A bluetooth wrapper for network programming: Bluenode

Netcentric Computing

Julius Wagt (11109602), Duncan Kampert (11287500),
Joey Lai (11057122), Ruben Stap (11269146)

February 1, 2019

1 Abstract

In this paper we look at the problems that come with ad hoc networking. We look at the possibilities of creating an everlasting network, while trying to keep the network stable with the help of routing- and communication protocols. For the link between clients, we use Bluetooth communication as a baseline, and expanded upon this. We created a chatting application to show that there is practical use of our network. With this application we have shown it is possible to communicate and share data with clients you are not directly connected with. We have also researched and compared different network maintenance- and routing protocols. We have found the distinct strengths and weaknesses of both researched algorithms.

2 Introduction

Ad hoc networks are self organized networks without any centralized control. In these networks, nodes aren't necessarily directly connected with all other nodes. Nodes which are not directly connected can communicate with one another with the help of multi-hop routing. This means that data passes multiple nodes before reaching the destination. To efficiently send data from one node to another without a direct connection it is therefore necessary to implement an efficient routing algorithm which determines the route the data takes. The goal of our project is to create an application with which an ad hoc network can be formed. This network should allow for indirect communication. There are multiple approaches to create such indirect communication, we wish to compare these approaches. It was decided to create a chat application on top of the logic necessary to form an ad hoc network as it is simple to implement and allowed us to shift our focus to the networking part. Furthermore, we will analyse two different algorithms for network maintenance.

2.1 Communication method

There are many different communication primitives methods to choose from. As the main goal of our application is to allow and compare between methods of indirect communication it was necessary that this wasn't already implemented by the chosen primitive. On the other hand it wasn't feasible to look at too underdeveloped primitives because of the limited amount of time.

Multiple protocols such as WiFi-direct, WiFi-adhoc and RFCOMM (Bluetooth) were considered. WiFi-adhoc is a protocol that implements a multi-hopping peer to peer (p2p) network for communication through WiFi. WiFi-direct is the same although only allowing for single-hop communication [3]. RFCOMM is a Bluetooth protocol which implements reliable single-hop communication between two clients. We decided to use RFCOMM as it is an ideal primitive for creating a robust p2p multi-hopping network. More specifically it does not implement a large portion of our goal even while not being too low level.

Another reason we chose to use Bluetooth over both the WiFi protocols is that Bluetooth supports multiple individual connections natively. At the time we thought that WiFi uses the device to either host a WiFi network or become a client of an existing network. This however is not the case as it can be both at the same time.

2.2 Bluetooth

According to [1] Bluetooth programming is very similar to Internet programming. For instance Bluetooth connections can be created, maintained and used with sockets just like Internet connections.

The Internet assigns MAC-addresses (Media Access Control address) to interfaces. In the same way Bluetooth assigns 48-bits addresses to Bluetooth chips. However in contrast to the Internet, this address is used at all layers in the Bluetooth protocol stack.

TCP and UDP are reliable and best-effort communication transport layer protocols for the Internet. Bluetooth has the protocols RFCOMM and L2CAP which achieve similar goals.

Classical Internet applications choose a port at design time. Transport layer protocols for Bluetooth offer fewer ports than the Internet transport layer protocols. Therefore it is not handy to adopt the same habit for Bluetooth applications. Instead Bluetooth provides dynamic port allocation to the applications. Applications can register themselves on a local Service Discovery Protocol (SDP) server with a Universally Unique Identifier (UUID), port, name and more possible fields. An SDP server runs on a fixed port. Therefore other Bluetooth devices can discover services on nearby devices by communicating with the SDP server of each device.

2.2.1 Optimized Link State Routing

Routing categories can be categorized by the algorithm used to determine the routes. Proactive routing protocols determine routes beforehand while reactive routing protocols determine routes on demand. [7] The latency for data transfer is the lowest for proactive protocols. The overhead however is the highest for such protocols as every topology change needs to be synced. Because it was decided to strive for low latency, the choice was made to implement a proactive routing protocol. Of all protocols that were considered, the Optimized Link State Routing (OLSR) protocol has the best scalability [9] and has a relatively easy implementation as it is a flat protocol. This means that every node has the same functionality.

OLSR is a internet layer protocol specifically designed for mobile ad hoc networks. An internet layer protocol has the function of routing data between networks over an unreliable link.

According to [12] OSLR starts by sending and receiving HELLO messages for every node in the network to detect it's 2-hop neighbourhood topology. Subsequently every node in the network chooses Multi-point Relay (MPR) nodes, which are located in the neighbourhood of each node. They are chosen in such a way that the entire 2-hop neighbourhood is reachable from the MPR nodes. The chosen MPR nodes of a node A are the only nodes allowed to rebroadcast data sent to and from node A.

[12] also notes that node broadcasting using MPR nodes places less stress upon the network. Instead of every node in the network resending a message to all neighbours only the MPR nodes of that node are allowed to do so. Therefore this mechanism is used when nodes share topology data with the rest of the network. More specifically, only nodes that have been chosen as MPR nodes, share topology data. This enables all nodes in the network to create a routing table for incoming data with a specific destination.

3 Implementation

3.1 Overview

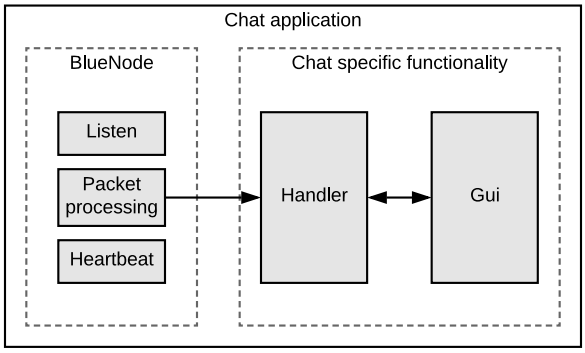


Figure 1: Structure of the application. Each grey block indicates a thread. Arrows indicate a communication channel.

