End-to-End Acknowledgements for Data Collection in Wireless Sensor Networks

by

John-Paul Arp

TR10-198, February 1, 2010

This is an unaltered version of the author's MCS thesis

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca
http://www.cs.unb.ca

Abstract

This thesis presents a novel method to improve the reliability of data collection in wireless sensor networks. The Disseminated ACKnowledgment protocol (DACK) builds on collection and dissemination protocols to provide end-to-end acknowledgement of data samples. A DACK protocol implementation was tested using simulations and experiments on TelosB motes running TinyOS 2.1. Experiments were carried out on three floors of a building, with 14 motes transmitting data samples continuously until battery exhaustion. Results show that the DACK protocol recovers all data samples that would have been lost using a collection protocol only. The benefit of increased data collection reliability comes at the cost of increased communication. In one experiment with 14 motes, 729 data samples were dropped from a total of 749,904 data samples sent over six days; all these dropped samples were recovered using the DACK protocol. This same experiment required an additional 720 collection packets to resend the dropped samples (in addition to the original 467,778 collection packets) plus 18,733 DACK packets.

Acknowledgements

I would like to thank the following people who helped make the completion of this thesis possible:

- My supervisor Brad Nickerson, for his detailed feedback, encouragement, patience, and financial support.
- My father Paul Arp, for his financial and emotional support, as well as my mother Maureen, and two brothers Peter and Alex.
- My friends Joel, Nash, Jeff, Natalie and Greg for distracting me from the stress of completing this thesis.
- The UNB Faculty of Computer Science for their continued support of my research, that stretched well beyond the ideal timeline.
- The TinyOS community, which provided all of the tools used to implement and test the DACK protocol.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

$ackwaiting$	aflag indicating that the base staion is waiting for it's last disseminated message to be acknowledged.
ASN	sequence number of the most recent consistantly acknowleged data sample
$ASNmismatch$	a flag indeicating whether a mote is reporting a differnt ASN than the base station's MI
B	th acknowledgment vector for mote M
BS	Base Station
C	a DACK collection packet of the form $(MID, DSN, SO, ASN, LSN, S[s])$
CTP	Collection Tree Protocol
$D1$	A packet for trasnporting $D1Ack$ messages.
$D1Ack$	A $D1$ type acknoweldgment message,
$D2$	A packet for trasnporting $D2Ack$ messages.
$D2Ack$	A $D2$ type acknoweldgment message,
$D3$	A packet for trasnporting $D3Ack$ messages.
$D3Ack$	A $D3$ type acknoweldgment message,
$DACK$	Dissemination ACKnowledgment protocol
DIP	DIssemination Protocol
$Drip$	Drip dissemination protocol
DSN	sequence number of the most recent dissemiation packet sent to mote M
E_D	Dissemenation event
E_R	Report event
E_S	Sample event
F	The number of data samples that fit inside $Storage$

fn	'false negatives': the number of data samples in nd the DACK protocol counts as dropped, but were actually received.
fp	'false positives': the number of data samples in nr that were counted as recovered after a *D1Msg* or *D3Msg*, but would have been recovered anyway.
G_C	Collection Delay
G_D	Dissemination Delay
I_D	Dissemination interval
I_R	Report interval
I_S	Sample interval
L	the bit length of B
LSN	sequence number of the most recently sent data sample
M_i	a mote in wireless network N with $MID = i$
M	a mote in wireless network N
MI	the Mote Index containing sequence numbers and mote status for mote M
MID	Mote Identifier
MT	the Mote Table is a hashtable on the base station that contains the MI for each mote in the network N.
n	number of motes in the wireless network N
N	a wireless network
nA	this is the number of samples the base station becomes aware of through the ASN and LSN values in collection packets. This number may be smaller than nS if a mote experiences storage overflows.
nC	the total number of collection packets sent by motes in the network during T.
nCR	the total number of collection packets sent by motes in the network during T containing data samples that were sent previously.
nD	the total number of DACK dissemination packets disseminated by the base station during T.
nd	the number of data samples sent by motes to the base station, but not received by the time the base station sends a $D1Ack$ or $D3Ack$ to recover the data sample. This can also be thought of as the number of data samples dropped by the collection protocol.

nl	the total number of data samples that the base station gives up on trying to acknowledge.
no	this is the number of samples the base staion sent a $D1Ack$ or $D3Ack$ for that has not yet been recovered or lost.
nr	the number of data samples that were recovered by the base station after sending a $D1Ack$ or $D3Ack$ message requesting a mote resend dropped samples.
nRX	the total number of data samples received at the base station.
nS	the total number of data samples sent by a mote.
nso	the number of storage overflows that occurred during an experiment.
nwo	the number of window overflows that occurred during an experiment.
R	a data sample reading
rr	the recover ratio, measured as $nr/(nd - no)$
RSN	the sequence number of the last data sample received by the base station
$RSNT$	the timestamp of the data sample with the sequence number = RSN
S	a data sample with the structure (R, SID, SN, TS)
s	the number of data samples that can fit in a collection packet.
SID	the sensor identifier
SID	a data sample sensor identifier
SN	a data sample sequence number
SO	the number of storage overflows that occured on mote M
$Storage$	the allocated storage space on mote M
T	the total amount of time that the network was running.
V_i	a sensor on mote M with $SID = i$
v	the number of sensors on more M
V	a sensor on mote M
W	the acknowledgment window size
$WindowOverflow$	a flage indicating mote M has a window overflow
WSN	Wirless Sensor Network

Chapter 1

Introduction

By the end of the 20th century, the continued miniaturization and mass production of micro-processors, micro-sensors, and radio-frequency integrated circuits (RFIC) invited earnest research into the problems and potential of building Wireless Sensor Networks (WSNs). A WSN is composed of autonomous computing nodes, or ”motes”. Each mote typically contains a microprocessor, an RFIC, non-volatile storage, an energy supply, and either integrated sensors or an expansion port for an external data acquisition board (DAQ). In 2003, the potential applications of WSNs lead the MIT Technology review to rate WSN as one of the top ten emerging technologies that will change the world [41].

Since then, WSN research has been rapidly expanding into research institutions around the world, with several hardware platforms [40, 16], RFICs [6, 7], operating systems [23, 11], programing languages [13], and standards [18, 25] emerging to better accommodate research and development. Applications that researchers have been working on include (but are not limited to): environmental monitoring (e.g. habitat [37, 52], geological [54], industrial [44], structural [21], etc.,); health [2]; object tracking [43]; surveillance [3]; hazard detection [47]; and distributed control systems (e.g. agricultural [1, 15], industrial [53]).

1.1 Communication in Wireless Sensor Networks

Communication in WSNs posed many new challenges for researchers. The limited energy, processing speed, storage, packet space, and memory of motes combined with bursty asymmetric radio links, proved to be a hostile environment for long tested network protocols like TCP/IP. Researchers typically aimed to make their WSN communication stacks follow the OSI model, however optimizing for the limited resources and harsh environment forced many teams to tightly couple the layers in their networking stack. This lead to a wealth of novel communication schemes (see [28]), but it also caused a fair bit of splintering in the research community. An brief overview of these techniques can be found in chapter 2. Recently, research efforts have been focused on building a modular architecture to facilitate the most popular communication patterns [48].

Research into networking protocols and modular network architectures has evolved to the point where WSN developers can download and combine different networking protocols into there applications. This paper explores building a novel transport layer acknowledgment protocol on top of two commonly used network layer protocols: collection and dissemination.

1.2 Data Collection and Dissemination in Wireless Sensor Networks

Collection protocols manage the task of delivering data contained within each node in the network to a base station. There are mote-centric, and gateway-centric methods for collecting data. In a gateway-centric method, the logic for selecting multihop communication paths is entirely on the gateway. The mote-centric method generally involves building a routing tree between all nodes in a WSN to one or more gateway nodes, and then making a best effort to deliver packets from each node to a base station. The Collection Tree Protocol

(CTP) [12] currently available in TinyOS 2.x makes a best effort to collect sampled data. Though it does not guarantee 100% delivery, it achieved 97% efficiency in harsh network environments [14].

Another common technique is dissemination. Dissemination protocols reliably and efficiently deliver a common data set to every node in a WSN. Dissemination is typically used for sending configuration parameters [30], or for sending large binary images for wireless in-network reprogramming [17]. Unlike collection, dissemination protocols aim to guarantee reliability to all connected motes.

1.3 Motivation and Contribution

The reliability of data dissemination creates an opportunity to reliably acknowledge packets delivered by a collection protocol. To our knowledge, no published work currently investigates this opportunity. We anticipate that the end-to-end data reliability will improve with an investment in extra dissemination messages.

To achieve this result we have implemented and tested the Disseminated end-to-end ACKnowledgment (DACK) Protocol. By "end-to-end" we mean that for each data observation sent in a collection packet by a mote, the mote receives an acknowledgment that the base station has received the data observation, and the base station receives an acknowledgment that the motes have received the data sample acknowledgment.

1.4 Thesis Structure

This thesis is broken up into 8 chapters. Chapter 2 gives an introductory overview of communications in wireless sensor networks, and provides the context within which the this thesis makes a contribution. Chapter 3 describes the various tools and libraries used to implement the experiments and simulations discussed in this thesis. Chapter 4 details the Disseminated end-to-end ACKnowledgment (DACK) Protocol in detail. Chapter 4 includes discussions about the algorithms used by the DACK protocol, and illustrates the various communication scenarios using time-lines, and provides a series of metrics which can be used to measure the effectiveness and cost of the protocol. Chapter 5 goes into greater detail of how the DACK protocol was implemented for the simulations and experiment, including descriptions of the source code and packet structures. Chapter 5 also presents tests that we use to verify that the DACK protocol is functioning properly. Chapter 6 analyses the results of two simulations, and Chapter 7 analyses the results of four experiments. Conclusions are presented in Chapter 8, along with a discussion of methods to improve the DACK protocol in future work. Appendices are also provided, containing the source code used in the simulations and experiment, and a more detailed discussion of the implementation.

Chapter 2

Communication in Wireless Sensor Networks

This chapter provids the context to which the DACK protocol is making a contribution. The aim of this chapter is to introduce the challenges of wireless communications in WSN, as well as the techniques that have been developed to address them.

2.1 Challenges

2.1.1 Energy Conservation

The principal constraint in outdoor WSNs is energy conservation. The consequences of minimizing energy use cascades into all aspects of WSN development. In the hardware, memory size, processing speed, and radio strength are all limited in order to minimize overall energy consumption. Wireless data transmission is typically the most energy consuming task a mote will have to perform. Wireless reception also creates an energy drain as it requires the mote to power its RFIC and CPU. Energy efficient communication protocols need to minimize the number of transmissions, and the amount of time spent listening for new messages to keep the hardware in a low power state, all while operating within the limited buffer and configuration memory space provided by the mote. The "duty cycle" of a mote, is the ratio of time the mote spends in a powered state over the time it spends in a sleep state.

