Automatic Topology Discovery in WiFi (IEEE 802.11) Multi-hop Ad Hoc Networks

Kiran Mathews

May 1, 2016

1 Introduction

IEEE 802.11 is a set of media access control and physical layer specifications for implementing wireless local area network communication in the 2.4, 3.6, 5, and 60 GHz frequency bands. They are created and maintained by the IEEE LAN/MAN Standards Committee. WiFi, wireless local area networks use these standards to communicate between the systems.
The goal of this project is the design, implementation and extensive testing of a protocol that automatically detects the communication topology of WiFi multi hop ad hoc networks. The protocol is tested on Raspberry pi nodes equipped with standard WiFi adapters. In Section 2, we describe the problem and in Section 3, we define our protocol in detail. In Section 4, we present the results obtained from testing the protocol on the Raspberry pi nodes. In Section 5, we propose features that can be added into the protocol in future, and in Section 6, we conclude our work.

2 Problem Description

A wireless multi hop ad hoc network is a decentralized type of wireless network which may use one or more wireless hops to convey information from a source to a destination. The network is ad hoc because it does not rely on a pre-existing infrastructure, such as routers in wired networks or access points in managed wireless networks. Instead, each node participates in routing by forwarding data to other nodes, so the determination of which nodes forward data is made dynamically on the basis of network connectivity [4]. Thus ad hoc networks are required to be self-organizing and self-configuring, which makes them easy to deploy. Even applications with rather static topologies, such as sensor/actuator networks for industrial automation, can benefit largely from the ease of deployment these systems provide [1].
For effective operation of multi-hop ad hoc networks, communication topology information is vital especially for routing purposes and for clustering of the nodes. Finding proper clusters in a network helps in network scaling, hierarchical routing and energy management improve the reliability and performance of the network. So the idea is to create, implement and test a protocol to detect the communication topology of WiFi multi-hop ad hoc networks when the nodes are powered on, before the actual operation starts. A similar approach for IEEE 802.15.4 was presented in [2].
In our protocol, we assume that each node has a unique ID and nodes are stationary while the protocol runs. One of the main challenges in wireless ad hoc networks is energy saving and it will help to find suitable energy levels for the transmission of each node. The protocol is tested with different transmission power to find the minimum transmission power for an existing reliable link. Another important purpose is that it will also help to reduce the collision of the frames. Reliability of the links will be defined based on the statistics on frame

delivery. Once the topology is detected and distributed, it is ensured that all nodes have consistent topology information.
The input of the protocol is the node ID of the current node. The output of the protocol is a matrix A, where $A[i,j] = 1$ iff there is a reliable communication link from node $i \rightarrow j$, $A[i,j] = 0$ iff there is no link from node $i \rightarrow j$ or $A[i,j] = 2$ iff the link from node $i \rightarrow j$ is unstable.
The matrix shows the communication topology of the network. In the matrix value 1, the parameter denotes the transmission energy level for the link where a is the highest and e is the lowest ($a > b > c > d > e$). In the initial phase, the protocol is tested by setting a constant transmission power which is the highest transmission power depending on the regulatory rules.

3 Protocol Description

This section describes the protocol. Starting by giving an overview about the protocol's mode of operation, this section describes the links types, how each link is determined, and how the protocol terminates.

3.1 Overview

In this protocol, the links are detected by doing statistics on frame reception. A node in the network is statically assigned as the master node such that the master node will decide when the protocol should terminate. Mainly there are two types of messages 1) Topology information messages and 2) Termination messages. The topology information message is used to determine the link quality between the sender of the message and all receivers and also to distribute the topology information within the network. Termination messages are used to start the termination process and to distribute the termination details. Both of them contain the node ID of the sender and a sequence number where the sending node increases the sequence number with each message it sends, i.e. (sender ID, Sequence No) uniquely identifies each message. Apart from the message sequence number, each link has a sequence number which is incremented when the link type changes. Nodes broadcast topology information messages repeatedly within a delay less than $DELAY_{max}$ with the local matrix A until they receive the termination message. When a node j receives topology information messages from its neighboring node i (where $i, j \in N \mid |N|$ = Total number of nodes in the network), the node starts the statistics on the link. After certain processing steps, depending on the Received Signal Strength (RSS) and $lossRatio[i]$, j will set the link type from its neighbors $i \rightarrow j$. $lossRatio[i]$ is an array at node j which stores the *frame loss ratio* of the link $i \rightarrow j$. The calculation steps used to find *frame loss ratio* will be explained in the following sections.

