Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Telecommunication Networks

Protocol Programming Lab

Development of a leader election algorithm in a two hop network with respect to wireless environment

Authors: Yannick Klose (381898)
Tim Schröder (382094)

Hand-in: 06. August 2021

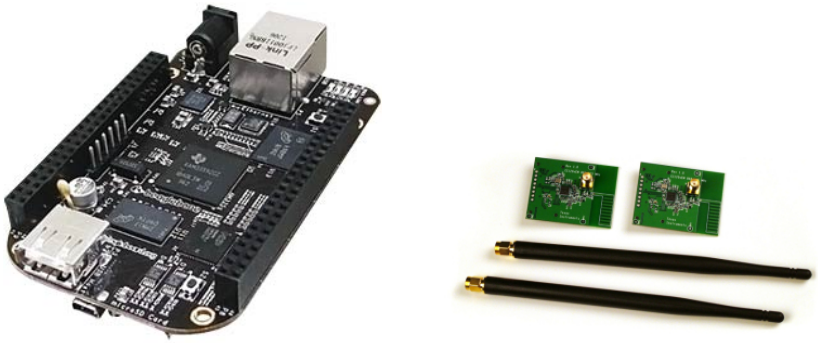
Contents

1 Idea

Wireless networks offer a wide range of communication possibilities and use cases. An important and essential part of this is the creation of a common consensus. This is often realized by a leader election, where a leader is determined after a predefined fair procedure. The leader then has certain privileges like the hosting and administration of databases etc. In this project we would like to implement this task in a wireless environment. Wireless environments have the great advantage that they do not require any infrastructure and are very flexible. Nevertheless, not only advantages occur with the implementation in wireless environments. In wireless environments, range or overloaded communication channels can lead to problems. In our project we would like to present a method how Beaglebones, equipped with a CC1200, can form a wireless network that autonomously performs a leader election and creating two hop system using forwarder mechanisms. The network should also function in the event of malfunctions and have redundancy against the failure of nodes. As soon as nodes fail, the network automatically regenerates itself and provides a new leader. If nodes are out of range of the leader, the messages are forwarded accordingly. To avoid overloading the communication channel, we have developed an alogrithm that performs RSSI-based forwarder determination. Since we do not have a central node acting as a server, our implementation is similar to a peer-to-peer network.

2 Setup

In the following section the setup for the Beaglebone and CC1200 is described (figure 1). The Beaglebone Black is from Texas Instruments and equipped with the Debian Linux operating system. The CC1200 (also by Texas Intruments) is used as a communication device that has a high performance RF transceiver included. In a first step the Beaglenbone has to be installed properly by using the instructions given on this website TKN PPL-LAB. For this project it is recommended to have 3 nodes, where each node is a Beaglebone with CC1200.



(a) Beaglebone Black (b) CC1200

Figure 1: Used Setup in the project

The project is stored on a Github repository and therefore has to be cloned on the beaglebone. This can be done by executing:

```
git clone https://github.com/yklose/RaftAlgo
```

In a next step the variable number of nodes has to be adjusted in the variables.h file. This is the only parameter that has to be adjusted for each scenario. In a standard configuration (with 3 nodes) the number of nodes is set by default to 3. After setting the parameter the code has to be compiled by executing:

```
cd /build
cmake ../.
make
```

Now everything is configured and each node can be started (at any time). To start the node please run the command:

```
./raftalgo
```

3 Development

The following section describes the development of the protocol for leader determination in a two hop process. First, the leader election is discussed and then the extension by a forwarder determination. For the implementation we are only using one frequency band at 876MHz with a transmit power of 14mW. A further discussion of using multiple frequency bands as well as an adjustment in the transmit power is shown in section 5.

3.1 Leader Election

Determining a leader in a group of nodes raises many challenges. How can we ensure that all nodes have the same leader? How do you determine a fair leader? What happens if a leader fails? Here we introduce how we achieve consensus in our system.

