To Hop or Not To Hop: Relay Protocol Design

Wireless Sensor Networks & Electronics [TI-WSNE]

Wireless Engineers, Mini Project, Q4 2017

Mads Møller (MM) au501466
Mathias Jessen (MJ) au500070
Nina Jensen (NJ) au574105
Søren Hansen (SH) au499306

May 31, 2017

Contents

1 Introduction

In this project, we consider a Wireless Sensor Network (WSN) with topology as shown in fig. 1.1. Here, node A needs to transmit packages to node C. Specifically, node C could be at an unfavourable position where it suffers fading and is unable to receive packages from node A. The roles of nodes in this network topology are described in section 3.3.

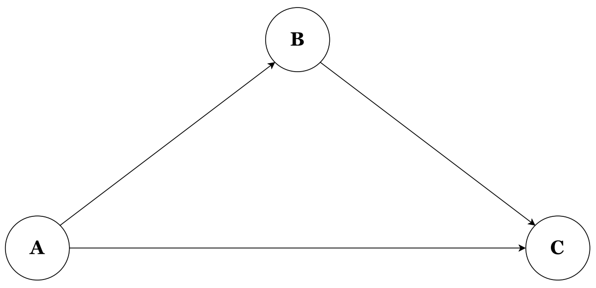

Figure 1.1. WSN with three nodes

This scenario gives motivation to use a relay node B, to increase the successful reception rate at node C. However, this set-up introduces extra energy cost at node B. This project focuses on scenarios wherein reliability must be 100%, i.e. node C must receive all packages sent, and power usage must be minimised. Throughput and latency are also worth considering. Scenarios are further described in section 3.

Throughout this project, TelosB motes are used for practical implementation.

Considering the network topology in fig. 1.1, several assumptions are made. These are discussed in sections 3.1 and 3.2. Some preliminary measurements were made; these are described in section 2.

A protocol for when to relay using node B has been designed and implemented. It is described in section 4, where design considerations, simulations, and implementation are described in detail. Finally, the protocol is tested in a practical setting, such that comparisons with the simulations can be made.

Finally, discussion of and conclusion on the results obtained in this project can be found in sections 5 and 6. The code implemented on the nodes and the code used in the simulations are available in appendices A and B.

2 Preliminary Measurements

To model current draw in a WSN as shown in fig. 1.1, several node measurements were made. It was decided to measure the following current draws:

- Idle listening
- Receiving
- Transmitting with varying power

The code used to obtain these measurements is listed in appendix A. In particular, idling is measured using code from appendix A.1, while receiving uses code from appendix A.2. Finally, the code allowing us to transmit with varying power is listed in appendix A.3.

To perform the measurements, the circuit displayed in fig. 2.1 is used. Using an oscilloscope, the voltage is measured over a 1Ω shunt resistor. This reading gave a voltage measurement that was then converted into a current through the resistor using Ohm's law.

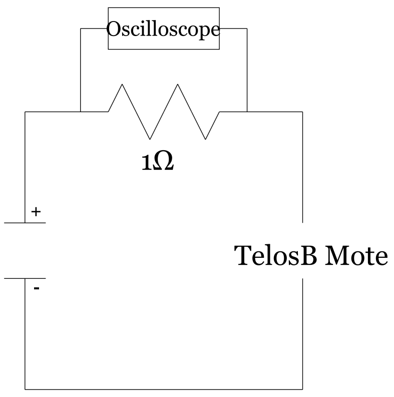

Figure 2.1. Circuit used for measurements

2.1 Measurement Results

The results of the current draw measurements can be found in table 2.1. It was decided to consider an average current.

It is evident from table 2.1 and fig. 2.2 that idle listening is a large part of the power consumption of a TelosB mote. Hence, it was decided that transmitting at full power and optimising other parameters would be of primary interest.

It should be noted that these preliminary results show that introducing a new mote in a WSN will be costly in large part due to idle listening. Scheduling protocols can be introduced to remedy this.

Table 2.1. Current measurements

Status	**Average**
Idle	16.84 mA
Receiving	17.25 mA
Transmit at -24 dBm	17.05 mA
Transmit at -16.8 dBm	17.4 mA
Transmit at -8.8 dBm	17.8 mA
Transmit at 0 dBm	18.3 mA

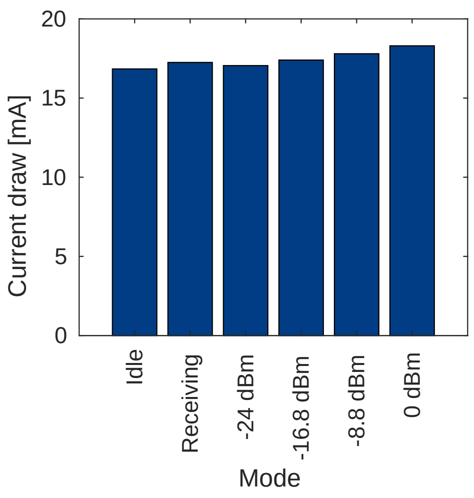

Figure 2.2. Current measurements. Labels stating dBm values are transmissions at differing power levels.

A problem in regards to these measurements is using average values over a certain period. However, using TelosB motes with TinyOS makes other approaches difficult, as the OS and its scheduling controls when certain tasks are run. This causes problems when deactivating the transceiver after a transmission is completed, and relying on these measurements for other programs where the scheduling might be done differently.

Using lower level software components with more control over the hardware could create a basis for better measurements. The increased difficulty in performing such measurements compared to the expected improvements in precision made this approach a low priority.

The measurements presented in table 2.1 are used in our simulations to implement a cost function, cf. listing B.1.4, line 54, and section 3.3.

3 Scenarios

This section describes the overall scenarios considered in this project. Assumptions and challenges are detailed, while node roles in the network topology are also examined. Finally, the channel types of interest are explored.

3.1 Challenges

When considering a WSN topology as described in the introduction, various challenges are met and must be taken into consideration. The challenges described in this section will form the basis of our assumptions in section 3.2 and discussion in section 5.

Overhearing and idle listening at relay

In a naive implementation of the network topology shown in fig. 1.1, node B will always listen for packages to relay immediately. This will introduce reception overhead as seen in the calculations made in section 3.3.

As briefly mentioned in section 2.1, idle listening can be reduced by using scheduling protocols, for instance S-MAC. An advantage of the S-MAC protocol as opposed to STEM, for example, is that S-MAC allows us to both receive and transmit packages during the listen period [1, p. 123]. This would also help reducing the overhearing, as a low duty cycle protocol such as S-MAC would put the relay to sleep if the package sent was not addressed for it.

It should be noted that scheduling protocols introduce a different kind of overhead, namely energy consumption when waking up the relay node and protocol/energy overhead in synchronisation and distribution thereof. Therefore, it should be considered whether this overhead will counteract the energy savings made. Further, it will need to be taken into account in our calculations for transmission costs.

Relay viability

In the project outline, an underlying assumption is that it is always viable to use a relay node when the primary channel suffers fading. However, if the entire network is in a bad state, for instance due to electromagnetic disturbances, it cannot be guaranteed that relaying is successful.

To simplify the design process, we assume that relaying is always possible and successful, cf. assumption 3. Otherwise, we would need dynamic information about the channel state which would increase the protocol complexity.

Network contention

When implementing WSNs the problem of network contention must be considered. Imagine that node A chooses to send via node B, but the next package can be transmitted directly to node C. Both A and B will attempt to transmit to C if the relay is not fast enough.

To avoid increasing protocol complexity, assumptions 3 and 6 are made. Together, these ensure that relaying does not introduce collisions at C.

Transmission at full power

In section 2, we argue that transmitting at full power is of primary interest due to little difference in current draws of different power levels. However, transmitting at full power introduces noise in the general environment, which can be a problem especially if the network topology is to be extended. A further discussion related to this subject can be found in section 5.

ACK Failure

Designing a relay protocol can for instance be done using acknowledgements to signify whether a package reaches its intended recipient. However, since any channel can suffer fading, ACK packages may get lost even though a data package was received successfully. This can cause data packages to be sent twice: once directly, and once using relay since no ACK was received from the sink. See fig. 3.1 for a graphical representation of the problem.

