Individual report

Ruben Stap (11269146)

January 2019

Our goal is to create a P2P ad hoc network with several routing and network maintenance algorithms which we wish to compare. To create a full proof of concept it was decided to build a chat on top of the P2P ad hoc network.

1 Protocol choice

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 [2]. 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.

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.

3 Disconnecting nodes

The application is seperated in a network and application layer. One of the functions of the network layer is to maintain a databese of active and directly connected nodes. Each entry in the databese is a socket. As other parts of the network layer use sockets from this databese for sending or receiving data a failure will indicate a disconnect.

Instead of placing logic to update the databese at each usage, a disconnection handler function was created. This is a function with which a socket and the usage (function) can be passed. Every send or receiving operation on a socket can now simply be passed to the disconnection handler, which takes care of a possible disconnect.

4 Specifying a bluetooth adapter

Our application currently only supports one present Bluetooth interface. This is because it chooses a random Bluetooth interface and uses the first Bluetooth address from the system call hciconfig as it's corresponding address. When more than one Bluetooth interface is on a system, the first Bluetooth address might not correspond with the randomly choosen Bluetooth interface.

Therefore code was added to the setup command in our Makefile to enable the user to choose between all available Bluetooth interfaces. This choice is written to a settings file which is read out in our application. In addition to this code was added to the run command in our Makefile to make sure that the Bluetooth interface is visible for other devices, which isn't the default setting of most interfaces.

5 Routing algorithms

Uptil now public communication inside the network is done by broadcasting. Private communication can also be implemented using broadcasts. When the network grows bigger this will cause unnecessary overhead. Therefore a routing protocol is needed.

There are many routing protocols for mobile ad hoc networks. In [4] multiple ways to categorize these protocols are discussed.

The most important categorization for narrowing down the choice of a routing algorithm is based on the manner in which routes are determined. This results in proactive, reactive and hybrid routing protocols. In proactive routing protocols nodes determine routes to all destinations at startup. Those routes are periodically updated. In reactive routing protocols routes are determined

on demand. A hybrid routing algorithm is a combination of both. Most of the time a hybrid routing algorithm is non-uniform where proactive routing and reactive routing are used at different levels i.e. for routing to nodes in a cluster and between cluster heads.

The overhead of a proactive routing protocol is high when compared to reactive routing protocols as every topology change needs to be transmitted across the entire network. Reactive routing has less overhead but more latency because of on demand route discovery. Because it was decided to strive for low latency, which proactive routing protocols offers, the choice was made to implement a proactive routing protocol. Note that a hybrid routing protocol could have offered the same low latency but with a bigger scalability. The choice was also made on the basis of ease of implementation.

There are many proactive routing protocols. To name a few: DSDV, WRP, GSR, FSR, STAR, DREAM and OLSR. Proactive routing protocols can be flat or hierarchical. Hierarchical protocols assign different functions to different nodes while a flat protocol assigns the same functionality across all nodes. The choice was made to implement a flat proactive protocol as flat protocols are the simplest to implement. Of the flat routing protocols, OLSR has the best scalability. [5] Therefore the choice was made to look further into OLSR.

6 OLSR

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 [8] 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.

[8] 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.

As mentioned earlier, OLSR is an internet layer protocol [3] 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.

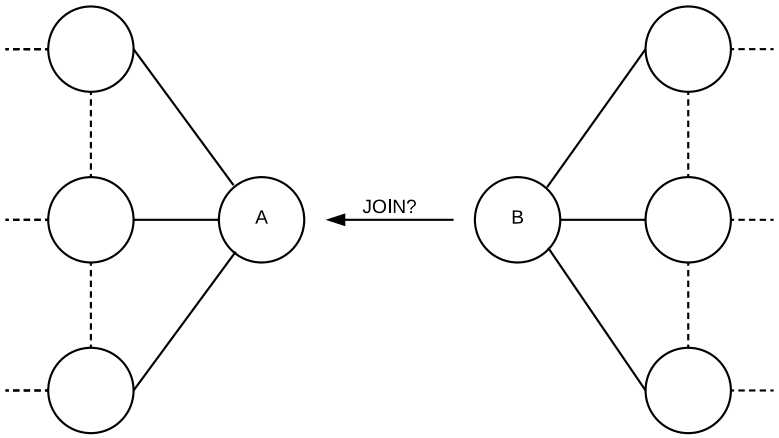
When a connection fails, the corresponding sockets will indicate this on usage. Therefore it is not necessary to implement polling on top of this, which OLSR does. Instead of implementing OLSR on top of RFCOMM, a link state proactive routing protocol was devised. This uses the earlier mentioned core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

6.1 A routing protocol based on OLSR and RFCOMM

6.1.1 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 [7] 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 nodes



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.

6.2 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. The heuristic to calculate MPR nodes is described in full detail on the following page and was taken from the RFC for OLSR. [3]

Algorithm 1 MPR nodes determination of a node A

```
1: Find shortest paths from A to all destinations uptil length 2.
2: two_ns = 2-hop neighbors list
3: is_ns = isolated 2-hop neighbors list
4: for paths in shortest paths do
5:     if path length is 2 then
6:         Add destination to two_ns.
7:     end if
8: end for
9: for two hop neighbor in two_ns do
10:    if two hop neighbor has degree 1 then
11:        Add neighbor to is_ns.
12:    end if
13: end for
14: nis_ns = non-isolated 2-hop neighbors = set difference two_ns and is_ns.
15: mpr_ns = mpr nodes = empty list
16: while is_ns is nonempty do
17:    Currently processed node = the first element is_ns
18:    The only neighbor of this node is now mpr node.
19:    Find neighbors of this mpr node.
20:    is_ns = set difference is_ns and neighbors of mpr node.
21:    nis_ns = set difference nis_ns and neighbors of mpr node.
22:    Add new mpr node to mpr_ns.
23: end while
24: ps_mpr_ns = possible mpr nodes = set difference neighborhood A & mpr_ns.
25: while nis_ns is nonempty do
26:    chosen mpr node = empty
27:    max reachability = 0
28:    for all possible mpr nodes in ps_mpr_ns do
29:        Find neighbors of node.
30:        Find the set union between neighbors and nis_ns.
31:        The size of this set is the 2 hop reachability of this mpr node.
32:        if reachability is greater than max_reachability then
33:            chosen mpr node = current node
34:            max reachability = reachability of current mpr node
35:        end if
36:    end for
37:    Add chosen mpr node to mpr_ns.
38:    nis_ns is set difference nis_ns and neighbors of chosen mpr nodes.
39:    Remove chosen mpr node from ps_mpr_ns.
40: end while
```

6.2.1 Routing protocol

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.

6.3 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 [6] 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.

References

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

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

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

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

[5] 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.

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

[7] 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/`.

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