3.2 Link Types and Configuration Parameters

In this section, we introduce the link types and configuration parameters of the protocol. The protocol classifies links as either *Communication link*, *Unstable link* or *No link*.

- *No link* ($A[i,j] = 0$) means that there has been no direct communication from node i to j.
- *Communication link* ($A[i,j] = 1$) means that the node i can send frames to node j which can be received by node j at least with signal strength above RSS_{Min} and the link is stable. In order to be part of the network, the nodes in the network should have at least one communication up and down link. A communication link from a node $i \rightarrow i$ denotes the presence of the node in the network.
- *Unstable link* ($A[i,j] = 2$) means that the link quality of the link from node i to j is changing from time to time, i.e. the link is unstable.
- *Processing link* ($A[i,j] = 3$) means node j still does not have enough data to finish statistics on the link $i \rightarrow j$.

For a node j, let N_{rx}^{j} be the set of neighbors that can send to node j and let N_{tx}^{j} be the set of neighbors that the node j can send to. Variables $RSS_{Avg}[i]$, $lostFrames[i]$ and $receivedFrames[i]$ are used to store the average of RSS, to count the number of lost frames and to count the number of received frames, respectively. To monitor message loss, the protocol uses message sequence numbers in the frames and timeout check. Once a link $i \rightarrow j$ is in *Processing link* state, node j is expected to receive successive frames with an upper bound delay of $DELAY_{max}$. $contLoss[i]$, $timeoutCheck[i]$ and $frameSeqCheck[i]$ are used to store the number of continuously lost packets for the link, the timer to check whether the link is broken and to store the message sequence number of the last received message, respectively. As mentioned earlier, the link types are set depending on $lossRatio$ and RSS_{Avg} value for each link. Both values are considered only if the total number of frames received is greater than $N_{analysis}$. $lossRatio$ for each link is recalculated when a new frame is received or when there occurs a time out. $lossRatio$ value for a link $i \rightarrow j$ is calculated as:

$$lossRatio[i] = \frac{lostFrames[i]}{lostFrames[i] + receivedFrames[i]} \tag{1}$$

Apart from the variables for each link, the protocol has other configuration parameters for measuring link quality. The constants $N_{analysis}$, T_{loss} and T_{cl} are used to set minimum number of frames (received plus lost) required for analysis, maximum tolerable frame loss (percentage) and maximum tolerable continuous frame loss, respectively. Apart from these, each node is expected to join the network within a time limit of $JoinTime_{MAX}$.