Similarly, node B can fail to receive ACKs from C. In both cases, the sink can adjust for duplicate data, however, overhead is still introduced from sending duplicates unnecessarily.

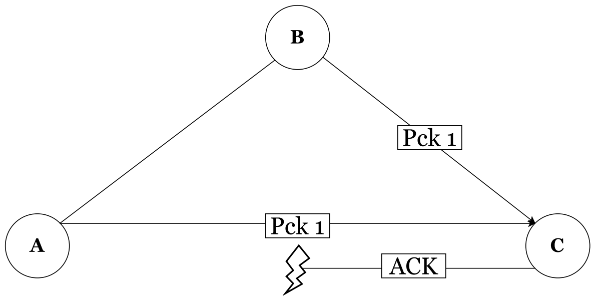

Figure 3.1. Network where ACK package fails.

3.2 Assumptions

In order to simplify modelling of scenarios when designing a relay protocol, several assumptions about the environment in which the WSN resides must be made. In the following, we detail the main assumptions made when the N-Relay Protocol, cf. section 4, was designed.

Assumption 1 ***The channel is bursty***.

This is in accordance with expectations for real-life channels suffering fading, as they will be in a bad state for a set amount of time and in a good state otherwise. A bursty channel can be described using the Gilbert-Elliot model as shown in fig. 3.2.

Assumption 2 ***Good state is longer than bad state***.

If the channel is perpetually in a bad state, the protocol will converge to a situation where it is always better to hop - therefore, a channel in a predominantly good state is of primary interest.

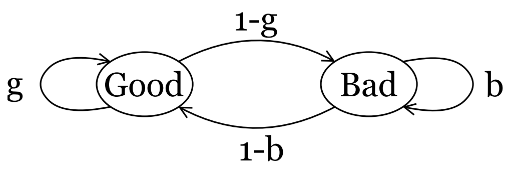

Figure 3.2. Gilbert-Elliott model

Assumption 3 ***Relay is always successful***.

This is obviously a simplification, as we cannot guarantee this to be the case. A relay node could be designed to send until successful transmission – this will introduce overhead, however.

Assumption 4 ***Required reliability is*** 100***%***

If a lower reliability is acceptable, one solution could be to drop packages instead of relaying them.

Assumption 5 ***Frequent transmissions***.

We must be able to reasonably predict the current channel state based on the previous one – a prediction will be more accurate when transmitting at frequent intervals.

Assumption 6 ***No contention***.

Node B is always able to relay fast enough such that no collisions are faced at node C.

3.3 Node Roles

The network topology as presented in the introduction has three nodes with different roles. In the following, these roles are explained in detail, and their influence on both the implementation and simulation are explored.

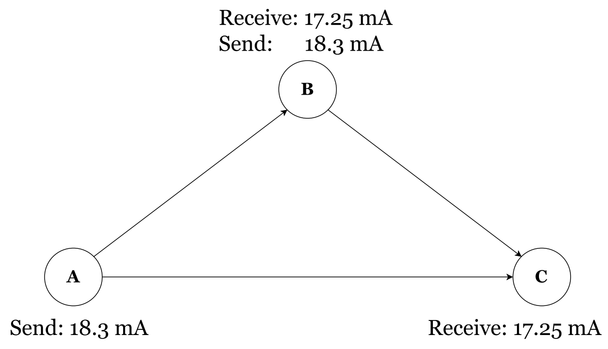

Figure 3.3. WSN with costs

Node A is the **information node**. It collects and senses data and must push it downstream. This data is of vital importance and should reach the sink with 100% reliability. It could for instance be a house alarm or fire detector.

Node B is the **relay node**. Its only purpose is to relay data to the sink to ensure the required reliability. In the naive implementation, it will always listen but only relay if the package is addressed to node B itself. As stated in assumption 3, it is assumed that node B is always successful in its relay endeavours. It sends ACKs upon successful package reception.

Node C is the **sink node**. It must receive data from node A with 100% reliability, and send ACKs upon successful package reception. Node C always listens for packages.

Figure 3.3 shows the network topology with added costs. Node B will, in our case, always listen, meaning it will receive all packages sent regardless of intended recipient. However, it will only relay if the package was addressed to node B. Hence, the cost for relaying can be approximated as follows.

Direct cost, $\mathbf{C}_d$:

- Node A sends: 18.3 mA
- Node B receives: 17.25 mA
- Node C receives: 17.25 mA

$$\begin{aligned}\mathbf{C}_d &= 18.3\text{ mA} + 2 \cdot 17.25\text{ mA} \\ &= 52.8\text{ mA}\end{aligned} \tag{3.1}$$

Relay cost, $\mathbf{C}_r$:

- Node A sends: 18.3 mA
- Node B receives: 17.25 mA
- Node B sends: 18.3 mA
- Node C receives: 17.25 mA

$$\begin{aligned}\mathbf{C}_r &= 2 \cdot (18.3\text{ mA} + 17.25\text{ mA}) \\ &= 71.1\text{ mA}\end{aligned} \tag{3.2}$$

3.4 Channels

Two channels will be used to simulate different scenarios in order to verify that the algorithm will adapt to different circumstances. The first channel $\text{ch}_{\text{office}}$ will illustrate an office building where the nodes will be placed at fixed positions. The second channel $\text{ch}_{\text{deep fade}}$ illustrates a more harsh environment, where deep fading occurs.

Examples of such a channel could be interconnected robots in a warehouse, or radio communication networks used by the military.

Office Channel

The first channel $\text{ch}_{\text{office}}$ is likely a channel where assumption 2 is correct. It can therefore be used to verify that it is correct, and that people occasionally breaking the line of sight, does not cause the channel to be in a bad state more often than in a good state. To verify this a random hallway with a length of approximately $n = 36$ meters has been used to estimate channel parameters for a Gilbert-Elliott channel.

To estimate the channel, two motes are used; a sender and a receiver. These correspond to the information and sink nodes, respectively, as discussed in section 3.3. The sender uses the program `CounterRadio`, see appendix A.7, sending consecutively numbered packages. The receiver is the `BaseStation` shipped with TinyOS, see appendix A.6. It receives the packages and forwards them to a computer.

When the computer receives the packages, we can simply pipe the results into a file and analyse the data. Table 3.1 shows a summary of these results. They are to be considered as time series, and can be used to derive parameters for the Gilbert-Elliott channel.

Table 3.1. Summary of hallway data.

Number of packages	50736
Received correctly	50221
Lost packages	515

The average length of good and bad state is 213.66 and 2.10, respectively. These numbers leads to the parameters

$$b_{\text{office}} = 1 - \frac{1}{2.10} \approx 0.524 \tag{3.3}$$

$$g_{\text{office}} = 1 - \frac{1}{213.66} \approx 0.995. \tag{3.4}$$

Deep Fading Channel

The channel $\text{ch}_{\text{deep fade}}$ is similar to $\text{ch}_{\text{office}}$, but with longer periods of unavailability of the sink node. Thus,

$$b_{\text{office}} < b_{\text{deep fading}} < 1. \tag{3.5}$$

At the same time, it is still assumed that assumption 2 holds, such that

$$b_{\text{deep fading}} < g_{\text{deep fading}}. \tag{3.6}$$

The exact values of b and g are less important, as long as the two conditions above are true. Their values are therefore chosen arbitrarily such that the conditions hold.

4 N-Relay Protocol

4.1 Design

Based on the measurements in section 2 it was found that only considering power when deciding whether to hop made little sense, as the power consumption of having the radio module active on three nodes exceed that of two nodes, even when transmitting at a lower dBm. Thus, if no other quality of service is required, simply dropping failed packages using the direct channel would be considered optimal.

However, taking into account the different characteristics of channels described in section 3.4, the objective of the relay protocol can be seen as sending packages reliably to the sink, wasting as little power as possible on failed transmissions. One additional quality of service could be allowed latency in package reception. Therefore, it becomes important to minimise both the number of failed attempts when transmitting directly to the sink, and the number of relays when the direct channel is in a good state.

