NTNU
Kunnskap for en bedre verden

Department of Mechanical and Industrial Engineering

TPK4560 - Robotics and Automation, Specialization Project

Sensor, Know Thyself

Author:
Kristoffer Susort Kvale

January, 2022

Contents

List of Figures

Listings

Sammendrag - Jeg har iløpet av dette master prosjektet laget en simulator for nettverk, som også har mulighet for å simulere arkitekturen jeg utviklet og presenterte i min prosjekt rapport i høst. Jeg viste på hvilke måter simulatoren ikke stemmer med virkeligheten. Jeg sammenlignet også resultater fra simulatoren for et refferansenettverk sammenlignet med en forenklet versjon av arkitekturen jeg presenterte i høst. Bassert på disse resultatene viste jeg at arkitekturen gir gode resultater. Jeg inkluderte også pseudo-kode og flere ideer for forbedring og implementering av arkitekturen under videre arbeid.

Abstract - In this master thesis, I have created a simulator for networks, with the possibility to simulate a new architecture I developed and presented in my project report. I showed in what ways the simulator differs from a real-world scenario. I also compared results from this simulator between a reference network and a simple simulation of the architecture I suggested. Based on these results I showed that the architecture I suggested is worth further research and work. I also included pseudo-code and ideas related to improving and implementing the architecture in the future.

1 Introduction

Interoperability and security are two of the important aspects to consider in industrial Internet of Things (IoT). Security is important, for instance ensuring access to data, i.e. the attack on Hydro Aluminium. It is also important that business secrets are not leaked to competitors. Interoperability is a counterweight to supplier locking, where if you start to buy equipment from a supplier, you are locked to buying from them later for the new equipment to work with the old equipment.

My project report suggests a new architecture to increase interoperability and efficiency in IoT. Interoperability is important to avoid supplier locking. Efficiency is in this case referring to Local Area Network (LAN) load, where this architecture uses the LAN to establish alternative communication channels that can be used to send the data. The architecture is based on the idea that the IoT devices know what communication technologies they can use.

At first, the plan was to implement a first version of the code for the devices to test it with a virtual surrounding system. This approach resulted in several problems, ultimately leading to implementing a simulation instead. After quickly checking existing solutions for simulating networks I concluded to make my own simulator. There are two result sets as part of this master project. The first one is the simulator itself, the second is the results of the simulator.

The implementation attempt was based on creating and using a version of an execute function. The execute function would be given the function to run and execute it. The functions would have been defined as a part of the method definition. The reason this approach failed is mainly the level of detail required. There are still several parts of the architecture that is not sufficiently defined for it to be implemented. There is no routing protocol for devices to take advantage of the alternative connections in an efficient way. In the project report, I presented several techniques and technologies that could be useful for implementing a first version of the architecture, but most of them would require a significant amount of work to use, and that was not feasible. After attempting to implement the methods structure, I decided to change the direction of the master project to a simulation.

Due to time constraints, I decided to create a simple simulator based on discrete event simulation, instead of learning how to use an existing network simulator. The main reason for this is that after looking into a couple of options it was apparent that the new architecture would be difficult to simulate in the options I found. I was also simply looking for a throughput result, and the simulators I found were significantly more advanced than what I wanted.

The simulator I made was based on a discrete event simulator I made a couple of years ago, and the first version used separate scripts for each device in the network. Packages in the network were simulated using JSON files and a directory structure representing the network structure. The first change was turning the scripts for a device into a class. After that, I changed packages from being JSON files into a class as well. Then I made a set of functions to connect the nodes into a network. The functions are based on the nodes having

a lookup table as a dictionary and an IP address structure. The IP addresses I use in this simulation have two parts for devices and one part for routers. The IP address of a device is the IP address of the router it is connected to followed by an identifier for that device.

Problem definition Can a new system architecture for IoT network devices utilizing fog-computing and self-awareness increase interoperability and efficiency in the network? By self-awareness, I specifically mean that the device has data about its own capabilities, as well as status and performance. In this master thesis, I have created a simple simulator to check if the project is viable for further work.

The rest of this thesis has the following parts; In section 2 I will introduce the simulator I created as part of this project. In section 3 I will show results, both those used to improve the simulator and those used to evaluate the alternative architecture suggested in my project report. In section 4 I will discuss the simulator and the results. In section 5 I describe potential future work on the architecture. Finally, in section 6 I conclude based on the results and the discussion section whether this project is worth further work and research.

2 The Simulator

In this section, I will present the simulation code used to simulate the reference network and the alternative network. The first version of the code was designed to run as several independent scripts, one for each node in the network. The first version also used JSON files in a folder structure as packages sent between the nodes. This was changed into a class and object structure with the simulation class designed to have one object per network to simulate. I created a user class for the simulated users. The user class is independent of the simulation class such that the same user traffic could be sent to more than one simulated network.

There are several things not included in the simulation. Nodes will not go off synchronization, nodes will not disconnect from the network, and packages will not time out, to mention a few. The alternative connections should also, depending on the communication technology used, have a different capacity compared to the LAN connections. On the other hand, I have not implemented a routing functionality that allows alternative connections to be used other than if the connection is specifically between the start node and the target node.

The simulation uses a variant similar to time division multiple access. In short, the simulation loops first through all devices and allow them to check received packages, process one package, and send one package if there is a package outgoing. After the simulation has looped through the devices it loops through routers in the same way.

The top level of the simulation is the simulation objects and user objects. The simulation objects handle the nodes and how packages are sent through the

network. The user objects generate task packages that it sends to the simulation. In addition to the simulation and user classes, there is a node class, a package class, and a set of network functions. Node objects simulate one node in the network, package objects represent packages that are sent through the network and the network functions are used to connect the nodes into a network.

2.1 Package class

The package class is the core of the simulation and is shown in listing 1. It is divided into a header and a payload. The header holds information about what node sent it, what node it is going to, the time it was created, and how many ticks of the simulation it has existed. The definition of the header class is shown in listing 2. The payload includes if it is a task package, or a ping package, and can also hold data. The definition of the payload class can be found in listing 3.

```
class package:
    def __init__(self, header, payload):
        self._header = header
        self._payload = payload

    def get_header(self):
        return self._header

    def get_payload(self):
        return self._payload

    def tick(self):
        self._header.tick()
```

Listing 1: Package class

```
class header:
    def __init__(self, sender, target, time):
        self._sender = sender
        self._target = target
        self._time = time
        self._ticks_alive = 0

    def get_time(self):
        return self._time

    def get_target(self):
        return self._target

    def get_sender(self):
        return self._sender

    def get_ticks(self):
        return self._ticks_alive

    def tick(self):
        self._ticks_alive += 1
```

Listing 2: Header class

```
class payload:
    def __init__(self, is_task, is_ping, data):
        self._is_task = is_task
        self._is_ping = is_ping
        self._data = data

    def get_is_task(self):
        return self._is_task

    def get_is_ping(self):
        return self._is_ping
```

Listing 3: Payload class

In addition to the class definitions of package, header, and payload there is a set of functions to create packages for different purposes. There is a function to create a ping, task, and data package. Ping packages were implemented to test the nodes, and are not used in the simulation. The function has sender and receiver as input, and returns the created ping package, as seen in listing 4. Task packages are sent from users to a node to simulate a request for data, the definition is seen in listing 5. The user selects random nodes as the sender node and the target node each time. When a node receives a task package it sends a data package back to the sender, the function to create a data package is shown in listing 6.

```
def ping(sender_ip_address, target_ip_address):
    t_header = header(
                    sender_ip_address,
                    target_ip_address,
                    get_now())
    t_payload = payload(False, True, {})
    return package(t_header, t_payload)
```

Listing 4: Ping function

```
def task(sender_ip_address, target_ip_address):
    t_header = header(
                    sender_ip_address,
                    target_ip_address,
                    get_now())
    t_payload = payload(True, False, {})
    return package(t_header, t_payload)
```

Listing 5: Task function

```
def data(sender_ip_address, target_ip_address, payload_data={}):
    t_header = header(
                    sender_ip_address,
                    target_ip_address,
                    get_now())
    t_payload = payload(False, False, payload_data)
    return package(t_header, t_payload)
```

Listing 6: Data function

2.2 Node class

Objects of the node class represent a device in the simulated network, either a router or a user device. There are two important parts of the node class. The first is the IP addresses and lookup table, and the second is package handling. The lookup table is implemented as a python dictionary with IP addresses as the keys and the values are references to the nodes that the packages should be sent to. The lookup method takes advantage of the IP address structure. If the specific IP address is not registered it will look for the IP address of its router. Through the network functions, all router IP addresses are registered for any node. The package handling is divided into three methods; check received, process package, and send package.