2.1.2 Hardware Constraints

As of 2010, there are a wide variety of motes on the market. Mote hardware is designed with energy consideration in mind. The main circuit board for a mote will typically include a micro processor and integrated circuits for radio, storage, and sensing. CPU's range from 8-bit Atmel Atmiga [8] in the Mica2 and MicaZ [19] motes, to the 32-bit ARM920T [29] in the SunSPOT [45]. The common requirement for all mote CPU's, is to be able to go into low power sleep states below 50 uA, when not being used. The other integrated circuits can be toggled on or off by the CPU to reduce energy consumption. For example, energy conservation requires applications to keep their radios switched of as much as possible. Motes usually have from 4KB to 10KB of RAM, 48KB to 128KB of program memory, and 512KB to 1MB of non-volatile memory [22] for storage.

2.1.3 Harsh Wireless Environment

Analysis done in [14] shows that links in WSN are asymmetric, and that signal strength could oscillate between periods of very good and very poor reception for each link. These conditions can lead to inaccurate link estimations, and inaccurate link estimations can lead to inaccurate neighbor and routing tables in multihop networks. Multihop protocols need to be able to adapt to dynamic wireless conditions in order to maintain robust communications.

2.1.4 Implementations of Traditional Networking Models are Too Heavy

The above constraints make WSN unsuitable for established but relatively resource heavy networking protocols such as TCP/IP. The 7-layer Open Systems Interconnection (OSI)

Reference Model for a layered communication protocol [20], and the the 4-layer TCP/IP Model [5], shown in Figure 2.1, demand a decoupling of layers that has proved difficult to do in WSN applications [38]. The 20-byte IPv4 header (or the 40-byte IPv6 header), plus the 20 byte TCP header, were not designed for early mote hardware like the Mica2 in which the default packet size is itself only 38 bytes. Later generations of Motes using IEEE 802.15.4 compatible protocols (see section 2.2.1) have larger maximum packet sizes that are limited to 123 bytes [18]. The TCP/IP header still occupies a significant amount of packet overhead. Further routing protocols like Open Shortest Path First (OSPF), the Border Gateway Protocol (BGP), and the Routing Information Protocol (RIP), require large routing tables that are not suitable for memory constrained motes. Recently progress has been made in migrating TCP/IP to WSN using compressed headers [32]. But the challenges to porting classical routing protocols to large networks or resource constrained motes remains.

2.1.5 Modularity

Because of the difficulty in using traditional networking protocols, developers of early WSNs often had to implement the entire communication stack from scratch. As new ideas were being developed at various research institutions, interoperability between the various protocols was as challenging as it was desirable.

Figure 2.1 contrasts a typical WSN stack scenario with the OSI and TCP/IP model stacks. The TCP/IP model defines protocols for layers 2 (Data Link) through 4 (Transport) of the OSI model. The key feature of TCP/IP model is the Internet Protocol (IP) layer which is common to all TCP/IP applications. The IP is a network addressing protocol which assigns a unique IP address to for up to 2^{32} hosts (using IPv4) or 2^{128} hosts (using IPv6) on a network. Below the IP layer, multiple Data Link layer protocols route data between hosts that are directly connected via a shared medium, such as the air, a wire, or Local Area Network (LAN). Above the IP layer transport protocols are used to facilitate

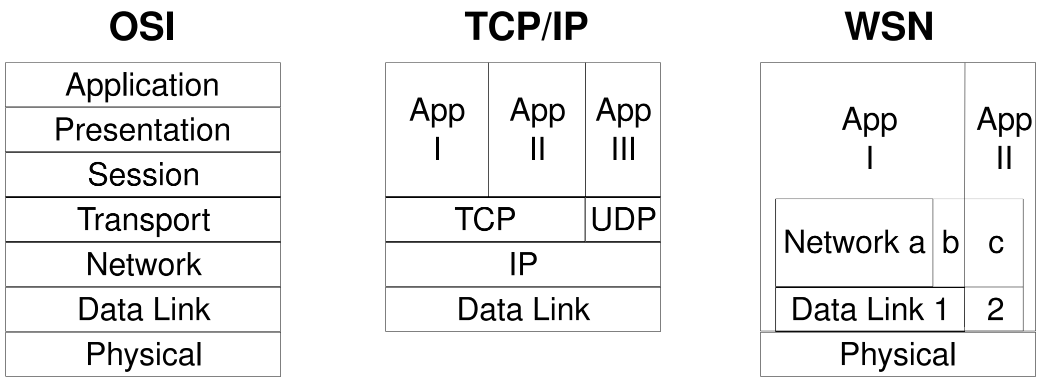


Figure 2.1: A “messy: WSN communication stack on the right, beside the more standardized 4 layer TCP/IP model in the middle, and the 7-layer Open Systems Interconnection (OSI) reference model on the left. In the TCP/IP stack, the common IP layer decouples transmission protocols from data link protocols. In the WSN stack on the right, there is no common networking layer. Transport and networking functionality are frequently combined, and may or may not be dependent on a specific data link layer protocol. The gap between the edge of the stack and the network and data link protocols is used to illustrate the interaction many WSN applications have with the lower layers.

the transport of packets from one host to any other host with an IP address, whether it is directly connected to it, or ten routers away and physically on the other side of the planet. The Transport Control Protocol (TCP) implements a reliable end to end link, in which all lost data is recovered, whereas the User Datagram Protocol (UDP) simply makes a best effort to deliver packets. It is up to applications to decide which transport protocols to use. Also in the transmission layer, protocols like the Border Gateway Protocol (BGP) and Routing Information Protocol (RIP) route IP packets across Wide Area Networks (WANs).

On the right hand side of Figure 2.1 we see the messier WSN stack. There is no common network addressing layer. Instead there are multiple networking protocols in which some use addressing schemes, whereas others are address-less (see sections 3.4.1 and 3.4.2 for examples of common WSN address-less networking protocols). Some network protocols may require a specific data link layer protocol be used, and it may be impossible for two networking protocols and/or two data link protocols to exist on the same mote. The Dozer application [4] requires a time synchronization and a Time Division Multiple Access (TDMA) scheme (see section) to function, whereas the Collection Tree Protocol, was tested to work with and without TDMA [14].

Application developers may want tighter control of the lower networking layers. If an application needs to conserve power, it may want to change the timing of the power cycling of the radio. In a dense and noisy network, the application may want to reduce its radio strength, or change the radio channel. A jam in one networking protocol, may provoke it to send data via another. A more exhaustive discussion of the challenges to modularity can be found in [38].

Recently the situation has improved. Many data link layer protocols and networking protocols have become popular and standardized, helping to focus the effort on modularity. Though many parallel efforts still exist, thanks to contributions by the TinyOS and other research communities, it is now possible to download and incorporate popular networking protocols into a single application. However, many non-compatible protocol stacks are still evolving, and a true modular architecture for sensornets still remains an evolving challenge [49].

2.2 Protocols

This section describes some of the more significant protocol developments at each layer in the WSN stack.

2.2.1 Physical Layer

Physical layer communication hardware in WSNs currently varies between hardware vendors. The original mica [16] used an RFM TR1000 radio, that used on-off keying (OOK) or amplitude shift keying (ASK). The mica2 radio supports 38.4 Kbaud radio interface using frequency shift keying (FSK) modulation in the 868/916 MHZ and 433 Mhz ISM bands. Manchester encoding is used at the bit level. Typical transmission range is between 10 and 100 meters. Other hardware platforms are also available.

In WSNs there is a significant trade-off with respect to where the boundary between hardware and software controlled communications is placed. Increasing the boundary on the hardware side can lead to increased transmission success rates, but also reduces fine grained control over power management and link level communications [28]. For example, the original mica's RFM TR1000 radio [42], had an interface allowing quick toggling of the radio. The mica2's CC1000, by comparison, has more advanced communication functions in hardware. This provided more robust communications, but also increased the radio toggling time by an order of magnitude. [40]

In 2003, the IEEE published the 802.15.4 standard for Wireless Personal Area Networks (WPAN) [18]. This standard described a physical and medium access control (MAC) layer specification that can be used in WSNs. Fourth generation motes, including the Micaz [19], and the TelosB [40] are ZigBee compliant using the Chipcon CC2420 [7] radio. 802.15.4 specifies usage in the unlicensed 868 MHz, 915 MHz, and 2.4 GHz ISM bands. The transmission rate in the 2.4 GHz band is up to 250 Kbits per channel using Direct Sequence Spread Spectrum (DSSS) and Offset Quaternary Phase-Shift Keying (0-QPSK) modulation. The ZigBee standard has strong industrial support, and will likely become a standard for future sensor networks. The 802.15.4 standard devices, including the CC2420, also includes hardware implemented schemes for encryption, authentication, link layer acknowledgments, and CRC checking of each packet.

2.2.2 Link Layer

Link layer communications in wireless networks must contend for access to a shared medium. There are many medium access control (MAC) protocols for wireless communications. Frequency Devision Multiple Access (FDMA) allows different links to share the same airspace by using different frequencies. 802.15.4 defines 16 5Mz channels; however, most WSN applications developed in the 916 Mhz band have been designed to operate in a single channel.

One method for sharing a channel is Time Division Multiple Access (TDMA), in which different motes are assigned different time slots for message transmission. TDMA does not scale very easily, and has difficulty adapting to changes in network topology. Wired Ethernet uses a MAC protocol called CSMA/CD (Carrier Sense Multiple Access with Collision Detection), which enabled scalable plug and play networking. In this scheme each connected device decides whether or not to transmit by sensing if the communication channel is currently being used. If the channel is free, the device will send a message, otherwise, the device will back off and then try again. Collision detection worked on wired Ethernet, because every message is designed to reach every node on the shared medium. This is not the case in wireless networks. It is possible that the transmission ranges of devices in a wireless network do not overlap. There is no way for a transmitting device to sense if other devices are transmitting within the range of the message destination. This is known as the hidden node problem. One common solution, known as Collision Avoidance, involves having transmitting nodes send Request-To-Send (RTS) packets, and receiving nodes send Clear-To-Send (CTS) packets to clear the air waves for communications. This system is employed by 802.11, as well as several sensor network MAC protocols.

