling and running the executables

By executing the command given in Listing A.1, the executeables will also be compiled. These can also be installed manually as follows (when current directory is '$HOME/go/src/github.com/fagerli93/dmni':

Listing A.3: Terminal command for compiling executables manually

```
1  go install ./...
```

All executeables in the DMNI will be compiled and will be located under '$HOME/go/bin/' - the following executeables will be compiled:

- dmni - DMNI process
  - Directory: '$HOME/go/src/github.com/fagerli93/dmni/'
- ou - OU process
  - Directory: '$HOME/go/src/github.com/fagerli93/dmni/cmd/ou'
- mule - MULE process
  - Directory: '$HOME/go/src/github.com/fagerli93/dmni/cmd/mule'
- simulator - simulator process
  - Directory: '$HOME/go/src/github.com/fagerli93/dmni/cmd/simulator'

Each executeable can also be individually compiled via the command found in Listing A.4. You need to be in the correct folder as seen in the list above where

there will be a single ‘main.go‘ file.

Listing A.4: Go build command

```
1 go build
2 ./dmni
```

A.4 Optional commandline parameters

The executables allow for the following commandline parameters:

- dmni
  - -ob <number of observation units in simulation>
  - -m <number of mules in simulation>
  - -ll <log level: 1 = Trace, 2 = Debug, 3 = Info, 4 = Warn, 5 = Error, 6 = Fatal>
  - -t <timeout (s) for mule>
- mule
  - -x <x coordinate of the mule in the virtual grid>
  - -y <y coordinate of the mule in the virtual grid>
  - -p <port of the mule HTTP server>
- ou
  - -x <x coordinate of the mule in the virtual grid>
  - -y <y coordinate of the mule in the virtual grid>
  - -p <port of the mule HTTP server>
  - -ll <log level: 1 = Trace, 2 = Debug, 3 = Info, 4 = Warn, 5 = Error, 6 = Fatal>
- simulator

B

DAO - Flying network infrastructure experiment

An experiment was conducted at the University of Tromsø (UiT) in December 2017 by members of the Distributed Arctic Observation (DAO) project where a UAV (quadcopter drone) was equipped with a radio and relayed LTE (4G) mobile network down to a ground station.

The purpose of the experiment was to see if LTE could be forwarded via the drone and effectively increasing the bandwidth for the basestation placed on the ground.

The speed which the basestation originally got was less than 0.1 Mb/sec.

With the drone flying 35 meters above the ground, the basestation could effectively achieve a speed of 25 Mb/sec download and 3 Mb/sec upload speed.

At the highest altitude, 118 meters above the ground, the basestation achieved a speed of approximately 65 Mb/sec download and approximately 10 Mb/sec upload speed.

The experiment was documented and uploaded to YouTube on the following link: `https://www.youtube.com/watch?v=CIrZgZWUgpE`

C

DMNI Visualization preview

Figure C.1 shows a screenshot of the visualizer whilst running with 50 OUs and 5 MULEs in a 2000x2000 virtual grid. Each device has a communication range of 150 grids.

A video of the visualizer can be seen by following the link:

`https://www.youtube.com/watch?v=ycHo1Egft74&feature=youtu.be`

It can also be seen in the delivery with the file name: '*dmni_visualizer.mkv*'

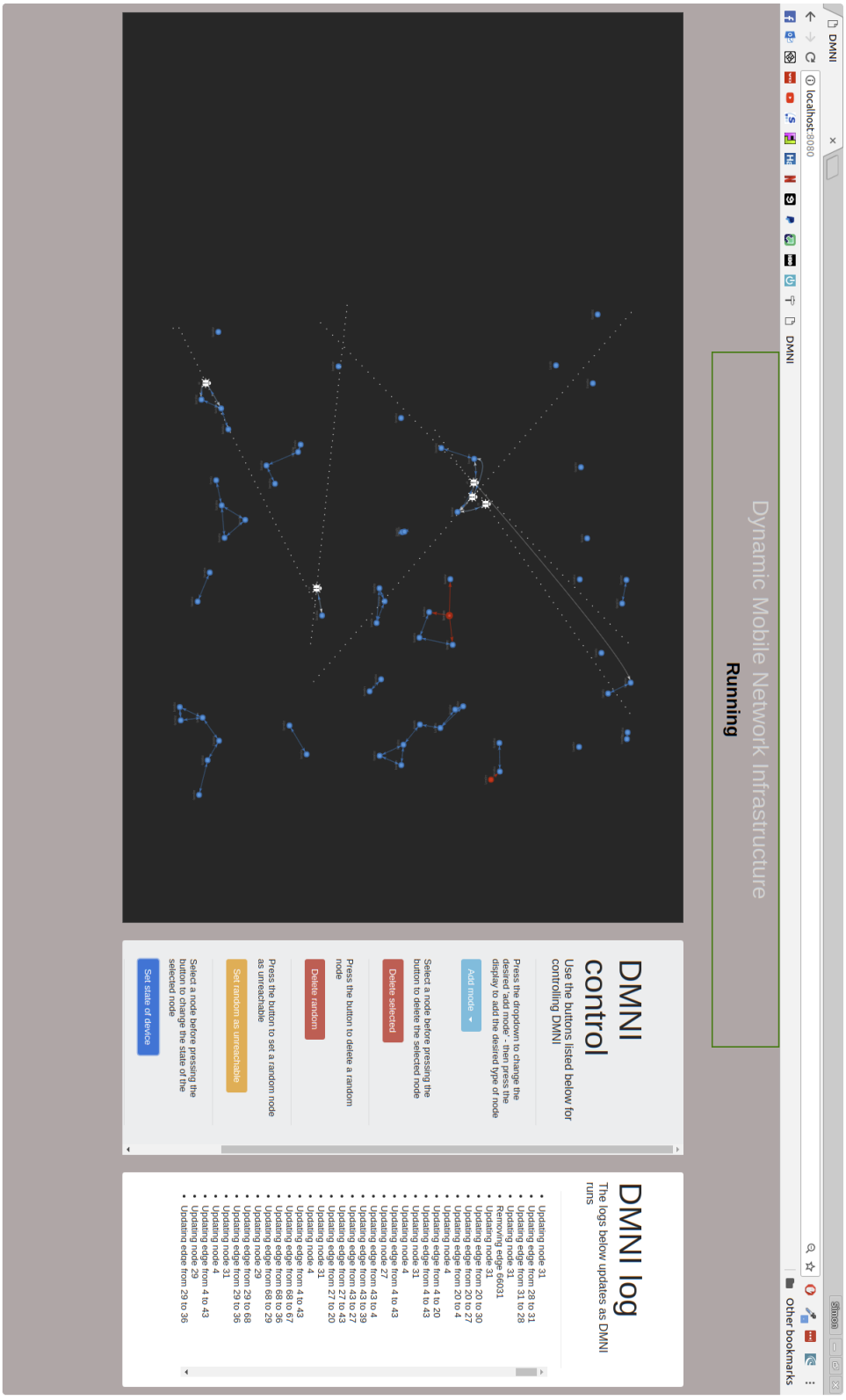


Figure C.1: DMNI Visualizer Screenshot