The IP addresses in the simulation are in essence the last part of actual IP addresses, used to identify a device in the network. The addresses have two parts; router and device. The lookup method checks if the address is registered in the lookup table, if not it attempts to look up the router IP, as seen in listing 7.

```
def lookup(self, ip_address):
    if "." in ip_address:
        try:
            return self._lookup_table[ip_address]
        except:
            last_dot_index = ip_address.rfind(".")
            return self.lookup(ip_address[:last_dot_index])
    else:
        try:
            return self._lookup_table[ip_address]
        except:
            return -1
```

Listing 7: Lookup method

Packages are placed in a package queue list when it is sent to the node. The first part of package handling is checking if any packages have been received, this is done by the check received method shown in listing 8. After this, if a package was found in the package queue, the current package is processed using the process package method found in listing 9. The process package method might put a package in the outgoing packages list, depending on the target of the processed package, and what type it is. If the package is not for the node it will be placed in outgoing packages directly. If the package is a data package it is moved to the simulations list of completed packages. If the package is a ping package a data package is created, and in the payload of the package the time used for the package to get to this node is included. If the package is a task package the node creates a set of data packages with the sender of the task package as the target. The number of packages to send is selected randomly between 1 and the maximum number of data packages, which is defined when the node object is created. When the process package method is completed the send package method is executed. The send package method selects one of the outgoing packages and places it in the package queue of the node found using

the lookup method, this is shown in listing 10.

```
def check_received(self):
    if len(self._package_queue) == 0:
        self._idle_cycles += 1
    else:
        # sort packages by time
        self._package_queue = sorted(self._package_queue,
            key=lambda p: p.get_header().get_time())
        self._current_package = self._package_queue.pop(0)
```

Listing 8: Check received method

```
def process_package(self):
    if self._current_package is not None:
        if self._current_package.get_header().get_target() == self.
    _ip_address:
            # package to me
            self._parent_simulation.add_to_sink(self.
    _current_package)
            if self._current_package.get_payload().get_is_task():
                # generate data package as response
                num_packages = randint(self._max_packages)
                for i in range(num_packages):
                    package = data(self._ip_address,
                        self._current_package.get_header().
    get_sender())
                    self._outgoing_packages.append(package)
            elif self._current_package.get_payload().get_is_ping():
                # ping, so send data package with time used to get
    here
                time_used = get_now() \
                - self._current_package.get_header().get_time()

                out_data = { "time_used": time_used }

                self._outgoing_packages.append(data(self.
    _ip_address,
                        self._current_package.get_header().
    get_sender(),
                        payload_data=out_data))
            else:
                self._data_packages_received += 1
                self.task_finished()
                # package of data sent to this device

        else:
            pass
            # package not to this node, forwarding
            # set outgoing_package to current_package
            self._outgoing_packages.append(self._current_package)
            # continue to send_package
    self._current_package = None
```

Listing 9: Process package method

```
def send_package(self):
    if len(self._outgoing_packages) > 0:
        outgoing_package = self._outgoing_packages.pop(0)

        target_ip_address = outgoing_package.get_header().
    get_target()
        target_node = self.lookup(target_ip_address)

        if target_node == -1:
            print("Node with ip: ", target_ip_address,
                    " not found in lookup table.")
        else:
            target_node.add_package(outgoing_package)
            self._packages_sent += 1
```

Listing 10: Send package method

2.3 Network functions

The network functions are used to connect nodes. The functions use the add lookup entry and remove lookup entry methods in the node class. The most basic functions are the; one way, indirect, and remove connection. One way takes two nodes, a and b, and adds node b with its IP address as an entry in the lookup table of node a. This represents a one-way connection from node a to node b. The one way function is shown in listing 11. The indirect function has node a, node b, and IP address as input. It adds an entry in the lookup table of node a with the IP address as key and node b as value. In other words, if node a tries to look up the given IP address, node b will be returned. The indirect function is shown in listing 12. Finally, there is a remove connection function. The remove connection function has a node and an IP address as input and removes the entry in the lookup table of the node with the IP address as the key. It is not used for the functions I use in the simulation but is added for completeness.

```
def one_way(node_a, node_b):
    if node_a.get_ip_address() == node_b.get_ip_address():
        return
    node_a.add_lookup_entry(node_b.get_ip_address(), node_b)
```

Listing 11: One way function

```
def indirect(node_a, node_b, ip_address):
    if node_a.get_ip_address() == ip_address:
        return
    node_a.add_lookup_entry(ip_address, node_b)
```

Listing 12: Indirect function

Building on the basic functions I have made several functions for connecting nodes; indirect list, both ways, one to all, one to all both ways, fully connected, circle connected and line connected. Indirect list has node a, node b, and IP addresses as input, it uses the indirect function to add a list of IP addresses to

the lookup table of node a, and all IP addresses in the list return node b. The both ways function connects two nodes both ways directly using the one way function. The one to all, and the one to all both ways functions have an origin node and a list of nodes as input. It connects the origin node to all the nodes in the list of nodes using the one way and both ways functions respectively. The fully connected function connects all combinations of nodes in a list of nodes both ways. The circle connected function connects a list of nodes in a circle one way. Each node has entries for all the IP addresses of the other nodes pointing to one other node in such a way that a package will travel in the same direction through the circle. The line connect function connects a list of nodes in a line, and all nodes have a lookup entry for the other nodes' IP addresses.

The functions I use to build the networks for the simulation are the router and nodes function and the routers and nodes alt function. These are built on the other functions defined in the network file. In short, the router nodes are connected in a line, the routers are connected to their device nodes and the device node has a lookup entry for each router's IP address pointing to its router. The router and nodes alt function uses the router and nodes function, and in addition, adds a number of both ways connection between device nodes directly. Both functions are explained in more detail below.

```
def routers_and_nodes(routers, list_of_list_of_nodes):
    # assumes nodes connected to a router has ip's as follows:
    # router ip i.e. "123"
    # nodes connected to that router has ip's "123.XXX"
    if not len(routers) == len(list_of_list_of_nodes):
        raise ValueError(
        "routers_and_nodes: length of routers list must be the same
     as length of list of list of nodes.")

    router_ips = [router.get_ip_address() for router in routers]

    line_connect(routers)

    for i in range(len(list_of_list_of_nodes)):
        router = routers[i]

        local_nodes = list_of_list_of_nodes[i].copy()
        for node in local_nodes:
            # router connected to its nodes
            one_way(router, node)
            # nodes map all router ips to its router
            indirect_list(node, router, router_ips)
```

Listing 13: Routers and nodes function

The routers and nodes function has routers and list of list of nodes as input. The function can be seen in listing 13. The list of list of nodes is a two-dimensional matrix, and the top-level dimension must be the same size as the length of the routers list. The content of the list of list of nodes is shown in table 2.3. Although not clear from the table, it is not a requirement for all routers to have the same number of device nodes.

The function starts by checking the dimension constraint described above.

Router 1	Router 2	...	Router n
Node 1.1	Node 2.1	...	Node n.1
Node 1.2	Node 2.2	...	Node n.2
...	...	...	...
Node 1.k	Node 2.k	...	Node n.k

Table 1: Structure of the list of list of nodes two dimensional matrix

After that, it connects the routers using the line connect function. Then for each node, it connects its router to the node. Finally, it uses the indirect list function to add lookup entries in the node for all router IPs to the nodes router.

```
def routers_and_nodes_alt(routers, list_of_list_of_nodes,
    other_connections):
    routers_and_nodes(routers, list_of_list_of_nodes)
    for i in range(other_connections):
        random_router = randint(len(routers))
        random_node = randint(len(list_of_list_of_nodes[
    random_router]))
        random_router2 = randint(len(routers))
        while random_router == random_router2:
            random_router2 = randint(len(routers))
        random_node2 = randint(len(list_of_list_of_nodes[
    random_router2]))
        node1 = list_of_list_of_nodes[random_router][random_node]
        node2 = list_of_list_of_nodes[random_router2][random_node2]
        both_ways(node1, node2)
```

Listing 14: Routers and nodes alt function

The routers and nodes alt function is shown in listing 14. In addition to the input for the routers and nodes function, it also takes in the number of other connections to create. First, it uses the routers and nodes functions to connect the nodes. After that, it repeatedly chooses two random nodes and uses the both ways function to connect them. The function ensures that the router is not the same for the nodes, however, it does not ensure that the same pair of nodes are chosen in a later iteration of the loop. This means that the number of alternative connections is actually the maximum number of alternative connections.

Because the routers and nodes alt function use the both ways function to create a connection between two device nodes it ensures that if a node has an alternative connection to another node a package is sent directly. The alternative connections are not used for routing as I have not made a routing protocol for the architecture, this is further discussed in the discussion section.

In table 2 I show what parameters are available for the simulator. These are used in the results section to describe the simulations.

Parameter	**Description**
Cycles	Number of cycles the simulation loops through before terminating
Routers	Number of router nodes in the simulated network
Devices per router	Number of devices connected to each router
Alternative connections	Number of alternative connections created (The simulator does not ensure that the same connection is not created several times)
Users	Number of users created
Waiting multiplier	Number multiplied with the exponentially distributed random number generated by each user to get number of cycles of the simulation to wait before next task is generated by that user
Task rate	Rate used by the users to generate the exponentially distributed random number for waiting time

Table 2: Simulator parameters

3 Results

I have looked at the number of packages processed as the result of a given simulation. The packages that are considered processed are those that have reached their target node and been processed with the process package method of the target node. The first section of results consists of results used to find and check for problems with the simulator. In the final section of results you will find the results, as they are with the newest version of the simulator.

3.1 Testing and problem discovery

The first round of simulations (5000 cycles): baseline of 30 routers, 10 devices per router, 10 percent of the total number of devices as the number of alternative connections, 50 percent of the total number of devices as the number of users. For each of the four variables, I also ran a simulation with one less to get an indication of how important that variable is compared to the others.

In order to discover potential mistakes in the simulator, I wrote down a set of assumptions. I was not sure of these, and could therefore not expect all of them to be true. However I would expect most of these to be true, and big deviations from these would indicate to me that I should check if a mistake had been made.

My assumptions before seeing the initial results are as follows;

1. I assume that each of the four variables are either beneficial or not for the alternative simulation. In other words, if it is beneficial for the alternative simulation to increase a specific variable from one to two, it will also be beneficial for the alternative simulation to increase the same variable further indefinitely.