Over the years, many researchers have studied this topic and come up with a wide variety of methods and algorithms. A very well known algorithm is the Paxos algorithm or the slightly modified Raft algorithm. This algorithm is able to perform a fair leader election, in which at any time at most one leader is determined. This leader must be elected by at least half of the nodes. This ensures that the leader reaches at least half of the nodes. The only requirement is the knowledge about the total number of nodes in the network. However the Raft Algorithm was designed for an IP based 802.11 communication and not a broadcasting one on the frequency level. Therefore we adjusted the Raft Algorithm for out needs.

3.1.1 Raft Algorithm

In the classic raft algorithm there exists 5 different message types. We will call them: PROPOSE, ACCEPT, DECLINE, LEADER and OK. Furthermore there exists 4 states: OPEN, FOLLOWER, PROPOSER and LEADER. The process of a leader election using the Raft Algorithm can be seen in Figure 2.

Each node starts with an initialization of a random timeout (Step 1). During this interval the node is open to receive packets from other nodes. Therefore during this period the state is defined as OPEN. When the node does not receive any message during the interval, the node will send a PROPOSE message (Step 2), indicating to candidate as a potential leader and therefore change their state to PROPOSER. When other nodes receive a PROPOSE message, they first check if their state is OPEN. If this is the case they will send an ACCEPT message (Step 3). The status changes respectively from OPEN to FOLLOWER. If the state happens to be FOLLOWER or PROPOSER, the node will send a DECLINE message. The PROPOSER constantly checks whether it has the majority of ACCEPT message. By reaching a majority the node will set its state to LEADER. During the leader interval the LEADER will send heardbeat messages, indicated as LEADER messages (Step 4). The FOLLOWERS will reply to these with a OK messages (Step 5). In contrast if a node reaches

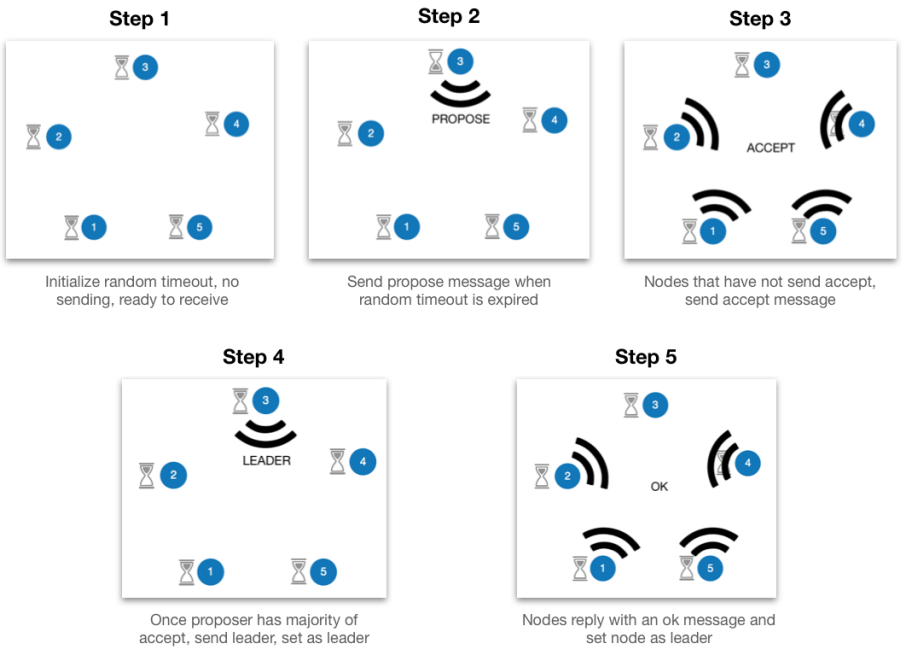


Figure 2: Example of Raft algorithm for 5 nodes at initialization

a majority of DECLINE messages it will set its state to OPEN again. Each time a node receives any packet, the timeout counter is reset. A more detailed description of the LEADER and FOLLOWER mode is made in section 3.1.2 and 3.1.3. The whole process of the leader election is shown in figure 2.