2.2.2.1 Link Layer Acknowledgments

Link Layer Acknowledgments can be used to detect the successful transmission of packets between two motes. However, due to the small size of the maximum transmission unit (MTU, the maximum size of a datagram packet), acknowledgments can constitute significant communication overhead. The lossy nature of the wireless medium, and the low transmission power of the radios involved, forces the MTU to be small in comparison to wired, or high powered wireless networks. 802.15.4 radios MTU is 128 bytes, and 916 Mhz based radios have an MTU of less than 50 bytes. A stop and wait acknowledgment protocol can result in a 40% overhead. [55]

2.2.2.2 Adaptive Rate Control

MAC protocols in sensor networks need to cooperate to increase the energy efficiency of the entire network. In [55], Woo and Culler investigated various schemes to minimize energy consumption in a WSN. Woo and Culler used a test case sensor network, in which every mote is both a data source and a data router, and all data must be sent to a single sink node. The topology of this test network can be visualized as a tree in which the sink is the root mote. Woo and Culler's analysis assumed the motes would listen for transmissions at all times. In addition to energy efficiency, Culler and Woo designed there solution such that bandwidth was allocated fairly to all motes. To meet these two design goals, Woo and Culler proposed an adaptive rate control scheme. In this scheme, each mote linearly increases the rate in which they transmit messages towards the sink after every successful transmission, and multiplicatively decreases the rate at which messages are sent for every failed (unacknowledged) transmission. Fairness was achieved by forwarding packets of child motes towards parent motes on a cyclical basis. Further, messages originating locally from a mote were given priority over messages originating from child motes. Woo and Culler investigated various CSMA schemes for this test network. The one they found that worked best was employing a random delay before every transmission, and phase shifting transmission times. In phase shifting, when a mote finds that its transmission period overlaps with the transmission period of another mote, it will shift its period in the future, such that the transmission times no longer overlap. The method described has limited energy efficiency because the motes are required to listen at all times, except when backing off to avoid collisions.

2.2.2.3 Time Sloted MAC

Slotted protocols rely on synchronizing the wake up and sleep times of each node in a sensor network. Examples of slotted protocols include SMAC ([57]) and the IEEE 802.15.4 standard. An illustration of the power consumption of a network of 4 nodes using a slotted protocol can be seen in Figure 2.2.

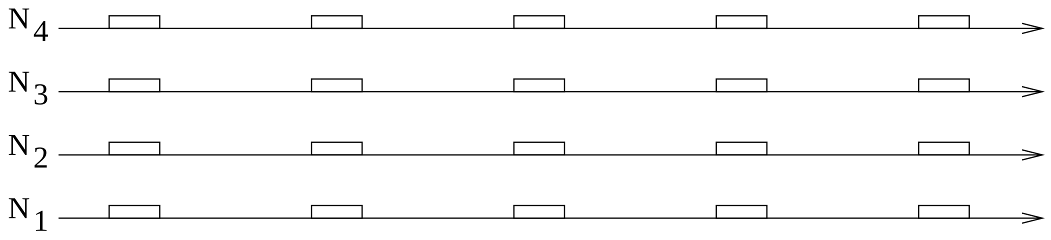


Figure 2.2: Illustration of a slotted protocol.

A low power MAC protocol called sensor-MAC (S-MAC) was proposed by Ye, Heidemann, and Estrich in [57]. The S-MAC protocol achieves low power consumption by periodically turning the radio off. For example, toggling the radio at regular intervals can reduce the duty cycle to 50%. During the period that the radio is on, communication uses CSMA/CA. Further energy efficiency is achieved by tuning the radio off during back off periods. S-MAC requires that neighboring motes synchronize their sleeping schedule with each other. S-MAC has many limitations. The sleep window increases latency at each hop on the sensor network, and, like TDMA, the scheduling scheme has difficulty scaling and adapting to changes in network topology.

2.2.2.4 Low Power Listening

Low Power Listening (LPL) is another MAC protocol technique that tries to minimize radio energy consumption. In LPL, the mote is generally in sleep mode. The mote will periodically wake up at a set interval and briefly check for radio activity. If activity is discovered, the mote will wake up and listen for a full packet. Otherwise, the mote will go back to sleep. Before sending data to a sleeping destination motes, the transmitting mote must first transmit a long preamble. The transmission time of this preamble must be at least as long as the sleep interval of the target motes in the network. Figure 2.3 illustrates motes using a LPL MAC protocol.

In [39] Polastre, Hill and Culler, proposed a MAC protocol called Berkley-MAC (B-

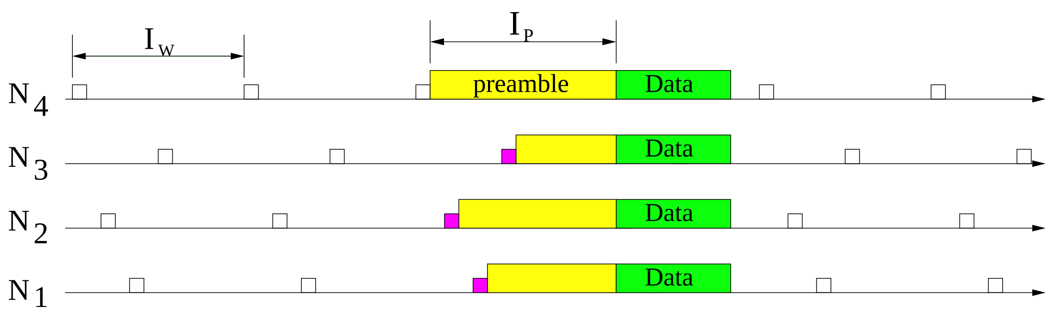


Figure 2.3: Illustration of a sampling protocol.

MAC) that can decrease the duty cycle of motes to 1% using low power listening (LPL) and an adaptive preamble sampling scheme. B-MAC is designed to provide a core minimum MAC functionality. More complicated MAC protocols, including S-MAC, could be built on top of B-MAC. B-MAC provides a methods for CSMA/CA, adaptable link level acknowledgments, packet back-off, and LPL. For carrier sensing B-MAC uses a clear channel assessment (CCA) algorithm. CCA involves estimating the noise floor to create a reference for carrier sensing. B-MAC parameters can be configures at higher layers to increase the efficiency of any given application. For example, link layer acknowledgments can be enabled or disabled on a per packet basis, and the sampling/preamble period for low power listening can be set at the application layer at any time. Like CSMA, B-MAC scales very well because there no scheduling is involved.

2.2.2.5 Link Quality Estimation

Good link quality estimations are required by higher layer protocols to adapt to changes in network topology. In [56] Woo, Tong, and Culler investigated the reliability of various link discovery and link estimation techniques. Due to the high variability of link quality in WSNs, signal strength has been found to be a poor estimator [10]. Due to the memory and processing constraints on motes, complex link quality estimation techniques such as linear regression and Kalman filters are unpractical [56]. The best link estimator that Woo, Tong, and Culler found in there investigation was the window mean with exponentially weighted moving average (WMEWMA). Woo et al. found that this estimator required as many as 100 packets to stabilize a link estimation with a 10% noise margin. This limits the rate of mobility in a sensor network.

2.2.3 Network Layer

As discussed in section 2.1.5, transport layer and network layer functionality are often mixed together in WSNs. One paper may refer to a protocol as being in the network layer, wheras another paper may refer to the same protocol as being in the transport layer. This is because the logic for transporting packets through the network, is often mixed with the logic for determining which mote to send the next packet to. The wealth of networking and transport layer protocols for WSN is too long to list here. A more exhaustive list can be found in [36]. The remainder of this section will focus on the more recent networking protocols, and the protocols that are most relevant to this thesis: dissemination and collection. These protocols are considered mature because: they have been in development for many years; they have been successfully tested in simulations and used in real world WSN deployments; and finally they have published source code that has been embraced and extended by the broader research community.

Network protocols in WSN can be divided into two planes: the control plane and the data plane. The control plane refers to messages and functions for managing the protocol parameters and network topology. The data plane refers to data being transported by the protocol. The ”overhead” of a network protocol is the ratio of control packet traffic to data packet traffic.

2.2.3.1 Dissemination

Dissemination protocols are designed to reliably deliver data to every node in a WSN. Dissemination protocols are "address-less" in that packets are not routed through the network based on a destination address. Instead, data (d) that motes wish to update via dissemination are assigned a unique key k, and a version number v. Message propagation is accomplished by having every mote periodically make sure it has the same v for k as other motes in the network.

For dissemination to work, every mote in the network must have the same k for the same d. Motes periodically broadcast their (k,v) information to check if they are up to date. To send a message reliably through the network, the mote changes the value for

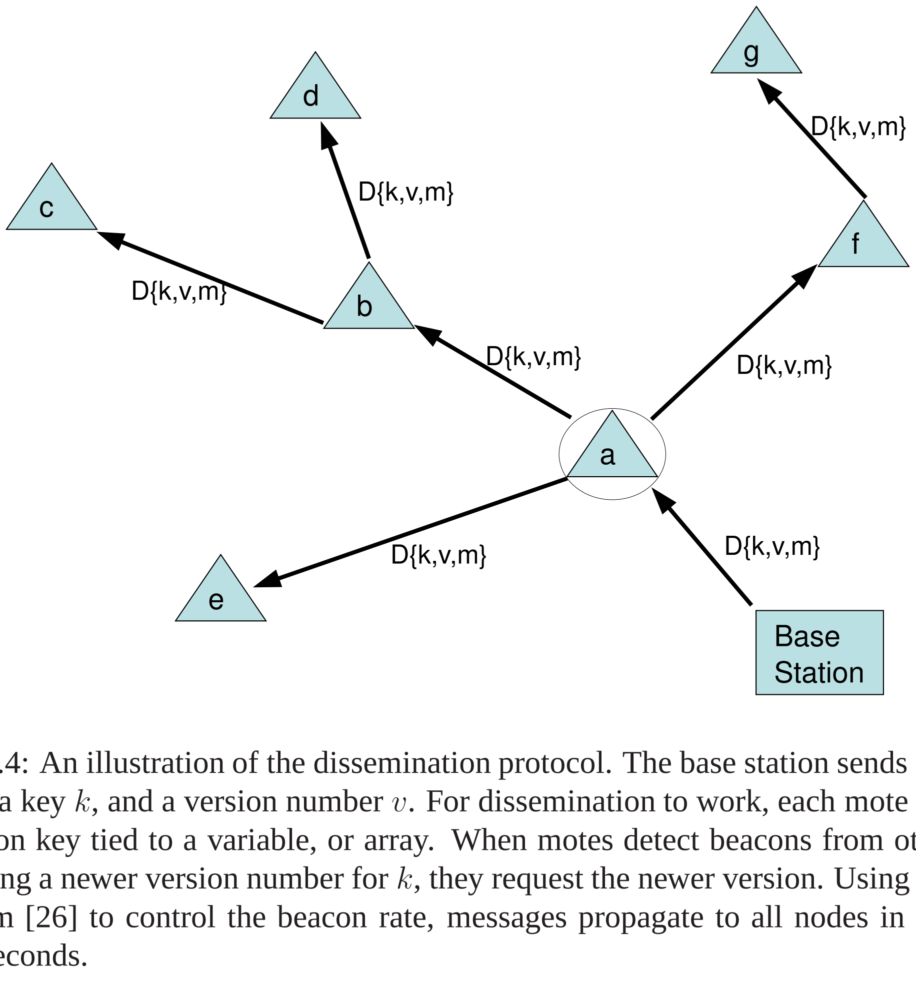


Figure 2.4: An illustration of the dissemination protocol. The base station sends a message m, with a key k, and a version number v. For dissemination to work, each mote must have a common key tied to a variable, or array. When motes detect beacons from other motes advertising a newer version number for k, they request the newer version. Using the trickle algorithm [26] to control the beacon rate, messages propagate to all nodes in a network within seconds.

d and increments v. If a mote receives a broadcast from a neighbor containing an older version number, the mote rebroadcasts updated value. This process continues until the entire network has the highest known version number. Done in a controlled fashion, this can lead to timely and reliable dissemination through the entire network.