2. Reducing the number of routers results in a larger number of short-distance transfers, and with that reduce the benefit of alternative connections between devices that are not close in the structured topology of these simulations.

3. Reducing the number of devices per router results in a larger average distance between devices, and that increases the benefit of alternative connections.

4. Reducing the number of alternative connections will reduce the difference between the reference simulation and the alternative simulation, the interesting part is how much compared to other changes.

5. Reducing the number of users should reduce queuing in the router devices, and therefore reduce the benefit of alternative connections.

The first set of results can be seen in table 3. It is important to note that these are results as part of discovering mistakes in the simulator, and several important mistakes were found after these results were generated by the simulator.

	Baseline	User	Router	Device per router	Alternative connection
Reference	13017	27367	13164	14161	42314
Alternative	13116	27564	13236	14111	13337

Table 3: First test results.

I realized that although reference and alternative simulations are comparable for one simulation run, they are not necessarily comparable across several simulations. Therefore as a reference, I added the total number of packages sent by users as a result of the simulations and ran them again. An alternative solution to fix this would be to run all the simulations at the same time, however, as it is only important that the comparison between reference and alternative simulation is comparable for this experiment I decided to only add the number of packages sent for now. The new results after adding packages sent can be seen in table 4. Several errors in how the simulator works are still present.

	Baseline	User	Router	Device per router	Alternative connection
Reference	12810	26998	13126	14117	40786
Alternative	12944	27246	13205	14311	13128
Packages sent	68250	42638	52952	56307	44027

Table 4: New results after adding packages sent to have a reference for the different runs.

Using base results there were around 13000 packages that got through the system of around 68000 packages sent. In contrast, the results when about 42500 packages were sent for the one less user test, there were around 27000 packages that got through. This I assumed was due to queuing in the network for both. To test if the number of packages sent was related to the results I tested using a different number of users. Based on the previous results showing improved throughput with reduced packages sent I decided to run tests with fewer users. I ran tests with 30, 60, 90, 120, and 150 users to see how the number of users affects throughput.

Number of Users	Reference	Alternative	Packages Sent
30	12223	12280	13168
60	25598	25944	24449
90	39183	39605	39860
120	52923	53345	53468
150	65976	66593	63432

Table 5: Testing results for different number of users

The results for testing a different number of users, seen in table 5, show that both the reference and alternative network simulation was able to get about the same number of packages through. Particularly for 150 users, the simulation did significantly better than in the initial tests, however, this was later found to be a mistake in the simulator, and still not real results. The same networks were used for all the tests, and as alternative connections are randomly assigned I next tested several simulations with the same parameters. I did a total of 100 simulations with the same parameters.

	Reference	Alternative	Both
Maximum	14870	15003	15003
Minimum	11404	11507	11404
Average	13967.44	14066.34	14016.89

Table 6: Maximum, minimum and average of the 100 simulations done with the same parameters.

In table 6 you can see that the maximums, minimums, and averages are not far from the base results. There are no outliers close to what was found for the one less user tests, the reference results for one less alternative connection (table 3 and 4) or the tests when changing number of users (table 5). After checking the scripts for the tests done it was clear that the outlier results happened for simulations that were run several times, and after further investigation, it was clear that the results were cumulative for all runs of the same instance of a simulation. All of the 100 simulation run results can be found in appendix A.

After discovering and fixing the mistakes described above I am now confident that the results presented in the next section are correct to the extent they are described. The results will only be presented and described shortly in the next section, and further discussed in the discussion section.

3.2 Proper results

In this section I show the results of the simulations. I started by running simulations with the values shown in table 7. Additional simulations were then done where number of users, routers devices per router, and alternative connections were reduced by one. The results, as the number of packages reaching and being processed by the target, are shown in table 8.

Parameter	Value
Cycles	5000
Routers	30
Devices per Router	10
Alternative Connections	30
Users	150
User wait multiplier	2000
User task rate	5

Table 7: Values for parameters for the initial test.

	Reference	Alternative
Base	13109	13342
One less user	11270	11360
One less router	13419	13556
One less device per router	14386	14500
One less alternative connection	13189	13248

Table 8: Results for initial simulations.

In these results, table 8, the alternative solution does better than the reference network for all cases. In addition, I have done some simulations where I have varied the number of users, routers, devices per router, and alternative connections. The results of these tests are shown in the graphs below. For these tests, the values of the parameters that are not changed during the simulations can be seen in table 9. These values are chosen to lower values than for the initial tests to reduce run-time.

Parameter	Value
Cycles	5000
Routers	5
Devices per Router	5
Alternative Connections	5
Users	5
User wait multiplier	1
User task rate	1

Table 9: Values for parameters when not varied during tests of varied values for one parameter at a time.

The first round of simulations was done for a different number of users. I chose 1, 5, 50, and 100 users as the values to test. Each of these users will on average have a waiting time of one cycle between each task generated. I chose these values to have a wide range of values while keeping the run-time of the simulation reasonable. The results for the reference network is shown in figure 2, and for the alternative network in figure 2.

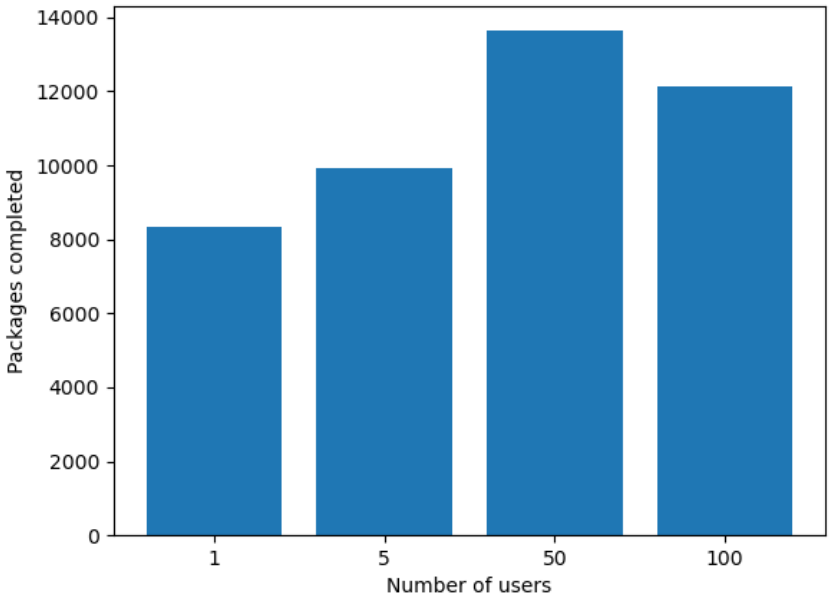

Figure 1: Results for the reference network with a varying number of users.

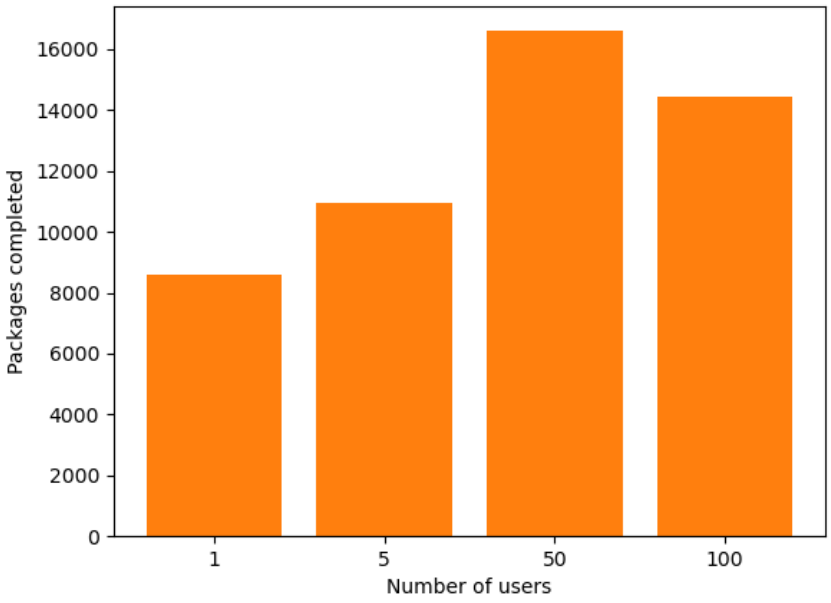

Figure 2: Results for the alternative network with varying number of users.

The second round of simulations had a varying number of routers. I chose 5, 50, and 500 as a broad variation in the number of routers. The results for the reference network are shown in figure 3. The results for the alternative network can be seen in figure 4.

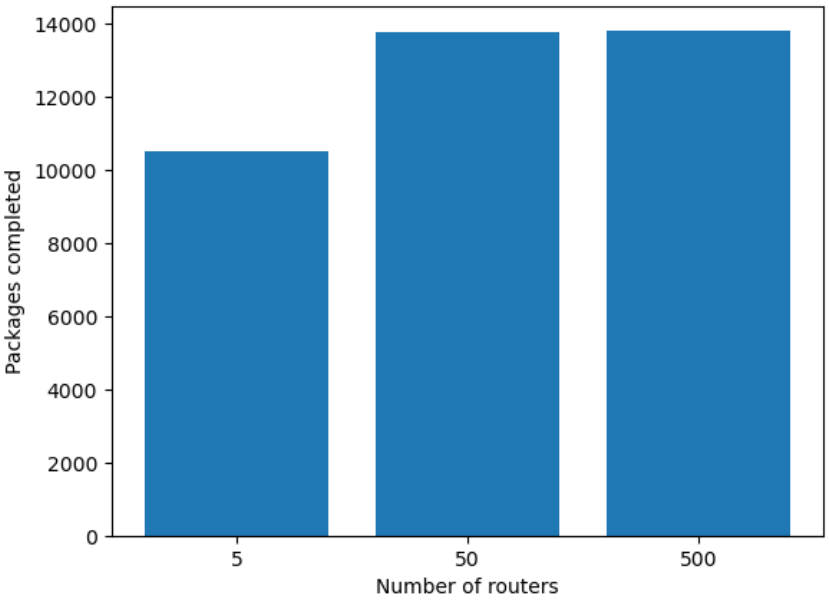

Figure 3: Results for the reference network with a varying number of routers.