We tried to make our application as modular as possible. More specifically we wanted to create a network functionality which could easily be used in the development of new applications. Therefore it was decided to split the application in network functionality and chat specific functionality. All network related functionality will be referred to as BlueNode. BlueNode functionality is handled in the listen, packet processing and Heartbeat threads while all chat related functionality is handled in the Handler and the Graphical User Interface (GUI) threads. This thread division was made because all operations performed in the threads are somewhat blocking. Any combination of functions executed in one thread would cause unacceptable latency.

BlueNode has two main functions: listening to new connections and handling incoming data on existing connections. These events are communicated to the Handler. It also provides functions to the Handler for nearby node searching, connecting and messaging (directly or a using a broadcast).

The GUI thread is purely responsible for a responsive chat, while the Handler thread handles incoming commands from the GUI as well as incoming packets from BlueNode. Communication from the packet processing thread to the Handler thread as well as communication between the Handler and the GUI thread is performed using queues.

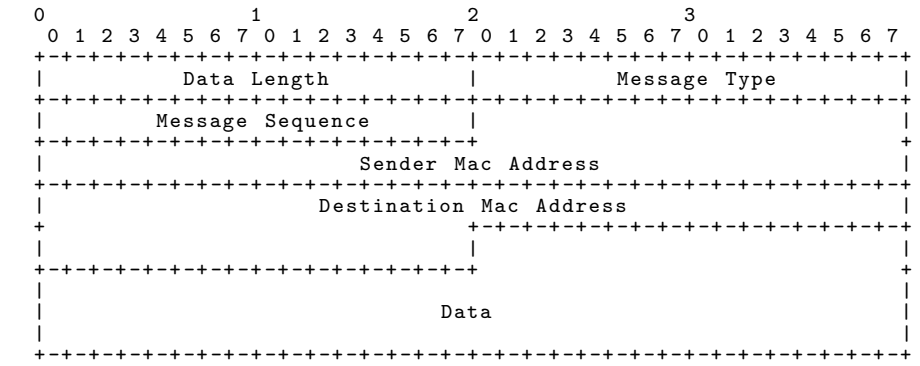
3.2 Packets

3.2.1 Network Layer

The network layer is what a node uses to communicate to other nodes in the network. This layer is used for a multitude of operations, a full list of operations

is given in 3.2.1.

The packet structure for the Network layer is as follows:



The packet header is defined as everything that is not the data segment of the packet. Data length is specifically the length of the data, excluding the size of the packet header.

Message type is used to tell the BlueNode what to do with a packet. There are a multitude of packets that are used for BlueNode to function. All the important packet types are explained below.

- Network Join Request
  This packet is sent by an unconnected BlueNode to a nearby node. This second node will then respond to this Network Join Request with a Network Join Response packet.

- Network Join Response
  This packet is sent as a reply to a Network Join Request. As we wanted to build a network with optional size limits, it is possible for a node to try to connect to a network which is full. In this case the responding node will send no data. In the case the requesting node may join the network, the responding node will send a serialized Python object containing all known data of the nodes in the network.

- Announce Node
  This packet is sent by the node who received a Network Join Response in the case the requesting node may join the network. This packet contains the MAC address of the new node, which is broadcasted through the network. All other nodes in the network then know when a new node joins.

- Topology Update Message
  When on-change topology sharing is on, this is a packet which contains updated network topology which each receiving node joins into it's local topology. It is sent from a joined node.

- Topology from joined/joining node Messages
  These are two messages. When the on-change topology sharing is on

this message is sent simultaneous with the join request/join response and contains the current topology from the joined/joining node. It gives the joined node a complete view of the new topology before flooding it through the network with the Topology Update Message packet.

- Broadcast Message
  This is a general packet for all data that needs to be broadcasted through the network. This is only used when the application needs to broadcast data.

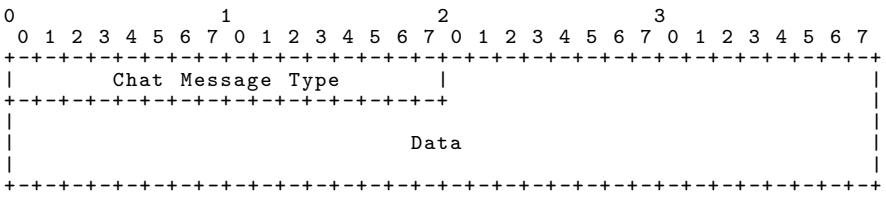
- Direct Message
  This is a general packet for all data that needs to be sent to a single user in the network. This packet is also only used when the application needs to send data directly to another node. To achieve this, we implemented multiple routing algorithms.

- Exit Message
  This packet is sent through the network when a node wants to leave it. This packet is used to update every other node's topology.

At the core, the Bluetooth socket works as a data stream. However we implemented a wrapper around the socket.recv(BUFFER) function which splits the stream into data frames. This means the programmer does not have to worry about keeping track of the length of the data. We made the choice to change the data stream into a data frame because it was more fit for our needs.