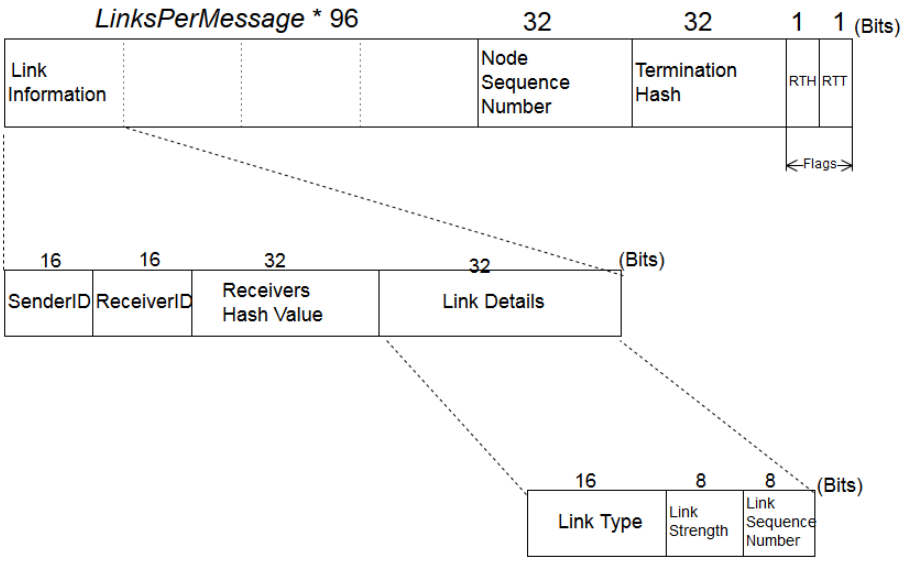

Figure 1: Structure of the *regular frame* used in the protocol

3.3 Frame Structure

In this section, we discuss the structure of the frames used in the protocol. In our implementation, the protocol runs on BiPs communication framework. Fig (1), shows the frame structure of a *regular frame* used by the protocol. A regular frame is sent along with other information like *senderID, checksum ...* etc. used by the framework. The protocol uses the *senderID* information for obtaining the sender ID of the topology information messages. As mentioned earlier, the protocol uses two type of messages, topology information messages and termination messages. Termination messages set the RTT and RTH flags to 1. The value of *Termination Hash* in topology information messages is set to 0. The total size of the regular frame is *LinkPerMessage**160 bits. Next, we explain the role of each field:

- *Link Information* stores the information about the links and is further subdivided into the following fields,
  - *SenderID* stores the source *nodeID* of the link,
  - *ReceiverID* stores the destination *nodeID* of the link,
  - *Receivers Hash Value* stores the hash of the local link matrix of the *SenderID* used for termination (see Section 3.5).
  - *Link Details* stores the details about the link and is further subdivided into:
    - *Link Type* stores the link type (0-3) (see Section 3.2),
    - *Link Strength* stores the average RSS,

    - *Link Sequence Number* stores the sequence number of the link information, in order to identify link updates.

- *Node Sequence Number* stores the sequence number of the frame, in order to monitor frame loss.
- *Termination Hash* stores the hash value that the master decided to terminate (see Section 3.5). This field is used along with termination messages.
- *RTH* flag used to indicate that the sender of the frame has started hashing. Used along with topology information messages and termination messages.
- *RTT* flag to indicate termination messages.

3.4 Protocol Description

In this section, we discuss the algorithm of the protocol. Algorithm (1), is the basic outline of our topology explorer. The protocol listens to the medium when it is not sending any frames and calls the procedure *onFrameReception* when a frame is received. To avoid repeating competition with frames sent by the neighbors, the time between two sent frames varies between $DELAY_{min}$ and $DELAY_{max}$. Nodes analyze the topology information messages from their neighbors to calculate the quality of the links. Moreover, when a node receives a frame from a neighbor, it parses through the frame and updates its local topology information matrix depending on the sequence number of the link. Apart from the sequence number for each link to check the updated information, each frame sent by a node has a sequence number. By comparing the sequence number of a frame with the last received sequence number, frame loss on this link can be calculated. *isNetworkStable()* function is used to check whether all nodes in the network have the same topology information. It compares the hash values sent by all nodes and returns the result. The function *handleTermination()* is used to handle the termination procedure and make sure that all termination criteria are satisfied (see Section 3.5). The *Termination Hash* is the input to the function *handleTermination()*. *handleTermination()* functions sends the termination message with necessary information using procedure *sendFrame*. *calculateHash()* function is used to calculate the hash value of the topology matrix.

Algorithm (2) defines the procedure to send a frame. When the timer for sending a frame expires, it broadcasts its topology information. In the case of the master node, it also checks whether all nodes have the same topology information by comparing the hash values of their matrices. If all nodes have the same topology information, it starts sending uni-cast termination messages to its neighbors. The function *hashStartCheck()* is used to check whether the node can begin hashing. Section 3.5 explains about the termination in detail.