3.1.2 Node in LEADER mode

In case of a general consensus, the winning node of the leader election switches to the LEADER mode. This section describes the features of the Leader mode visualized in figure 3 more detailed. The LEADER mode has a fixed heartbeat interval of 50ms. During this interval the node is in receive mode (RX) and waits for the responses of the LEADER messages, described by the OK messages. This allows the leader to be aware of the number of unique nodes within its communication range. The Leader can differentiate the messages by reading the ID within the OK message. The exact message format of all messages are described in section 3.3. Each unique message ID is written into the local list of the leader. Later this local leader list will be important to figure out which nodes need a forwarder. The process of forwarding is described in detail in section 3.2 and shall be summarized here only briefly. The leaders task in the forwarding process is to determine the most suitable forwarder. Therefore the leader first has to send its local leader list to all nodes (within its communication range). If one of the follower nodes detects that there is a node within its

communication range that is not in the one from the leader, it will send a REQUEST FORWARD message including the RSSI of the node the follower wants to forward. The Leader stores these forward request messages and selects the Forwarder according to the highest RSSI.

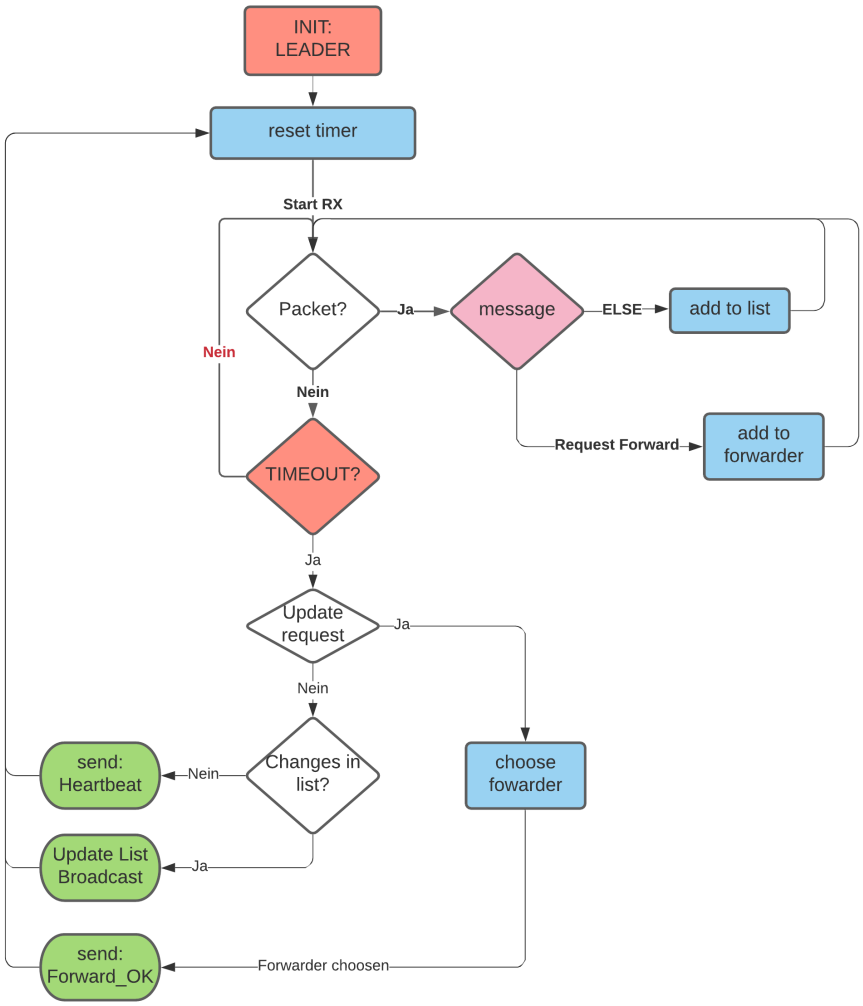


Figure 3: Flowchart of LEADER MODE

3.1.3 Node in FOLLOWER mode

Now we will look at the functionality of the follower nodes. Beside providing a common consensus, which we discussed in 3.1, the follower nodes are responsible for the forwarding of nodes that are not in the range of the LEADER. In this section we will review the responsibilities of the follower nodes. The details of the technical implementation follow in the next chapters.