The most popular algorithm, Trickle [26], uses a policy of “polite gossip” to control the rate at which dissemination packets are sent. Motes broadcast more frequently when they overhear old version numbers, and less frequently overhear the same version number, exponentially backing off to a maximum beacon interval under stable network conditions. In practice the trickle algorithm is able to scale dissemination time and energy consumption logarithmically with network size.

Protocols like Drip [51] and DIP [30], are designed to disseminate a lot of small values (typically control parameters) efficiently. Other protocols like Deluge [17] and Mate [27] use dissemination to transfer large multi-packet datasets for mote reprogramming.

2.2.3.2 Mote Centric Collection

In WSN, collection protocols are designed to reliably deliver messages from multiple sources (motes) to a common sink (a gateway). In mote centric collection protocols, the logic for the deciding the next hop address for a packet is located on the motes. Gateway centric collection, in which the logic for determining the next hop is processed entirely on the gateway or base station, is discussed in the next section.

Figure 2.5 illustrates a mote centric collection algorithm. Mote centric collection protocols form tree-like topologies, passing packets from child nodes to parent nodes, until packets arive at the root node. Mote centric collection can also be 'address-less'. For example in the Collection Tree Protocol (CTP), motes decide the next hop destination using a routing metric called the Expected number of Transmissions (ETX). ETX is an approximation of the expected number of times a packet will have to be transmitted before it reaches the gateway. A mote that is one hop away from the gateway, with ideal signal strength,

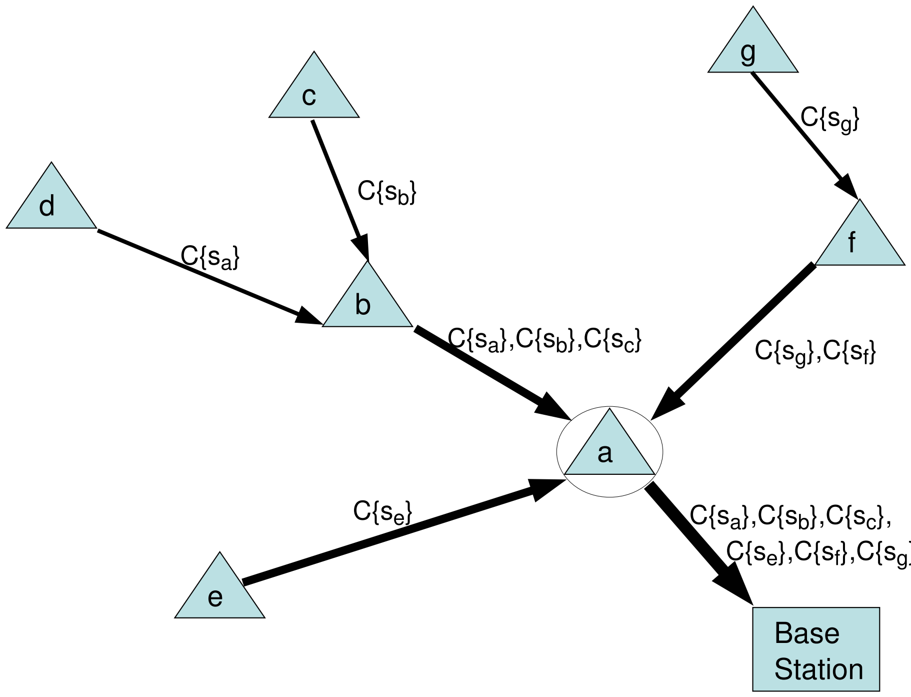


Figure 2.5: As its name implies, collection protocols transport packets from motes to a base station.

and little interference, may have a ETX of 1. Another mote with a noisier connection to the gateway may have an ETX of 1.5 or worse. A mote with no connection to a gateway will search for a neighbor that has the smallest ETX. It then calculates its own ETX as the expected number of transmission required to reach the neighbor, plus the neighbors ETX.

Collection routing was one of the first routing problems tackled by WSN researchers. The first major mote centric collection algorithm was MintRoute [56]. MintRoute uses the WMEWMA link quality estimator, discussed in section 2.2.2.5, and a minimum transmission routing metric similar to ETX. Link quality estimates are calculated using received signal strength indicator (RSSI) readings from the mote hardware. The MultiHopLQI protocol adapted the MintRoute protocol to use the link quality indicator on the CC2420 radio to improve link estimations. However, the protocol struggles to deliver packets reliably. MintRoute was tested (in conjunction with other protocols) in a large scale environmental sensor networks in the Great Duck Island experiment [46], and in a Redwood tree in Berkeley [52]. In both of these experiments, more than half of the motes failed to deliver any data, and many other motes had poor reception rates.

The Collection Tree Protocol (CTP) is built off of the research of the MintRoute algorithm, and as of 2009 has demonstrated high reliability in experiments and simulations. In [14] the CTP achieved 99% reliability in 4 test beds. CTP improved on MintRoute in three significant ways. First, the hardware link estimators used by MintRoute and MultihopLQI suffered from “sampling bias”: they measure the quality of the received packets, but they do not take into account packets that were lost altogether. The bias is significant because low power wireless links tend oscillate between a strong in weak signal. CTP improved on this by adding sequence numbers to routing beacons, allowing motes to take lost packets into consideration when estimated link quality. Second, the trickle algorithm (discussed in section 2.2.3.1) was applied to the timing of routing beacons to minimize the control packet overhead. Third CTP snoops on data traffic to detect routing loops and inefficiencies in network topology.

CTP was designed, and has been tested, to work on top of various link layer protocols. Using BMAC (described in section 2.2.2.4) for low power listening, CTP is theorized to achieve long battery life times. In [14] Gnawali et al, calculated that a mote using AA batteries that measures a sample and sends a collection packet once every 6 minutes could last for 400 days.

Another low power mote centric collection protocol that has achieved success is the Dozer protocol [4]. Dozer achieved high reliability on a 40 mote network using a 0.2% duty cycle. Unlike CTP, Dozer is not portable to multiple link layer protocols, instead it contains a customized stack developed to meet the goal of the ultra low duty cycle. Dozer uses a TDMA scheme coordinate communications. Dozer does not have a network wide time synchronization. Instead, time synchronization happens locally between child nodes and parent nodes on the routing tree.

Mote centric collection protocols do not provide end-to-end acknowledgments, and focus on making a best effort to deliver packets to the sink. Reliability is improved by establishing reliable links and paths, collision avoidance, and hop by hop acknowledgments.

2.2.3.3 Gateway Centric Collection

In gateway centric collection protocols, all routing logic is handled at the gateway or base station. Motes collect data at a regular interval, but do not transmit the data until instructed to do so by the gateway (or base station). When the gateway is ready to receive new data from a specific mote, it generates one or more reliable paths to that mote. It then uses source routing, embedding the path to the destination into the packet, such that it can be relayed by intermediate motes. The mote then sends all archived data down the reliable path. The one mote at a time methodology make it easier to establish a reliable link to motes, but as a consequence data is retrieved in a less timely fashion than in mote centric routing.

In [50] Stathopoulos et al. introduced the gateway centric collection protocol CentRoute. In their tests, CentRoute was able to achieve 95% delivery of all data sent by motes, and used 60% less overhead than MintRoute. In [33] Musaloiu-E et al. introduce the Koala protocol. Koala improves on the ideas of CentRoute in may ways, including: adding optional end-to-end acknowledgments for data delivered down the reliable path; implementing channel switching on 802.14.5 based radios; and implementing a new link layer technique called Low Power Probing (LPP). LPP is a protocol to proactive wake up an entire network. Using this technique Koala was able to achieve high reliability and very low duty cycles.

Chapter 3

Implementation Context

This chapter describes the various components used in the implementation of the DACK protocol.

3.1 Motes

Development and testing was done with TelosB [40] motes. TelosB motes have 10 Kbytes of RAM and 1 Mbyte of storage. The TelosB motes use the CC2420 [7] radio, which is compatible with th IEEE 802.15.4 specifiation.

3.2 TinyOS and nesC

TinyOS is a free and open source Operating System for tiny embedded devices published under the BSD license. TinyOS was designed to have to low energy footprint. Threads and dynamic memory allocation were not included in the core design, because they could quickly overwhelm the limited stack space in a motes RAM. TinyOS uses component oriented architecture, in which programs are sets of Components that interact with each other through Interfaces. Components use a static amount of stack space, allowing the programmer to precisely control the RAM usage of their program. Concurrency is handled through a $Task$ scheduler. A $Task$ is run to completion and then the stack is freed, before the Scheduler posts the next $Task$. It's up to the application developer to place code sensitive to concurrency errors inside a $Task$, and then post the $Task$ to the scheduler.

Components allow the hardware software boundary in TinyOS to vary from platform to platform. An encryption component may be implemented as software on one platform, and as hardware on another platform. This fine grained control over hardware gives TinyOS fine grained control over the energy consumption on the motes. TinyOS knows to only supply power to the hardware that requires it to process a certain function, and then turn it off after the process completes.