Next, we discuss the process of determining the links in the protocol. Initially, the protocol assumes that all links are of type *No link*. Once a topology

Algorithm 1 Topology Explorer Algorithm

```
1: procedure TOPOLOGY EXPLORER(NODEID, MASTER)
2:     terminate ← FALSE
3:     topologyMatrix [ ][ ] ← 0
4:     rxMatrix[][];                          ▷ information received from neighbor
5:     lastReceived[] ← 0                          ▷ last time message received
6:     lastSequenceNo[] ← 0
7:     receivedFrames[] ← 0
8:     timeoutCheck[] ← 0
9:     frameSeqCheck[] ← 0
10:    contLoss[] ← 0
11:    rxNeighbor[] ← 0
12:    txNeighbor[] ← 0
13:    RSS_avg[] ← 0
14:    TEXFrame f;
15:    sequenceNo ← 0
16:    onframeReception(processMsg(message M));      ▷ run frame reception
17:                                                  process in background
18:    while (terminate ≠ TRUE) do
19:        randomDelay = random(DELAY_min,DELAY_max);
20:        wait(randomDelay);
21:        if (isNetworkStable() == TRUE & nodeId == MASTER) then
22:            terminate ← handleTermination(calculateHash())
23:        else
24:            sendFrame(topologyMatrix);
25:        end if
26:        sequenceNo ++;
27:    end while
28: end procedure
```

Algorithm 2 sendFrame

```
1:  procedure sendFrames(topologyMatrix)
2:      if (nodeId ==MASTER) then
3:          if (isNetworkStable()) then
4:              frame.regFrame.terminationHash←calculateHash(topologyMatrix)
5:              frame.regFrame.RTT ← 1
6:          end if
7:      else
8:          if (hashStartCheck()) then
9:              frame.regFrame.terminationHash←calculateHash(topologyMatrix)
10:             frame.regFrame.RTT ← 0
11:             frame.regFrame.RTH ← 1
12:         else
13:             frame.regFrame.terminationHash ← 0
14:             frame.regFrame.RTT ← 0
15:             frame.regFrame.RTH ← 0
16:         end if
17:         frame.regFrame.linkDetails ← topologyMatrix
18:         frame.regFrame.messageSequenceNo ← sequenceNo
19:         send(frame);
20:     end if
21: end procedure
```

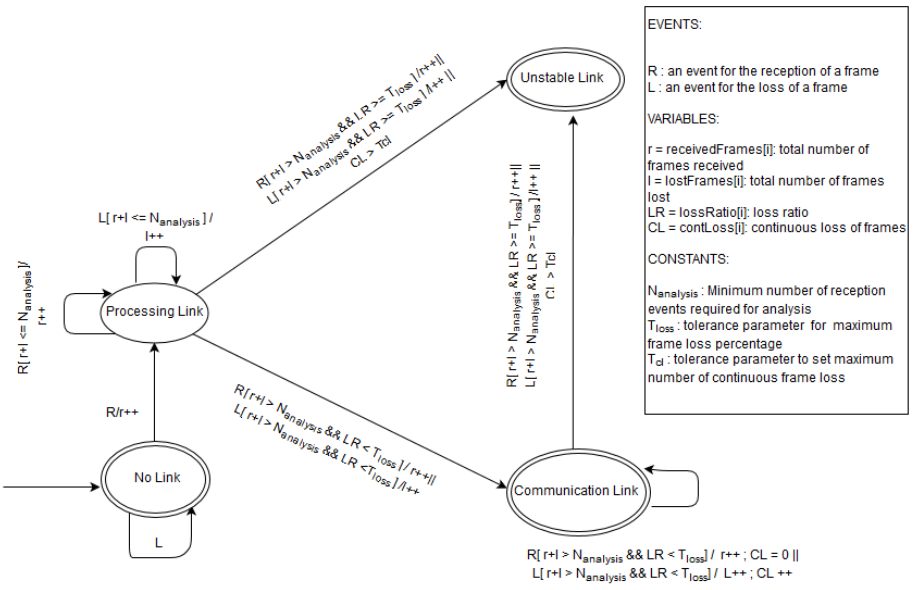

Figure 2: State graph representation of link processing of link $i \rightarrow j$

information message from node i is received by node j, the link $i \rightarrow j$ is set to *Processing link* and the node i also sets the link $i \rightarrow i$ as *Communication link* to denote the presence of the node in the network. The link $i \rightarrow j$ is either set to *Communication link* or *Unstable link* depending on the statistical result done after $N_{analysis}$ number of frames are received (or lost).