As in LEADER mode, a random timer defines the interval of receiving messages from other nodes. The random length of the timeout is in the range of 400 to 1000ms. During this interval the CC1200 is in receive mode (RX). When receiving a message the first step is to check its validity by using the checksum. If the message is considered a valid, the message is processed according to its message type. If a node does not receive a message during the interval the node sends a propose message and changes its state from OPEN/FOLLOWER to PROPOSER. This starts a new LEADER election. To understand the different message types, let's now assume that we are a node in the network, our timer has expired without receiving any messages, and we send PROPOSE message. By chance another of the 5 nodes in the network had a similar timer and sent a PROPOSE message before our PROPOSE message reached them, which we will receive. Since we are not in the state OPEN, but PROPOSER, we send this node a DECLINE message back. After that the messages of the other nodes arrive. Two ACCEPTs and two DECLINES including one from the other PROPOSER and a node, which has received the other PROPOSE message first. At each ACCEPT or DECLINE we check whether we have the absolute majority or minority. When we have the majority of ACCEPT messages we can switch to the LEADER mode, with the majority of DECLINE messages we switch back to the OPEN mode. In our example with two ACCEPTs, we can switch to LEADER mode, because we vote for us as well and thus stand at 3 out of 5 votes. As described in the previous section the leader node is sending heartbeat LEADER messages in predefined intervals. The follower nodes will send an acknowledgement OK message, after they received the heartbeat. It should be noted that the OK messages are not sent directly, but only after a short random delay to avoid collisions. More details will be described in 3.3.2.

3.2 Forwarder Determination

In wireless networks, range can often be a problem. While it is possible to increase the transmitter power on the one hand, on the other hand this causes high interference and consumes more energy. For this reason, many wireless protocols use so-called forwarders to cover large distances. These have the purpose of forwarding the traffic of nodes that are normally out of range. This allows the transmission power of the individual nodes to be reduced.

In Figure 5 we can see a simple scenario where the node 4 is not in the communication range of the leader and therefore does not receive any messages. However node 2 and 3 can both see node 4. We implemented an algorithm for a two-hop method that allows individual nodes to be selected as forwarders. The