One way to achieve this could be drawing inspiration from the back-off algorithm, used by Ethernet as well as WLAN to avoid packet collisions. These protocols employ collision detection and back off for a set amount of time to avoid further collisions.

Using these principles information node A will attempt to send to the sink C. If transmission fails, the node will back-off from transmitting directly to the sink and will instead transmit using the relay node to forward the message to the sink.

Like the back-off algorithm, the N-Relay algorithm will, for each failed attempt at sending directly to the sink, further increase the number of times it transmits using the relay. This will decrease the number of failed attempts to transmit directly to the sink in case the channel is suffering fading. The number of relays required is reset to zero every time a direct transmission is successful.

This design is simplified because of the assumptions described in section 3.2. One of the main influences here is that messages sent over relay will always be successful. This makes the choice of sending packages over relay simpler, as the cost of doing so will be known beforehand.

A basic algorithm constructed from this idea can be seen in algorithm 1.

Algorithm 1 N-Relay Protocol Algorithm

```
1: Counter = 0
2: Attempt sink transmission
3: if Transmission fails then
4:     Increase Counter
5:     Relay Counter number of times
6:     Goto 2
7: else
8:     Counter = 0
9:     Goto 2
10: end if
```

Two parameters in the algorithm can be modified and thereby be used to influence the overall power consumption when sending a number of packages over a given channel. One parameter is the number of times, t, the node attempts to transmit directly to the sink before it starts using the relay. Increasing this will make the algorithm handle small fading better, as it will not directly start using the relay after a failure.

The other parameter is how much the counter is increased each time a direct transmission fails. This increase could be chosen to be linear, exponential or etc. Having a higher growth will make the algorithm use less failed attempts when encountering deep fading.

These two parameters also align with the assumptions described in section 3.2, as both of them will influence how the algorithm reacts to a transition from a good state to a bad, as well as bursts from the channel.

A further exploration into how these two parameters affect the N-Relay algorithm will be discussed in section 4.2.

4.2 Simulations

The two channels described in section 3.4 are used for simulations. The purpose of these simulations is to find good values for the parameters of algorithm 1. This includes which growth function to use for the step size s, the value of s, and the threshold t describing how many direct transmission attempts that are necessary before relaying is allowed. These values can then be used to show the efficiency of the algorithm. The efficiency will be compared to algorithm 2 – we refer to this as the utopian algorithm from now on.

The utopian algorithm will always make the best choice; this is only possible because it knows the exact sampling of the channel ahead of time, which obviously is impossible in reality.

Algorithm 2 Utopian Algorithm

```
1: state = get state of channel
2: if state is good then
3:     Send directly
4: else
5:     Send via relay
6: end if
```

Algorithm 3 Genetic Algorithm

```
1: Generate initial population P
2: Evaluate population P
3: if stopping criteria is satisfied then
4:     Stop
5: else
6:     Create mating pool M
7:     Evolve M into a new P
8:     Goto 2
9: end if
```

A genetic algorithm is used to optimise s and t for both $\text{ch}_{\text{office}}$ and $\text{ch}_{\text{deep fade}}$ where s is increased both linearly and exponentially. The family of genetic algorithms behave like Darwin's survival of the fittest theory. The algorithm is described in algorithm 3 – each method is described in more details below.

In this project, the population used in the genetic algorithm is a collection of one thousand vectors containing s and t values. The initial population is created by initialising one thousand vectors with uniformly distributed random values. The distributions of the values are: $s_{\text{lin}} \sim U(0, 25)$, $s_{\text{exp}} \sim U(1, 25)$, and $t \sim U(1, 50)$. The bounds are chosen to allow the algorithm to explore a large search space.

Evaluation of the population is performed using the cost function in eq. (4.1). `direct` is the number of direct transmits from node A to node C regardless of success or failure, and `relays` is the number of transmits that are relayed over node B. The weights are set according to eqs. (3.1) and (3.2).

$$c = \texttt{direct} \cdot 52.8 + \texttt{relays} \cdot 71.1 \qquad (4.1)$$

The goal is to minimise c by evolving the vectors into better vectors. To select vectors for the mating pool, we pick the best five hundred that result in the lowest cost. This is a kind of tournament selection. After selecting the best five hundred, we create five hundred new vectors by a simple crossover function in eq. (4.2).

$$\mathbf{u} = \begin{bmatrix} v_t \\ w_s \end{bmatrix} \quad \text{where } \mathbf{v} = \begin{bmatrix} v_t \\ v_s \end{bmatrix} \text{ and } \mathbf{w} = \begin{bmatrix} w_t \\ w_s \end{bmatrix} \qquad (4.2)$$

The crossover function selects two random vectors (**v** and **w**) from the best five hundred and creates an offspring by taking the s part of **v** and the t part of **w**. This is done five hundred times resulting in the mating pool reaching a size of one thousand vectors. The last step is mutation. In this step, five percent of the vectors in the mating pool receive either a new s part or t part. These are selected like the initial values, the single best vector will not be mutated.

The algorithm we use runs twenty five times and return the best vector. Hence, the stopping criteria is simply the number of iterations performed. To provide a statistically better result, the parameters are estimated for ten samplings of the same channel.

To alleviate the problem of starting with a random population, the genetic algorithm is run one hundred times per channel sampling and only the best result is saved from each sampling. These results $\{s_i\}$ and $\{t_i\}$ for $i \in [1, 10]$ can then be used to show tendencies where the parameters converge and calculate the channel efficiency. Only the results for the most efficient of the two growth functions will be shown for each channel, since several graphs are used.

Simulations of the Office Channel

The simulation results for $\text{ch}_{\text{office}}$ is shown in table 4.1. It can be seen that linear and exponential growth of s results in virtually the same cost in current draw. This is due to the fact that the length of fading in $\text{ch}_{\text{office}}$ is relatively small, so neither of the methods increase the step size in large increments. It can also be seen that t converges at a value of 2. This results in node A trying to send directly two times before relaying over node B when a direct transmission fails.

The overhead presented in table 4.1 describes how much extra current is used in the system per package on average. An overhead below one percent is very good. Since the results of increasing s linearly or exponentially is virtually the same in the simulations of $\text{ch}_{\text{office}}$, we will only present the linear results in more detail.

Table 4.1. Simulation results of office channel

Office Channel	**Min.**	**Avg.**	**Max.**
Simulated (mA) [lin]	53.37	53.49	53.61
Simulated (mA) [exp]	53.22	53.52	53.74
Optimal (mA) [lin]	53.00	53.04	53.09
Optimal (mA) [exp]	52.94	53.05	53.13
Threshold [lin]	1	2	4
Threshold [exp]	1	2	5
Step Size [lin]	0.00	0.43	0.51
Step Size [exp]	1.00	1.00	1.00
Overhead (%) [lin]	0.36	0.45	0.54
Overhead (%) [exp]	0.28	0.45	0.61
Overhead (%) [lin]	0.69	0.85	1.02
Overhead (%) [exp]	0.53	0.87	1.14

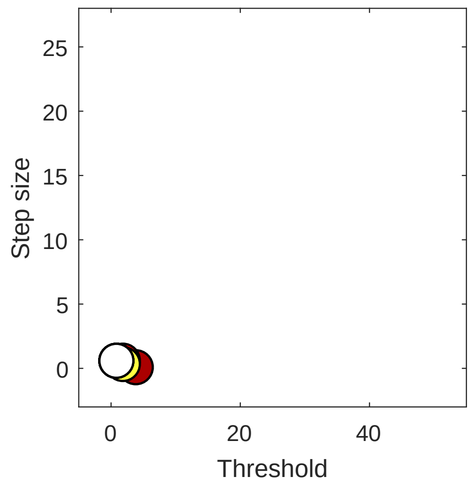

Figure 4.1. Heat map of the best parameters for linear incrementation for each sampling of ch_{office}. The x-axis show t and the y-axis show s. White colour indicate lowest power usage required for transmitting five thousand successful packages.