3.2.2 Application Layer

The application layer is what the handler uses, and may be changed to fit whatever needs the program has. For our chat application we created a simple packet structure which can be used to send chat messages and binary files over the network. The packets have a very simple structure as follows:



In the socket.recv function we made sure all data gets combined into frames as how the original sender created it. This allows for the the programmer to send any length data over the network as long as the length in bytes fits in the Data Length field in the network packet. For our implementation of this packet we allocated two bytes for the data length, which means the data for the chat has a maximum length of $2^{16} - 2$.

3.3 BlueNode

BlueNode is the core of our project, and does all managing of the network internally. It was decided to run BlueNode in a separate thread from the GUI so all blocking communication could be done without interrupting the interface for the user.

As described earlier the two main functions of BlueNode are: handling incoming data and listening for new connections. Handling new connections is blocking. Because we wanted to support multiple connection to a single node we added a thread for listening to new connections. This means a node can simultaneously add new connections to the network, and handle messages from known nodes in the network.

Disconnection handling is done with a wrapper around socket.send and socket.recv. As Bluetooth connections are volatile, a disconnection might not be cleanly handled by the disconnecting client. In those cases the Bluetooth socket throws an error, this error is then unpacked and checked for the specific disconnection error code (104). In that case exclusively will the node be handled as a disconnecting node. All other error codes will not be handled.

To connect to a new node, as a currently detached node, there are two functions necessary. Firstly we need a function that scans the area for currently active nodes. This function uses the builtin find_service function from PyBluez [13]. This function returns a list of nearby nodes for which the UUID is equal to the node's own UUID. We then put this data in the queue so the handler can use this information to tell the user about all nearby nodes. Sadly this find_service function blocks all other uses of the Bluetooth device in the implementation of PyBluez. For that reason we only scan the area on request of the user.

The second function we need is the actual connection function. To connect to a Bluetooth socket, the only two parameters required are the MAC-address, and the port of the host socket.

3.4 Connecting to the network

In order to join a network, we implemented a simple handshake method instead of an auto-join protocol. This decision was made because we wanted to be able to put restraints on the maximum size of our network. The handshake method makes use of three different types of packets. These packets are all highlighted in the packet section 3.2.1.

Algorithm 1 Joining Peer

```
1: Send Network Join Request
2: Receive Network Join Response
3: if Accepted then
4:     Sync network/topology
5: else
6:     Close connection
7: end if
```

Algorithm 2 Entry Peer

```
1: Receive Network Join Request
2: if Current network size < max allowed network size then
3:     Send accepted Network Join Response
4:     Sync network/topology
5:     Announce new node to other peers in the network
6: else
7:     Send refused Network Join Response
8:     Close connection
9: end if
```

3.5 Network maintenance and communication

Mode	Network maintenance	Broadcast method	Routing method
0	Heartbeat	Naive	Naive broadcasting
1	on-change	Naive	Direct
2	on-change	MPR	Direct
3	on-change	MPR	MPR broadcasting

Table 1: Possible modes of chat maintenance and communication in BlueNode.

As indicated in 1, BlueNode can run in 4 modes of chat maintenance and communication. This was done to allow for experimental comparisons. Heartbeat floods the network to keep track of active nodes in the network. As it doesn't share topology data, the broadcasting and routing method available to the handler are both built on normal broadcasting. Modes 1-3 share network topology data when a change in the network is detected. For this reason it allows direct routing, and an optimized form of broadcasting. This optimized method uses selectively forwarding for broadcast messages.

3.5.1 Heartbeat algorithm

The auto-self discovery algorithm (Heartbeat) of Haixia Liu [5] describes a method of finding and discovering peers in a pure wireless ad hoc peer-to-peer network. This method makes use of a network flooding technique to maintain it's network. We used certain elements of this algorithm to compare its performance with a different network maintainability technique, namely the on-change method. Our heartbeat algorithm implements the following components:

- Heartbeat mechanism:
  Each node in the network continuously broadcasts 'Exist' type messages with the current local time.
- Heartbeat list:
  Each node in the network maintains a list of the addresses of peers with its latest timestamp.

- Detector:
  Main function which periodically checks if every peer in the heartbeat list is still alive.

The heartbeat mechanism and detector are always running in the background in a separate thread. When nodes in the network receive 'Exist' messages with a new address, it will be added to the heartbeat list. When the address already exists in the list, it will update the timestamp of the corresponding address.

Algorithm 3 Peer detector

```
1: while True do
2:     loop through each peer in heartbeat list:
3:         if current_timestamp − timestamp_in_list > time_out then
4:             remove peer from network
5:         end if
6:     end loop
7: end while
```

With this mechanism, it is also possible to keep track of two different heartbeat lists, namely for peers in its local and remote ranges. Using two different heartbeat list would be advantageous for using different time out values for local and remote peers.

Algorithm 4 Peer detector with local and remote lists

```
1: while True do
2:     loop through each peer in local heartbeat list:
3:         if current_timestamp − timestamp_in_list > local_time_out then
4:             remove peer from network
5:         end if
6:     end loop
7:
8:     loop through each peer in remote heartbeat list:
9:         if current_timestamp − timestamp_in_list > remote_time_out then
10:            remove peer from network
11:        end if
12:    end loop
13: end while
```

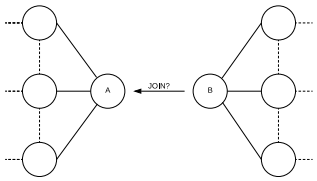
3.5.2 Routing protocol based on OLSR and RFCOMM

As mentioned earlier, OLSR is an internet layer protocol [6] which tries to achieve best effort multi-hop data delivery across a network over unreliable links. Accordingly it periodically checks the status of links and thereby the reachability of neighbours. RFCOMM, the currently used protocol in our application is a transport layer protocol and already ensures reliable data delivery between two nodes.

