Reliable Data Delivery over Wireless Sensor Networks

Per-Erik Björkstad

2007-01-18
Abstract

In this thesis we present ReDaR, a software and hardware environment for reliable data retrieval from multiple remotely located wireless sensor networks (WSNs). The software is programmed in Java and NesC over TinyOS, and allows for direct interaction with individual sensors using a graphical user interface running on a remote client. The hardware platform includes a WSN deployed with Telos wireless sensors.

ReDaR implements a control algorithm over a real WSN, which allows to keep under control the overhead communication bandwidth between the sensors and the remote client. Specifically, we implement a control algorithm based on the extremum seeking control theory, which enables an adaptive forward error correction mechanism to ensure reliable real-time data retrieval. The amount of redundancy is controlled via a switching controller that uses a feedforward and a feedback mechanism. The switching controller ensures an optimal trade-off between maximizing the bandwidth and maximizing the quality of service.

Experiments performed in our environment confirm theoretical results about performance of the controller, i.e. the feedback mechanism provides a robust controller that finds the optimum without model reliance, while the feed forward one improves the transient behavior.
Contents

1 Introduction to WSN 4
   1.1 Advantages and Limitations .................................. 4
   1.2 Usage of Sensor Networks ..................................... 4
   1.3 Routing in WSN .................................................. 5
   1.4 WSN Research Challenges ...................................... 8
   1.5 Tenets - An example of a network structure .................. 9

2 Problem Formulation 11
   2.1 The Problem ..................................................... 11
   2.2 The System Model ............................................... 12
   2.3 The Solution .................................................... 12

3 Error Controller 14
   3.1 Feedforward Control .......................................... 16
   3.2 Feedback Control .............................................. 17
   3.3 Error Controller .............................................. 18

4 Test Bed 19
   4.1 Setup ............................................................ 19
      4.1.1 TMote Sky Hardware ....................................... 19
      4.1.2 TinyOS ..................................................... 20
      4.1.3 NesC ...................................................... 20
      4.1.4 Aegis ..................................................... 22
   4.2 Getting Started ................................................ 22
      4.2.1 Installing Cygwin ......................................... 22
      4.2.2 Installing the Motes ....................................... 22
      4.2.3 Compiling and Running the GUI ......................... 23
   4.3 Using ReDaR ..................................................... 25
4.3.1 The Control Panel ........................................... 27
4.3.2 The Info Panel ................................................ 28
4.4 Beneath the shell .................................................. 30
  4.4.1 The Main Class and the Servers ......................... 30
  4.4.2 The Main Frame, the Control Panel and the Info Panel 30
  4.4.3 How the Communication works ......................... 31
  4.4.4 The Tmote software ....................................... 33

5 Experiments ....................................................... 36
  5.1 How the tests were performed ............................. 36
    5.1.1 The Packet Loss and the Block Length ............... 36
    5.1.2 The Change Detection Filter ......................... 37
    5.1.3 The Controllers .................................... 38
    5.1.4 How we managed to control the Packet Loss ........ 38
  5.2 The Experiments ............................................ 39
    5.2.1 Experiment 1: Constant Redundancy .................. 39
    5.2.2 Experiment 2: Constant Packet Loss .................. 40
    5.2.3 Experiment 3: With the Controller ................... 41

6 Conclusion and Future Work .................................. 42
Chapter 1

Introduction to WSN

Wireless Sensor Networks (WSN) are networks of nodes equipped with limited sensing, processing and communication functionalities. They involve deploying a large number of small nodes that senses environmental changes and reports them to other nodes over a flexible network architecture. Sensor nodes are an appealing technology for monitoring hostile environments, large geographical areas, industrial environments, healthcare, etc.

1.1 Advantages and Limitations

A WSN consists of hundreds or thousands of low cost nodes which could either have a fixed location or be randomly deployed to monitor the environment. Due to their small size and numbers, a sensor network can supply highly detailed in depth-sensing. However, the small size also imply limitations as the power supply. A small wireless device, not larger than a few centimeters, can only hold a small battery, solar panel or other very small and limited power supply. This affects the hardware that each sensor can keep. The sensor nodes, will therefore have limited computational power, memory size and sensing capability etc.

1.2 Usage of Sensor Networks

Sensor networks can be useful in a variety of domains. The primary domains at which sensors are deployed are:
• **Environmental Observations:** Sensor networks can be used to monitor environmental changes. An example could be water pollution detection in a lake that is located near a factory that uses chemical substances. Sensor nodes could be randomly deployed in unknown and hostile areas and reveal the exact origin of pollution so that precise countermeasures can be taken. Other examples include forest fire detection, air pollution, animal habitat monitoring and monitoring of snowfall to prevent avalanches.

• **Military monitoring:** Military uses sensor networks for battlefield surveillance; sensors could monitor vehicular traffic, track the position of the enemy or even safeguard the equipment of the side deploying sensors.

• **Building monitoring:** Sensors can be used in large buildings or factories monitoring climate changes. Thermostats and temperature sensor nodes are deployed all over the building's area. In addition, sensors could be used to monitor vibrations that could damage the structure of a building.

• **Health Care:** Sensors can be used in biomedical applications to improve the quality of the provided care. Sensors can be implanted in the human body to monitor medical problems like cancer and help patients maintain their health.

### 1.3 Routing in WSN

Sensors networks are organized such that the routing of data ends at special nodes called base stations (sometimes they are also referred to as sinks). A base station links the sensor network to another network (like a gateway) to disseminate the data sensed for further processing. Base stations have enhanced capabilities over simple sensor nodes since they must do complex data processing. These typically have workstation/laptop class processors, and enough memory, energy, storage and computational power to perform their tasks well. Usually, the communication between base stations is initiated over high bandwidth links.