Interacting with hardware components in TinyOS usually requires split phase operations. Software components can have split phase operations as well. Split phase operation are operations that have an initialization method to start the operation, and a completion method that is triggered when the operation has finished. For example, to write a message to storage in TinyOS, you call a $write()$ operation. After the $write()$ operation completes, it will trigger a $writeDone()$ event. Split phase operations can split up the flow of logic in a TinyOS program, and requires attention to synchronization issues. For example, only one message can be written to storage at a time. To avoid collisions writing to storage, you can set a $storage-busy$ flag to $true$ before calling the $write$ operation, and set $storage-busy$ to $false$ in the $writeDone$ event.

TinyOS 2.x was used for the implementation described in this thesis. TinyOS 2.x is a mature operting system for sensor network research, and comes bundled with a number of libraries, tools, and demo applications. There is also an active research community that contributes to the TinyOS source code repository on Sourceforge, as well as in discussion forums, and working groups.

3.3 TOSSIM-Live

TOSSIM [24] is a TinyOS SIMulator that simulates entire TinyOS applications. TOSSIM for TinyOS 2.x currently simulates the hardware of the MICAz mote hardware platform, such that code written for the MICAz can be recompiled to run on the simulator instead. At the time this research was conducted, TOSSIM for TinyOS 2.x had a short coming in that both serial communications and non-volatile storage emulation were not yet implemented.

A recent masters thesis [31] has contributed a fork of TOSSIM called TOSSIM-Live, that includes serial communications, as well as a throttling mechanism to slow down simulation times. TOSSIM-Live was used for the simulations of the DACK protocol discussed in chapter 6.

3.4 Libraries

As discussed in the introduction, the DACK protocol is built on top of a collection protocol and a dissemination protocol. This section describes the TinyOS implementations of these protocols that were used for the simulation and experiment described in chapters 5 and 6.

3.4.1 Collection Tree Protocol (CTP)

An implementation of the Collection Tree Protocol (CTP) [14] described in section 2.2.3.2 is available as a core network library for TinyOS 2.

3.4.2 DIssemination Protocol (DIP)

DIP [30], is also available as a core network library in TinyOS 2. DIP is a dissemination protocol (described in section 2.2.3.1) designed for disseminating numerous small values, such as control parameters. For example, it could be used to change sampling intervals or switch digital I/O channels. In previous dissemination protocols, overhead scaled linearly with the number of data items being disseminated. In DIP, overhead as scales $O(\log(\log(T)))$, where T is he number of data items.

3.4.3 Drip

The Drip [51] is an alternative dissemination protocol, that can be found in the TinyOS 2.x libraries. Though we were able to use DIP in our simulations, we were unable to get it to work in our experiment. For this reason the Drip protocol was used for both the experiment and the simulation.

3.5 Sensor Web Language (SWL)

The Sensor Web Language (SWL) is a programing language with a set of compilers and code templates that can be used to simplify the development of deployable WSNs [34]. The DACK protocol is designed to be compatible with SWL, such that it can be used on WSNs built with SWL.

Chapter 4

Disseminated end-to-end ACKnowledgment (DACK) Protocol

This chapter describes the Disseminated end-to-end ACKnowledgment (DACK) protocol, a method for improving the reliability of data collection in Wireless Sensor Networks. This method is designed to sit on top of a best effort data collection protocol as described in section section 3.4.1, and a reliable data dissemination protocol as described in section 3.4.2. If the collection protocol can deliver most packets from the sensor network to the base station, and the dissemination protocol can reliably disseminate packets from the base station to every mote in the sensor network, then the DACK protocol can be used to deliver end-to-end acknowledgments, for the purpose of retrieving data lost by the collection protocol. Figure 4.1 shows how the DACK protocol is built on top of a collection protocol as described in Figure 2.5 and a dissemination protocol as described in Figure 2.4.

Conceptually, each mote in the network tags all data samples with a contiguous sequence number before sending them to the base station. As these samples are collected at the base station, the base station evaluates the sequence numbers to determine if there are any dropped data samples. The base station then periodically creates a list of all the unreceived sequence numbers from every mote in the network. This list is then compressed and

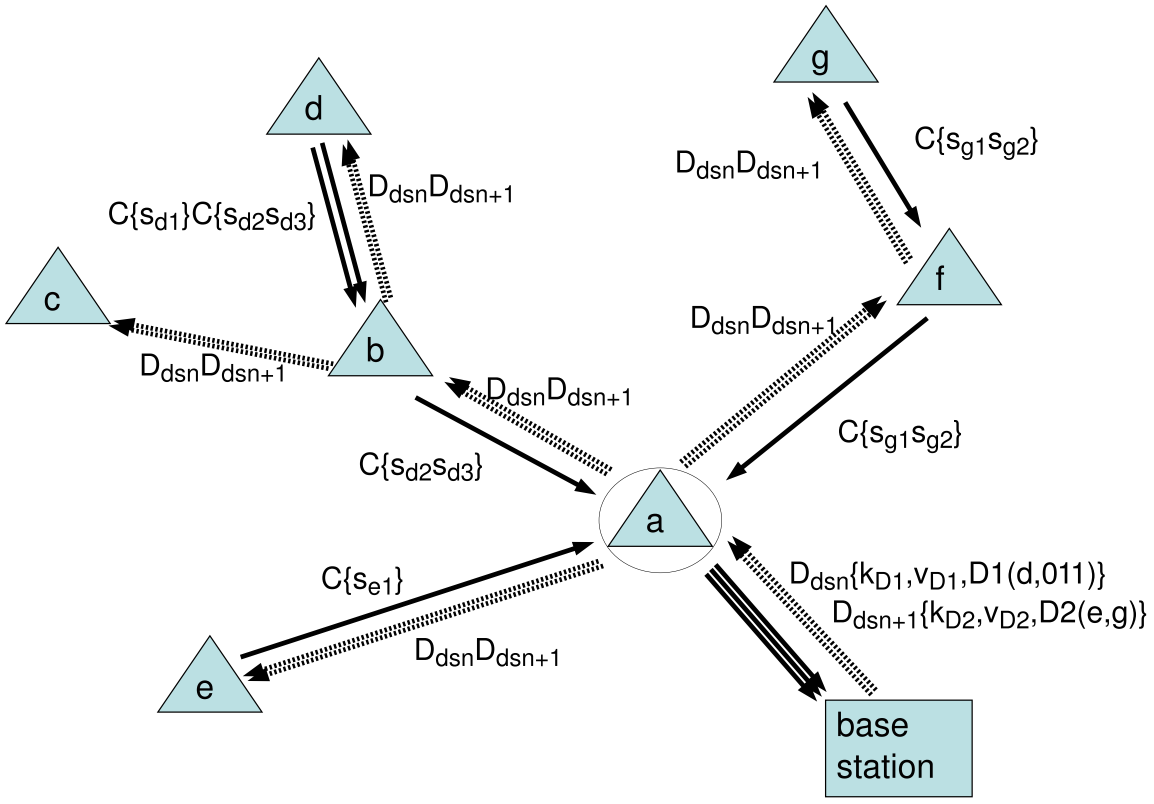


Figure 4.1: A network overview of the DACK protocol. In the above example, solid lines represent the path of a collection packet from the mote to the base station. Dashed lines represent dissemination packets being disseminated from the base station to each mote in the network. Mote a is a $gateway$ mote connected to a $base\ station$. In this snapshot, collection packet containing sample s_{d1} from mote d is lost on the path to the gateway. At the beginning of the next dissemination interval (I_D), the base station disseminates a negative acknowledgment for recent dropped samples in the network in a $D1$ packet, as well as a positive acknowledgment for the samples that have been received from the network in a $D1$ and $D2$ type packet. Once the dissemination packets are received by mote d, mote d notes that it needs to resend the data sample s_{d1} at the beginning of the next report interval I_R.

disseminated out to the entire network, letting each mote know which data was received and which was not. Dissemination is reliable, but expensive, so compression is vital to the efficiency of the method. Figure 4.2 illustrates the various time epochs considered for the dissemination process. The sample intervals I_S, report interval I_R, and dissemination interval I_D, trigger the sample event E_S, report event E_R, and dissemination event E_D respectively, as shown in Figure 4.3. In the present implementation, the network operator

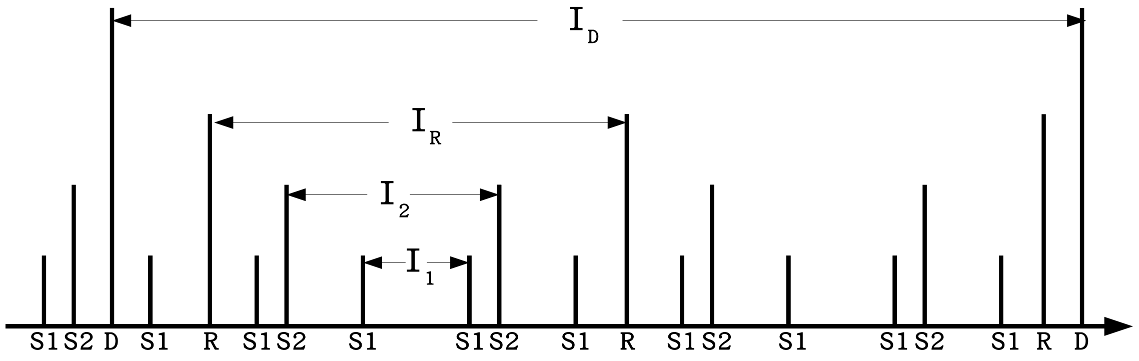


Figure 4.2: An example of the timing intervals required on a mote using the DACK protocol. I_1 and I_2 indicate two sampling time intervals tied to two separate sensors on a single mote. I_R indicates the mote's report interval, and I_D the interval at which disseminated data should be received on the mote from the base station. Sampling intervals can be changed at any time to achieve the data frequency the network operators require.

must tune I_S, I_R, and I_D to match the scale of there network. Potential methods to automatically adapt I_D to minimize the number of disseminations required to provide a network with acknowledgments is discussed in Section 8.2.1.