When a connection fails, the corresponding sockets will indicate this on usage. Therefore it is not necessary to implement polling on top of this, which OLSR does. Instead of implementing OLSR on top of RFCOMM, a link state proactive routing protocol was devised. This uses the earlier mentioned core ideas of OLSR, but doesn't concern itself with the objectives RFCOMM already achieves.

Link state sharing
In a link state routing protocol it is crucial that every node keeps track of the network topology. This is done in a NetworkX [11] graph where nodes are represented with their Bluetooth addresses. This graph is updated when nodes join or disconnect from somewhere else in the network. So, in other words the topology sharing method is on-change. As nodes can only detect changes to their local neighborhood, it is crucial that this information is relayed across the entire network.

Joining a network



In the final step of the joining procedure Alice sends a JOIN_RESPONSE to Bob indicating acceptance in the other network. It also adds the remote node to it's local topology. In response to the JOIN_RESPONSE message, Bob sends it's local network topology to Alice. This is done because Bob might be in a network himself. Node Alice merges this topology with it's local topology and finally broadcasts the complete network topology across the entire network with a TOP_UPDATE packet. On reception of such a packet, a node merges the topology in the message with it's locally saved topology. Thus, after the TOP_UPDATE message has been received by each node in the network, each node has the newly updated network topology.

Disconnecting nodes
When a node detects a disconnecting node in its neighborhood through the disconnection handler, it broadcasts this event with the MAC address of the node. All other nodes in the network will then receive a NODE_DISCONNECTED message. On reception of this message a node updates it's local topology.

Optimized link state sharing
Just like OLSR, the devised protocol is made more efficient by using MPR nodes for broadcasting. As described earlier only the MPR nodes of a node are eligible

of broadcasting data. This reduces stress on the network. For the optimized broadcasting functionality, a forward field was built into the packet format.

When a node wishes to send a broadcast message, MPR nodes are calculated from the locally saved network topology using an heuristic approach. Subsequently the forward flag is set to 1 for packets which have a MPR node as it's destination, while packets heading to non-MPR nodes have the forward flag set to 0. Nodes receiving broadcast packets can subsequently determine whether they have to forward a packet or not.

This heuristic approach to calculate MPR nodes is as follows. First a node determines all two-hop neighbors reachable through only one neighbor. All those neighbors are set as a MPR node. Subsequently neighbors are added to the set of MPR nodes iteratively based on which neighbor reaches the most neighbors within a 2-hop distance. [6]

3.5.3 Routing

In contrast to the link state sharing, the routing protocol is rather simple. DIRECT_MESSAGES are routed to the destination by calculating the next hop in the shortest path to the destination from the local network topology. This is done using a shortest path algorithm from NetworkX on the locally saved network topology. When a node receives a DIRECT_MESSAGE that is not meant for himself, the same procedure is repeated.

3.6 Monitoring

We wanted to monitor our topology and network traffic. For this we used the AdHoc-Monitor application which was created by Erik Landkroon [8]. This application makes it possible to see how devices are connected and the amount of data that is sent over the connection between these devices. This project consists of a java module and a web interface which gives a representation of the network. The java module is made specifically for android, and sends data to the webserver in a json format. This means that even though our application is programmed in Python, we can still monitor our topology using this monitoring tool. We ran the webserver on a single computer to provide a graphical visualization of the network. All clients can then send data to this webserver to show a lot of important information about the network.

3.7 Simulation

Testing and debugging our application turned out to be difficult with physical devices. To speed up testing an attempt was made to create a local simulation. The idea was to have an application in which one can draw the network topology as desired. Subsequently all nodes, chat applications, launch as processes which communicate locally to other processes with their RFCOMM sockets.

The problem is that RFCOMM sockets don't support local interconnection. TCP sockets do support this. Moreover both provide similar functionality.

Therefore code was written which emulates a RFCOMM socket with a TCP socket. By using a conditional import in BlueNode for either the real or simulated RFCOMM socket a simulation was made possible with minimal changes to the networking code.

A simple graph drawing tool was taken from Github [10] and adjusted to launch chat processes on a button press. The tool passes a randomly generated virtual Bluetooth address, node name and server connection listening port number to each chat process. The same information is passed about all neighboring chat processes according to the graph. Each chat process shows these processes as nearby nodes in the chat GUI.

The TCP socket wrapper library is simply a wrapper for most socket functions. Emulating the accept and send functions required some more logic.

The accept function accepts a client connection on a server socket. For RFCOMM it returns the Bluetooth address of the client. As the port of the client is dynamically allocated and the host is 'localhost', it is not possible to determine the virtual Bluetooth address of the client chat process without communication. Therefore the connect function was modified to send the virtual Bluetooth address after establishing a connection.

When an RFCOMM connection is broken and an attempt is made to send or receive data, an exception is thrown immediately. For TCP sockets this only happens when data is received. For this reason the simulated socket internally tries to receive 1-byte before sending data.

4 Experiment

We did two different experiments with our application. The first experiment checks the performance of our network compared to the theoretical maximum of Bluetooth. This experiment was split into two sub-experiments, namely latency, and throughput.

The second experiment we did was to check the performance of our implemented algorithms. We can toggle between multiple algorithms when starting up our application. These modes are described in the table 1. We experimented to compare the performance between two modes. To ensure the results have as little differences between them as possible, we did all experiments on the same network topology.

4.1 Performance

We measured both the throughput, and the latency of our network per hop. For the throughput we sent a large file (300 MB) of data through the network using direct routing. We observed a maximum throughput of approximately 700 KB/s, averaged over multiple seconds of communication.