Figure 4.1 shows a heat map of the best s and t parameters for each of the ten samplings. Each axis correspond to a parameter, and the colouring indicate the cost according to fig. 4.2.

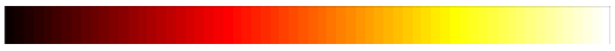

Figure 4.2. Left is highest cost and right is lowest cost.

The heat map in fig. 4.1 is scaled to show the entire search space, which indicate a clear clustering regardless of the sampling. This makes us believe that the genetic algorithm converges at a global minimum and not just a local one.

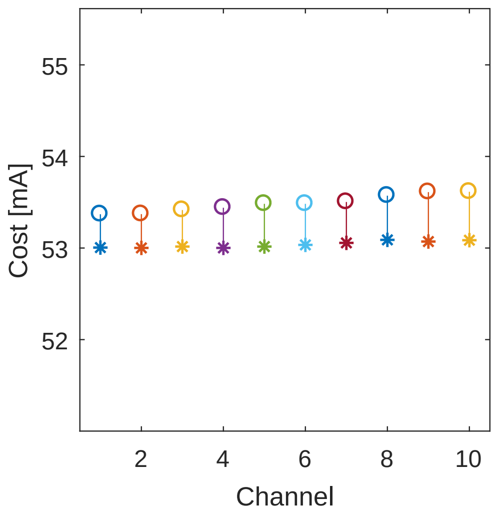

Figure 4.3. Difference between the cost of using the best estimate of s and t versus the cost of the utopian algorithm, for each sampling of the channel, using ch_{office}. Star indicates an optimal cost and circle the cost of the parameter estimates. Each colour indicates a different sampling.

The parameters result in current draws close to the utopian algorithm simulated by the cost function. The difference in the cost of using the utopian algorithm and the estimated parameters is shown in fig. 4.3. Stars show the optimal and circles the simulated, for each sampling of the channel.

Simulations of the Deep Fading Channel

The algorithm also provides the best results using linear growth for the channel with deep fading ch$_{\text{deep fade}}$ as shown in table 4.2. The two methods are close to each other performance wise as was the case in the simulations of ch$_{\text{office}}$.

Table 4.2. Simulation results of deep fading channel

Deep Fading Channel	**Min.**	**Avg.**	**Max.**
Simulated (mA) [lin]	58.86	60.65	62.96
Simulated (mA) [exp]	59.41	61.13	62.26
Optimal (mA) [lin]	57.33	58.88	60.79
Optimal (mA) [exp]	55.27	58.60	59.61
Threshold [lin]	1	1	1
Threshold [exp]	1	1	3
Step Size [lin]	2.94	5.42	9.75
Step Size [exp]	1.38	1.58	1.87
Overhead (mA) [lin]	1.40	1.77	2.17
Overhead (mA) [exp]	2.03	2.53	2.92
Overhead (%) [lin]	2.40	3.00	3.58
Overhead (%) [exp]	3.44	4.32	4.92

Compared to the costs in table 4.1, the cost in the simulations of ch$_{\text{deep fade}}$ is higher both for the N-Relay protocol and the utopian algorithm. This is the result of deep fading which requires more packages to be relayed. The overhead in these simulations indicate that it is harder to estimate the current state when considering a channel subjected to deep fading.

The linear increase of s still provides slightly better results than exponential growth, so we present the simulated results of the linear increase in details below. There is still a clear clustering of the parameters as shown in fig. 4.4. The cluster is more scattered than for ch$_{\text{office}}$ in regards to s, but t is always one.

Further, the values of s are much higher for ch$_{\text{deep fade}}$ than for ch$_{\text{office}}$. This is the result of larger periods of failed transmissions – higher values of s result in determining the size of fading more quickly, and therefore minimise the number of failed direct transmissions. The nature of sampling deep channels is what causes the cost to vary more in fig. 4.5 and the overhead is correspondingly higher as indicated by the longer distance between stars and circles.

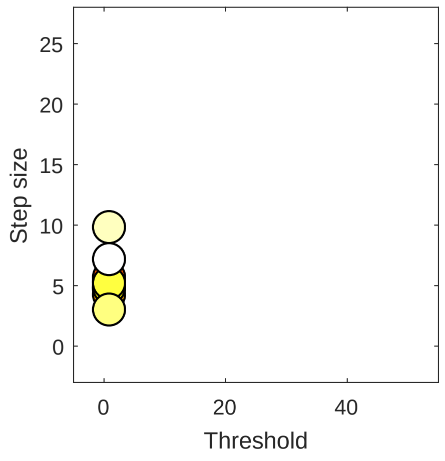

Figure 4.4. Heat map of the best parameters for each sampling using ch$_{\text{deep fade}}$. The x-axis show t and the y-axis show s. Whiter colour indicates low power usage required for transmitting five thousand successful packages.

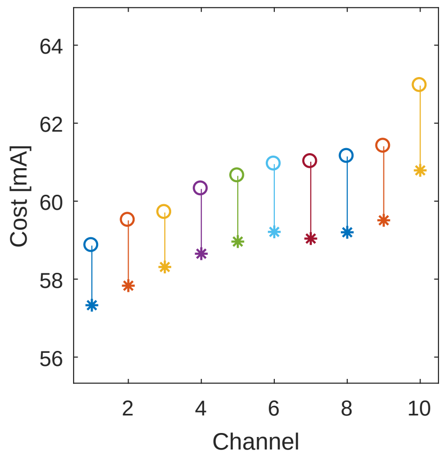

Figure 4.5. Difference between the cost of using the best estimate of s and t versus the cost of the utopian algorithm, for each sampling of the channel, using ch$_{\text{deep fade}}$. Star indicates an optimal cost and circle the cost of the parameter estimates. Each colour indicates a different sampling.

4.3 Implementation

All nodes in the network topology are implemented on the TelosB motes provided by AU Engineering. The motes are each running TinyOS providing basic functionality such as sending, receiving, scheduling, and event handling.

The main objective of node A is to execute the algorithm described in section 4.1. Using results from simulations done in section 4.2, a baseline for good parameters for the algorithm is provided. As each of these channels used in the simulation resemble a channel built on either measurements or expected behaviour, the parameter values provided from the simulations are suitable for use in the implementation.

The implementation makes use of the `AMSender` component provided by TinyOS. This provides the necessary interface for sending packages using the wireless transceiver. To ensure a correct evaluation of the outcome of each transmission, the sender requests acknowledgements for each message sent out. This is done using the `requestAck` function provided by TinyOS. After each package is sent and the event `sendDone` is fired, the result of the sent message can be evaluated using the `wasAcked` function. This function returns a value describing whether or not an acknowledgement was received.

Node A will review whether the channel is in a good state based on previous transmissions before attempting to send a package. This review allows the node to decide if it should send over node B.

To enable straightforward testing, the addresses of the three nodes are set statically. This makes it easier to determine if a message is received from node A or node B when reviewing the messages received by node C. Another measure implemented is the use of a timer in order to send massages at a certain interval. The code for node A can be seen in appendix A.4.

The message used by the system is provided by the `ActiveMessageC` component, and contains 80 bits of data. These 80 bits are currently used for ease of test, providing 16 bits with the original sender address, 32 bits with the number of previous consecutive messages sent over the relay, and the last 32 bits for the remaining number of messages to be send over the relay.

For node C and node C to fulfil the functionality matching the topology, two simple programs are implemented. Like node A, node B and node C use `AMSender` and `AMReceiver` to send and receive messages using the wireless transmitter. Turning on the relay node will have it enter receive mode, where it will wait for any messages sent to its address. Any messages received by the relay node will be forwarded directly to the receiver node, copying the payload of the received message into the message to be transmitted. The code for the relay note can be seen in appendix A.5.