4.1 Overview

The DACK protocol negotiates end-to-end acknowledgements between a base station BS, and a wireless network N of n motes in a *continuous near-realtime data collection network*. In a *data collection network*, n motes collect samples at preprogrammed intervals, and then periodically relay this information through the network to a common base station. A network is *continuous* if it prioritizes collecting and relaying new data, over preserving old data. A *noncontinusous* network will stop collecting new data from a mote if old data readings have not been acknowledged. *Noncontinuous* operation is more common when a network is designed to log a transient event, rather than continuously monitor an environment over a long period of time. A network is *near-realtime* when motes report readings frequently relative to sampling time. For example, if a mote collects one sample per minute, and relays the sample in a report every five minutes it is *near-realtime*. If the same mote

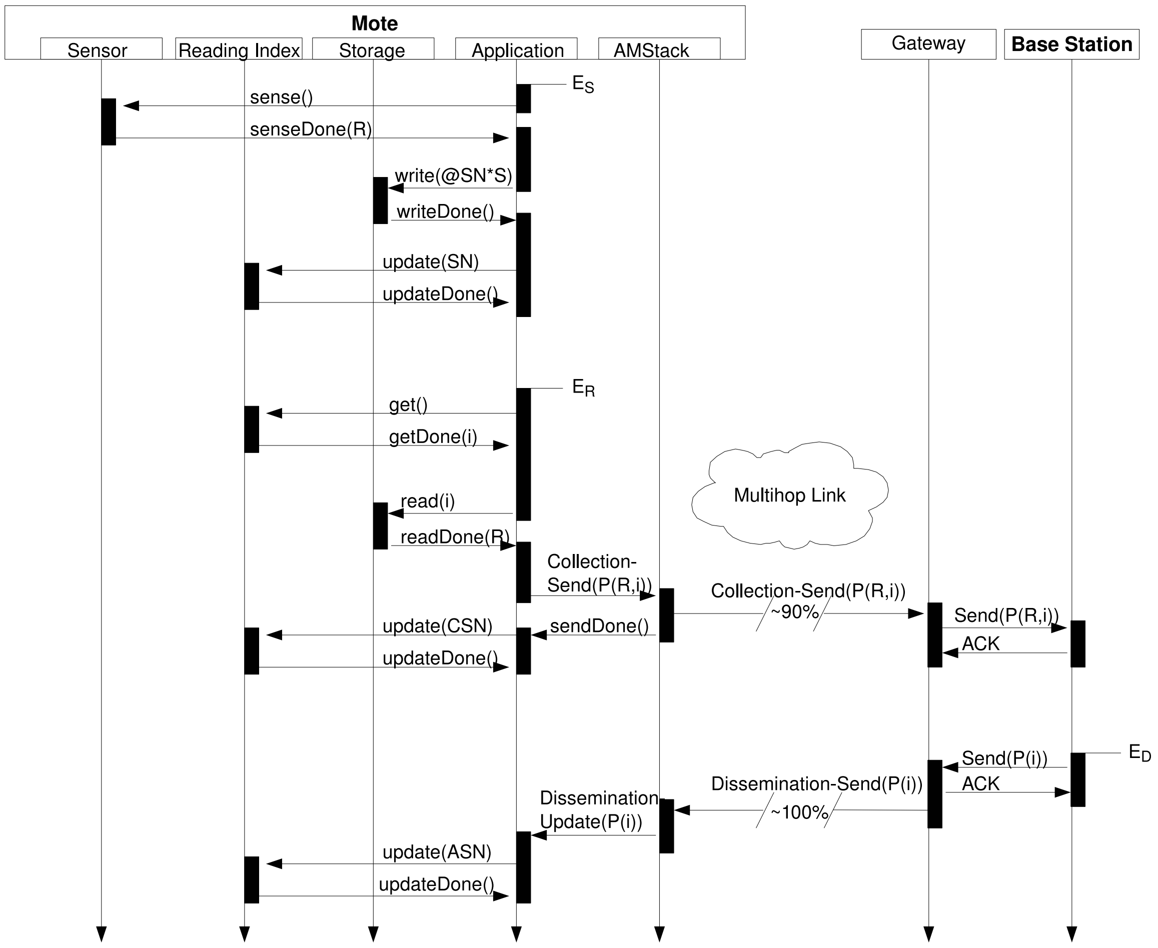


Figure 4.3: Sequence diagram of a mote and a base station using the DACK protocol to acknowledge a data sample. AMStack indicates the TinyOS active message radio stack [35]. The events E_S, E_R, and E_D are shown. Sending and writing to storage in TinyOS is a split phase operation (see section 3.2). Percentages given in the send commands over the multihop link indicate an example probability of successful end-to-end delivery of the message.

instead reported its readings once every 24 hours, then the network readings would be much farther from realtime.

Collection protocols have recently achieved very high reliability using best effort mechanisms [14]. The Collection Tree Protocol (CTP) discussed in 3.4.1 provides a very reliable protocol for building *continuous near-realtime data collection networks*. CTP is able to achieve greater than 99% reliable delivery of packets in many network configurations.

The DACK protocol uses dissemination to add an end-to-end acknowledgment mechanism to help recover the data samples lost by CTP. The DACK protocol, as described below, acknowledges samples, and not messages. This focuses DACK on acknowledging a continuous stream of data samples, rather than insuring the delivery of a specific message. The DACK protocol does not guarantee reliability, but makes a best effort to recover dropped data samples before they get too old. The DACK protocol has a maximum window size W that limits how old a dropped data sample can be before the protocol gives up trying to recover the sample.

4.1.1 Collection Process

Each mote M_{MID} in network N has a unique ID number MID, and v sensors. Each sensor V_{SID} on mote M_{MID} is identified by a unique sensor ID SID. Mote M_{MID} will trigger a sampling event E_{SID} for the sensor V_{SID} at the end of interval I_{SID} for each connected sensor. Each mote M_{MID} also triggers a report event E_R at the end of every report interval I_R. In the report event E_R, mote M_{MID} will send all data samples collected in the preceding report interval I_R. If the mote is in $aggressive$ mode, the mote will also send all unacknowledged data samples. If the mote is in $passive$ mode, the mote will also send all negatively acknowledged data samples. The definition for $passive$ mode and $aggressive$ mode can be summarized as:

- $aggressive$ mode: always resend data samples until a positive acknowledgment of successful receipt by the base station is received by the mote.
- $passive$ mode: resend data samples only when a negative acknowledgment for the samples from the base station is received by the mote.

In each sample event E_{SID}, mote M_{MID} collects one new sample reading $S(R, SID, SN, TS)$ from V_{SID}. R refers stores the data reading on the sensor, TS is a timestamp of the time that the sample was taken, and SN is a sequence number that is incremented in each sample event. Each sampling event must run atomically. This causes some sampling events to be delayed while waiting for other sampling events to complete, but insures that each sample S is given a contiguous SN.

Each mote M_{MID} allocates enough storage space $Storage$ to store F samples. Samples $S(R, SID, SN, TS)$ are stored in $Storage[SN]$ relative to their sequence number SN. To keep track of which samples have been sent, and which samples have been acknowledged, the mote keeps an index MI_{MID} containing the values: $(F, SN, ASN, LSN, DSN, B, L, W, SO)$, which are defined as follows:

- F: the total number of samples that can be stored on the mote.
- SN: the sequence number SN of the newest data sample in storage.
- ASN: the SN of last consecutively acknowledged data sample. A data sample is consecutively acknowledged if all data samples with an older timestamp have been either acknowledged or lost.
- LSN: the SN of the last data sample that mote M_{MID} sent in a report event.
- DSN: the sequence number of the last acknowledgment message received. DSN also doubles as a status flag that the mote can write to indicate a window overflow or a storage overflow.
- B: an L-bit vector referred to as the acknowledgment vector. Bits in B correspond to the data samples with a sequence number SN between ASN and LSN. For each bit b at position i in B, a value of 0 indicates $SN \leftarrow (ASN + i) \textbf{ mod } F$ requires a negative acknowledgment, and a 1 indicates $SN \leftarrow (ASN + i) \textbf{ mod } F$ requires a positive acknowledgment. B is generated by the base station, and is received in acknowledgment messages.
- L: the number of bits in B.
- W: the maximum size of the acknowledgment vector B, called the acknowledgment window. When B grows larger than W, it causes an acknowledgment window overflow error.
- SO: the number of storage overflows that have occurred on the mote. A storage overflow occurs whenever a mote has to overwrite a data sample before it is acknowledged.

Algorithm 1 shows the values that the parameters in the mote index MI are initialized to when the mote boots up. The values for F and W are not shown. These can vary at the discretion of the network operator, and the storage capacity of the motes being used.

```
Trigger: $E_B$ invoked when the mote boots up
Data: Global MoteIndex MI
Result: Initialize the mote's index.
$MI.DSN \leftarrow 250$;
$MI.SN \leftarrow 0$;
$MI.ASN \leftarrow -1$;
$MI.LSN \leftarrow -1$;
$MI.L \leftarrow 0$;
$MI.SO \leftarrow 0$;
```

Algorithm 1: Mote Boot Event E_B

Algorithm 2 shows a procedure for handling a sample event E_S. Each incoming data sample is tagged with a contiguous SN. The sequence number SN is calculated as $SN \leftarrow (SN + 1)\ mod\ F$, such that, if necessary, the newest sample always overwrites the oldest sample in storage. The sampling event also catches storage overflow errors when the SN overwrites the oldest data sample ASN that is still waiting to be acknowledged. If a storage overflow occurs, the mote resets the values in the mote index.

```
Trigger: $E_S$ invoked by a clock interrupt every $I_S$
Data: DataReading R, SensorID SID
Result: Insert new sample into storage.
$MI.SN \leftarrow (MI.SN + 1)$ mod $MI.F$;
if ($MI.SN$ = $MI.ASN$) or (($MI.ASN$ = $-1$) and ($MI.SN$ = $0$)) then
    $MI.DSN \leftarrow$ STORAGEOVERFLOW;
    $MI.SO \leftarrow MI.SO + 1$ $MI.L \leftarrow 0$;
    $MI.SN \leftarrow 0$;
    $MI.ASN \leftarrow -1$;
    $MI.LSN \leftarrow -1$;
end
$S \leftarrow$ new $DataSample$;
$S.SN \leftarrow MI.SN$; $S.SID \leftarrow SID$; $S.R \leftarrow R$; $S.T \leftarrow NOW()$;
$Storage[MI.SN] \leftarrow S$;
```

Algorithm 2: Mote Sampling Event E_S

Algorithm 3 shows a procedure for handling a report event E_R. The mote loops through the data samples in storage, from ASN to SN, and packs all new data samples and negatively acknowledged data samples into collection packets $C(MID$, DSN, SO,