For latency measurement we used a ping round trip time (RTT) measurement over multiple hops. We used the direct routing algorithm using the shortest path from NetworkX. We measured the ping RTT over one, two, and three hops.

We however could not do measurement for more than three, as we did not have any more computers to run our application.

4.2 Comparisons

4.2.1 Mode 0 vs 1

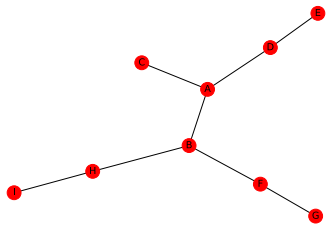


Figure 2: Network topology used for comparison between mode 0 and 1.

We wanted to use a larger network for this comparison between algorithms, and opted to use our simulation for this experiment. The reason we did not use real devices was that we did not have enough devices supporting Bluetooth and Python to test this. This however should not make much of a difference in our results, as the size of the data we sent was far below the maximum throughput of Bluetooth. This means the only difference between the simulation and the physical Bluetooth communication was the latency. As we did not use latency or timeliness for this experiment, the results should be indicative of how a real network will behave.

There are two distinct differences between the Heartbeat algorithm implemented in mode 0, and the on-change algorithm implemented in mode 1.

The first difference between these two is the data required to connect this new node to the network. The Heartbeat algorithm requires very little data to connect a new node to the network. It sends a list of all nodes in the network from the host node to the client node. Then, the network is informed of a new client joining the network with a broadcast message containing the new client's MAC address and his username. Compared to this, the on-change algorithm sends more data through the network when connecting a new node. This algorithm initially sends an entire NetworkX graph of the current network to the new client. Then, the new client combines this with his current network, and sends the combined network back to the host. The host then informs the network of the new client by broadcasting the new graph through the network.

The second difference between these two nodes is in the data required for upkeep in the network. As the Heartbeat algorithm does not know anything about the network structure, it sends broadcast messages through the network announcing the presence of the sending node. Every node sends one of these messages on a small interval, which causes a moderate amount of strain on the network. The on-change algorithm does not require any sort of upkeep to

maintain the network. For this reason it is expected for the heartbeat algorithm to initially have a lower network usage, but over time it should exceed the amount of data sent when compared to the on-change algorithm.

To generate this data we polled the I/O of the BlueNode every 150ms. This interval was chosen to be easily compatible with the existing monitoring tool we already used.