For the receiver node, the `BaseStation` program provided with TinyOS is modified slightly to match the required behaviour. The build process of the `BaseStation` set two different `cflags` in the makefile, namely `-DCC2420_NO_ACKNOWLEDGEMENTS` and `-DCC2420_NO_ADDRESS_RECOGNITION`. The first flag, `-DCC2420_NO_ACKNOWLEDGEMENTS`, ensures that `BaseStation` will not send acknowledgements to any node from which it receives a package. The second flag, `-DCC2420_NO_ADDRESS_RECOGNITION`, enables `BaseStation` to receive any package on the wireless transceiver, regardless of whether the MAC address matches that of the `BaseStation`. The code for the receiver node can be seen in appendix A.6.

Both these flags are removed from the makefile in order to ensure that `BaseStation` will only receive messages addressed to it, as well as send an acknowledgement to the sender of the package. This makes the nodes able to act as specified in the network topology, in addition to enabling viewing any received message on a connected computer.

4.4 Testing Results

To test the implementation, the three programs discussed in section 4.3 are used in a set-up very similar to that of the estimation of $\text{ch}_{\text{office}}$. Node A was placed in the same place as `CounterRadio`, and node C in the same place as `BaseStation`. The main difference was that node B was added and placed with equal distance to the other two nodes. Due to similar placement of nodes, the channel was expected to be similar to when it was first estimated. Therefore the growth function and parameters of the simulation that performed best on $\text{ch}_{\text{office}}$ was used in the implementation of `SenderC`. The best result obtained from simulations are with linear growth and the parameters $s = 0.43$ and $t = 2$.

During the test, 6838 packages were sent successfully with a frequency of ten packages per second. Of these, 6594 was sent directly to node C and 244 was relayed through node B. Other than the successful packages, another 242 was sent directly where either the package or the acknowledgement was lost.

This causes an average cost of successful packages of approximately 55.32mA. With the optimal cost from the simulated utopian channel shown in table 4.1, we can calculate the overhead. Using this, we get a overhead of approximately $.3\%$. It is necessary to use the simulated utopian cost, since there is no way of finding that for the real channel. The cost and overhead is summarised in table 4.3.

Table 4.3. Realised results of office channel

Cost (mA)	55.32
Overhead (%)	4.3%
Overhead (mA)	2.28

The algorithm performs slightly worse than the simulation where the overhead is only 0.85%, but the obtained result is still decent for the realisation. In the realisation, we have 69 duplicate packages that have been removed before the overhead calculations. This is done to adjust for assumption 3 and ACK failure described in section 3.1. This ensures that an ACK failure when transmitting from node A directly to node C will count as a failed transmission, and an ACK failure when transmitting from node B to node C will be ignored. In doing so, the results can more fairly be compared with the simulation.

5 Discussion

Overhearing

One of the main issues influencing the result of the simulations is overhearing. The problem here is that the relay node will always will be listening for a package addressed for it. Therefore the relay node will also be receiving the packages send directly for the sink, which will be discarded after reception as the address did not match.

One approach to reduce overhearing would be to introduce a low duty cycle protocol such as S-MAC. This would reduce both idle listening and overhearing, as the relay node would only have to listen in the slot in which it is possible for the sender node to transmit to it. The relay node can go to sleep immediately if the package is addressed to the sink.

Using S-MAC would also introduce the possibility of using negative acknowledgement, where the sink or relay node would only send a negative acknowledgement when nothing was sent for it in the expected time slot. This also enables caching of multiple packages to be transmitted to the relay node consecutively in a given time slot to further reduce idle listening.

A disadvantage of using S-MAC would be the introduced overhead required ensure proper scheduling and synchronisation.

Reliability

One of the main assumptions in this project is that the package reception reliability must be 100%. This is primarily relevant in applications where every package contains critical data, for instance fire alarms or similar. However, other applications may have different reliability requirements. If data aggregation is used, for instance to monitor an agricultural environment, some package loss might be acceptable since aggregation functions could be designed to adjust for missing data.

In these cases, a choice between relaying and simply dropping the package is subject to consideration. Large holes in transmission may still not be desirable, however, to save energy it might be feasible to drop packages when the channel is bad, and only start relaying if the length of the bad state exceeds a certain threshold.

Extended Topology

While the topology described in section 3.3 is suitable to simplify the problem of whether to relay, it is not a suitable topology for all applications. Often, in a real application, knowing your neighbours might not be straightforward, and therefore finding a valid relay node might not be trivial.

To solve this in an extended topology, neighbour discovery protocols and routing tables could be used to build a virtual graph of the network. This would also introduce overhead, which in turn would increase the contention in the channel, increasing the chance of package loss if TDMA is not being used. Another problem with extending the topology is the use of maximum transmission power at 0 dBm, which introduces a lot of noise in the channel.

If the topology was to be extended, introducing dynamic transmission power could be favourable, but would also add additional complexity as hopping could be used in an attempt to reduce noise.

Simulation Parameters

While the values for the two parameters found during the simulations are suitable for the two different channels specified in section 3.4, these may not be applicable for all other channels. This issue arises due to the wide range of channels possible in practice, as well as the difficulty of carrying out realistic simulation. This effect can also be observed in the results from section 4.4 – despite basing one of the simulated channels on measurements of a real channel, there are still some deviations in the results.

6 Conclusion

Making decisions whether to send directly or relay generally require nodes in the network to have information about the channel. As seen throughout this report, obtaining channel information usually requires several assumptions to be made, as it is no trivial task. These assumptions are rarely accurate for practical scenarios, and therefore deviations between the implementation and simulations are observed.

Although several assumptions are made to simplify the design of the relay protocol, cf. section 4, both the simulations presented in section 4.2 and testing results described in section 4.4 show that the N-Relay protocol design works; packages are relayed when the information node senses a bad state.

Some additional overhead is observed when running tests of the implementation in comparison to simulated results. The real channel likely exhibits different characteristics than the simulated channel, and therefore the overhead percentage differs.

When testing the practical implementation, ACK packages were sometimes lost as described in section 3.1. However, during data analysis the duplicate packages were ignored and not used for overhead calculations. This was done to facilitate comparison with simulated data, as our simulation does not take lost ACKs from node B into account. These duplicates will contribute to overhead in practice, though.

Even though the overhead observed in testing of the implementation is higher than that of the simulation, it is still reasonable. In conclusion, the N-Relay algorithm performs very well in the tested scenario.

References

[1] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, 2005. ISBN 9788126533695.

A nesC Code

A.1 Sleeper

Listing A.1.1. SleeperC.nc

```
#include <UserButton.h>

module SleeperC {
  uses {
    interface Boot;
    interface Notify<button_state_t>;
    interface Leds;
    interface SplitControl as AMCtrl;
    interface Timer<TMilli> as Timer0;
    interface Timer<TMilli> as Timer1;
  }
}
implementation {
  int isOn = 1;
  event void Boot.booted() {
    call Notify.enable();
    call AMCtrl.start();
  }
  event void Notify.notify( button_state_t state ) {
    if ( state == BUTTON_PRESSED ) {
      if (isOn) {
        call AMCtrl.stop();
      } else {
        call AMCtrl.start();
      }
    }
  }
  event void AMCtrl.startDone(error_t t){
      call Leds.led2On();
      isOn = 1;
      call Timer0.startOneShot(500);
  }
  event void AMCtrl.stopDone(error_t t){
      call Leds.led0On();
      isOn = 0;
      call Timer1.startOneShot(500);
  }
  event void Timer0.fired(){
    call Leds.led2Off();
  }
  event void Timer1.fired(){
    call Leds.led0Off();
  }
}
```

Listing A.1.2. SleeperAppC.nc

```
#include "Radio.h"

configuration SleeperAppC {}
implementation {

  components SleeperC;

  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  SleeperC.Timer0 -> Timer0;
  SleeperC.Timer1 -> Timer1;

  components MainC;
  SleeperC.Boot -> MainC.Boot;

  components UserButtonC;
  SleeperC.Notify -> UserButtonC.Notify;

  components LedsC;
  SleeperC.Leds -> LedsC.Leds;

  components ActiveMessageC;
  SleeperC.AMCtrl -> ActiveMessageC;
}
```