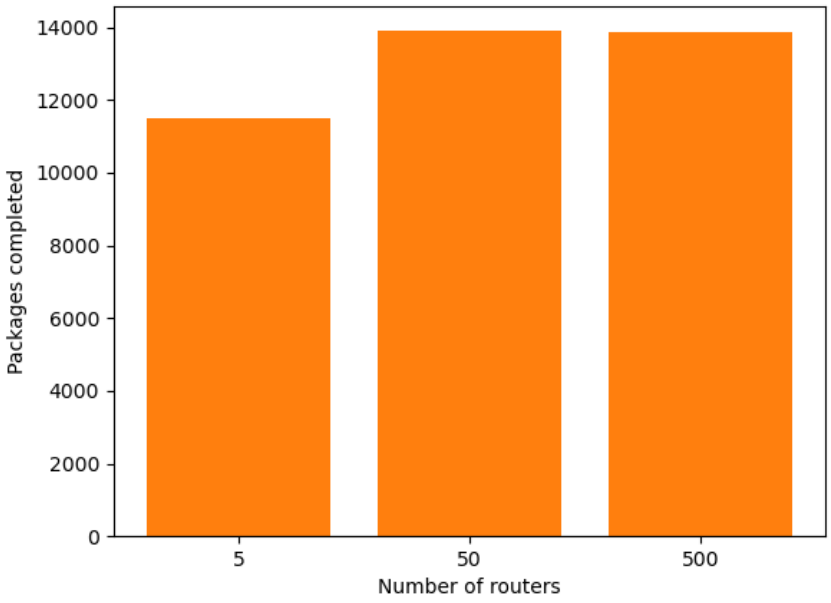

Figure 4: Results for the alternative network with a varying number of routers.

The third round of simulations included a variation in the number of devices per router. Figure 5 show the results for the reference network, and figure 6 show the results for the alternative network. It is worth noting that in the figure for the alternative network there is a blue line on top of the bar for 500 users. This shows that in this case, the reference network got better results than the alternative network.

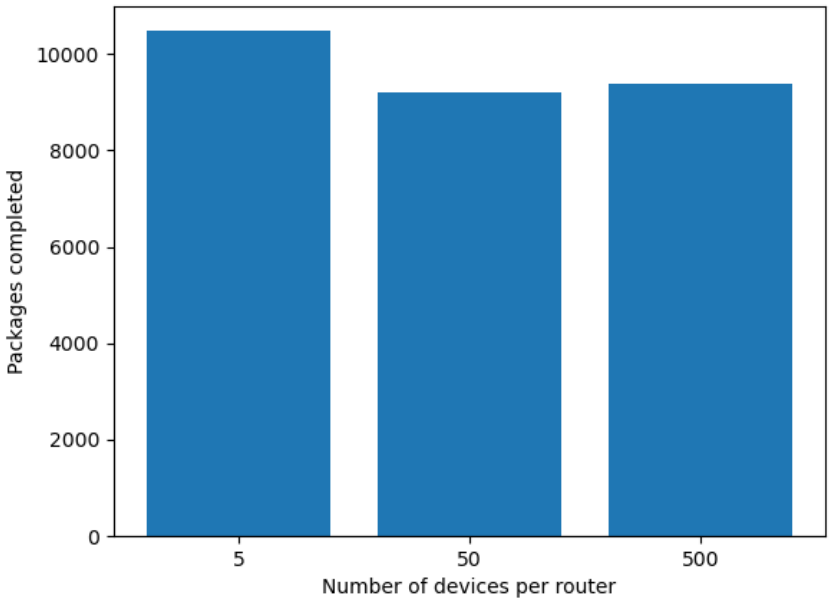

Figure 5: Results for the reference network with a varying number of devices per router.

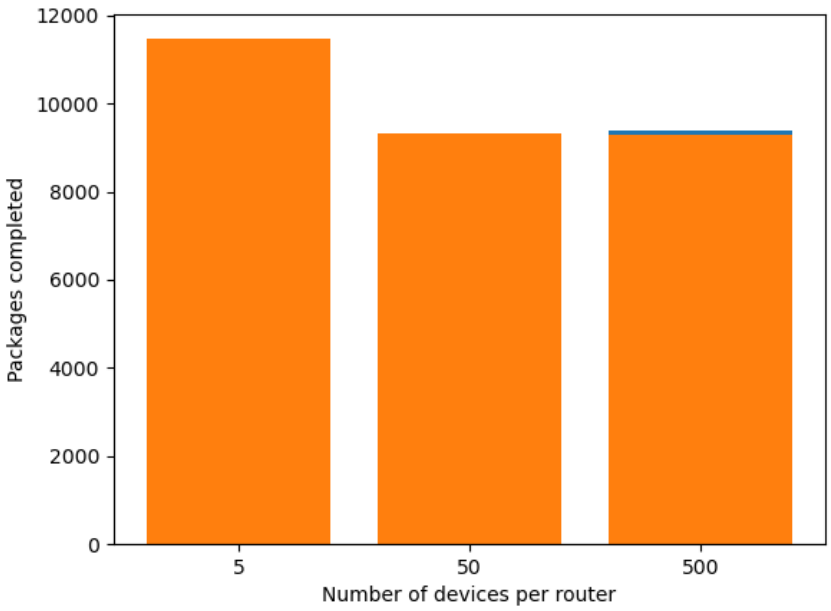

Figure 6: Results for the alternative network with a varying number of devices per routers.

The final round of simulations had a varying number of alternative connections. The reference network results can be used to compare with the results of the alternative network in this case. The reference network results are shown in figure 7. The results for the alternative network can be seen in figure 8.

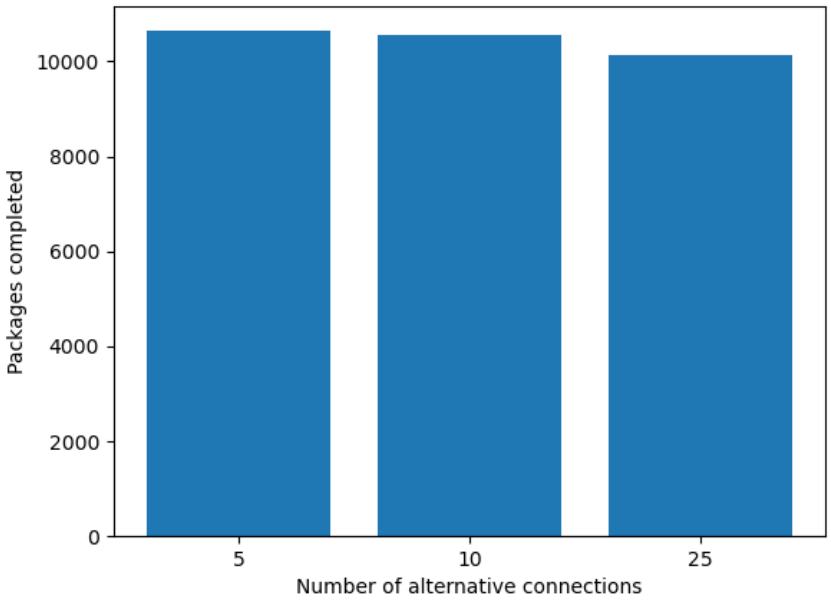

Figure 7: Results for the reference network with a varying number of alternative connections.

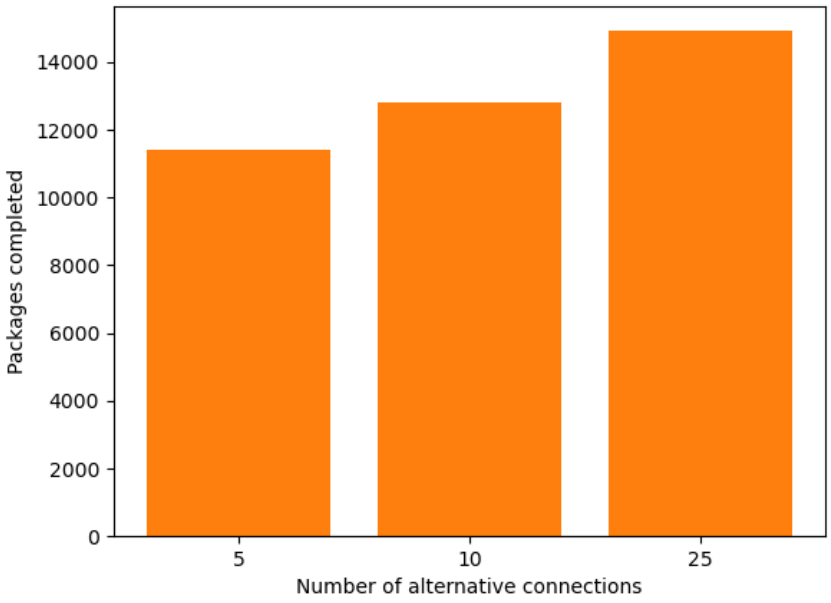

Figure 8: Results for the alternative network with a varying number of alternative connections.