Sensors that can’t communicate directly with the base station, can send their data to nearby sensors, in a multi-hop fashion, until the data reaches
its base station. Thereby a multi-hop data chain is formed and sensors that are out of reach from the base station can reach it.

Since power consumption is one of the major problems of sensor networks, multi-hops needs to follow certain rules. For example, one could imagine a structure where every node broadcasts its data to every neighboring node until it reaches the base station. This kind of routing, however, greatly increases the traffic load of the network, since network traffic load is strongly related to the life-time of the nodes, routing must be carefully designed. Specifically, packets should be routed such that the nodes consumes a minimum of energy during the routing operation.

![Figure 1.1: Figure of Multi-Hop Broadcasting in WSN](image)

In general, aggregation points are introduced in the network. This reduces the total number of messages exchanged between nodes and saves energy. Aggregation points are regular nodes that receive data from neighboring
nodes, perform some kind of processing on the data, and then forwards the aggregated data to the next node.

Figure 1.2: Figure of Multi-Hop Routing in WSN

Clustering is a Technique that allows to implement the aggregation, Sensor nodes can be organized into clusters, each cluster having a ”cluster head” as the leader. The cluster head can perform the aggregation of packets coming from the nodes within the cluster. The cluster head then communicates directly, or via other cluster heads, with the base station.

Another method for saving energy is setting the nodes to go idle (into sleep mode) if they are not needed and wake up when required. The challenge is to find a pattern at which energy consumption is made evenly for all the nodes in the network. In this report we will not consider aggregation or in-network pre-processing of collected data, but rather discuss a method that secures data delivery and minimizes data losses without increasing network load and power consumption significantly.
1.4 WSN Research Challenges

Due to the sensors limited capabilities, there are a lot of design issues that must be addressed to achieve an effective and efficient operation of wireless sensor networks.

- **Energy saving algorithms:** Sensor nodes use batteries for power. These can be difficult to replace when consumed, so it is critical to design algorithms and protocols in such a way that minimal energy is utilized. To do so, implementations must reduce communication between sensor nodes, simplify computations and apply simple security solutions.

- **Location discovery:** In many applications tracking of an object require knowing the exact or approximate physical location of the sensor nodes. Furthermore, many geographical routing protocols need the location of sensor nodes to forward data among the network. Location discovery protocols must be designed in such a way that minimum information is needed to be exchanged among nodes to discover their location. Additionally, cost is another factor that influences design; it has to be kept at a minimal level to allow for mass deployment. Therefore, solutions as with GPS are not recommended, since they are expensive and energy consuming. New ways to discover location are needed.

- **Security:** It is not possible to introduce a new technology without addressing security. However security is not the top priority when designing something new. Security solutions are constrained when applying them to sensor networks. For example, cryptography requires complex processing to provide encryption to the transmitted data. Secure routing, secure discovery and verification of location, key establishment and trust setup, attacks against sensor nodes, secure group management and secure data aggregation are some of the many issues that need to be addressed in a security context.
1.5 Tenets - An example of a network structure

WSN systems today are small to medium sized networks. Typically data are gathered continually at the nodes and forwarded to a central location, where most of the data processing takes place. The systems are tailored for the sensing task at hand, which typically makes them complex, expensive and hard to manage. What is needed is a system standard that allows mass production and deployment of larger systems. Tenets has been proposed as a possible standard.

![Figure of the Tenet architecture](image)

Figure 1.3: Figure of the Tenet architecture

In a Tenet the network is layered, and the lowest layer holds the sensing and actuating nodes. These are small, low power, wireless sensing devices, but they are generic and they all look and work in the same way. They can be pre-programmed in the factory, which lowers the price and simplifies system deployment and manageability. On a higher layer there are Star-gate class nodes called Masters. These can be wired or wireless, but they typically have 1-2 magnitudes higher power supply (if not infinite). This allows them to have a higher computing power, memory size, transmission rate and radius etc. In Tenets, the Masters are programmed to handle application specific data processing. End users connected to a master node can interact with the network via it.

This standard simplifies node functionalities to handle simple requests.
from masters. Examples of such requests can be: get a reading or activate a switch. All application specific processing are handled by Masters. However, generic, local sensor data processing, such as compression or transformation, can be implemented on the lowest layer motes. This can help reduce traffic and data transfer size.

By excluding complex data processing from the low power nodes, it is possible to gain two major advantages: first, the application runs in a less resource constrained environment, which simplifies development and enables higher system robustness; and second, the node layer can be made generic, only needs to be implemented once and can be reused for a number of other applications. The drawback is a small loss of efficiency.

The appealing aspect of this standard is that it allows mass production, reduces system manage costs and increases system robustness, but still allows the sensing properties that WSN systems can offer. The price we pay - a potential loss of efficiency - is small compared to the gains. Note, however, that future technological revolutions in energy storage (ie miniaturized fuel cells) or vastly more energy effective processors, transmitters and/or memory storage may come to question the use of Tenets. Tenets are designed and developed for the technology we have today. ?are leigble offer has half today?
Chapter 2

Problem Formulation

2.1 The Problem

In the near future, the expected wide deployment of WSNs risks to flood existing communication infrastructures by transmitting tremendous amounts of information. Consequently, the management and control of the data generated by WSNs, while meeting the quality of service required by real-time decision making applications, is an important research problem. The management of the amount of information produced by WSNs in end-to-end communications over heterogeneous (wired and wireless) networks involves several techniques, which may be implemented at various layers of the communication protocol stack. The management is complicated by the fact that modern networks are characterized by a multiplicity of architectures, platforms and protocols, where several existing complex interfaces are intended to intercorporate. Moreover, heterogeneous networks exhibit extreme bandwidth variability and burst traffic behaviors. In this context, the control of the amount of information produced by WSNs is a truly challenging task.