Fig (2) represents the state graph for link processing. As mentioned earlier, $N_{analysis}$ is the minimum number of required *event*s to start the statistics of a link. An *event* is either the reception of a frame (R) or the detection of a frame loss (L). A link $i \rightarrow j$ has initial state *No link*. If a topology information message is received by node j, the link state moves to *Processing link* and it waits in that state until $N_{analysis}$ events have occurred. Then, it calculates the *lossRatio[i]* and the RSS_{Avg} for the link. Depending on those values, link state moves to either *Communication link* or *Unstable link* state. If it is in *Unstable link* state, it remains in that state, even if the *lossRatio[i]* falls below T_{loss}. If it is in *Communication link* state, it continues to recalculate the *lossRatio[i]* with every event and if the recalculated *lossRatio[i]* is above T_{loss}, then the link state changes to *Unstable link* state and remains there. Next, we discuss the algorithm (3) which handles the frame reception.

Algorithm (3) defines *processMsg* procedure. Its main purpose is to monitor the medium whenever the node is not sending. It uses each received message to measure the link quality between the sender and updates the link information if there is any new information. It also detects message loss from its neighbors with help of timers and node sequence numbers. When a message is received, the timer function *timersSetExpiry()* will update the timer for particular link with the value $DELAY_{max}$ + *processing delay* i.e. the timer only expires when there is a frame loss. When a frame is received, it checks whether it is a termination message or topology information message by checking the RTT flag. If it is a topology information message, it checks whether its a new link and if it is a new link, the information is stored for termination process. For example, when a new link $i \rightarrow j$ is detected, the node j adds node i into the set N_{rx}^{j}. Topology information messages are also used to measure the link quality and also for updating local topology information about the network. If it is a termination message, it will stop sending topology information message and proceed to termination. The function *isNewlink()* is used to check whether the frame is received for the first time or not.

3.5 Protocol Termination

In this section, we discuss the challenges faced in the termination process and solutions to solve it. As mentioned earlier, the master node decides the termination of the protocol. When the master decides to terminate, the master must assure two things:

Algorithm 3 processMsg

```
1: procedure processMsg(Message M)
2:     if (M.regFrame.RTT == TRUE) then          ▷ Check the message type
3:         terminate ← handleTermination(M.regFrame.terminationHash)
4:     else
5:         if (isNewlink(M.senderID) == TRUE) then
6:             N_rx ← M.senderID
7:         end if
8:         receivedFrames[M.senderID]++
9:         lossTimeout[M.senderID] ← timersSetExpiry(DELAY_max +
10:                currentTime() + processingDelay)
11:        if (M.regFrame.messageSequenceNo ≠ lastSequenceNo[M.senderID]
12:                + 1) then                          ▷ Checking frame loss
13:            lostFrames[M.senderID]←(lostFrames[M.senderID]+
                                       (M.regFrame.messageSequenceNo
14:                                    - lastSequenceNo[M.senderID]))
15:            contLoss[M.senderID] ← M.regFrame.messageSequenceNo -
16:                                    lastSequenceNo[M.senderID]
17:        else
18:            contLoss[M.senderID] ← 0
19:        end if
20:        lastSequenceNo[M.senderID] ← M.regFrame.messageSequenceNo
21:        for (i=0 ; < sizeof(M.regFrame.links) ; i++) do
22:            for (j=0 ; j < sizeof(M.regFramelinks) ; j++) do   ▷ Update the
23:                                                  link information inside the message
24:                if (M.regFrame.linkMatrix[i][j].sequenceNo >
25:                        TopologyMatrix[i][j].sequenceNo) then
26:                    TopologyMatrix[i][j].linkType ← M.regFrame.
27:                                                  linkMatrix[i][j].linkType
28:                end if
29:            end for
30:        end for
31:        totalFrames← lostFrames[M.senderID]+receivedFrames[M.senderID]
32:        RSS_avg[M.senderID] ← (M.RSS + RSS_avg) / totalFrame
33:        if (totalFrames≥ N_analysis) then
34:            lossRatio[M.senderID] ← lostFrames[M.senderID] / totalFrames
35:            if (lossRatio[M.senderID] ≥ t_c ‖ (contLoss[M.senderID] ≥ T_loss
36:                        ‖ RSS_avg[M.senderID] ≤ RSS_Min) then
37:                TopologyMatrix[M.senderID][nodeID].linkType ←
38:                                                  Unstable Link
39:                TopologyMatrix[M.senderID][nodeID].sequenceNo ++
40:            else
```