4 Discussion

In this section I will discuss the functionality of the simulator I created, and the results presented in the results section. The simulator discussion includes how the functionality that is included compares to a real-world version and functionality that is not included in the simulator. The result discussions focus on the implications of the results, particularly whether the results are promising for further work on the suggested architecture.

4.1 The simulator

The users in the simulator have a randomly generated waiting time in number of cycles between generating tasks for the networks. In statistics, it is known that a Poisson process can be used for random discrete events and that when using a Poisson process the time between the events is exponentially distributed [4]. This is the reason for using an exponential distribution for the waiting time between tasks generated by the users.

The simulator cycles through all nodes of the network one by one, allowing them to send one package for each cycle. This is similar to Time Division Multiple Access (TDMA), where the devices of the network have an allocated time to send data over the network. IEEE 802.11ax, colloquially known as wifi-6, uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CS), where a node ready for transmission starts a countdown until transmission, and the counter stops while it detects traffic on the network. When the countdown is completed transmission starts [2]. TDMA is used in 2G schemes like GSM [3]. This is one of the ways the simulator I created is not realistic. The consequences of using TDMA on the results should be further investigated.

The routing scheme used in both the reference networks and the alternative networks is based on a lookup table. The lookup tables are generated with the two-level hierarchy of routers and devices as a basis, and the alternative connections are added between two devices. This results in alternative connections only being used if a package is sent between two devices that have an alternative connection. The alternative connections are not used unless the connection is specifically between the start and target nodes. This will clearly reduce the benefit of the alternative connections on the total flow through the network. However as the results are interpreted qualitatively in this thesis, and they suggest that the alternative architecture has a beneficial effect on flow, this will only have to be taken into account for quantitative tests in the future.

When the alternative connections are created in the simulator they are not ensured to not already exist. This means that when a simulation has set the number of alternative connections to i.e. 5, this is the maximum number of actual alternative connections created. This should be fixed before results are used quantitatively.

The simulator does not include timing out packages. It also does not include nodes getting disconnected. The effects of including this have not been investigated. However, as suggested in the project report the use of alternative connections in combination with routing could improve the network flow by circumventing disconnected nodes. The effects of adding timeouts could initially be investigated by looking at how many cycles the packages are in the network.

The alternative connections can send the same packages as the core network packages. There are however reasons to suggest that the alternative connections should have a reduced capacity. For instance, the data rate of Bluetooth is 1 Mbps [7], and 802.11g has a data rate of 54 Mbps [8].

4.2 The results

The initial test results shown in table 8, show that the alternative network perform better in all those simulations. The parameter values are here arbitrarily chosen. These simulations were done to get an indication of what happens to the results when one of the four central parameters of the simulation was changed. The most significant results in the initial test were that the results of both the reference and the alternative network were reduced when the number of users was reduced, and both networks did better when the number of devices per router was reduced.

Following the initial simulations, I ran simulations with a varying number of users, routers, devices per router, and alternative connections. The results of these tests are shown in figures 2 - 8. For the number of users test, both the reference network and the alternative network got better results for 50 users. The alternative network got better results than the reference network for 5, 50, and 500 users, and similar results for 1 user. These results also show that the first assumption I made in section 3.1 is clearly wrong. There was an increase in performance for both networks up to 50, and then a drop in results for 100 users. The assumption that reducing the number of users would also reduce the benefit of alternative connections appears correct, the alternative network did significantly better compared to the reference network when having more users.

The results for a different number of routers are very similar for 50 and 500 routers. The alternative network is significantly better in the simulation with 5 routers. These results are opposite to the assumption I made in section 3.1, where I suggested that reducing the number of routers would reduce the benefit of alternative connections. The results show that the alternative network did better for a low number of routers, and very similarly for 50 and 500 routers.

The alternative network did significantly better with 5 devices per router, and similarly for 50 and 500 devices per router. The results indicate that my assumption for the number of devices per router was correct. The assumption was

that when the number of devices per router is low, the benefit of the alternative network was higher. The results for the alternative network were significantly better for 5 devices per router and similar for 50 and 500 devices per router.

The alternative network got significantly better results with 10 and 25 alternative connections. The benefit per connection appears to be reduced as connections are added, however, this can also be related to the simulator not checking if the connection already exists.

With the exception of the simulation with 500 routers, seen in figure 4, the alternative network perform as well or better than the reference network. This indicates that the suggested architecture is worth further investigation.

5 Future work

In this section I describe potential future work on the architecture I suggested in the project report and simulated as a part of this master thesis. The first part is dedicated to potential implementations of the architecture as it is now. The second part describes suggestions for extending the architecture.

5.1 Implementation

During my attempt to implement a part of the architecture I found that using either the Arduino programming language, micropython or C are the most viable choices for programming language. Arduino is based on C++, and designed for use with microcontrollers [1]. Micropython is specifically designed to be used for low-level programming [5]. C has been used for low-level programming for many years. The choice of specific language is up to the developer but should be informed by research on what language is used in the targeted IoT device. The architecture describes methods that are externally savable and possible to load during run-time. This suggests object files in the case of using C. These can be chosen, loaded, and executed by the main program loop of the device.

Conversation templates is a quite complex structure with each message having several possible answers. A solution to implement this in a low-level language is to aggregate the conversation structure from message templates, and for each of them define possible answers. The possible answers must as a minimum be identifiable for the device, for instance using a regular expression or formatted string solution.

Message templates should be a format string that can be filled with the relevant data and have defined possible answers as pointers to other message templates. This also requires an extract data function for the message. This could either be done using a header and payload solution where the header includes information about how to extract data from the payload, or other solutions that allow extractions of data from a message.

In most cases, the use of different communication technologies will require using libraries to extend the functionality of the base programming language. The methods must either include this code as part of the method or have a reference to the library to allow the required library to be loaded when needed. The main program loop must also abstract the use of the specific methods in a way that allow different communication technologies to be used. One solution to work around this is that all methods define a wide range of functions i.e. scanning, connecting, starting transmission, and ending transmission. The specific method would then either define the function or simply pass control on to the next function in the list. There would have to be some form of error handling to ensure that the errors reach the main program.

A decision that must be made during implementation is whether it should

replace an operating system or be dependent on an operating system. The two choices both have advantages and disadvantages. The choice will likely be dependent on the target devices.

5.1.1 Pseudo-code

Below I show some pseudo code written for the different roles in the architecture. This suggests what functions should be included for the different roles, and how they might work. First is a suggestion for class definitions of the classes; method (listing 15), conversation (listing 16), Message (listing 17) and Communication Technology (listing 18). These are here defined as classes, but could also be templates or similar, depending on how they are implemented.

5.1.2 Classes

- method
- conversation templates/objects
- message templates/objects
- communication technology

```
class Method:
    def __init__(self, communication_technology,
    conversation_template):
        self.id = auto_id()
        self.communication_technology = communication_technology
        self.conversation_template = conversation_template

    def checksum(self):
```

Listing 15: Method Class/Template

```
#need some relational graph of conversation structure
class Conversation:
    def __init__(self, messages):
        self.id = auto_id()
        self.messages = messages
```

Listing 16: Conversation Class/Template

```
class Message:
    def __init__(self, formated_string):
        self.id = auto_id()
        self.formated_string = formated_string

    def SendMessage(self, data_to_send):
        return self.formated_string, with data_to_send filled in
```

Listing 17: Message Class/Template

```
class CommunicationTechnology:
    def __init__(self, proper_id, human_identifier):
        self.id = proper_id
        self.human_identifier = human_identifier
```

Listing 18: Communication Technology Class/Template

5.1.3 Method User

The method users in the architecture will require a set of functions. They must be able to start a conversation, initialize methods, and decode conversations and messages. These four functions have pseudo code suggestions below, each with further descriptions.

Following is pseudo-code for the conversation starter function 19. It needs a list of methods available, a map between the target device and method used, the target device, the type of data to send and the data to be sent. The function will first check if there is a method that has been used to communicate with the target device before, and send the same type of data. If that exists it will try to use this method again. If a method was not found or the previously used method failed it will send available methods to the target device using the already established network. If a reply is received with a chosen method it will attempt to send the data to the target device using this method.

```
#Conversation starter function
def ConversationStarter(my_methods, methods_used, target_device,
    data_type, data_to_send):
    if exists methods_used[target_device, data_type]:
        try:
            ExecuteMethod(method_used[target_device, data_type],
    target_device, data_to_send)
        except:
            delete methods_used[target_device, data_type]
            ConversationStarter(my_methods,
                                methods_used,
                                target_device,
                                data_type,
                                data_to_send)

    send my_methods to target_device
    wait for reply
    if reply not none:
        try:
            ExecuteMethod(methods_used[target_device],
    target_device, data_to_send)
        except:
            reply = none

    if reply not none:
        methods_used[target_device, data_type] = reply
```

Listing 19: Conversation starter pseudo code

The execute method is used to send data from a device to a target, shown in listing 20. The function takes a method, the target, and the data to send as

input. The function first extracts communication technology and conversation template from the method. It then creates a conversation object using the decode conversation template function on the conversation template. After this, it will attempt to connect to the target using the communication technology and use the conversation object to send the first message on the connection. If it fails the connection will be broken and an error is thrown.

```
#Execute Method function
def ExecuteMethod(method, target, data_to_send):
    communication_technology = extract communication_technology
    from method
    conversation_template = extract conversation_template from
    method
    conversation_object = DecodeConversationTemplate(
    conversation_template)

    try:
        connection = connect to target with
    communication_technology
        execute conversation_object on connection (uses
    DecodeMessageTemplate)
    except:
        end connection if open
        throw error "connection failed"
```

Listing 20: Execute Method pseudo code

The decode conversation template function (listing 21) is used in the execute method function. The function simply uses a conversation template to create a conversation instance. The conversation instance can then be used to create messages with data, and extract data from answers.

```
#Decode Conversation Template function
def DecodeConversationTemplate(conversation_template):
    create conversation_object from conversation_template
```

Listing 21: Decode Conversation Template pseudo code

The decode message template function, seen in listing 22, is used to create a message instance from a message template. It takes a message template and data to send as input. The function creates a message that can be sent to another device.

```
#Decode Message Template function
def DecodeMessageTemplate(message_template, data_to_send)
    create message_object from message_template
```

Listing 22: Decode Message Template pseudo code

5.1.4 Registry Mediator

In addition to the functions described for the method user role, the registry mediators must have some additional functions. There are three main capabilities the registry mediators must have; check for updates, get a method from the registry and verify a method. Each of these is further described and has pseudo code below.