When real-time applications retrieve data produced by multi-hop sensor networks, end-to-end delay and packet error rate are typical network state variables to take into account to ensure an adequate quality of the data received by the application. Techniques to counteract these communication imperfections may introduce additional traffic load. So in order to avoid a blow up of the amount of information generated by the WSNs, a design that enables a dynamic tradeoff between data quality and network load should be considered.
Suitable packet error rate can be ensured with channel coding techniques and automatic repeat request (ARQ) protocols. However, for real time applications (as in WSN), a low packet error rate cannot be achieved by retransmissions. Moreover, retransmissions heavily reduces the lifetime of the WSN.

The problem of reducing the amount of traffic generated by WSN while ensuring an adequate quality of the received data, requires several approaches. In this Thesis, we consider an efficient forward error correction (FEC).

2.2 The System Model

Figure 2.1 shows the system scenario studied in this Thesis. What we want to keep reliable is the communication between the nodes and the Client (Application). We have several sensor nodes that acts as routing nodes and builds a link from the data generating node to the base station. The base station is then linked to the client computer via an IP network (e.g. the internet). In the IP network the TCP/IP protocol is used. In the WSN we use multi hop routing for messages going to the base station from the nodes and multi hop broadcast for messages going from the base station to the nodes. This is not the most energy conservative way to send data, but it will suffice and not affect our results. To secure the delivery of packets the receiver needs to detect faulty and lost packets.

2.3 The Solution

The solution to cope with network losses presented in this thesis, involves Error Correction Codes (ECC). This technique increases the quality of the data received by the client without additional delay, but increases the network load by adding redundant packets (containing the ECC) to the data packets. If the quality is associated with a cost, such that the higher the quality reduction the higher the cost, the cost will decrease when redundancy is increased. The redundancy should thus be high to give a low cost. However, increasing redundancy will increase the network load since additional redundant packets are sent. From this perspective, the redundancy should be low to reduce the cost of network load. Thus, the number of redundancy packets should vary depending on how much packet loss there is i.e. how bad the link
is at the moment. There will be a tradeoff between quality and performance when adjusting the redundancy. The goal of the Error Controller in Chapter 3 is to find this optimal tradeoff.

If we can find the optimal it will allow us to receive data at a high rate with a good connection but it will automatically decrease the data rate if the connection gets bad (since we need to send more redundancy packets) so we can minimize packet loss. With the right choice of error correction codes, it will be possible to fully recover received faulty packets without requesting a resend.
Chapter 3

Error Controller

With the technique discussed in this thesis we need error correction codes (ECC). These codes provide redundancy packets, denoted \( u_b \), which are added into a block \( b \) together with \( N - u_b \) application data packets, where \( N \) denotes the total block length. During the transmission some packets may be lost (referred to as the loss process). Some of these packets can, via the ECCs, be restored in the correction process at the receiver. Packets that cannot be fully restored are considered distorted packets, here denoted as \( d \).

\[
d = (N - u) - z
\]

In this formula \( z \) denote the number of restored packets. This distortion measure captures the amount of lost data after the error correction process, and thereby the rate of distortion in the receiver. See figure 3.1

It may be, that all application data packets can not be successfully restored and some packets may be lost even with the optimal choice of redundancy. With many redundancy packets, we have a higher probability to restore lost and damaged application data packets. However, we do not want to add too many of these packets, since that will increase the network load and increase the energy consumption in the network. To help us find the optimal number of redundancy packets we study a cost function that is based on the distortion \( d \) and the number of redundancy packets \( u \) used in the previous block:

\(^1\)Here, we restrict our attention to block codes
This cost function should punish high distortion as well as high redundancy and it must be kept low in order to find the optimal choice of redundancy packets to be sent. The global minimum of the cost function will give the optimal choice of redundancy. However, to find the minimum is not an easy task. Therefore we will study two different strategies. One is based on Feedforward Control and the other on Feedback Control.

Considering that increasing $u$ decreases the throughput linearly, and that a large $d$ is not desirable, we have used the following cost function in the test bed:

$$c(u) = \frac{d(u)^2}{N} + \rho \frac{u}{N}$$

Where $\rho$ is a positive constant. The objective is then to minimize the expected value of $c$. The problem is finding a controller that uses both information on the network loss process and the distortion after decoding to adapt the redundancy amount.

The sending node transmits coded sequences of packets over the network. These messages will be received with some distortion at the server, which can be communicated back to the sender node. The receiver may also estimate the loss process $p$ and provide the sender node with that information. Based on the information about $d$ and $p$, the sender adjusts the redundancy $u$ via...
Thus, the function $f$ represents the control algorithm used by the node sender, while the information $p$ and $d$ is produced by the receiver. We will only consider the choice of $f$. The receiver algorithms for how to get $d$ and $p$ will not be discussed. Even though both $p$ and $d$ in figure 3.2 are sent back from the receiver to the sender, they represent different types of information from a control point of view. The system to be controlled is the error correction process, with redundancy $u$ as input and the cost $c$ as output. While $u$ has a direct influence on $d$, it has little or no influence on the loss process.