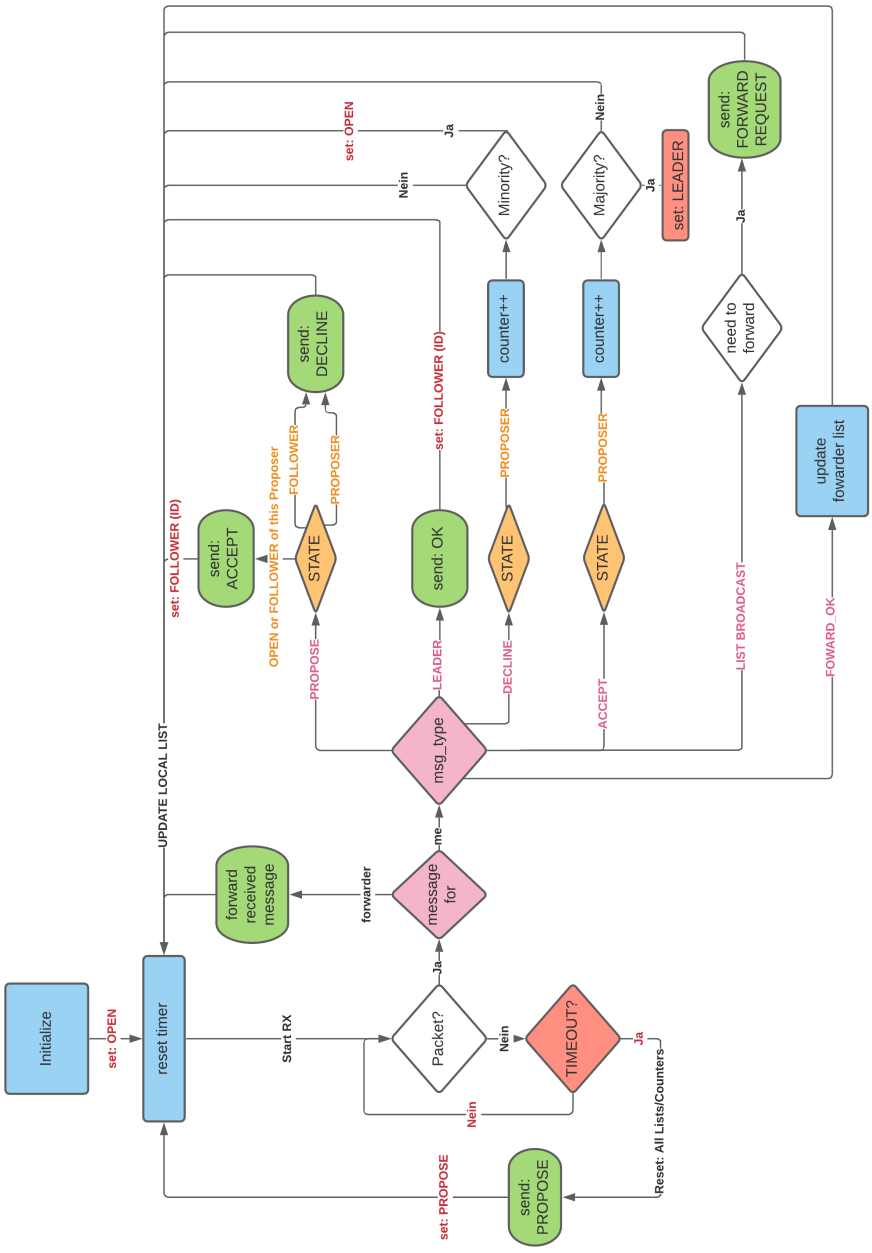


Figure 4: Flowchart of FOLLOWER MODE

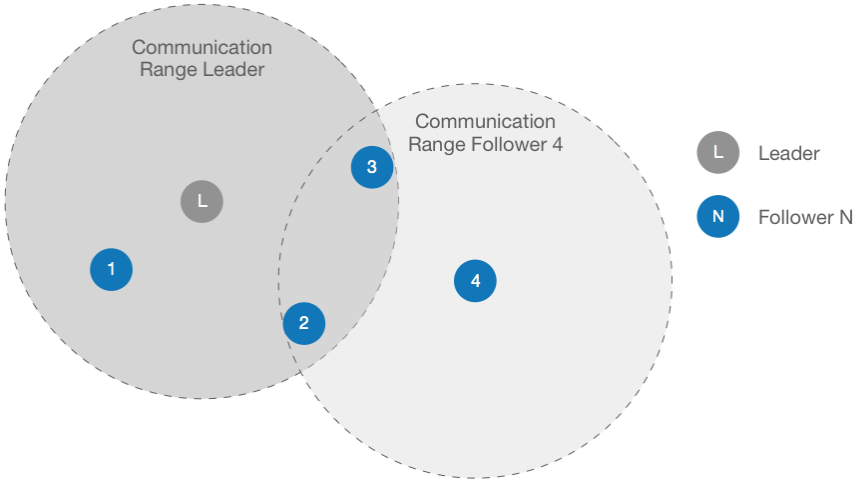


Figure 5: Example of a forwarder situation

determination is guided by the leader and ensures that only one node is chosen as a forwarder at a time. Figure 3 and 4 we can see the implementation of handling the forwarding from the point of view of the leader and the follower. The general idea is that each node maintains a local list of nodes that it can see. The local list is updated by reading all message, even though their will not be processed. Furthermore the leader broadcasts his local list to all nodes in his range. If on the local list of the individual node is an ID that is not on the one of the Leader, the node will send a request to be a forwarder for this specific ID. In order to pick the best node, the request forward message includes the RSSI from node to forward. The leader then waits for 3 Leader iterations (total 150ms), then compares the RSSIs and chooses the node with the highest RSSI of the node to forward. In the example in Figure 5 this would mean that node 2 and 3 send a request and the leader would then choose node 3, because it has a higher RSSI (e.g. is closer in this configuration). By using this type of forwarder determination we can ensure that only one node will forward the data for node 4 and therefore reduce the overall load compared to a native approach where all nodes would forward the data. Furthermore this implementation is robust against node failures. When node 3, being the forwarder for node 4, would fail, the leader would not receive messages from 3 and 4. This would cause that the leader sends in the next broadcast interval not node 1,2,3 and 4 in the list but only node 1 and 2. Then node 2 would send a forward request message for node 4 and becomes the forwarder.