The check for updates function, shown in listing 23, is used to check if the registry has a new version of a method in a list of methods. The function takes a list of methods as input. The registry mediator will try to connect to the method registry and send the list of methods. If it gets a response this will be a list of the methods that have updated versions available. The check for updates function will then request these methods from the method registry.

```
#Check For Updates function
def CheckForUpdates(my_methods)
    init list_methods_with_updates_available
    try:
        connect to Method Registry (or use POST or whatever
    depending on technology)
        send my_methods
        wait for response = list_methods_with_updates_available
    except:
        log not working, try again later???
        throw error?

    for method_id in list_methods_with_updates_available:
        GetMethod(method_id)
```

Listing 23: Check For Updates pseudo code

The get method function, shown in listing 24, is used in the check for updates function. It takes a method id as input. The function will attempt to connect to the method registry and request the method with the given id.

```
#Get Method function
def GetMethod(method_id):
    try:
        connect to Method Registry
        get method with id = method_id
        if OK:
            add new method to my_methods
            delete old method
    except:
        error
```

Listing 24: Get Method pseudo code

The verify method, shown in listing 25, is used to verify a method. The use of this is for devices to ensure that another device has not had the method changed. The function takes a method id and a checksum generated by the device's version of the method. This function will then send both to the method registry, and the registry will answer whether the checksum is correct or not.

```
#Verify Method function
def VerifyMethod(method_id, check_sum)
    try:
        connect to Method Registry
        send method_id and check_sum
        if response OK:
            return True
        elif response NOT OK:
            return False:
    except:
        return "-1"/error
```

Listing 25: Verify Method pseudo code

5.2 Extensions

In this section I describe some suggestions to extend the existing architecture. In particular, I have looked at options for routing. I have two suggestions for how routing can be done where alternative connections are utilized. The new architecture does not include a routing scheme for utilizing alternative connections. I have come up with two general suggestions for routing schemes that could be used in the future. The first one is based on the A*-algorithm used for finding the shortest path in a graph. The other is based on the devices having knowledge of the topology of the core network. The two suggestions are further described below.

The A*-algorithm uses a combination of the cost already made to get to a node in the network, with a heuristic function to estimate the cost of getting from the current node to the goal [6]. The costs would in this case be the time used, and my suggestion is to use historical data saved in the devices for different connections as the heuristic of cost from the node to the goal. This approach would either require sending several copies of the initial packages through different paths, until the shortest path is found, or the shortest paths found during setup and/or low load times for the network. Other heuristic functions could also be used and might improve this approach further.

If the devices have knowledge of their physical position, as well as the physical position of the other nodes in the network. In this case, the package could be sent to the connected node that is closest to the goal node. There is however an issue with this approach illustrated in figure 9. The larger shape illustrates a building, the circles are nodes in the network and the red line is a wall disrupting connections. In this scenario the node marked S is the start node, and it is sending a package to the goal node marked G. Although if the start node will not directly try to send the package to the goal node, when it sends the package to the node marked N1 then that will send it back to the start node. A way of attempting to solve this is finding the entire path, but that will be more similar to the A*-algorithm suggestion using physical location as the heuristic function.

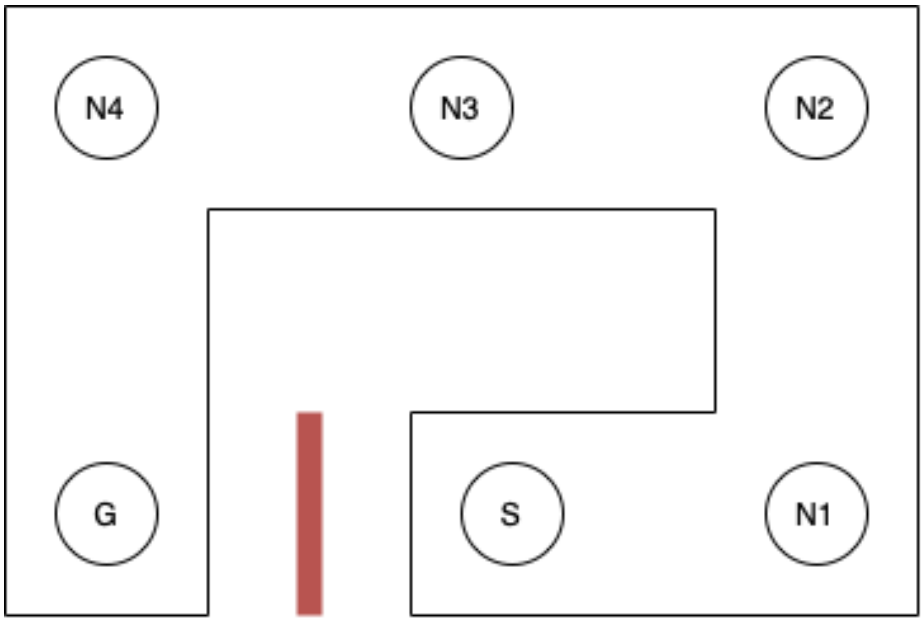

Figure 9: Illustration of closest physical node issue.

These are only two suggestions for routing schemes that could be used to utilize the alternative connections in the architecture. There could be several other existing routing schemes that could also be used for the architecture. In order to simulate the architecture more accurately, a routing scheme should be used. This is further discussed in the discussion section.

6 Conclusion

In this section I will present conclusions based on the results and the discussion sections. I will first present conclusions for some of the partial issues discussed. Finally, I will make a conclusion on whether the project as a whole is worth further research and work.

The most significant improvement in results for the alternative network compared to the reference network was found for a higher number of users. Although more testing should be done, this suggests that the alternative network is more capable of handling high data loads. This is consistent with the benefit of efficiency described in the project report.

The results for the alternative network compared to the reference network was either better or similar in all the simulations done as part of this thesis. This is a clear indication that there should be done further work and research on the suggested architecture.

Acronyms

CSMA/CS Carrier Sense Multiple Access with Collision Avoidance 27

IoT Internet of Things 5, 6

LAN Local Area Network 5, 6

TDMA Time Division Multiple Access 27

References

[1] Y. A. Badamasi. *The Working Principle Of An Arduino*. 2014. ISBN: 9781479941063. DOI: 10.1109/ICECCO.2014.6997578.

[2] B. Bellalta. "IEEE 802.11ax: High-efficiency WLANS; IEEE 802.11ax: High-efficiency WLANS". In: *IEEE Wireless Communications* 23 (2016). DOI: 10.1109/MWC.2016.7422404.

[3] G. Gu and G. Peng. *The Survey of GSM Wireless Communication System*. 2010. ISBN: 9781424485987. DOI: 10.1109/ICCIA.2010.6141552.

[4] M. Holický. "Selected Models of Discrete Variables". In: *Introduction to Probability and Statistics for Engineers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 53–62. ISBN: 978-3-642-38300-7. DOI: 10.1007/978-3-642-38300-7{_}5. URL: https://doi.org/10.1007/978-3-642-38300-7_5.

[5] *MicroPython - Python for microcontrollers*. URL: https://micropython.org/.

[6] S. J. 1. Russell and P. 1. Norvig. *Artificial intelligence a modern approach*. Third. Pearson, 2010. ISBN: 978-1292153964.

[7] S. Al-Sarawi et al. *Internet of Things (IoT) Communication Protocols : Review National Advanced IPv6 Centre (NAV6)*. 2017. ISBN: 9781509063321.

[8] D. Vassis et al. *The IEEE 802.11g standard for high data rate WLANs; The IEEE 802.11g standard for high data rate WLANs*. Tech. rep. 2005. DOI: 10.1109/MNET.2005.1453395.