5 Results

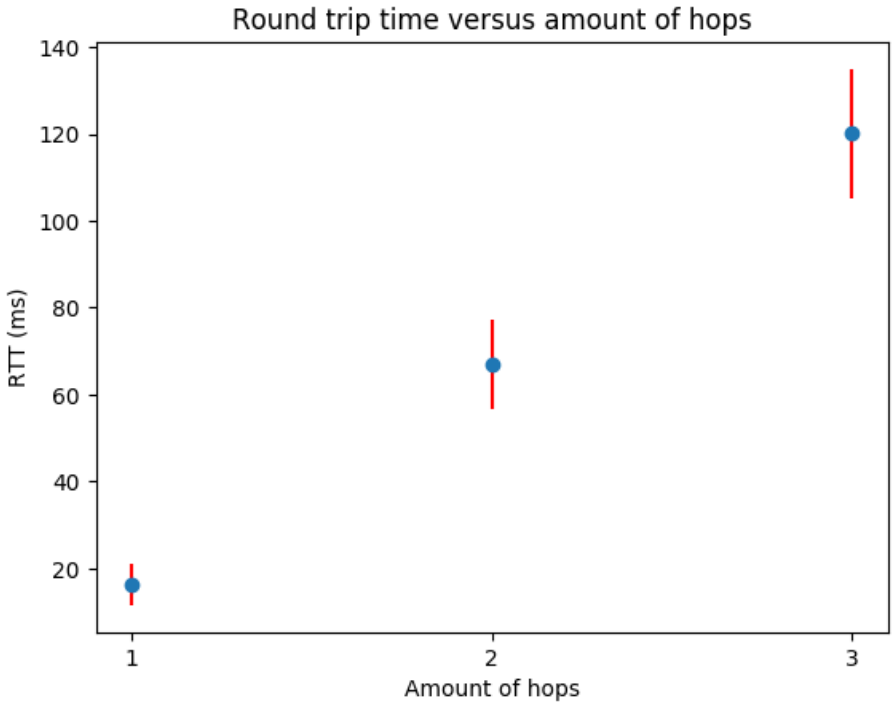


Figure 3: RTT of a packet per hop

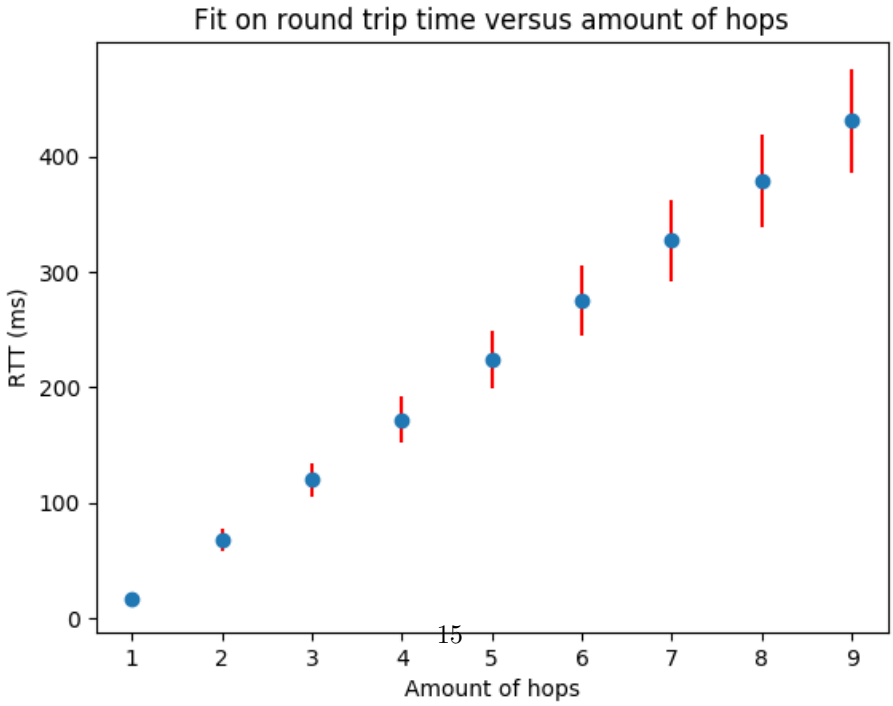


Figure 4: RTT data fit for more than three hops