Algorithm 3 processMsg(continued)

```
41:                 if (TopologyMatrix[M.senderID][nodeID] ≠
42:                       Unstable Link) then
43:                     TopologyMatrix[M.senderID][nodeID].linkType ←
44:                                                   Communication Link
45:                     TopologyMatrix[M.senderID][nodeID].sequenceNo ++
46:                 end if
47:             end if
48:         end if
49:         if (contLoss[M.senderID] ≥ T_cl) then
50:             TopologyMatrix[M.senderID][nodeID].linkType ← Unstable Link
51:             TopologyMatrix[M.senderID][nodeID].sequenceNo ++
52:         end if
53:     end if
54: end procedure
```

1. The topology is stable and
2. All nodes in the network have the same topology information.

To tackle this problem, all nodes calculate the hash value of their current topology matrix and send this value with termination messages. Nodes only start calculating the hash value of their local topology matrix once all the links it analyzes are not in *Processing link* state i.e. for a node j, $\forall i \mid i \in N_{rx}^{j}$, with links $i \rightarrow j$ are either in *Unstable link* or *Communication link* state. The master node monitors the hash values of all nodes and once all these hash values are the same, it starts to send the termination message. Termination messages are uni-cast messages, i.e. each sender waits for the acknowledgment from its neighbors and then terminates. This ensures that each node receives the termination message.
Nodes that receive the termination message compare the termination hash with their local hash value. If the node has a different hash, the node should roll back the topology information that corresponds to the received hash. Once a node receives a termination message, it should ensure two things before terminating:

1. a node should forward the termination message to all neighbors, i.e. for a node i, it should forward the termination message to all links $i \rightarrow j$, $(\forall j \mid j \in N_{tx}^{i})$
2. a node should receive termination message from all neighbors, i.e. for a node j, it should receive the termination message from all links $i \rightarrow j$, $(\forall i \mid i \in N_{rx}^{j})$

Apart from this, a node terminates if any of the following condition occurs:

- if any one of the node in the network cannot communicate with the node. For a node j, the node terminates if $N_{rx}^{j} = \emptyset$. In such a case, a node j sends

the topology information by updating the link $j \rightarrow j$ as *No link* which denotes the node j is inactive. Also, it sets the links where the node j as either a sender or receiver as *No link* and broadcast the information to its neighbors. Since topology information messages are broadcast message, there is not guarantee in the reception of the message. Thus, topology information messages sent before the termination of a node may not be received by any other node in the network. Since the node has terminated, the links where a node j as the sender will change into *Unstable link* state due to continuous frame loss. The master node handles such nodes by eliminating them from the network. For this, the master node verifies whether every node in the network had a communication up and down link before starting the termination process.

- if it cannot communicate with any one of the node in the network, i.e. the links from the nodes in set N_{tx} are unstable links ($N_{tx}^{j} = \emptyset$). In such a case, the node follows the same procedure explained in the above case. In this situation, the probability for receiving the topology information messages with the node dead indication is less since the links to its neighbors are unstable. But the nodes will identify about the node death due to continuous frame loss.

- if it cannot detect the master node after a certain time limit. Currently, it verifies the presence of master $JoinTime_{MAX}$ seconds after hashing is started. A more precise analysis of end to end delay for the exchange of link information is needed to be calculated for finding a proper upper bound.