Thus, the signal $p$ can be considered as a disturbance acting on the system, and the cost can be seen as a function of $u$ parameterized by $p$, i.e., $c = -h(u, p)$. Since the sender knows $u$, it only needs information on $d$ to find $c$. Hence basing the control signal on the information given by $p$ results in a feed forward controller, while basing the control on $d$ results in a feedback controller. Combining these two components, the error controller uses the feed forward information for fast transient behavior and the feedback information to get good performance during steady state.

### 3.1 Feedforward Control

Basing the control signal only on the current estimated packet loss probability gives that

\[ u = f(d; p) \]

\(^2\)Large sudden changes in the average packet loss, e.g. the average packet loss increases from 10% to 20%.

\(^3\)Stable average packet loss.
\[ u = f(p) \]

The estimation of \( p \) is made as the average loss probability over a time window \( w_2 \). Based on the this estimate, the applied control signal is given by an online table look-up from the optimal redundancy. The optimal redundancy can be found offline given an assumption of the loss process\(^4\). This strategy is combined with a change detection filter, to detect sudden changes in packet loss probability. The filter is a Sequential Probability Ratio Test (SPRT) and works in the way described in [1]. At each time step (i.e., for each packet) a residual is calculated as

\[ \epsilon_t = p_t - \hat{p}_{t-1} \]

where \( p_t \) is the estimated packet loss probability over a (shorter) time window \( w_1 \), and \( \hat{p}_{t-1} \) is the packet loss probability over a longer window \( w_2 \). Then the filter state \( g_t \) is updated according to

\[ g_t = g_{t-1} + \epsilon_t - \nu \]

The parameter \( \nu \) is used to compensate for slow drifts. If \( g_t \) exceeds a certain threshold, a new packet loss probability is estimated. The internal state of the estimator is then reset together with both windows.

Using only feed forward information \( p \) to determine the control \( u \) has two drawbacks: the need for accurate estimation of the packet loss probability and the reliance on a loss model. If the estimate or the model is incorrect the applied redundancy will not be optimal, resulting in suboptimal performance.

### 3.2 Feedback Control

To address the drawbacks with the feed forward controller, a feedback control algorithm is added. It uses feedback information on the distortion to find the current value of the cost function, why in this case

\[ u = f(d) \]

Since the sender knows \( u \), it can calculate \( c \) given \( d \). The purpose of the controller is to find the optimum of the function \( c \), which is unknown since

\(^4\)Independent losses are assumed.
no loss model is assumed. To find the optimum, we utilize an extremum-seeking controller. Such a controller aims to maximize the output of a system, without the need for a model. The proposed controller for this application is given by

\[ u_{b+1} = u_b - \beta \text{sgn}(u_b^d c_b^d) \]

where \( \beta \) is the control parameter and \( u_b^d \) and \( c_b^d \) denote derivative estimates at block \( b \) of the signals \( u \) and \( c \). This controller is inspired by the switching controllers [2] and aims to drive the system towards the optimum. The basic principle is to estimate on which side of the peak the system is, and then move towards that peak. If the relation between \( c \) and \( u \) is given by a unimodal static function, the system will always converge to the optimum. A more thorough analysis can be found in [3].

### 3.3 Error Controller

The error controller uses the feedback algorithm during steady state, but also utilizes the change detection filter of the feed forward controller. Since both information on the network and the distortion determines the control signal,

\[ u = f(p, d) \]

If a sudden change in packet loss probability is detected, the redundancy is reset to what is optimal given the new estimate. The optimality here is with respect to independent losses. The extremum seeking algorithm then continue seeking for the optimal steady state value. By combining these ideas it is possible to achieve both fast responses during transient and stabilizing on the optimal value during steady state. The control algorithm will then be

\[
\begin{align*}
    u_{b+1} &= \begin{cases} 
    u^*(p_b) & \text{If change detected (i.e. feed forward control)} \\
    u_b - \beta \text{sgn}(u_b^d c_b^d) & \text{Otherwise (i.e. feedback control)}
    \end{cases}
\end{align*}
\]

where \( u^*(p_b) \) is the optimal redundancy amount for packet loss probability \( p_b \), when the losses are assumed to be independent.
Chapter 4

Test Bed

4.1 Setup

The test bed consists of a stationary computer with internet connection. To such a computer a Telos Mote Sky sensor node, also called TMote, is connected via the USB port. This TMote together with the stationary computer is the Base station. Apart from it there are 1-4 other TMotes that are wireless connected to the base station. Additionally there is a Laptop with internet connection. This Laptop will act as the Client. The program that implement the error controller and supply the Graphical User Interface (GUI) is called ReDaR (Reliable Data Retrieval).

Before we look at ReDaR and how it works, we will have a look on the TMote hardware and what the program is based upon.

4.1.1 TMote Sky Hardware

Telos motes are ultra-low power consumption devices, they have an integrated antenna for wireless communication, and the version used with ReDaR has 4 onboard sensors; 2 Light Sensors, 1 Temperature Sensor and 1 Humidity Sensor. Every mote is an independent node that runs an application and can interact with other nodes, sending and receiving sensor data from other nodes. A more in depth description with features etc. can be found at the Moteiv homepage [10].
4.1.2 TinyOS

To operate, the TMote, as well as any kind of sensor node, needs an Operating system. Existing operating systems such as UNIX, Linux and Windows runs well on 32-bit processors at hundreds of Mhz with several megabytes or gigabytes of RAM. A TMote, however, only have a fraction of these resources, thus these operating systems cannot run on a TMote. Therefore a new open source operating system called TinyOS has been developed, tailored especially for small to medium class motes. [?] TinyOS is an event based operating system. Programs written for TinyOS are built up by components and each component has one or several tasks to do. For example, a component can fill the purpose of retrieving data from a light sensor. This component is formed by two sub-components. One component is in charge of sending the request for data to the light sensor and the other is for telling the main program that data retrieval has been performed. Furthermore, these two components can have subcomponent of their own and form several layers of program components. The idea is that the lowest-level components abstract the physical hardware and deliver physical interrupts as asynchronous events to the higher levels.