A Results 100 simulations with same parameters

Simulation	Reference	Alternative	Packages Sent
1	13479	13567	63584
2	12425	12545	45283
3	13101	13072	56567
4	14072	14322	55503
5	13157	13246	44506
6	13912	14165	52089
7	14032	13962	49123
8	13922	13986	46092
9	13407	13547	50395
10	13742	13859	54074
11	13408	13368	47869
12	14038	14029	52508
13	14558	14671	48117
14	14257	14414	49751
15	14331	14425	53883
16	14338	14473	48071
17	13926	13986	45360
18	14228	14312	50246
19	14224	14369	48955
20	14114	14232	48536
21	13416	13422	65602
22	11404	11507	41715
23	13208	13249	58331
24	14030	13965	52846
25	12736	12859	50573
26	13969	14012	50223
27	14405	14554	59713
28	13727	13865	51276
29	13761	13915	50420
30	13771	13867	52838

Simulation	Reference	Alternative	Packages Sent
31	14142	14198	48881
32	14107	14273	52596
33	14456	14417	57728
34	13862	13918	53596
35	14076	14215	51500
36	13470	13532	47789
37	14528	14559	47013
38	13834	13907	49856
39	14407	14441	54694
40	14293	14342	51998
41	14144	14334	46969
42	14220	14240	52627
43	14417	14534	49641
44	13920	14215	45004
45	14141	14255	52137
46	13548	13798	48426
47	14053	14164	57220
48	14273	14309	53056
49	14053	14094	49163
50	13985	14022	48407
51	14022	14227	52409
52	13354	13425	51137
53	14429	14527	56385
54	14870	15003	53226
55	14136	14294	54327
56	14313	14493	54039
57	13594	13650	53820
58	13992	14030	50061
59	14395	14460	48613
60	14252	14294	50222
61	13801	13847	49366
62	13994	14061	53932
63	13822	13834	54491
64	13721	13847	52139
65	14149	14482	52909
66	14335	14337	50419
67	13875	13981	50651
68	14535	14663	51134
69	14120	14176	50980
70	14254	14370	46411

Simulation	Reference	Alternative	Packages Sent
71	13832	13992	51269
72	13626	13844	53849
73	13882	13981	52928
74	14334	14459	51629
75	14162	14254	49722
76	13905	14135	51618
77	14546	14836	54488
78	14082	14044	47356
79	14517	14682	54100
80	14427	14531	49553
81	14446	14630	51007
82	13933	14029	49779
83	13883	13946	52943
84	14549	14676	51536
85	13922	14033	44565
86	14318	14370	47691
87	14221	14294	49126
88	13580	13675	55954
89	14482	14494	56940
90	14211	14326	54749
91	14080	14005	48668
92	14195	14386	51094
93	13640	13726	59184
94	13712	13750	56138
95	13826	13993	50775
96	13758	13803	50409
97	13854	13954	50272
98	13685	13853	50549
99	13475	13595	49446
100	14671	14805	53479

B Simulator code

B.1 Simulation.py

```
from imports.node import node
from imports.user import user
from imports.network import routers_and_nodes, routers_and_nodes_alt
from imports.network import network_results
from imports.package import package_results
from imports.datetime import get_now
from numpy import sum
from numpy.random import randint

"""
File imports/simulation.py
class simulation
options:
is_alt: boolean, wether alternative connections should be added.
alt_connections: int, number of alternative connections to make (if is_alt=true)
number_of_routers: int, number of router nodes to make
number_of_nodes_per_router: int, number of nodes to make per router
max_send_data_packages: int, maximum number of data packages generated per task

description:
The simulation class handles one simulation of a network of nodes. It has a set
of routers and devices. The network created has the routers connected in a line,
and the nodes are connected to all other nodes for the same router as well as
the router. For nodes not connected to the same router it will send it to its
router.

Methods:
Tick:
All nodes get a tick call
All routers get a tick call
All nodes call check_received, process_package and send_package
All routers call check_received, process_package and send_package


Function simulation_results:
Returns a string with results from the routers, nodes and packages that got to
the package_sink. The package sink is where data packages are placed when
recieved by the goal node.
"""

class simulation:
    def __init__(self, name="Simulation",
```

```
        is_alt = False, alt_connections = 0,
        number_of_routers = 1, number_of_nodes_per_router = 1,
        max_send_data_packages = 5):
        self._name = name
        self._number_of_routers = number_of_routers
        self._number_of_nodes_per_router = number_of_nodes_per_router
        self._max_send_data_packages = max_send_data_packages

        self._routers = [node(str(i), self, pretty_name="Router " + str(i))
                    for i in range(number_of_routers)]

        self._devices = [[
            # node
            node(str(i)+"."+str(j), self,
                    max_send_data_packages=max_send_data_packages,
                    pretty_name="Device " + str(i)+"."+str(j))
                # inner list of nodes (one list per router)
                for j in range(number_of_nodes_per_router)]
                    # outer list of list of nodes
                    for i in range(number_of_routers)]

        if is_alt:
            routers_and_nodes_alt(self._routers, self._devices, alt_connections)
        else:
            routers_and_nodes(self._routers, self._devices)

        self._devices = [
            node for list_of_nodes in self._devices for node in list_of_nodes]

        self._package_sink = []

    def __str__(self):
        return self._name

    def add_to_sink(self, package):
        self._package_sink.append(package)

    def get_sink_packages(self):
        return self._package_sink

    def tick(self):
        for device in self._devices:
            device.tick()

        for router in self._routers:
            router.tick()
```

```
            for device in self._devices:
                device.check_received()
                device.process_package()
                device.send_package()

            for router in self._routers:
                router.check_received()
                router.process_package()
                router.send_package()

        def get_devices(self):
            return self._devices

        def get_routers(self):
            return self._routers

        def get_device_by_ip(self, ip):
            for device in self._devices:
                if device.get_ip_address() == ip:
                    return device
            return -1


    def run_sim(users, ref_simulation, alt_simulation, simulation_cycles):
        simulations = [ref_simulation, alt_simulation]
        end_of_simulation = False
        packages_sent_before = sum([user.get_packages_sent() for user in users])
        while not end_of_simulation:
            for user in users:
                if user.tick():
                    random_start_node = randint(len(simulations[0].get_devices()))
                    random_goal_node = randint(len(simulations[0].get_devices()))

                    start_node_sim1 = simulations[0].get_devices()[random_start_node]
                    goal_node_sim1 = simulations[0].get_devices()[random_goal_node]
                    user.create_package(start_node_sim1, goal_node_sim1)

                    for i in range(1, len(simulations)):
                        sim = simulations[i]
                        start_node = sim.get_device_by_ip(
                                            start_node_sim1.get_ip_address())
                        goal_node = sim.get_device_by_ip(
                                            goal_node_sim1.get_ip_address())
                        user.create_package(start_node, goal_node)
```

```
                user.package_sent()

                user.set_new_task_time()

            for sim in simulations:
                sim.tick()

            simulation_cycles -= 1
            if simulation_cycles <= 0:
                end_of_simulation = True

        packages_sent_after = sum([user.get_packages_sent() for user in users])
        packages_sent = packages_sent_after - packages_sent_before

        return len(ref_simulation.get_sink_packages()),\
                len(alt_simulation.get_sink_packages()), \
                packages_sent


def simulation_results(sim, sim_name):
    result_string = "############################################################\n"
    result_string += "Results for simulation "
    result_string += sim_name + "\n"
    result_string += "############################################################"
    result_string += network_results(sim.get_routers())
    result_string += network_results(sim.get_devices())
    result_string += package_results(sim.get_sink_packages())
    result_string += "------------------------------------------------------------"
    result_string += ""
    return result_string
```

B.2 Package.py

```
from imports.datetime import get_now
from numpy import sum

class header:
    def __init__(self, sender, target, time):
        self._sender = sender
        self._target = target
        self._time = time
        self._ticks_alive = 0

    def get_time(self):
        return self._time

    def get_target(self):
        return self._target

    def get_sender(self):
        return self._sender

    def get_ticks(self):
        return self._ticks_alive

    def tick(self):
        self._ticks_alive += 1


class payload:
    def __init__(self, is_task, is_ping, data):
        self._is_task = is_task
        self._is_ping = is_ping
        self._data = data

    def get_is_task(self):
        return self._is_task

    def get_is_ping(self):
        return self._is_ping


class package:
    def __init__(self, header, payload):
        self._header = header
        self._payload = payload
```

```
    def get_header(self):
        return self._header

    def get_payload(self):
        return self._payload

    def tick(self):
        self._header.tick()


def ping(sender_ip_address, target_ip_address):
    t_header = header(
                    sender_ip_address,
                    target_ip_address,
                    get_now())
    t_payload = payload(False, True, {})
    return package(t_header, t_payload)

def task(sender_ip_address, target_ip_address):
    t_header = header(
                    sender_ip_address,
                    target_ip_address,
                    get_now())
    t_payload = payload(True, False, {})
    return package(t_header, t_payload)

def data(sender_ip_address, target_ip_address, payload_data={}):
    t_header = header(
                    sender_ip_address,
                    target_ip_address,
                    get_now())
    t_payload = payload(False, False, payload_data)
    return package(t_header, t_payload)

def package_results(packages):
    if len(packages) == 0:
        return "No packages in list"
    result_string = "### Package Results ###\n"
    num_packages = len(packages)
    result_string += "Number of packages = " + str(num_packages) + "\n"
    package_ticks = [package.get_header().get_ticks()
                        for package in packages]
    avg_ticks = sum(package_ticks) / num_packages
    result_string += "Average ticks alive = " + str(avg_ticks) + "\n"
    return result_string
```