4 Experiments

The protocol was evaluated by simulating different topologies. Eight WiFi adapters were connected to an extended USB2.0 port HUB. Fig (3) shows the topology used in the experiment. The reason for choosing this topology is because it covers most of the test cases. For example, before termination a node should make sure that it has forwarded the termination message to its neighbors (i.e. nodes in the set N_{tx}) and received termination message from all its neighbors (i.e. in the set N_{rx}). In the selected topology, node 16 should check whether it has received a termination message from nodes 18 and 19, and also should forward this termination to nodes 15 and 18. Similarly, node 15 should check whether it has a received termination messages from nodes 4, 16 and 19, also should forward this termination to node 4,18 and 19.
For the experiment, node 6 was assigned as the master node and simulated with different configuration parameter settings. Table (1) shows the parameter values for different settings. Configuration 1 was mainly used for less complicated topologies. Configuration 2 was used for simulating the main topology as shown Fig (3). The reason for choosing more relaxed configuration 2 is because the WiFi adapters were placed so close with a few centimeters apart and chance of

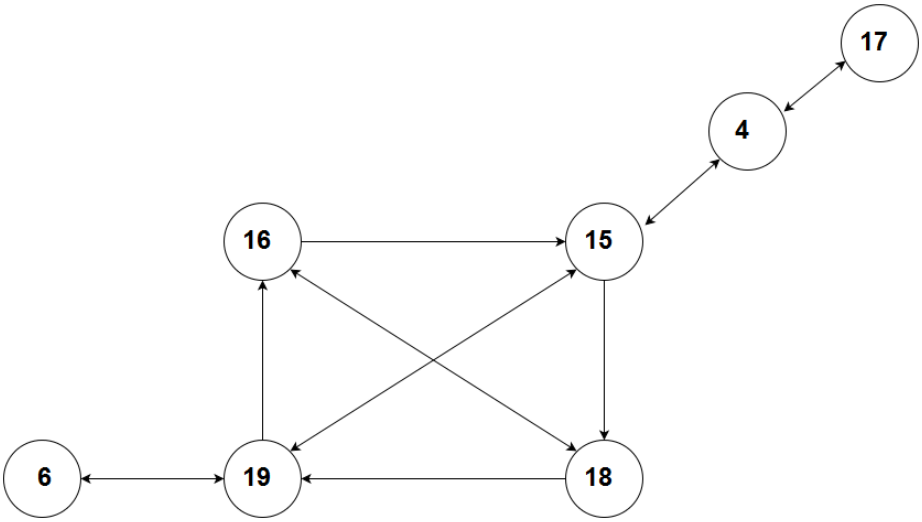

Figure 3: Topology used for most of the experiments

collision is very high.

Parameters	Configuration 1	Configuration 2
$N_{analysis}$	15	30
T_{loss}	15%	30%
T_{cl}	5	10
$JoinTime_{MAX}$	5 sec	5 sec

Table 1: Values of the configuration parameters for each configuration

During our first test of the protocol, we saw nodes crashing after the termination has started which was due to a bug. But in real world, nodes could also crash, e.g. due to empty battery. The nodes can crash due to different reasons in real world scenario, but in the testing period the nodes crashed due to the bug in our messaging system. The protocol can detect the node crash during the run of the protocol, but the detecting node crash once the termination process has begun was not possible in this version. For example, consider the topology in Fig (3) and the situation where node 15 crashes during the termination process. Node 16 sends the termination message to its neighbor nodes 15 and 18. Node 16 waits for the termination acknowledge from node 15 for a certain time period and terminates. But that's not the case with node 18. Node 18 expects the termination 15 and unaware of the fact that node 15 has crashed. Since node 16 already sent node 18 a termination message, it will ignore the