4.1.3 NesC

Programs running under TinyOS are written in NesC. NesC is in many ways a lightweight version of C. Syntaxes are similar but tools for graphical use, advanced mathematical operations and advanced memory handling, etc, has been excluded.
Programs written in NesC are often built up by preprogrammed existing component, that are linked together in a link file usually called *program-name*.nc. Components can, of course, be programmed and designed, but this is usually a very heavy task to do. The link file is a file that specifies to which subcomponents what other subcomponents shall send their events to. So simply, this file defines the program structure. A sample picture of a the highest level linking topology in Redar can be seen in figure 4.2.

Beside the link file, there is the file where the actual program code is written. This file is usually called *programnameM*.nc. Here instructions defies
the mote behavior when a component generates an event. After compilation, this file can be compared to the executable in a normal program running under a conventional operating system.

4.1.4 Aegis

ReDaR is based upon the program Aegis [5]. Aegis purpose is to set up a wireless sensor network, where every TMote Sky Mote reads data continually from its onboard sensors and sends the readings to a base station. It builds up a dynamical multihop routing tree, so it is able to recover from node failure. It supplies a GUI to visualize and control the network. It can also connect to a remote server for remote sensor data retrieval. However, Aegis does not supply the functionality for remote sensor data retrieval and remote network control that is desired and implemented in ReDaR. Neither does it implement the error controller described in Chapter 4.

A more detailed description of Aegis can be found in the Aegis manual [5].

4.2 Getting Started

4.2.1 Installing Cygwin

If you run Windows you will have to install Cygwin in order to install and run applications on the TMotes. Cygwin is a Unix Emulator for Windows. The Cygwin install files can be downloaded at www.cygwin.com.

4.2.2 Installing the Motes

Before you can use the motes you will have to install the appropriate USB drivers. They can be found on the TinyOS CD or on the Web at:

http://www.ftdichip.com/Drivers/VCP.htm

A guide how to install these drivers can be found at:

http://www.ftdichip.com/Documents/InstallGuides.htm

The ReDaR program for the motes can be found in ReDaR Tmote.zip in the ReDaR package. Unzip this file to a suitable location in your TinyOS installation folder. A suitable location would be
To compile it you have to start cygwin, then go to that same directory where you unzipped ReDaR Tmote.zip, and type

```
make tmote
```

For some reason this does not seem to work on all computers. If it does not, you can try with "make telos" or "make telosb". Compiling should not be necessary since the ReDaR Tmote application is already compiled.

The compiler will create an image that can be written to a Tmote. To do that, first plug in a mote to the USB and type

```
make tmote reinstall, 1
```

This will install the image on the mote and give it address 1. Thus the number represent what address the mote will have. You will need at least one mote with address 0 for ReDaR to work properly. This mote will be the base node and has to be physically connected to the base station (plugged into the USB port) during runtime. If you are experiencing problems with having ReDaR on the basenode you can try with TOSBase.

### 4.2.3 Compiling and Running the GUI

Before you can use the Graphical User Interface you will have to start *SerialForwarder*. This program is written in Java and it is listening on the USB port for messages from the Tmote base station. It forwards these messages to other running Java applications. To start SerialForwarder open a cygwin window and type

```
java net.tinyos.sf.SerialForwarder -comm serial@COM#:tmote
```

where # is the USB port number. To find out what port number your tmote has been assigned to type "motelist" and a list of currently connected motes will be shown.

To start the GUI you have to unzip the "Redar Java.zip" to

```
CygwinPath/Cygwin/opt/tinyos-1.x/tools/java/net/tinyos/aegis
```

---

1 TOSBase is included in the TinyOS installation. It can be found in *Cygwin-Path/cygwin/opt/tinyos-1.x/apps/TOSBase*, compiled with "make tmote" and installed on a mote with "make tmote reinstall".

23
Note that it has to be exactly this folder, so if you previously have an installation of Aegis, it is highly recommended that you make a backup of that folder. To compile the program type ”make” while in the aegis directory. You can also compile each .java file individually. Unless you plan on modifying ReDaR, this shouldn’t be necessary, since all the files are already compiled.

To start the ReDaR GUI type

```
java net.tinyos.aegis.Redar.
```

This will start a local server and the GUI client. Redar can also be started with the client only. To start the client only add ”client” to the line above. When started as a client only remote connections are allowed and SerialForwarder doesn’t have to be started.

Note that ReDaR does not support any type of network tunneling or equally. So if you want to set up a server behind a firewall or a router you have to open port 3337 and 3338 in the firewall and configure your router to forward the same ports to the pc where the server is running.
4.3 Using ReDaR

When the GUI client starts up you will see:

![ReDaR GUI](image)

Figure 4.3: Start up

After closing the Credits window, you can see 4 buttons in the main list.

- Control Panel: This button will open a window that allow commands to be sent to the WSN.

- Info Panel: This button will open a window that allow monitoring of the feedback control parameters for the implemented error controller.
• Close Connections: This button will close the current connection.

• Connect: This will open a window that allow you to connect to a remote or local server.

Additionally there are two zoom buttons: Zoom In and Zoom Out.

Whether you run a local server or not, the first thing you should do is to connect to a server. To do this press the "Connect" button. A popup window will show up where you can specify the IP or WWW address of your server. If you are running a local server type "localhost" and press enter to connect.