3.3 Implementation in wireless networks

The Raft algorithm was implemented for IP based system. In our setup we are not using IP and the 802.11 protocol and therefore needed to adjust and the existing algorithm. The changes made for our wireless CC1200 setup are documented in this section.

3.3.1 Data types

Since IP headers are not feasible for our application, we first of all had to develop a method to transmit messages to specific nodes. In order to implement the Raft algorithm we came up with a specific data structure and added some features to the existing algorithm. We first assign each node a unique randomly generated ID (more in 3.3.3). Next we define a data structure shown in figure 7 that allows to distinguish different message types as well as different senders and receivers. The type describes the message type, encoded in an integer. A full overview of all message types can be found in table 1. The Receiver ID indicates the ID where the message should be send to, the sender ID the ID it came from. This ensures that messages can be addressed individually. For certain message types such as propose or Leader message the receiver ID is not relevant, since all nodes should receive this message. Regarding broadcast messages we have to consider that the message lengths can be longer than two ids. Therefore the datatype has to be dynamic, as shown in figure 8. Furthermore the datatype of the request forward is defined as shown in Figure 9. When a node is selected by the leader as the forwarder the datatype described in Figure 7 is used. To make sure that not corrupted messages are processed, we use in our data structure a checksum that is the modulo of the sum of all send bits. In order to process the message the node first reads the type and then decide based which datatype should be used to read the rest of the message. Furthermore the node can directly decide whether the message is important or if the message can be discarded. E.g. for a node being in LEADER mode propose, accept and decline message are not relevant. For a node being in follower mode the ok messages are not relevant.

Figure 6: Datatype for Accept, decline, ok and forward ok messages

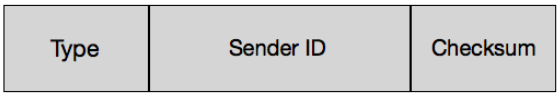


Figure 7: Datatype for propose and leader messages

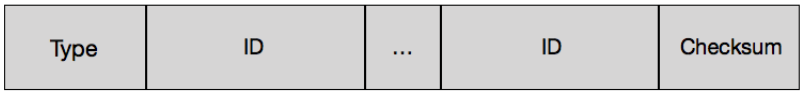


Figure 8: Datatype for list broadcast messages

Figure 9: Datatype for request forward messages

Encoding	Type name	Description
1	PROPOSE	Node wants to become a leader
2	ACCEPT	Node accepts the propose message
3	DECLINE	Node declines the propose message
4	LEADER	Indicate that node is the leader
5	OK	Response to leader message
6	FORWARD_OK	Chooses Forwarder (from leader)
7	REQUEST_FORWARD	Node wants to become forwarder
8	LIST_BROADCAST	List of nodes the leader reaches

Table 1: Type Encoding for messages

3.3.2 Timing

Timing is incredibly important in our application, this concerns both the leader election and the heartbeat message, as well as the handling in case of failure of nodes. Therefore, this topic will get its own section in this report.
The first timing starts immediately after the first initialization, because at the very beginning a random timeout is generated. As a reminder, when this timeout expires, a propose message is sent out and the leader election begins. Timing plays a crucial role already here, because if the timers are close to each other, several PROPOSE messages may be sent out by different nodes. Since it is a random selection of the timeouts, we can only limit this problem by increasing the allowed span of the timeouts. Of course, this is not possible indefinitely, because the leader selection should of course be as fast as possible. In addition, we can generate a new random one after each timeout, so we avoid that one random setup can corrupt the whole system. In the original Raft algorithm these timeouts are 150-300ms. The choice of timeout length is furthermore strongly linked to the choice of leader heartbeat interval length.
At least one heartbeat is expected in the follower timeouts. It must be taken into account that sometimes a message may not arrive, but also that we use the transmission channel economically and do not waste capacity. To find the optimal follower timeouts and heartbeat intervals, we implement our system with different time spans and investigate the effects. The results of this experiment can be found in Figure 10. It can be seen that different heartbeat intervals are represented over certain follower timeouts. In the graph, the two main findings are marked in the corresponding sections. First, there are unwanted propositions from followers when their timeout is below 250ms seconds. Second, if the leader heartbeat is below 25ms, the list broadcast may not be transmitted properly in some cases. With these results, we find the optimal heartbeat interval at 50ms and the follower timeouts between 400-1000ms.