ASN, LSN, $S[s]$), and sends the packets to the base station via the collection protocol. MID identifies the sender of the message; DSN reports the ID of the last ACK received; SO reports the number of storage overflows the mote experienced; LSN and ASN report the mote's current sampling and acknowledgment status; and $S[s]$ is an array of s samples. If the mote is in $passive$ mode, then negatively acknowledged samples and new samples are appended to C for delivery. If the mote is in $aggressive$ mode, then all unacknowledged samples are also appended to collection packets for delivery.

```
Trigger: $E_R$ invoked by a clock interrupt every $I_R$
Data: Global MoteIndex MI, Global Storage S
Result: Send all new and unacknowledged messages to the base station.
$C \leftarrow$ new $CollectionPacket$;
$b \leftarrow 0$;
$MI.LSN \leftarrow MI.SN$;
foreach $S$ in $Storage$ between $Storage[MI.ASN]$ and $Storage[MI.SN]$ do
    if ($MODE = PASSIVE$) then
        if (($S.T > Storage[MI.LSN].T$) or (($b < L$) and (bit $b$ of $B$ = 0))) then
            append $S$ to $C$;    // append negative acked samples and new samples
        end
    end
    if ($MODE = AGGRESSIVE$) then
        if (($b > L$) or (($b < L$) and (bit $b$ of $B$ = 0))) then
            append $S$ to $C$;    // append all unacked samples
        end
    end
    $b \leftarrow b + 1$;
    if ((C is full) or ($S.SN$ = $MI.SN$)) then
        $C.MID \leftarrow MI.MID$; $C.DSN \leftarrow MI.DSN$;
        $C.ASN \leftarrow MI.ASN$; $C.LSN \leftarrow MI.LSN$;
        $C.SO \leftarrow MI.SO$; collectionsend(C);
        $C \leftarrow$ new $CollectionPacket$;
    end
end
```

Algorithm 3: Mote Reporting Event E_R

To process incoming collection messages, the base station creates a local mote index object MI for each mote in the network. Each MI object is stored in the mote table MT, in which motes are referenced by MID. On the base station each MI object contains the following values:

- MID: The mote ID of the mote this MoteIndex object is keeping track of.
- ASN: The last consecutive acknowledged data sample SN received from mote M_{MID}.
- LSN: The last known data sample SN that was sent in a report interval from mote M_{MID}, but not necessarily received.
- RSN: The last data sample SN actually received from mote M_{MID}.
- $RSNT$: The timestamp of the last data sample with sequence number RSN.
- DSN: The sequence number of the last dissemination packet sent containing an acknowledgment for M_{MID}.
- $ASNmismatch$: Set to $TRUE$ when the $MoteIndex$ object and incoming collection packets from the corresponding mote disagree on the value of ASN.
- B: the most recent L-bit acknowledgment vector sent to mote MID.
- L: the size of the vector B.
- F: The max number of samples that can be stored on mote M_{MID}.
- W: is the maximum size of the acknowledgment vector B.
- SO: the number of storage overflows that have occurred on mote MID.
- $WindowOverflow$: Set to $TRUE$ for mote MID when the DSN value inside an incoming collection packet from mote M_{MID} is set to 255. This indicates that the mote has detected an Acknowledgment Window Overflow Error, and is alerting the base station.
- $ackwaiting$: An integer value indicating how many more dissemination intervals I_D the base station will wait before it sends a new ack message to a mote. If a mote includes the DSN of the previous dissemination message in a collection packet before $ackwaiting = 0$ then the base station will set $ackwaiting \leftarrow 0$ and send an ack message to mote i at the next E_D.
- $Storage[]$: This is a mirror of the Storage on the mote M_{MID} that is used for reference. It stores F data samples of the form $S(R, SID, SN, TS, Dropped, Count)$. R, SID, SN, and T reflect the most recently received values from the mote M_{MID}. $Dropped$ is a flag that is used to indicate whether the base station believes the data sample SN was sent by mote M_{MID} but not received. $Count$ is the number of samples that have been received with the same sequence number SN, but different timestamps.
- $dropped$: The number of data samples that have been dropped and required negative

acknowledgments.

- $lost$: The number of data samples that have been permanently lost.
- $received$: number of data samples received.
- $recovered$: number of data samples that were marked as dropped, but eventually recovered.

When collection messages are received on the base station, the base station triggers the collection message received event E_C, and the message is added to a queue called $messagequeue$. The $messagequeue$ is processed at the beginning of the dissemination event E_D. Algorithm 5 shows where the message queue is processed in the dissemination event. Algorithm 4 shows the procedure for processing a single message in the queue. For each message, the base station checks the MID to find out which MI object in the MoteTable to update. If the message contains a new MID not found in the MT, then a new MI object is created for it. Otherwise, the appropriate MI object is selected. The base station then checks to see if the mote is reporting a storage overflow error, by checking if the storage overflow value in the collection packet $C.SO$ against the storage overflow value in the mote index $MI.SO$. The current reaction to a storage overflow error is to treat MID as a new mote and resume sample collection from LSN. After this, the base station loops through every sample contained in the packet. For each sample, the base station checks if the sample is new by comparing the timestamp $S.T$ of the data sample S, with the timestamp of the previous sample stored at $Storage[S.SN]$. The $Storage[S.SN].Dropped$ flag is checked to see if the database was waiting for an acknowledgment of this sample. If this sample contains the most recent timestamp that the base station has ever seen for mote MID, then RSN and $RSNT$ are all updated to the values of the sample, and LSN is updated to the value contained in the collection packet.

```
Trigger: Invoked in the queue processing section of the dissemination event $E_D$
Data: DACKCollectionPacket C, MoteTable MT
Result: Update MoteTable
if ($MT.containsKey(C.MID)$) then
    $MI \leftarrow MT.get(MID)$;
else
    $MI \leftarrow$ new $MoteIndex$;
end
if ($C.SO > MI.SO$) then
    $MI.resetCounters()$;  $ASNmismatch \leftarrow$ TRUE;  $ASN \leftarrow LSN$;
end
if ($C.DSN = MI.DSN$) then
    $MI.ackwaiting \leftarrow 0$;
    if $\neg(MI.ASNmismatch)$ then  $MI.ASN \leftarrow C.ASN$ ;
    if ($MI.DSNtype = D3$) then
        $MI.ASNmismatch \leftarrow$ FALSE;
    end
end
foreach  $S$ in $C$ do
    if $\neg(MI.Storage[S.SN].T < S.T)$ then
        $MI.Storage[S.SN] \leftarrow S$;  $MI.received$++;
        $MI.Storage[S.SN].Count$++;
        if ($MI.Storage[S.SN].Dropped$) then
            $MI.recovered$++;
            $MI.Storage[S.SN].Dropped \leftarrow$ FALSE;
        end
    end
    if ($S.T > MI.RSNT$) then
        $MI.RSNT \leftarrow C.T$;  $MI.RSN \leftarrow C.SN$;  $MI.LSN \leftarrow C.LSN$;
    end
end
$MI.DSNmote \leftarrow C.DSN$;
$MT.put(MI.MID, MI)$;
```

Algorithm 4: Base station processMessage() function for processing a collection packet

4.1.2 Dissemination Process

As discussed in section 2.2.3.1, there are many possible ways to disseminate acknowledgments in a wireless sensor network. The method described here is tailored to make use of either the DIP or Drip dissemination protocols discussed in Sections 3.4.2 and 3.4.3, which were designed to disseminate single packets.

The DACK protocol consists of three types of acknowledgment messages, and corresponding packet types for bundling sets of acknowledgment messages. The three DACK message types are defined as follows:

- $D1Ack$: A $D1Ack$ message contains the values (MID, L, B) to acknowledge mote M_{MID} with ackvector B of size L, and indicates that mote M_{MID} needs to resend the samples referenced by the B vector.
- $D2Ack$: A $D2Ack$ message is a blanket acknowledgement indicating that all samples have been received. A $D2Ack$ can either be a single MID, or two MIDs MID_1 and MID_2 indicating a range of contiguous MIDs identified by the smallest (MID_1) and largest (MID_2) MID in the range.
- $D3Ack$: A $D3Ack$ message contains the values (MID, ASN, L, B) to acknowledge mote M_{MID} with ackvector B of size L starting from the data sample after the sequence number of the last consecutively acknowledged data sample ASN.

A $D1$ packet consists of a list of $D1Ack$ messages, a $D2$ packet consists of a list of $D2Ack$ messages, and a $D3$ packet consists of a list of $D3Ack$ messages. The $D1$, $D2$, and $D3$ packets required to acknowledge samples collected from every mote in the network are generated and sent in each dissemination event E_D. The base sation triggers a dissemination event E_D at the end of every dissemination interval I_D.

Algorithm 5 shows a procedure for handling a dissemination event E_D. First, the base station processes all queued colelction messages using the $processMessage()$ function described in Algorithm 4. Next the following five function prepare and send the dissemination packets: $updateMoteStatus()$ is defined in algorithm 6; $generateD1()$ is defined in algorithm 7; $generateD2()$ is defined in algorithm 9; and $generateD3()$ is defined in algorithm 11. $disseminate()$ involves disseminating all the packets generated by the preceding functions in the network. An implementation of the disseminate function can be found in Appendix A.7.

```
Data: MoteTable MT
Result: Prepare and then disseminate acknowledgments into the network
while (messagequeue is not empty) do
    $processMessage(messagequeue.pop());$
end
$updateMoteStatus();$
$generateD1();$
$generateD2();$
$generateD3();$
$disseminate();$
```