![Connection Panel](image)

Figure 4.4: The Connection Panel

If all went well you should now be connected to the base station and be able to interact with the wireless sensor network. As soon as a connection has been established, a command will be sent to all the sensors in the network to tell them to start transmit data. A red dot will show up on the map where the server is located and by zooming in on that dot you can see the sensors in the network and the network topology. See figure 4.5.
4.3.1 The Control Panel

With the Control Panel you can send commands to the WSN. These commands are:

- **Leds On**: it turns on the green light on all sensors.
- **Leds Off**: it turns off the green light on all sensors.
- **Start all Sensors**: it commands all sensors to start transmit data at 4 packets per second.

Figure 4.5: Sample picture of the network topology which can be seen by the zoom panel
- Start at rate: it commands the selected sensor to start transmit data and wait the specified number of milliseconds between each packet.
- Set rate to: it sets the transmission rate, in wait time between each packet, for the specified sensor.
- Fix redundancy at: it allows to fix the redundancy.
- Auto ECC: it turns on automatic redundancy adjusting using the error controller described in this paper. This is on by default.
- Stop all Sensors: it commands all sensors to stop transmitting data.

![Figure 4.6: The Control Panel](image)

4.3.2 The Info Panel

With the Info Panel you can see how the error controller works.

It is possible to specify the sensor that you want to supervise by inserting the sensors identification, then press select to get the data. In figure
you can see that there are four sub-diagrams. All sub-diagrams shows data retrieved and calculated over the last hundred blocks. More specifically, the first sub-diagram shows the packet loss, calculated as an average over the last 100 packets received. The second sub-diagram shows the distortion, calculated as described in chapter 3. The third sub-diagram shows the redundancy, and the forth sub-diagram shows the cost function.
4.4 Beneath the shell

This section will describe how the java part and the NesC part of Redar is built up.

4.4.1 The Main Class and the Servers

The main class is called Redar.java. From the main class two servers are started and a frame window is created. The two server separately handles commands from and to the network. The code for them can be found in MyFwdServer.java and MyCmdServer.java. Each of these include a simple java Server Socket and a mote listener. To connect to one of these servers a client socket is necessary. The code for the clients can be found in MyFwdClient.java and MyCmdClient.java. We will describe the details of the connection later.

4.4.2 The Main Frame, the Control Panel and the Info Panel

The window that you see when ReDaR starts is a Java Frame and the code can be found in MyFrame.java. As we have seen in the previous section this contains a menu bar and a map with graphics. The map field is a JPanel and the code for it is in MyGraphPanel.java. The code for the Control Panel can be found in ControlPanelDialog.java and the code for the Info Panel can be found in InfoPanel.java.

To work properly these classes needs information from the sensors. Whenever a message is received, it is stored in an object of the class Node. The Node class is a queue, where each Node object holds information about one sensor, its parent, its latest sensor values and an address in the memory to the next Node object. The Node class also holds the implementation of the Error Controller. At each message received, a calculation for the packet loss as an average over the last 100 samples is performed. Whenever 50 packets is received (1 block), an update of the error controller parameters is made and stored to a files called Node#.txt, where # stands for the sensor identity.

With the information stored in the node queue, MyGraphPanel and InfoPanel has enough information to display the nodes position on the map.

\[2\text{At this moment, ReDaR does not include any type of positioning algorithm. Therefore}\]
and show the diagrams in the Info Panel.

4.4.3 How the Communication works

First let us see what the ReDaR packets looks like. If you start ReDaR, open a connection and start to receive packets you can open a listener that will print out these in hexadecimal code on the screen. The program is called Listen and is included in the sf packet (which also include SerialForwarder). To start it, open a cygwin window and type:

```
java net.tinyos.sf.Listen
```

The ReDaR Packet Format

Figure 4.8 shows a sample picture of what ReDaR packets looks like.

![Sample picture of Redar packets](image)

Figure 4.8: Sample picture of a few Redar packets

A ReDaR packet consist of 38 Bytes of data. Note that the least significant byte is to the left in each field.

- The first 20 bytes are the AM header (10 Bytes) and the multihop header (10 Bytes). These two holds information about source, destination, parent sensor node, etc. They are the same as are used in Aegis. The actual sensor positions are not shown.
A detailed description of these two headers can be found in the Aegis Manual [5] on page 31.

- The next 2 bytes belong to the ReDaR header and describes what type of message this is. In figure 4.8 these are 0x00 0x00, which means that this is a sensor reading.

- The following 8 bytes are the sensor readings. In ReDaR actual sensor readings are not sent therefore these are constants set to 0x6F 0x00 = 111, 0xDE 0x00 = 222, 0x4D 0x01 = 333 and 0xBC 0x01 = 444.

- The last 8 bytes are the ReDaR tail. The first 4 bytes are a flag field used to tell if this is an application packet (0x00 0x00 0x00 0x00 = 0) or a redundancy packet (0x01 0x00 0x00 0x00 = 1). The last 4 Bytes is a sequence number spanning from 0 to 30000.