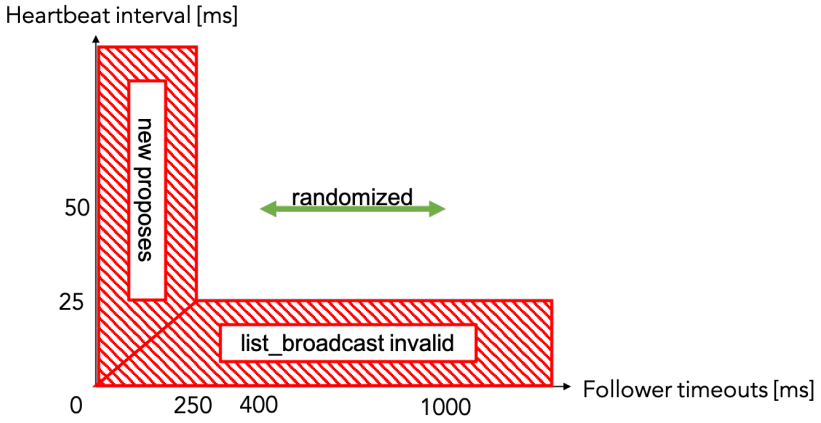


Figure 10: Problems with message timing

3.3.3 Random numbers

Our modifications of the Raft algorithm rely heavily on generated random numbers, first for the IDs of the Tods and second for the random delays. At first we generated the random numbers technically, but during testing we found out that in very rare cases different nodes have the same IDs or delays. This is due to the random seed, which depends on the computation time. This leads to a big problem, because without unique IDs individual nodes can not be addressed unambiguously. To avoid this, we use the hardware of the CC1200 to generate random numbers. One can read a random 8 bit number from the register RND_GEN. Additionally you can XOR the result with the local RSSI.

$$rand\_int = RND\_GEN \oplus RSSI(8Bit) \tag{1}$$

3.3.4 Error handling

Even though we tried to implement many features like random numbers, timing parameters and checksums to avoid collisions and uncontrolled behaviour, in every real world scenario these will occur. Therefore we implemented special features to resolve collisions and handle invalid messages.

One problem that we encountered during the development was that after receiving an invalid message the nodes counter was reset, leading to endless loops with no progress. Therefore we adjusted the protocol in such a way to ignore any invalid message, which we detect with invalid checksum. Furthermore we made the leader message an override message, meaning regardless of the state of the node, whenever a leader message is received, the node will reset itself and sets itself as a follower. Finally one issue that we encountered was related to random timeouts. Even though we made the random timeouts random, there is still a chance that two or more nodes have a very similar timeout and will be stuck in a propose loop. That means each node sends a propose message at the same time and declines the others propose message. Since there is no majority, this will go forever. Our approach here is to build a random "give up" counter. This timer sets a upper bound of a maximum number of decline messages. When this counter is expired the node will be open for one time slot, allowing to accept the propose from the other node. Using these techniques we were able to handle most of the errors and achieve progress of the protocol.

3.4 Debugging and Testing

In order to be able to test the individual functions during the development process, we used the interface of 4C. This allows us to send our own messages to the Beaglebones and thus artificially simulate a large number of nodes. For this we have to make sure that the configurations are chosen correctly and that the checksum is correct. In figure 11 you can see the 4C interacting with the two Beaglebones (shown by the two terminals). The left terminal represents the leader, sending the heartbeat messages. The right terminal shows a follower, responding to the heartbeat messages. By sending an artificially created forward request message, it can be observed that the leader (left terminal) receives the forward request message and also chooses a forwarder. Therefore the artifically created node would now forward all data that is addressed to the forwarding node as well as rebroadcast all broadcast message such that the forwarding node receives all messages. When debugging the code with 4C one has to make sure that the lists are not reset after a timeout occurs, because with 4C we have a much lower reaction time compared to the protocol.