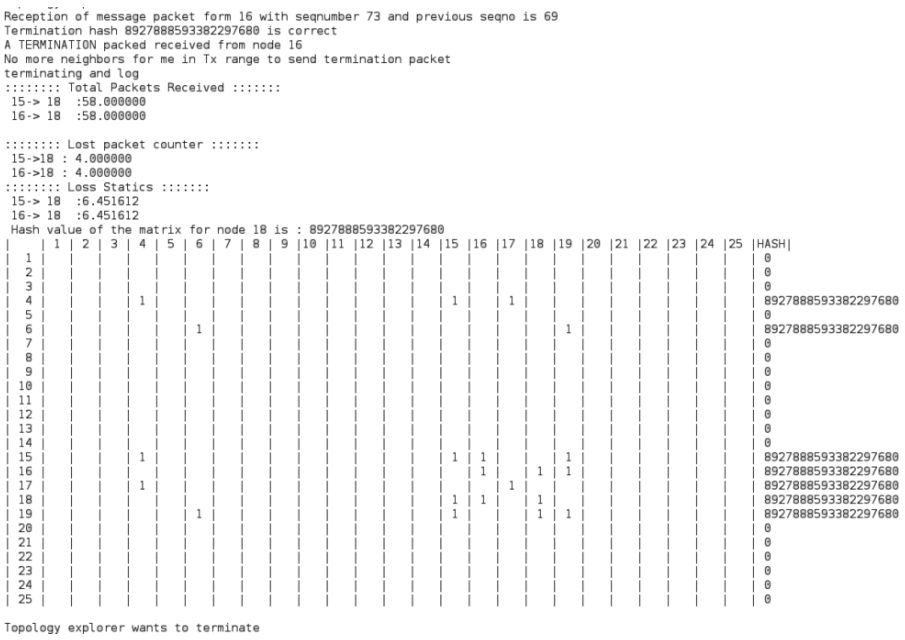

Figure 4: A sample output

topology information messages from node 16. In such case node 18 will remain as unterminated.

The protocol was simulated by setting the transmission power to the maximum power allowed by the regulatory rules which is 30dBm. But the protocol was simulated by setting the WiFi adapters to operate on different channels. The end result depends on the traffic on each channel and depends on the environment. The run time of the protocol depends on the configuration parameter $N_{analysis}$ and data rate allocated for each node in a network. As $N_{analysis}$ is larger, the protocol need more frames to calculate the link quality and increases the run time of the protocol. Parameters T_{cl} and T_{loss} can be used to adjust to the desired link quality. Fig (4) shows a sample output of node 18 in the simulated topology. It also shows the stats about the links in the network where the node 18 is the receiver. The first row of the topology matrix indicates the sender node ID and the first column indicates the receiver node ID. For example, a total of 58 frames were received and 4 frames were lost for the link $15 \rightarrow 18$ and has loss ratio of 6.45%, which is below the tolerance value and results in *Communication Link*.

5 Future Work

In this section, we discuss future work that can be done to optimize the protocol. One thing that must be done is the implementation of the solution for detecting crashed nodes after the termination process has begun. One way to overcome this issue is by continuously (in all phases) checking the routing to every node. If there is a route to every node in the network, it implies that nodes can forward and receive a termination message. It can be done by implementing Dijkstra's path finding algorithm [3] for repeatedly checking the route to every node. Another possibility is to continuously monitor the links with some kind of *ping*, and to notify the network with *LinkError* messages, if a link is broken. If all links from a node is reported as broken, then it leads to the conclusion of crashed node.
Another concept to be implemented is the method to transmit frames with different transmission power. It will help to choose the least possible transmission power for sending frames between nodes. Apart from those important updates, more upgrades can be introduced to optimize the protocol.
Current version assumes that when the protocol runs, the nodes are stationary and the protocol terminates with a stable topology information of that particular time. The protocol can be extended for dynamic topology exploration, i.e instead of terminating with a topology information, it will update the nodes with latest topology information. For this, each node have its local topology information, and global topology information agreed by all other nodes in the network. The master node decides the network topology by comparing the hash values of each node and updates the nodes with new network topology.

6 Conclusion

The main goal of this guided research was to come up with a protocol to detect the topology of the network with nodes connected via IEEE 802.11. We came up with a protocol, which terminates with consistent topology information among nodes. One of the main tasks was the termination of the distributed protocol, which was solved by introducing a master node to compare the hash values of the topology matrix of each node. As a part of guided research, the protocol was implemented and tested.

References

[1] Johan Åkerberg, Mikael Gidlund, and Mats Björkman. Future research challenges in wireless sensor and actuator networks targeting industrial automation. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 410–415. IEEE, 2011.

[2] Christopher Kramer, Dennis Christmann, and Reinhard Gotzhein. Automatic topology discovery in TDMA-based ad hoc networks. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2015 International*, pages 634–639. IEEE, 2015.

[3] S Skiena. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pages 225–227, 1990.

[4] A Srinivas, G Vasavi, B Kavitha Laxmi, and K Rama Krishna. Interference revelation in mobile ad-hoc networks and confrontation.