The Mote Communication

To communicate with the WSN via SerialForwarder, the two servers that are created in Redar.java implement the Java interface MessageListener. MessageListener provides methods to send and retrieve messages from a WSN via SerialForwarder. Furthermore an object of the class MoteCom and a listener for it is necessary for the communication to work properly. These two lines creates a MoteCom object and registers a listener for it:

```java
mote = new MoteIF(PrintStreamMessenger.err, 125);
mote.registerListener(new RedarMsg(), this);
```

This Listener is the class itself and requires the implementation of MessageListener. When messages are received from a WSN, the method messageReceived is called upon. The message will be of the class Message and its payload will be composed of the data sent from the sensor except for the AM heather. Therefore it is necessary to extract the important information from it, i.e. the Redar message payload. This can be done via

```java
messageReceived(...) {
    MultihopMsg multiMsg =
    new MultihopMsg(recievedMsg.dataGet());
    RedarMsg tmpAMsg =
    new RedarMsg(multiMsg.dataGet(),multiMsg.offset_data(0));
}
```
The code to send packets to the sensors can be found in RedarInject.java. The command packets are of the class SimpleCmd, which is part of the Surge application that is included in the Moteiv package. A command packet of this type is built up via the following lines:

```java
SimpleCmdMsg packet = new SimpleCmdMsg();
packet.set_seqno(sequenceNo);
packet.set_hop_count((short)0);
packet.set_source(0);
packet.set_type(messageType);
packet.set_addr(target_sensor_address);
packet.set_value1(Value1);
```

This will be a broadcast packet and sent to all sensors in the network.

Once the mote communication is established, the following line can be used to send the command message:

```java
MyCmdServer.getMoteIF().send(TOS_BCAST_ADDR, packet);
```

Note that this line uses the MoteCom of MyCmdServer.

### 4.4.4 The Tmote software

As all programs written in NesC, Redar has a linking file, ReDaR.nc, and an implementation file, ReDaRM.nc. The linking file tells the compiler what components shall be included, and what names they will go under. The implementation file ReDaRM.nc holds the application code. Additionally, in the Redar Tmote directory you will find a file called ReDaR.h. This file holds global variables and the Redar packet format.

Redar basically work after a timer. Each time the timer is fired it sends an alarm (event) and a method named timer.fired() is called. In this method the sequence number is updated and the flag field is set to either 0 (application packet) or 1 (redundancy packet).

```c
event result_t Timer.fired() {
    seqno++;
    if(seqno > 29999) seqno = 0;
    redpointer = seqnoif(redpointer > N-redundancy-1)
    flags = 1;  // redundancy packet
    else flags = 0;  // application packet
}
```
Finally the packet is sent via SendData().

```c
TOS_Msg gMsgBuffer; //define a TinyOS message

task void SendData() {
  RedarMsg *pReading;
  uint16_t Len;
  pReading = (RedarMsg*)call Send.getBuffer(&gMsgBuffer,&Len)
  pReading->type = ReDaR_TYPE_SENSORREADING;
  pReading->parentaddr = call RouteControl.getParent();
  pReading->readingT = 111;
  pReading->readingH = 222;
  pReading->readingLtsr = 333;
  pReading->readingLpar = 444;
  pReading->seq_no = seqno;
  pReading->flags = flags;
}
```

Whenever a command message is received the method ReceiveCmdMsg.receive(...) is called. To process the command, first ProcessCmd.execute() is called and then the message is interpreted in cmdInterpret(). Here follows a cutting from ReDaRM.nc:

```c
event TOS_MsgPtr ReceiveCmdMsg.receive(TOS_MsgPtr pmsg){
  call ProcessCmd.execute(pmsg);
}

command result_t ProcessCmd.execute(TOS_MsgPtr pmsg) {
  post cmdInterpret();
  return SUCCESS;
}

  task void cmdInterpret() {
    switch (cmd->type) {
      case ReDaR_TYPE_LED_ON:
        // do whatever...
```
break;
case ReDaR_TYPE_LED_OFF:
    // do whatever...
    break;
case ReDaR_TYPE_START:
    // do whatever...
    break;
case ReDaR_TYPE_STOP:
    // do whatever...
    break;
case ReDaR_TYPE_SETRATE:
    // do whatever...
    break;
case ReDaR_TYPE_SET_REDUNDANCY:
    // do whatever...
    break;
default:
    default:
    }
signal ProcessCmd.done(msg, SUCCESS); // Action Performed
}

default event result_t ProcessCmd.done(...) {
    post forwarder(); // rebroadcast the packet
    return SUCCESS;
}
Chapter 5

Experiments

This chapter will describe the implemented parameters and show how we tested the implementation.

5.1 How the tests were performed

To test the implementation of the error controller three types of experiments were done. In the first experiment we kept the redundancy constant and let the packet loss vary. In the second experiment, we tried to keep the packet loss constant and vary the redundancy manually. In both of these experiments the error controller was inactivated, so the last experiment was run with the controller activated and varying packet loss. In all experiments only one sensor node was used besides the base station. The reason is explained in 5.1.4. In figure 5.1 the test bed setup can be seen.

Before we address the experiments, we will have a look on how the controller was implemented.

5.1.1 The Packet Loss and the Block Length

The packet loss was calculated as an average over the last 100 received packets. The block length was 50 packets. After each block received, a number of calculations is performed\(^1\) to calculate the redundancy in the next block.

\(^1\)The calculations are described in chapter 3.
Figure 5.1: The test bed setup. Only one free sensor node was used. It was wireless connected to a sensor node that was attached to the USB port of a laptop and formed the base station. With another laptop (the client) it was possible to connect to the base station via an IP network. With Redar running on both computers it was possible to perform the experiments on the client side.