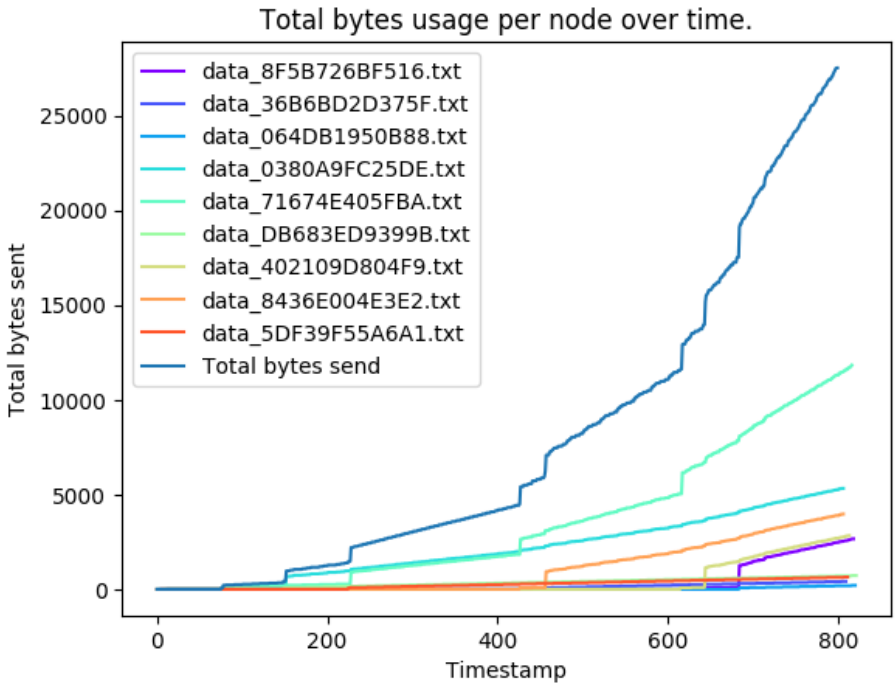


Figure 5: Heartbeat Algorithm

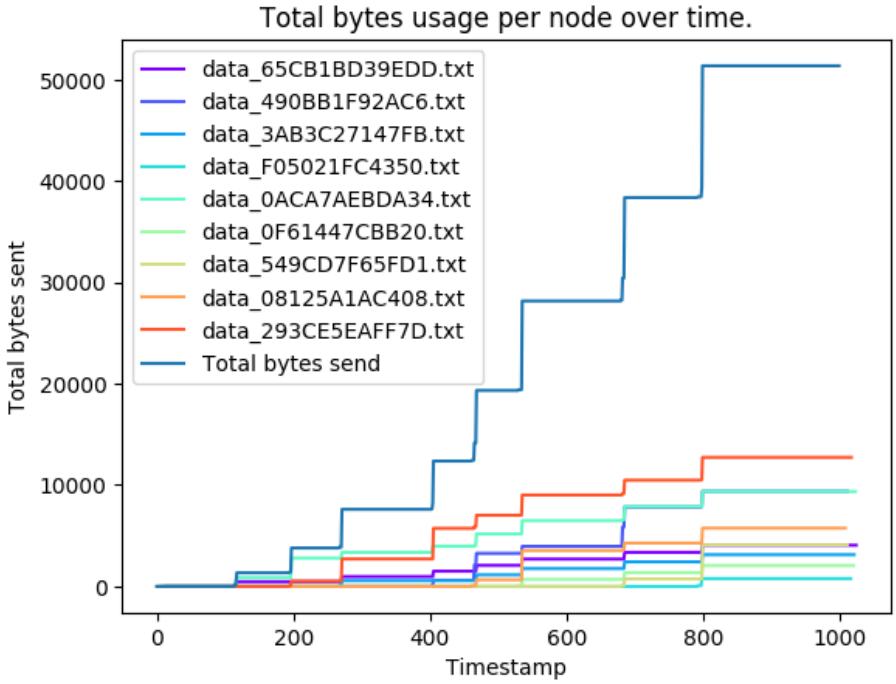
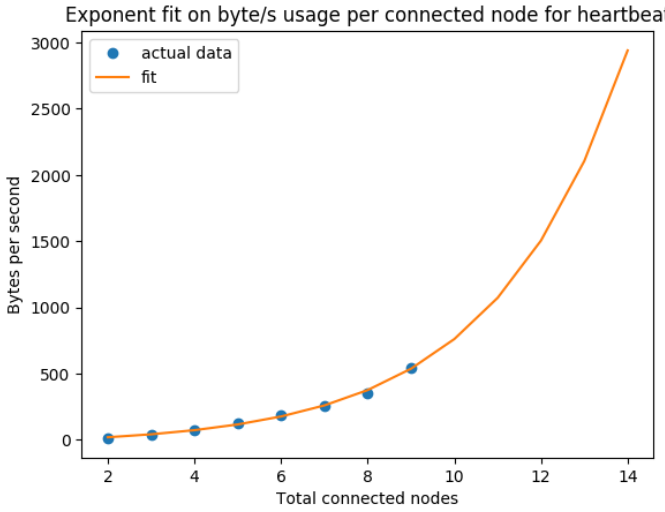
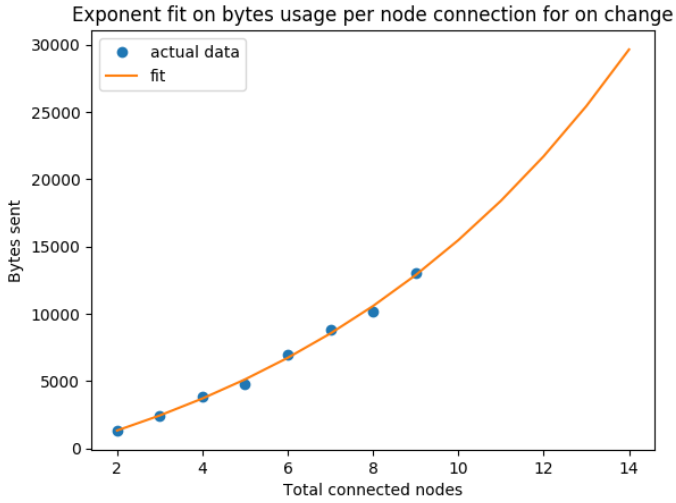