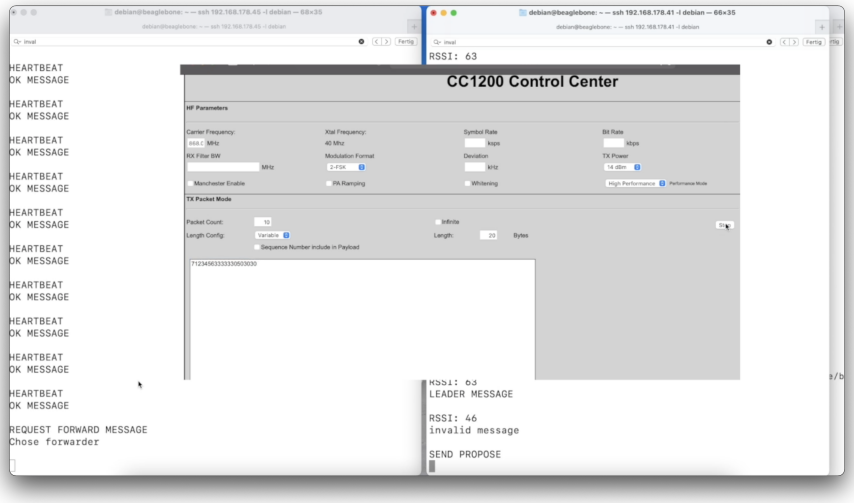


Figure 11: Interaction with two Beaglebones (using our protocol) and 1 Beaglebone using 4C to inject packets

4 Results

In order to measure the performance of our protocol, we used a configuration where 3 nodes tried to elect a leader. We started each node randomly and observed if the nodes would agree on a leader. The first observation was that often the IDs of the nodes were the same and therefore the protocol crashed. This was the reason why we implemented hardware generated random numbers based on the observed noise of the individual beaglebone. The second observation was that sometimes there were many collisions at the leaders node. This was caused by the fact that for sending the data there was no random timer implemented. The random timer for sending the data has to be dependent on the time a node takes to read a message from the fifo and eventually respond on this message. We therefore measured the processing time for each incoming message. The result is shown in table 2.

Incoming packet	busy time	response with message
PROPOSE	7ms	Yes (Accept/Decline)
ACCEPT	3ms	No
DECLINE	3ms	No
LEADER	6ms	Yes (Ok)
LIST BROADCAST	3ms/8ms	No/Yes (Request)
FORWARD OK	3ms	No
REQUEST FORWARD	3ms	No

Table 2: Busy time for different incoming packets

What generally can be observed by the results in table 2 is that the busy time for incoming packets that require no response message are 3ms and incoming packets that do require a response from 6-8ms. This means the worst case a node is busy for 8ms before being able to receive a new message. Therefore our timing delay for each message should be in the order that message are send with a delay of approximately 10ms each. However since we are not in a synchronous model we use probability to fix this issue. The timing delay for each message is therefore defined in section 3.3.2, where the random numbers are again computed by the hardware.

5 Improvements

While the implemented protocol tries to solve many issues, there still remain some points that can be improved. A main issue that will be discussed further is scalability. Since we are using one frequency channel the higher the number of nodes the higher are the random timeout for the individual nodes. This means it will take on average longer to receive a message and the protocol gets slower. Furthermore the probability of collisions increases. One way to solve such issue is to use multiple frequency channels as well as using two antennas to transmit and receive messages at the same time.

Another improvement addresses the issue of agreeing on a leader. While we implemented a protocol that requires 50% of positive responses it could happen that the topology is unfavourable for this protocol. When no node is able to reach at least 50% of the nodes the consensus is not possible which would result in an endless loop. A possible solution for this problem would be to individual adjust the transmit power in order to reach more beaglebones while keeping the interference low for those who are close to each other with a lower transmit power.

Lastly an important improvement for real world implementations is security. Currently message are not encrypted. Furthermore the protocol is not resistant against Byzantine behaviour. While the first problem can be solved by encrypted messages, the last problem is a general problem of the Raft Algorithm. That means that our algorithm can only be used in a scenario in which we trust all other nodes, like an internal network.

Overall, we have developed a protocol that is able to organise itself and thereby perform a leader determination and automatically determine forwarders if necessary. We have considered many problems such as interference, scalability and failure of nodes. However, in order to develop a robust protocol, some improvements as listed above are necessary.