A.2 Reader

Listing A.2.1. ReaderC.nc

```
#include "AM.h"

module ReaderC {
  uses {
    interface Boot;
    interface SplitControl as AMCtrl;
    interface Receive as Radio[am_id_t id];
  }
}
implementation {
  event void Boot.booted() {
    call AMCtrl.start();
  }

  event void AMCtrl.startDone(error_t t){
  }

  event void AMCtrl.stopDone(error_t t){
  }

  event message_t *Radio.receive[am_id_t id](message_t *msg,
                void *payload,
                uint8_t len) {
    //return receive(msg, payload, len);
    return msg;
  }


}
```

Listing A.2.2. ReaderAppC.nc

```
#include "Radio.h"

configuration ReaderAppC {}
implementation {

  components ReaderC;

  components MainC;
  ReaderC.Boot -> MainC.Boot;

  components ActiveMessageC;
  ReaderC.AMCtrl -> ActiveMessageC;
  ReaderC.Radio -> ActiveMessageC.Receive;
}
```

A.3 Transmission Power

Listing A.3.1. TransmissionPowerC.nc

```
#include <UserButton.h>
#include "Radio.h"

module TransmissionPowerC{
  uses interface Boot;
  uses interface Leds;
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface SplitControl as AMControl;
  uses interface CC2420Packet;
  uses interface Notify<button_state_t>;
}

implementation{
  bool busy = FALSE;
  message_t pkt;
  uint16_t counter = 0;

  event void Boot.booted() {
    call AMControl.start();
    call Notify.enable();
  }

  event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
    }
    else {
      call AMControl.start();
    }
  }

  void sendSomething(void)
  {
    Msg* btrpkt = (Msg*)(call Packet.getPayload(&pkt, sizeof(Msg)));
    btrpkt->nodeid = TOS_NODE_ID;
    btrpkt->counter = 3 + 4 * counter;
    call CC2420Packet.setPower(&pkt,3 + 4 * counter);
    if (call AMSend.send(RECIEVER_ADDR, &pkt, sizeof(Msg)) == SUCCESS) {
      busy = TRUE;  }
  }

  event void Notify.notify(button_state_t state) {
    if (state == BUTTON_PRESSED) {
      if (!busy) {

        Msg* btrpkt = (Msg*)(call Packet.getPayload(&pkt, sizeof(Msg)));
        btrpkt->nodeid = TOS_NODE_ID;
        btrpkt->counter = 3 + 4 * counter;
        call CC2420Packet.setPower(&pkt,3 + 4 * counter);
        if (call AMSend.send(RECIEVER_ADDR, &pkt, sizeof(Msg)) == SUCCESS) {
          busy = TRUE;
        }
      }
    }
    else if (state == BUTTON_RELEASED) {
      counter = (counter + 1) % 8;
      call Leds.set(counter);
    }
  }


  event void AMControl.stopDone(error_t err) {
  }

  event void AMSend.sendDone(message_t* msg, error_t error) {
    if (&pkt == msg) {
      busy = FALSE;
      sendSomething();

    }
  }
}
```

Listing A.3.2. TransmissionPowerAppC.nc

```
#include "Radio.h"

configuration TransmissionPowerAppC {
}
implementation {
  components MainC;
  components LedsC;
  components TransmissionPowerC as App;
  components ActiveMessageC;
  components new AMSenderC(AM_BLINKTORADIO);
  components UserButtonC;
  components CC2420PacketC;

  App.Boot -> MainC;
  App.Leds -> LedsC;
 App.Packet -> AMSenderC;
 App.AMPacket -> AMSenderC;
 App.AMSend -> AMSenderC;
 App.AMControl -> ActiveMessageC;
 App.Notify -> UserButtonC.Notify;
 App.CC2420Packet -> CC2420PacketC;
}
```

A.4 Sender

Listing A.4.1. SenderC.nc

```
#include <UserButton.h>
#include <math.h>

#define SEND_INTERVAL 50

module SenderC {
    uses {
        interface Boot;
        interface Leds;
        interface AMSend;
  interface Packet;
        interface SplitControl as AMCtrl;
        interface Timer<TMilli> as Timer0;
        interface Timer<TMilli> as Timer1;
        interface PacketAcknowledgements as PkAck;
    }
}
implementation {
    message_t pkt;
    float th = 1;
    float stepSize = 0.81;
    float relayCount = 0;
    uint8_t remainingRelays = 0;
    uint8_t numFailed = 0;
    am_addr_t toAddr;


    event void Boot.booted() {
        call AMCtrl.start();
        call Timer0.startPeriodic(SEND_INTERVAL);
    }

    int8_t isChannelGood(void){
        return numFailed < th && remainingRelays < 0.5 ? 1 : 0;
    }

    event void AMSend.sendDone(message_t* msg, error_t error){
        if (call PkAck.wasAcked(msg)) {
            if (toAddr == NODE_C_IDX) {
                relayCount=0;
                numFailed=0;
            } else
                remainingRelays--;

        } else {
            if (toAddr == NODE_C_IDX) {
                relayCount += stepSize;
                numFailed++;
                remainingRelays = relayCount;
            }
        }
    }

    event void Timer0.fired(){
  Msg* btrpkt = (Msg*)(call Packet.getPayload(&pkt, sizeof (Msg)));
  btrpkt->nodeid = TOS_NODE_ID;
  btrpkt->data1 = relayCount;
  btrpkt->data2 = remainingRelays;
        call PkAck.requestAck(&pkt);
        if (isChannelGood())
        {
            call AMSend.send(NODE_C_IDX, &pkt, sizeof(Msg));
            toAddr = NODE_C_IDX;
        } else {
            call AMSend.send(NODE_B_IDX, &pkt, sizeof(Msg));
            toAddr = NODE_B_IDX;
        }
    }

    event void Timer1.fired(){
    }

    event void AMCtrl.startDone(error_t t){
    }

    event void AMCtrl.stopDone(error_t t){
    }
```

```
}
```

Listing A.4.2. SenderAppC.nc

```
#include "Radio.h"

configuration SenderAppC {}
implementation {

  components SenderC;

  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  SenderC.Timer0 -> Timer0;
  SenderC.Timer1 -> Timer1;

  components MainC;
  SenderC.Boot -> MainC.Boot;

  components LedsC;
  SenderC.Leds -> LedsC.Leds;

  components ActiveMessageC;
  SenderC.AMCtrl -> ActiveMessageC;
  SenderC.PkAck -> ActiveMessageC;


  components new AMSenderC(AM_BLINKTORADIO);
  SenderC.AMSend -> AMSenderC;
  SenderC.Packet -> AMSenderC;
}
```

A.5 Relay Node

Listing A.5.1. RelayNodeC.nc

```
 #include "Radio.h"

 module RelayNodeC {
  uses interface Boot;
  uses interface Leds;
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface SplitControl as AMControl;
  uses interface Receive;
  uses interface PacketAcknowledgements;
}
implementation {
  bool busy = FALSE;
  message_t pkt;



  event void Boot.booted() {
    call AMControl.start();
  }

  event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
    }
    else {
    call AMControl.start();
    }
  }

  void forwardMessage (Msg* payload)  {
    Msg* btrpkt = (Msg*)(call Packet.getPayload(&pkt, sizeof (Msg)));
    btrpkt->nodeid = payload->nodeid;
    btrpkt->data1 = payload->data1;
    btrpkt->data2 = payload->data2;
    btrpkt->data3 = payload->data3;
    call PacketAcknowledgements.requestAck(&pkt);
    if (call AMSend.send(0x0C, &pkt, sizeof(Msg)) == SUCCESS) {
      busy = TRUE;
    }
  }

  event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
    if (len == sizeof(Msg)) {
      Msg* btrpkt = (Msg*)payload;
      call Leds.led2Toggle();
      forwardMessage(btrpkt);
    }
    return msg;
  }

  event void AMControl.stopDone(error_t err) {
  }


  event void AMSend.sendDone(message_t* msg, error_t error) {
    if((call PacketAcknowledgements.wasAcked(msg)))
    {
        busy = FALSE;
        call Leds.led2Toggle();
    }
    else
        if (call AMSend.send(0x0C, msg, sizeof(Msg)) == SUCCESS) {
          busy = TRUE;
        }

  }
}
```