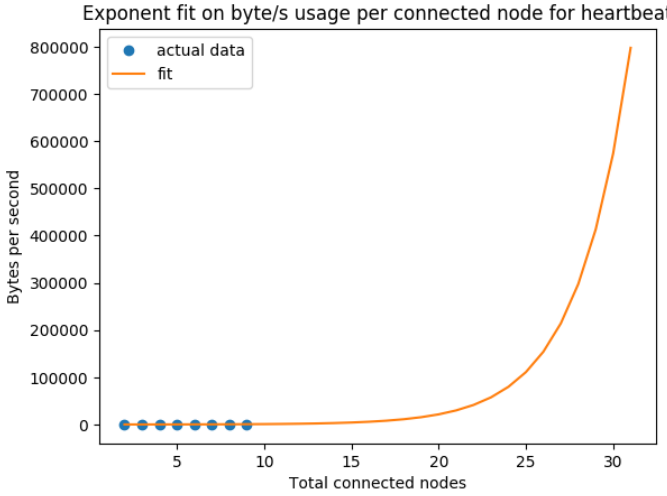
Figure 6: On-change Algorithm



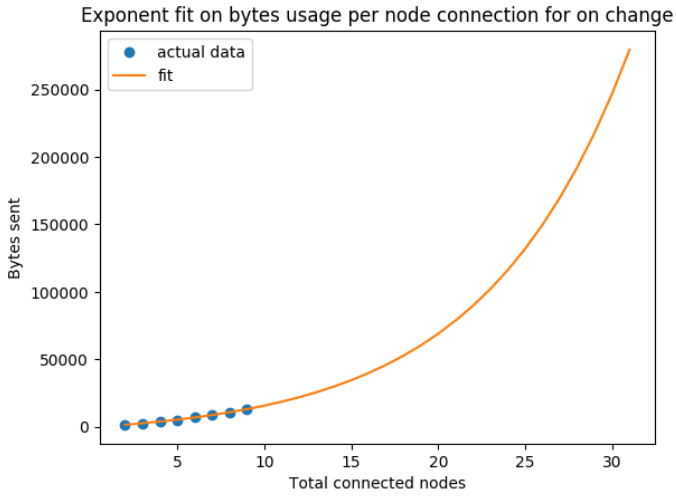
(a) Heartbeat algorithm



(b) On-change algorithm



(c) Heartbeat algorithm



(d) On-change algorithm

Figure 7: Fitted data usage per network maintenance algorithm

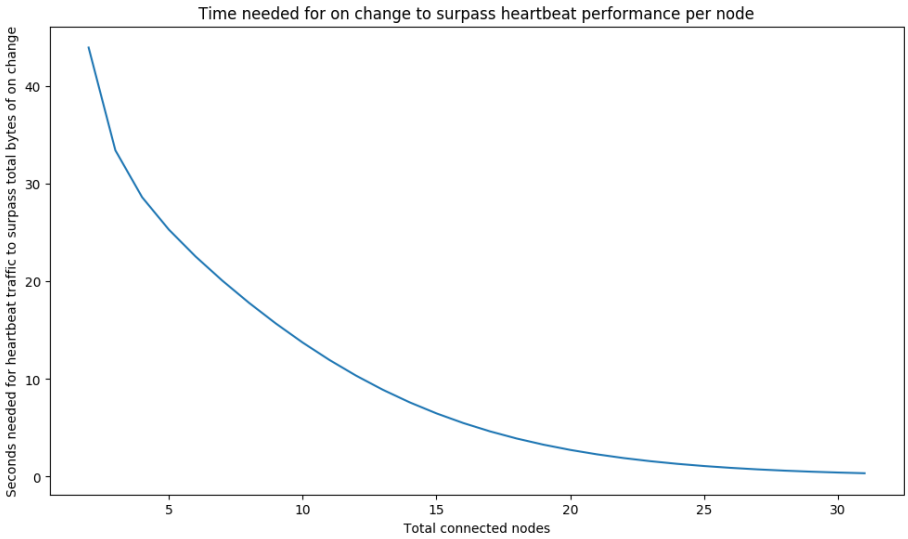


Figure 8: Performance comparison between Heartbeat and on-change

6 Discussion

It was a great challenge to develop a Bluetooth network, where there is no centralised control. Right now the application is tested and proven to work with a small dynamic network without many redundant connections and an efficient multi-hop algorithm. However there are still a lot of things we want to do. We have yet to test for when there are thousands of people connected to a network and see if there are any redundant connections.

6.1 Network maintenance

A quick glance at the results of (Figure 6), we can see that the total bytes usage per node with the on-change algorithm is much higher than the Heartbeat algorithm in the first 800 timestamps. This does not mean that the performance of the Heartbeat is better than the on-change, since Heartbeat continuously sends out data while the on-change only sends data on a new connection.

By looking at the exponent fit of the data usage per node connection(Figure 7), we can see that the exponential growth of bytes of the Heartbeat algorithm is higher than the exponential growth of the total bytes send with on-change. This is because Heartbeat algorithm floods the network with a exist message from every node periodically, while the on-change algorithm only floods the network with a updated topology from the joined node.

Finally, in the comparison figure(Figure 8) we notice that the time taken for the traffic of Heartbeat to surpass on-change gradually decreases with each connected node. This means that the Heartbeat algorithm is not as scalable

as the on-change algorithm. We can therefore conclude that the Heartbeat algorithm is better for short small networks while the on-change algorithm is better for long large networks.

6.2 Latency

We observe from Figure 3 that the relation between the RTT and amount of hops seems to be linear. By fitting our data to calculate the RTT of a larger amount of hops, we can see that by 9 hops we reach a RTT value of more than 400ms. This means that our implementation of a peer-to-peer network is not sufficient for an application that needs low latency. Low latency applications like FPS games have a threshold of 100ms which our application already exceeds at 3 hops [14].

It should be pointed out that the fit was only done on 3 data values. This means that the fit is not a 100% accurate. For future research, it would be possible to increase the accuracy of the fit by using more devices to create extra hops.

6.3 Security

We have thought about implementing a security protocol in our connection- and networking layer. But this seemed neigh impossible. First of all the network requires certain nodes to act as links between segments of the network, these nodes are essentially man-in-the-middle attackers since there might be new nodes linked to the network due to this certain node. This can be circumvented by having each new node make a connection with two existing nodes in the network making it more difficult to impersonate a node trying to join the network. Multiple papers have been written about security and authentication in ad hoc wireless networks [2] [4] and the challenges they offer. Implementing all of this seemed ridiculously excessive and out of the scope of this project. We also did not know if implementing such a protocol might make the application inherently slow.

7 Conclusion

Our final work shows it is possible to create a mobile and dynamic ad hoc network. Our implementation may not be perfect and leaves room for further improvement, but it proves that with only short range wireless technology such as Bluetooth we can replicate most of the functionality that we find in other communication protocols. Furthermore, we have concluded what the optimal network maintenance algorithms are for different networks. That is to say, Heartbeat for short small networks and on-change for long large networks.

References

[1] Albert Huang. An Introduction to Bluetooth Programming, 2005.

[2] Dirk Balfanz, Diana K Smetters, Paul Stewart, and H Chi Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *NDSS*. Citeseer, 2002.

[3] Colin Funai, Cristiano Tapparello, and Wendi Heinzelman. Supporting Multi-hop Device-to-Device Networks Through WiFi Direct Multi-group Networking. Technical report, 2015.

[4] Pravin Ghosekar, Girish Katkar, and Pradip Ghorpade. Mobile ad hoc networking: imperatives and challenges. *IJCA Special issue on MANETs*, 3:153–158, 2010.

[5] Liu Haixia. *Peer-to-Peer Application in Ad Hoc WIreless Networks*. PhD thesis, University of Dublin, 2002.

[6] Philippe Jacquet. Optimized link state routing protocol (olsr). 2003.

[7] Subir Kumar Sarkar, Puttamadappa, and T Basavaraju. *Ad Hoc Mobile Wireless Networks*. Auerbach Publications, 2007.

[8] Erik Landkroon. Ad hoc Monitoring Tool. `https://github.com/ERLKDev/AdHoc-Monitor`.

[9] Shima Mohseni, Rosilah Hassan, Ahmed Patel, and Rozilawati Razali. Comparative review study of reactive and proactive routing protocols in MANETs. In *4th IEEE International Conference on Digital Ecosystems and Technologies*, pages 304–309. IEEE, apr 2010.

[10] Npanov. npanov/simple-qt5-graph-editor, Feb 2017.

[11] Open source 3-clause BSD license. Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. `https://networkx.github.io/`.

[12] Andreas Tønnesen. *Impementing and extending the Optimized Link State Routing Protocol*. PhD thesis, University of Oslo, 2004.

[13] vlovich and rgov. Bluetooth Python extension module. `https://github.com/pybluez/pybluez`.

[14] Rasa Bruzgiene Peter Pocta Lea Skorin-Kapov Andrej Zgank Zhi Li, Hugh Melvin. Lag Compensation for First-Person Shooter Games in Cloud Gaming. In *Lecture Notes in Computer Science*. Springer, may 2018.