B.3 User.py

```
from imports.datetime import get_now
from numpy.random import exponential
from math import floor
from imports.package import task, ping, data

class user:
    def __init__(self, wait_multiplier=1, task_rate=1, pretty_name=""):
        self._pretty_name = pretty_name
        self._wait_multiplier = wait_multiplier
        self._task_rate = task_rate
        self._next_task = 0
        self._packages_sent = 0
        # self._packages_recieved = 0

    def __str__(self):
        if self._pretty_name == "":
            return "Unnamed User"
        else:
            return "User: " + self._pretty_name

    # def add_package(self, package):
    #     self._packages_recieved += 1 # sink for packages

    def set_new_task_time(self):
        self._next_task = self._wait_multiplier *\
                          floor(exponential(1/self._task_rate))

    def tick(self):
        self._next_task -= 1
        return self._next_task <= 0

    def package_sent(self):
        self._packages_sent += 1

    def get_packages_sent(self):
        return self._packages_sent

    def create_package(self, start, target):
        # Setting sender ip address to start node, simulating that as the
        # device the user sends the request from
        outgoing_package = task(start.get_ip_address(), target.get_ip_address())
        start.task_started()
        start.add_package(outgoing_package)
```

```
    def write_results(self):
        result_string = "### Results for " + str(self) + "###\n"
        result_string = "Packages sent: " + str(self._packages_sent) + "\n"
        return result_string
        #print("Packages recieved: ", self._packages_recieved)

def user_results(users):
    result_string = "### User Results ###\n"
    for user in users:
        result_string += user.write_results()
    return result_string
```

B.4 Node.py

```
from imports.datetime import get_now
from imports.package import data
from numpy.random import randint


class node:
    def __init__(self, ip_address, parent_simulation,
                 max_send_data_packages=1, is_test=False, pretty_name=""):
        self._pretty_name = pretty_name

        self._parent_simulation = parent_simulation

        self._ip_address = ip_address
        self._max_packages = max_send_data_packages

        self._lookup_table = {}
        self._package_queue = []
        self._current_package = None
        self._outgoing_packages = []

        self._idle_cycles = 0
        self._packages_sent = 0
        self._data_packages_received = 0

        self._tasks_started = 0
        self._tasks_finished = 0

        if is_test:
            pass
            # file = open(join(unique_string, "test.json"), "w")
            # test_package = ping(self, self)
            # file.write(create_json(test_package))
            # file.close()

    def task_started(self):
        self._tasks_started += 1

    def task_finished(self):
        self._tasks_finished += 1

    def __str__(self):
        if self._pretty_name == "":
            return "Unnamed Node"
        else:
```

```
        return "Node: " + self._pretty_name

    def get_ip_address(self):
        return self._ip_address

    def add_package(self, package):
        self._package_queue.append(package)

    def remove_lookup_entry(self, ip_address):
        self._lookup_table.pop(ip_address)

    def add_lookup_entry(self, ip_address, node):
        self._lookup_table[ip_address] = node

    def lookup(self, ip_address):
        if "." in ip_address:
            try:
                return self._lookup_table[ip_address]
            except:
                last_dot_index = ip_address.rfind(".")
                return self.lookup(ip_address[:last_dot_index])
        else:
            try:
                return self._lookup_table[ip_address]
            except:
                return -1

    def tick(self):
        for package in self._package_queue:
            package.tick()

        for package in self._outgoing_packages:
            package.tick()

    def check_received(self):
        if len(self._package_queue) == 0:
            self._idle_cycles += 1
        else:
            # sort packages by time
            self._package_queue = sorted(self._package_queue,
                key=lambda p: p.get_header().get_time())
            self._current_package = self._package_queue.pop(0)


    def process_package(self):
        if self._current_package is not None:
```

```
            if self._current_package.get_header().get_target() == self._ip_address:
                # package to me
                self._parent_simulation.add_to_sink(self._current_package)
                if self._current_package.get_payload().get_is_task():
                    # generate data package as response
                    num_packages = randint(self._max_packages)
                    for i in range(num_packages):
                        package = data(self._ip_address,
                            self._current_package.get_header().get_sender())
                        self._outgoing_packages.append(package)
                elif self._current_package.get_payload().get_is_ping():
                    # ping, so send data package with time used to get here
                    time_used = get_now() \
                    - self._current_package.get_header().get_time()

                    out_data = { "time_used": time_used }

                    self._outgoing_packages.append(data(self._ip_address,
                            self._current_package.get_header().get_sender(),
                            payload_data=out_data))
                else:
                    self._data_packages_received += 1
                    self.task_finished()
                    # package of data sent to this device

            else:
                pass
                # package not to this node, forwarding
                # set outgoing_package to current_package
                self._outgoing_packages.append(self._current_package)
                # continue to send_package
        self._current_package = None


    def send_package(self):
        if len(self._outgoing_packages) > 0:
            outgoing_package = self._outgoing_packages.pop(0)

            target_ip_address = outgoing_package.get_header().get_target()
            target_node = self.lookup(target_ip_address)

            if target_node == -1:
                print("Node with ip: ", target_ip_address,
                        " not found in lookup table.")
            else:
                target_node.add_package(outgoing_package)
```

```
            self._packages_sent += 1

    def write_results(self):
        result_string = "### Results for " +  str(self) + "###\n"
        result_string += "Packages sent: " + str(self._packages_sent) + "\n"
        result_string += "Packages received: " +\
                            str(self._data_packages_received) + "\n"
        result_string += "Idle cycles: " + str(self._idle_cycles) + "\n"
        result_string += "Length incoming queue: " +\
                            str(len(self._package_queue)) + "\n"
        result_string += "Length outgoing queue: " +\
                            str(len(self._outgoing_packages)) + "\n"
        result_string += "Tasks started: " + str(self._tasks_started) + "\n"
        result_string += "Tasks finished: " + str(self._tasks_finished) + "\n"
        return result_string
```

B.5 Network.py

```
from numpy.random import randint

"""
File imports/network.py
Top-level functions: routers_and_nodes, routers_and_nodes_alt
and network_results

Routers_and_nodes:
Uses the line_connect function to connect the routers in a line, fully connects
the set of nodes for a router with themselves and their router, and forwards all
other ip addresses to their router. This esentially creates a network with
"""

def remove_connection(node, ip_address):
    node.remove_lookup_entry(ip_address)

def indirect(node_a, node_b, ip_address):
    if node_a.get_ip_address() == ip_address:
        return
    node_a.add_lookup_entry(ip_address, node_b)

def one_way(node_a, node_b):
    if node_a.get_ip_address() == node_b.get_ip_address():
        return
    node_a.add_lookup_entry(node_b.get_ip_address(), node_b)

def indirect_list(node_a, node_b, ip_addresses):
    for ip_address in ip_addresses:
        indirect(node_a, node_b, ip_address)

def both_ways(node_a, node_b):
    one_way(node_a, node_b)
    one_way(node_b, node_a)

def one_to_all(origin_node, node_list):
    for node in node_list:
        one_way(origin_node, node)

def one_to_all_both_ways(origin_node, node_list):
    for node in node_list:
        both_ways(origin_node, node)

def fully_connect(node_list):
    for node in node_list:
```

```
        copied_node_list = node_list.copy()
        copied_node_list.remove(node)
        one_to_all(node, copied_node_list)

def circle_connect(node_list):
    for i in range(len(node_list)):
        # if i == 0:
        #     one_way(node_list[-1], node_list[0])
        # else:
        one_way(node_list[i-1], node_list[i])

def line_connect(node_list):
    future_ips = [node.get_ip_address() for node in node_list]
    past_ips = []
    for i in range(len(node_list)):
        future_ips.remove(node_list[i].get_ip_address())
        if len(past_ips) > 0:
            indirect_list(node_list[i], node_list[i-1], past_ips)
        if len(future_ips) > 0:
            indirect_list(node_list[i], node_list[i+1], future_ips)
        past_ips.append(node_list[i].get_ip_address())

def routers_and_nodes(routers, list_of_list_of_nodes):
    # assumes nodes connected to a router has ip's as follows:
    # router ip i.e. "123"
    # nodes connected to that router has ip's "123.XXX"
    if not len(routers) == len(list_of_list_of_nodes):
        raise ValueError(
        "routers_and_nodes: length of routers list must be the same as length\
        of list of list of nodes.")

    router_ips = [router.get_ip_address() for router in routers]

    line_connect(routers)

    for i in range(len(list_of_list_of_nodes)):
        # router connected to all its nodes
        # one_to_all(routers[i], list_of_list_of_nodes[i])

        # a router is connected to one other router, and all router ips map to
        # that other router
        #indirect_list(routers[i-1], routers[i], router_ips)

        router = routers[i]

        local_nodes = list_of_list_of_nodes[i].copy()
```

```
        for node in local_nodes:
            # router connected to its nodes
            one_way(router, node)
            # nodes map all router ips to its router
            indirect_list(node, router, router_ips)

def routers_and_nodes_alt(routers, list_of_list_of_nodes, other_connections):
    routers_and_nodes(routers, list_of_list_of_nodes)
    for i in range(other_connections):
        random_router = randint(len(routers))
        random_node = randint(len(list_of_list_of_nodes[random_router]))
        random_router2 = randint(len(routers))
        while random_router == random_router2:
            random_router2 = randint(len(routers))
        random_node2 = randint(len(list_of_list_of_nodes[random_router2]))
        node1 = list_of_list_of_nodes[random_router][random_node]
        node2 = list_of_list_of_nodes[random_router2][random_node2]
        both_ways(node1, node2)

def network_results(node_list):
    # function to print results after a simulation
    result_string = "### Network Nodes Results ###\n"
    for node in node_list:
        result_string += node.write_results()

    return result_string
```

B.6 Datetime.py

```
from datetime import datetime, timedelta

def datetime_to_int(datetime_obj):
    diff = datetime_obj - datetime.min
    return diff.days * 24 * 60 * 60 + diff.seconds

def int_to_datetime(int):
    return datetime.min + timedelta(0, int)


def get_now():
    return datetime_to_int(datetime.now())
```