Listing A.5.2. RelayNodeAppC.nc

```
#include "Radio.h"
```

```

configuration RelayNodeAppC {
}
implementation {
  components MainC;
 components LedsC;
  components RelayNodeC as App;
  components ActiveMessageC;
  components new AMSenderC(AM_RADIO);
  components new AMReceiverC(AM_RADIO);

  App.Boot -> MainC;
  App.Leds -> LedsC;
 App.Packet -> AMSenderC;
 App.AMPacket -> AMSenderC;
 App.AMSend -> AMSenderC;
 App.AMControl -> ActiveMessageC;
 App.Receive -> AMReceiverC;
 App.PacketAcknowledgements -> ActiveMessageC;
}
```

A.6 Base Station

Listing A.6.1. BaseStationC.nc

```
// $Id: BaseStationC.nc,v 1.6 2008/09/25 04:08:09 regehr Exp $

/*                  tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

/**
 * The TinyOS 2.x base station that forwards packets between the UART
 * and radio.It replaces the GenericBase of TinyOS 1.0 and the
 * TOSBase of TinyOS 1.1.
 *
 * <p>On the serial link, BaseStation sends and receives simple active
 * messages (not particular radio packets): on the radio link, it
 * sends radio active messages, whose format depends on the network
 * stack being used. BaseStation will copy its compiled-in group ID to
 * messages moving from the serial link to the radio, and will filter
 * out incoming radio messages that do not contain that group ID.</p>
 *
 * <p>BaseStation includes queues in both directions, with a guarantee
 * that once a message enters a queue, it will eventually leave on the
 * other interface. The queues allow the BaseStation to handle load
 * spikes.</p>
 *
 * <p>BaseStation acknowledges a message arriving over the serial link
 * only if that message was successfully enqueued for delivery to the
 * radio link.</p>
 *
 * <p>The LEDS are programmed to toggle as follows:</p>
 * <ul>
 * <li><b>RED Toggle:</b>: Message bridged from serial to radio</li>
 * <li><b>GREEN Toggle:</b> Message bridged from radio to serial</li>
 * <li><b>YELLOW/BLUE Toggle:</b> Dropped message due to queue overflow in either
   direction</li>
 * </ul>
 *
 * @author Phil Buonadonna
 * @author Gilman Tolle
 * @author David Gay
 * @author Philip Levis
 * @date August 10 2005
 */

configuration BaseStationC {
}
implementation {
  components MainC, BaseStationP, LedsC;
  components ActiveMessageC as Radio, SerialActiveMessageC as Serial;

  MainC.Boot <- BaseStationP;

  BaseStationP.RadioControl -> Radio;
  BaseStationP.SerialControl -> Serial;

  BaseStationP.UartSend -> Serial;
  BaseStationP.UartReceive -> Serial.Receive;
  BaseStationP.UartPacket -> Serial;
  BaseStationP.UartAMPacket -> Serial;

  BaseStationP.RadioSend -> Radio;
  BaseStationP.RadioReceive -> Radio.Receive;
  BaseStationP.RadioSnoop -> Radio.Snoop;
  BaseStationP.RadioPacket -> Radio;
  BaseStationP.RadioAMPacket -> Radio;

  BaseStationP.Leds -> LedsC;
}
```

Listing A.6.2. BaseStationP.nc

```
// $Id: BaseStationP.nc,v 1.10 2008/06/23 20:25:14 regehr Exp $

/*                  tab:4
 * "Copyright (c) 2000-2005 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."
 *
 * Copyright (c) 2002-2005 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

/*
 * @author Phil Buonadonna
 * @author Gilman Tolle
 * @author David Gay
 * Revision:  $Id: BaseStationP.nc,v 1.10 2008/06/23 20:25:14 regehr Exp $
 */

/*
```

```
 * BaseStationP bridges packets between a serial channel and the radio.
 * Messages moving from serial to radio will be tagged with the group
 * ID compiled into the TOSBase, and messages moving from radio to
 * serial will be filtered by that same group id.
 */

#include "AM.h"
#include "Serial.h"

module BaseStationP @safe() {
  uses {
    interface Boot;
    interface SplitControl as SerialControl;
    interface SplitControl as RadioControl;

    interface AMSend as UartSend[am_id_t id];
    interface Receive as UartReceive[am_id_t id];
    interface Packet as UartPacket;
    interface AMPacket as UartAMPacket;

    interface AMSend as RadioSend[am_id_t id];
    interface Receive as RadioReceive[am_id_t id];
    interface Receive as RadioSnoop[am_id_t id];
    interface Packet as RadioPacket;
    interface AMPacket as RadioAMPacket;

    interface Leds;
  }
}

implementation
{
  enum {
    UART_QUEUE_LEN = 12,
    RADIO_QUEUE_LEN = 12,
  };

  message_t  uartQueueBufs[UART_QUEUE_LEN];
  message_t  * ONE_NOK uartQueue[UART_QUEUE_LEN];
  uint8_t    uartIn, uartOut;
  bool       uartBusy, uartFull;

  message_t  radioQueueBufs[RADIO_QUEUE_LEN];
  message_t  * ONE_NOK radioQueue[RADIO_QUEUE_LEN];
  uint8_t    radioIn, radioOut;
  bool       radioBusy, radioFull;

  task void uartSendTask();
  task void radioSendTask();

  void dropBlink() {
    call Leds.led2Toggle();
  }
```

```

void failBlink() {
  call Leds.led2Toggle();
}

event void Boot.booted() {
  uint8_t i;

  for (i = 0; i < UART_QUEUE_LEN; i++)
    uartQueue[i] = &uartQueueBufs[i];
  uartIn = uartOut = 0;
  uartBusy = FALSE;
  uartFull = TRUE;

  for (i = 0; i < RADIO_QUEUE_LEN; i++)
    radioQueue[i] = &radioQueueBufs[i];
  radioIn = radioOut = 0;
  radioBusy = FALSE;
  radioFull = TRUE;

  call RadioControl.start();
  call SerialControl.start();
}

event void RadioControl.startDone(error_t error) {
  if (error == SUCCESS) {
    radioFull = FALSE;
  }
}

event void SerialControl.startDone(error_t error) {
  if (error == SUCCESS) {
    uartFull = FALSE;
  }
}

event void SerialControl.stopDone(error_t error) {}
event void RadioControl.stopDone(error_t error) {}

uint8_t count = 0;

message_t* ONE receive(message_t* ONE msg, void* payload, uint8_t len);

event message_t *RadioSnoop.receive[am_id_t id](message_t *msg,
            void *payload,
            uint8_t len) {
  return receive(msg, payload, len);
}

event message_t *RadioReceive.receive[am_id_t id](message_t *msg,
            void *payload,
            uint8_t len) {
  return receive(msg, payload, len);
```

```
}

message_t* receive(message_t *msg, void *payload, uint8_t len) {
  message_t *ret = msg;

  atomic {
    if (!uartFull)
{
  ret = uartQueue[uartIn];
  uartQueue[uartIn] = msg;

  uartIn = (uartIn + 1) % UART_QUEUE_LEN;

  if (uartIn == uartOut)
    uartFull = TRUE;

  if (!uartBusy)
    {
      post uartSendTask();
      uartBusy = TRUE;
    }
}
    else
dropBlink();
  }

  return ret;
}

uint8_t tmpLen;

task void uartSendTask() {
  uint8_t len;
  am_id_t id;
  am_addr_t addr, src;
  message_t* msg;
  atomic
    if (uartIn == uartOut && !uartFull)
{
  uartBusy = FALSE;
  return;
}

  msg = uartQueue[uartOut];
  tmpLen = len = call RadioPacket.payloadLength(msg);
  id = call RadioAMPacket.type(msg);
  addr = call RadioAMPacket.destination(msg);
  src = call RadioAMPacket.source(msg);
  call UartPacket.clear(msg);
  call UartAMPacket.setSource(msg, src);

  if (call UartSend.send[id](addr, uartQueue[uartOut], len) == SUCCESS)
    call Leds.led1Toggle();
  else
    {
failBlink();
post uartSendTask();
    }
}

event void UartSend.sendDone[am_id_t id](message_t* msg, error_t error) {
  if (error != SUCCESS)
    failBlink();
  else
    atomic
if (msg == uartQueue[uartOut])
  {
    if (++uartOut >= UART_QUEUE_LEN)
      uartOut = 0;
    if (uartFull)
      uartFull = FALSE;
  }
  post uartSendTask();
}

event message_t *UartReceive.receive[am_id_t id](message_t *msg,
                        void *payload,
                        uint8_t len) {
  message_t *ret = msg;
  bool reflectToken = FALSE;

  atomic
    if (!radioFull)
{
  reflectToken = TRUE;
  ret = radioQueue[radioIn];
  radioQueue[radioIn] = msg;
  if (++radioIn >= RADIO_QUEUE_LEN)
    radioIn = 0;
  if (radioIn == radioOut)
    radioFull = TRUE;

  if (!radioBusy)
    {
      post radioSendTask();
      radioBusy = TRUE;
    }
}
    else
dropBlink();

  if (reflectToken) {
    //call UartTokenReceive.ReflectToken(Token);
  }

  return ret;
}

task void radioSendTask() {
  uint8_t len;
  am_id_t id;
  am_addr_t addr,source;
  message_t* msg;

  atomic
    if (radioIn == radioOut && !radioFull)
{
  radioBusy = FALSE;
  return;
}

  msg = radioQueue[radioOut];
  len = call UartPacket.payloadLength(msg);
  addr = call UartAMPacket.destination(msg);
  source = call UartAMPacket.source(msg);
  id = call UartAMPacket.type(msg);

  call RadioPacket.clear(msg);
  call RadioAMPacket.setSource(msg, source);

  if (call RadioSend.send[id](addr, msg, len) == SUCCESS)
    call Leds.led0Toggle();
  else
    {
failBlink();
post radioSendTask();
    }
}

event void RadioSend.sendDone[am_id_t id](message_t* msg, error_t error) {
  if (error != SUCCESS)
    failBlink();
  else
    atomic
if (msg == radioQueue[radioOut])
  {
    if (++radioOut >= RADIO_QUEUE_LEN)
      radioOut = 0;
    if (radioFull)
      radioFull = FALSE;
  }

  post radioSendTask();
}
}
```