5.1.2 The Change Detection Filter

The first of these calculations is the filter to detect quick changes in the packet loss. This filter is a SPRT as described in section 3.1. It consist of two windows, one short window $w_1$ (the packet loss over the last 500 packets) and one large window $w_2$ (the packet loss over the last 2500 packets). If we subtract the average of the small window with the average of the large window, we will get a value that we can call $s$. By adding $s$ and +1 or -1 respectively to two separate functions, that we call $g_u$ and $g_d$, we get a detection filter for quick changes upwards and downwards.

$$g_d = g_d + s + 1$$
$$g_u = g_u + s - 1$$

Once either of these functions reaches a certain threshold they send an alarm and a sudden change in the packet loss is detected.
5.1.3 The Controllers

Once an alarm is triggered the feed forward controller\(^2\) kicks in. It gets the optimal redundancy from the look-up table and resets the feed back controller variables. Since the small window in the change detection filter is 500 packets long and \(s\) in the functions above will be 0 then, changes in the packet loss can’t be detected with a better precision then each 500 packet i.e. each 10th block.

While the packet loss is stable and no alarm is triggered, the feed back controller\(^3\) is active. After each block received it calculates the estimates \(u^d_b\) and \(c^d_b\) and updates the redundancy via the formula in section 3.2.

5.1.4 How we managed to control the Packet Loss

To perform the experiments described in the beginning of this chapter, we had to have a good influence on the packet loss process. The packet loss had to be kept fairly constant over 100 or more blocks, then it had to be changed on a given command to another fairly constant value. This proved to be surprisingly easy once we detected that if we increased the packet transmission rate, we got packet losses even with only one sensor connected to the base station. At 50 packets per second, we had a packet loss of about 40% and at 20 packets per second we had about 5-10% packet loss.

The bandwidth that was used for these experiments was 38.4 kbps. Remembering that each packet consisted of 38 bytes, we can calculate that it takes

\[
38 \times \frac{8}{38400} = 7.9\text{ms}
\]

7.9 milliseconds for each packet to be sent and the wait time between each packet is

\[
\frac{1}{50} = 20\text{ms}
\]

20 milliseconds. This gives the CPU 12 milliseconds to process new data. That span should be more than enough to keep the CPU from over-flooding, considering that the CPU works at 8Mhz. Additionally the transmission of data is a separate process and doesn’t load the CPU, which means that the

\(^2\)Described in section 3.1
\(^3\)Described in section 3.2
CPU actually has 20 milliseconds to compute new data. So the packet losses is not in the CPU or in the sensor itself, but rather in the communication with the base station.

5.2 The Experiments

5.2.1 Experiment 1: Constant Redundancy

In this Experiment the redundancy $u$ was kept constant at 10 packets per block, while we let the packet loss vary between 0 and 50%. The results can be seen in figure 5.2. On the x-axis are the block numbers, which corresponds to the time. On the y-axis, the first diagram shows the packet loss $p$, the second diagram shows the distortion $d$, the third diagram shows the redundancy $u$ and the fourth diagram shows the cost function $c$.

![Figure 5.2: Experiment 1: Constant redundancy and varying packet loss.](image)

Looking at figure 5.2, as expected the number of lost packets (the distortion) is 0 at 0% packet loss. When the packet loss increases to 35% we get a distortion of 5-10 packets per block. Since we have 10 redundancy packets here, hence 40 application packets per block, a distortion of 5-10 packets per block equals an application packet loss rate of 12.5-25%. As the packet
loss is reduced to 25%, the distortion, with the exception of a few spikes, is reduced to 0. At 30% packet loss the algorithm is getting into trouble and a few spikes becomes a regular distortion of 5 packets per block. When the packet loss reaches 45%, it gets even worse and the distortion is increased to around 15 lost application packets per block or almost 40% lost application packets.

So what we can see is that with 10 redundancy packets the algorithm is effective at 25% packet loss or less, but as the packet loss increases it slowly gets worse until it has no effect at all.

5.2.2 Experiment 2: Constant Packet Loss

In Experiments 2 the packet loss was kept constant at 30-35% and we let the redundancy vary by setting it to user given values. See figure 5.3.

![Figure 5.3: Experiment 2: Constant packet loss and varying redundancy.](image)

As you can see in figure 5.3 a redundancy of 5 is not enough to secure a good communication with 30% packet loss. With this redundancy we have a distortion of about 10 and this is unacceptable. However, when the redundancy is increased to 20 packets, the distortion is reduced to 0. If we try to lower the redundancy to 15, we can see that we get a few spikes but
this is acceptable. If we try to lower the redundancy further, we are getting
a regular distortion again - not acceptable.

This shows us that, with 30% packet loss a redundancy of about 15 pack-
ets is needed to secure a good communication. More then that is unnecessary.

5.2.3 Experiment 3: With the Controller

In Experiments 3 the error controller was reactivated. With the error con-

controller activated, we can see that the redundancy follows the packet loss. When we have 0% packet loss (perfect communication) the redundancy is reduced to 0 packets, so the line can be used to its full potential. When the packet loss increases to 30% the forward controller quickly reacts and raises the redundancy to 15, but this proves not to be enough and the redundancy is again raised (to 20 this time). At this point the backward controller is trimming the redundancy to a good value, slightly lower then 20. When the communication gets better and the packet loss drops, the forward controller reacts again and lowers the redundancy. As before the backward controller trims the redundancy to a good value.
Chapter 6
Conclusion and Future Work

In this thesis, we have presented a new way to cope with packet losses in Wireless Sensor Networks. This technic has been implemented on a test bed with real wireless sensor nodes. The test bed includes a feedback-feedforward network coding control that allow for reliable reception of delay sensitive data, while reducing the traffic overhead of the network coding.

Experimental results show that the control mechanisms are actually able to track the variations of the packet losses introduced by the wireless transmission. Moreover, the experimental results confirm those obtained via earlier analytical simulations \[4\].
Bibliography


[10] The Moteiv Homepage:  

[11] The TinyOS Homepage:  
http://www.tinyos.net