A.7 Radio Counter

Listing A.7.1. CounterRadioC.nc

```
#include <Timer.h>
#include "Radio.h"

module CounterRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface SplitControl as AMControl;
}
implementation {
 bool busy = FALSE;
 message_t pkt;
  uint32_t counter = 0xffffffff;

  event void Boot.booted() {
    call AMControl.start();
  }

   event void AMControl.startDone(error_t err) {
   if (err == SUCCESS) {
     call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
   }
   else {
     call AMControl.start();
   }
 }

 event void AMControl.stopDone(error_t err) {
 }

 event void Timer0.fired() {
    counter++;
   if (!busy) {
     BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)(call Packet.getPayload(&pkt,
   sizeof (BlinkToRadioMsg)));
     btrpkt->counter = counter;
     if (call AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(BlinkToRadioMsg)) ==
  SUCCESS) {
       busy = TRUE;
     }
   }
  }


event void AMSend.sendDone(message_t* msg, error_t error) {
   if (&pkt == msg) {
      busy = FALSE;
    }
  }
}
```

Listing A.7.2. CounterRadioAppC.nc

```
#include <Timer.h>
#include "Radio.h"

configuration CounterRadioAppC {
}
implementation {
  components MainC;
  components LedsC;
  components CounterRadioC as App;
  components new TimerMilliC() as Timer0;
  components ActiveMessageC;
  components new AMSenderC(AM_BLINKTORADIO);

  App.Boot -> MainC;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
 App.Packet -> AMSenderC;
 App.AMPacket -> AMSenderC;
 App.AMSend -> AMSenderC;
 App.AMControl -> ActiveMessageC;
}
```

B MATLAB Code

B.1 Simulation Code

Listing B.1.1. crossOver.m

```matlab
function [crossed] = crossOver(input)
    for i = 1:length(input)
        vec1 = input(randi([1 length(input)]),:);
        vec2 = input(randi([1 length(input)]),:);
        crossed(i,:) = [vec1(1) vec2(2)];
    end
end
```

Listing B.1.2. genetic.m

```matlab
function [summary] = genetic(method, channel,run,ch)
    clc;
    %method = "lin"

    % Init sizes
    pSize = 1000;
    iter = 25;

    % Limits
    thrMax = 50;
    thrMin = 0;
    modMax = 25;
    modMin = 0;

    % Initialize parameters
    thr = randi([thrMin thrMax],pSize,1);
    % for exponential 1-10 | for linear 0-10
    if method == "exp"
        modMin = 1.001;
        mode = rand(pSize,1)*(modMax-modMin)+modMin;
    else % method == "lin"
        mode = rand(pSize,1)*(modMax-modMin)+modMin;
    end
    % Make vectors
    p = [thr mode];

    % Summary matrix
    summary = zeros(iter,5);
    % Main Loop
    for i = 1:iter
        cost = zeros(pSize,1);
        hops = zeros(pSize,2);
        for j = 1:pSize
            [cost(j), hops(j,1), hops(j,2)] = nrelay(method,p(j,:),channel);
        end
        m = [cost p];
        m = sortrows(m);
        s = sortrows([cost p hops]);
        summary(i,:) = s(1,:);
        s(1:10,:);
        m = m(:,2:end);
        m(pSize/2+1:end,:) = crossOver(m(1:pSize/2,:));
        % Mutate 5 % of the vectors (Keep the best)
        for j = 2:pSize
            if rand() < 0.05
                par = randi([1 2]);
                % Mutate threshold
                if par == 1
                    m(j,par) = randi([thrMin thrMax]);
                else
                % Mutate modifyer
                    m(j,par) = rand()*(modMax-modMin)+modMin;
                end
            end
        end
        p = m;
        clc
        fprintf("Channel: %d\nRun: %d\nIteration: %d of %d\n",ch,run,i,iter);
    end
    summary = summary(end,:);
end
```

Listing B.1.3. gilbertelliot.m

```matlab
function channel = gilbertelliot(pGood,pBad)
state = 1;
    staygood = pGood;
    staybad = pBad;

    N = 10000;
    channel = zeros(N,1);

    for i = 1:N
        if state == 1
            channel(i) = 1;
            if rand() > staygood
                state = 0;
            end
        elseif rand() > staybad
              state = 1;
        end
    end

    %stem(channel)

    %save('channel','channel')
end
```

Listing B.1.4. nrelay.m

```
function [cost, directs, relays] = nrelay(method, x, channel)
% Extract from vector
thr = x(1);
mode = x(2);

numOfFailed = 0;
relayLength = 0;
relayCounter = 0;
tryGood = false;

directs = 0;
relays = 0;

c = 0;
i = 0;
while i < 5000
    c = mod(c,length(channel)-2) + 2;
    % Send directly
    if numOfFailed < thr || tryGood == true
        tryGood = false;
        directs = directs + 1;
        % if transmission fails
        if channel(c)+channel(c+1) ~= 2
            numOfFailed = numOfFailed + 1;
            % increase relay length based on method
            if method == "exp"
                if relayLength == 0
                    relayLength = 1;
                end
                relayLength = relayLength * mode;
            else % method == "lin"
                relayLength = relayLength + mode;
            end
        % if direct transmission succeeds
        else
            relayLength = 0;
            numOfFailed = 0;
            i = i + 1;
        end
    % Send via relay
    else
        if relayCounter == 0
            relayCounter = round(relayLength);
        end
        relays = relays + 1;
        relayCounter = relayCounter - 1;
        if relayCounter == 0
            tryGood = true;
        end
        i = i + 1;
    end
end
```

```
% Cost = number of hops
cost = (directs*52.8+relays*71.1)/i;
end
```