Algorithm 5: Dissemination event E_D

Before disseminating acknowledgments into the network, the base station loops through every MI object to check each mote's status, set the appropriate flags, and generate the latest ack vector B. Algorithm 6 shows the mote status update process. The base station first checks to see if the value for ASN that was most recently reported by the mote matches the ASN value in storage. The function $findASN$ is not defined here, but an implementation is in appendix A.8. If the values for ASN do not match, then the base station sets the $MI.ASNmismatch$ flag, indicating that the mote requires a $D3Ack$ type acknowledgment. Next the base station generates the ack vector B for each mote. The ack vector is created by checking each sample in reverse from the sample at LSN (the sequence number of the last known data sample collected on the mote) to the sample after ASN. In this way the oldest sample becomes the least significant bit of the ack vector. If every sample is acknowledged, B and L are both set to 0, indicating that the mote needs a $D2Ack$ type acknowledgment. If $ASNmismatch$ is false, but one or more samples are dropped, a $D1Ack$ is sent. Next the base station checks for windows overflow errors. If there is a window overflow error then B and L are set to 0, ASN is set to LSN and $ASNmismatch$ is set to true. This forces the the mote to stop resending samples older than LSN. This can create some false negatives (samples assumed to be lost, when in fact they are not) if the samples do arrive later.

```
**Trigger:** invoked by $E_D$
**Data**: MoteTable MT
**Result**: Decide which type of ACK is required for each mote
**foreach** $(MI$ **in** $MT)$ **do**
    $MI.ackwaiting \leftarrow MI.ackwaiting - 1$;
    **if** $(MI.ASN \neq MI.findASN())$ **then**
        $MI.ASN \leftarrow MI.findASN()$;
        $MI.ASNmismatch \leftarrow TRUE$;
    **end**
    $MI.B \leftarrow 0$; $MI.L \leftarrow 0$; $allclear \leftarrow TRUE$;
    **foreach** $S$ **in** $MI.Storage$
                **from** $MI.Storage[(ASN + 1)$ **mod** $F]$
                **to** $MI.Storage[LSN]$ **do**
        $MI.B \leftarrow MI.B << 1$; $L$++;
        **if** $(S.Count < MI.Storage[RSN].Count)$ **then**
            $S.Dropped \leftarrow TRUE$; $allclear \leftarrow FALSE$;
            $MI.B \leftarrow MI.B$ **&** $0$;
        **else**
            $MI.B \leftarrow MI.B$ **&** $1$;
        **end**
    **end**
    **if** $(allclear)$ **then** $MI.B \leftarrow 0$; $MI.L \leftarrow 0$;
    **if** $(MI.L > MI.W)$ **then**                         /* window overflow error */
        **for** $b \leftarrow 1$ **to** $L$ **do**
            **if bit** $b$ **in** $B$ **=** $0$ **then**
                $S \leftarrow MI.Storage[(ASN + b + 1)$ **mod** $F]$;
                $S.Dropped \leftarrow FALSE$; $MI.dropped$++;
                $S.Count \leftarrow MI.Storage[RSN].Count$;
                $S.T \leftarrow MI.RSNT$;
            **end**
        **end**
        $MI.B \leftarrow 0$; $MI.L \leftarrow 0$; $ASN \leftarrow LSN$; $RSN \leftarrow LSN$;
        $MI.ackwaiting \leftarrow 0$; $MI.ASNmismatch \leftarrow TRUE$;
    **end**
**end**
```

Algorithm 6: Generate B vectors and set flags for each mote at the base station.

After generating the B vectors and setting flags for each mote, the base station generates the three types of DACK dissemination packets. Algorithm 7 shows the procedure for generating $D1$ type packets. DSN is incremented to give the packet a unique sequence number. DSN is also used by the mote to indicate a storage overflow error by setting DSN to $STORAGEOVERFLOW$. The enumerated value for $STORAGEOVERFLOW$ is greator than $MAXDSN$. To complete the $D1$ packet, the base station loops through every mote in the mote table. If B indicates dropped data samples, there is no $ASNError$, and the base station is not currently waiting for a response to a previous dissemination, then the mote's MID, L, and B are appended to the packet.

```
Trigger: invoked by $E_D$
Data: MoteTable MT
Result: Populate a D1 message
$D1 \leftarrow$ new $D1DACKMsg$;
$DSN \leftarrow (DSN{+}{+})$ mod $MAXDSN$;
$D1.DSN \leftarrow DSN$;
foreach ($MI$ in $MT$) do
    if ($MI.B \neq 0$) and ($\neg MI.ASNmismatch$) and ($MI.ackwaiting \leq 0$)) then
        $D1.append(MI.MID, MI.L, MI.B)$;
        $MI.ackwaiting \leftarrow 2$;
    end
end
```

Algorithm 7: Generating a D1 packet on the base station.

Algorithm 8 shows the process for handling an incoming $D1$ packet on the mote. The mote loops through the $D1$ looking for its MID. If the mote finds its MID, the local values for DSN, L, and B, are updated. Next the mote loops through the bits of B starting from the least significant bit. If the least significant bit of B is a 1, then ASN gets incremented, B loses its least significant bit, and L is decremented to reflect the shrinking size of B. The loop terminates whenever the least significant bit of B becomes 0.

```
Trigger: $D1$ packet is received on mote $MID$
Data: D1Packet D1
Result: Update MI
foreach ($D1Ack$ in $D1$) do
    if ($D1Ack.MID$ = $MI.MID$) then
        $MI.DSN \leftarrow D1Ack.DSN$;
        $MI.L \leftarrow D1Ack.L$;
        $MI.B \leftarrow D1Ack.B$;
        while ($MI.B$ & $1$) do
            $MI.ASN{+}{+}$;
            $MI.B \leftarrow MI.B >> 1$;
        end
    end
end
```

Algorithm 8: Processing an incoming $D1$ packet on a mote.

Algorithm 9 shows the procedure for generating $D2$ type packets on the base station.

The base station increments DSN and creates an ordered array of all the MI objects in the mote table MT, sorting by MID. This allows the base station to create a short ACK message for motes with contiguous MIDs and no dropped data samples. The $lookahead()$ function looks ahead in the ordered array of MIDs, returning the largest MID it can find from its current position that is fully acknowledged. A range is generated as a sequence of bytes in the form '$MID1$ 250 $MID2$'. This creates the side effect that no mote in the sensor network can have the value 250 in the most significant byte of its MID.

```
Data: MoteTable MT
Result: Populate a D2 message
$D2 \leftarrow$ new $D2DACKMsg$;
$DSN \leftarrow (DSN++)$ mod $MAXDSN$;
$D2.DSN \leftarrow DSN$;
$MoteIndex[MT.numElements()]\ MI \leftarrow getSortedArray(MT)$;
$nummotes \leftarrow MI.size()$; $i \leftarrow 0$;
while $(i < nummotes)$ do
    if (($MI[i].B$ = 0) and ($\neg MI[i].ASNmismatch$) and
        ($MI[i].ackwaiting \leq 0$)) then
        $D2.append(MI[i])$;
        $MI[i].ackwaiting \leftarrow 2$;
        $j \leftarrow lookahead()$;
        if ($j - i > 1$) then
            $D2.append(250, MI[j])$;
            for ($k \leftarrow i$ to $j$) do $MI[k].ackwaiting \leftarrow 2$;
            $i \leftarrow j$;
        end
    end
    $i$++;
end
```

Algorithm 9: Generating a D2 packet on the base station

Algorithm 10 shows the process for handling an incoming $D2$ packet on the mote. The algorithm loops through the $D2$ packet looking for an explicit reference to the mote's MID, or a range that includes an implicit reference to the mote's MID. The bit structure of a $D2$ packet is illustrated in Figure 5.7.

```
**Data**: D2DACKPacket D2
**Result**: Update MI
$ACK \leftarrow FALSE$;
$tMID, tMID1, tMID2 \leftarrow 0$;
$mB \leftarrow sizeInBytes(MID)$;
$dB \leftarrow sizeInBytes(D2)$;
$D2b \leftarrow$ **byte array of** $D2$;
$i \leftarrow 0$;
**while** $(i + mB * 2 + 1 < dB)$ **do**
    **if** $(D2b[i + mB] = 250)$ **then**
        $tMID1 \leftarrow D2b[i : i + mB - 1]$;
        $tMID1 \leftarrow D2b[i + mB + 1 : i + 2 * mB]$;
        **if** $(tMID1 \leq MID \leq tMID2)$ **then**
            $ACK \leftarrow TRUE$;
            **break**;
        **end**
        $i \leftarrow i + mB * 2 + 1$;
    **else**
        $tMID \leftarrow D2b[i : i + mB - 1]$;
        **if** $(tMID = MID$ **then**
            $ACK \leftarrow TRUE$;
            **break**;
        **end**
        $i \leftarrow i + mB$;
    **end**
**end**
**if** $(ACK)$ **then**
    $MI.ASN \leftarrow MI.LSN$;
    $MI.L \leftarrow 0$;
    $MI.B \leftarrow 0$;
**end**
```

Algorithm 10: Processing an incoming D2 packet on the mote

Algorithm 11 shows the procedure for generating $D3$ type packets on the base station. Generating a $D3$ type packet is much the same as generating a $D1$ type packet, except in this case the value for $ASNmismatch$ is $TRUE$, and the value for ASN is included after the MID.

```
Data: MoteTable MT
Result: Populate a D3 message
$D3 \leftarrow$ new $D3DACKMsg$;
$DSN \leftarrow (DSN{+}{+})$ mod $MAXDSN$;
$D3.DSN \leftarrow DSN$;
foreach ($MI$ in $MT$) do
    if ($MI.ASNmismatch$) and ($MI.ackwaiting \leq 0$)) then
        $D3.append(MI.MID, MI.ASN, MI.L, MI.B)$;
        $MI.ackwaiting \leftarrow 2$;
    end
end
```

Algorithm 11: Generating a D3 packet on the base station

Algorithm 12 shows the process for handling an incoming $D3$ packet on the mote.

```
Data: D3DACKPacket D3
Result: Update MI
foreach ($D3Ack$ in $D3$) do
    if ($D3Ack.MID$ = $M3.MID$) then
        $MI.DSN \leftarrow D3Ack.DSN$;
        $MI.ASN \leftarrow D3Ack.ASN$;
        $MI.L \leftarrow D3Ack.L$;
        $MI.B \leftarrow D3Ack.B$;
        while ($MI.B$ & $1$) do
            $MI.ASN{+}{+}$;
            $MI.B \leftarrow MI.B >> 1$;
        end
    end
end
```

Algorithm 12: Processing an incoming D3 packet on the mote

4.2 End-to-End Messaging Scenarios and Timelines

This section discusses how the DACK protocol defined in the previous section operates under a variety of communication conditions. Various end-to-end timing scenarios for the DACK protocol are illustrated in each section. In each timing scenario, motes operate in $passive$ mode. Recovering dropped samples is discussed in Section 4.2.2. Sending a blanket acknowledgment is discussed in Section 4.2.3. Section 4.2.4 shows how an $ASNmismatch$ can be caused by message latency. For the DACK protocol to be effective in recovering lost samples, it is necessary for the dissemination protocol to be significantly more reliable than the collection protocol. Poor dissemination reliability can lead to window errors as discussed in Section 4.2.5, and storage overflow as discussed in section 4.2.6.

There are two major messaging delays in the end-to-end operation of the DACK protocol, each of which is described below, and illustrated in figure 4.4:

- **Dissemination Delay** G_D: the time it takes to disseminate a DACK Message from the base station to the target mote.
- **Collection Delay** G_C: the time between when a report event is invoked on the mote, and the time that the base station receives the first of any of the samples in the report. (The first collection packet that is received may not be the first packet that was sent.)

The DACK protocol requires the network operator to fine tune dissemination interval I_D to meet the needs of their network. If disseminations are sent too frequently, this can lead to a significant energy drain on the network. If disseminations are sent too infrequently, then we increase the chance of permanently losing data samples through acknowledgment window overflow errors.

For the DACK protocol to function, I_D must be greater than G_D, and I_R must be greator than G_C. The CTP protocol described in [14], is able to deliver packets from a mote to the base station over multiple hops in milliseconds. The collection process is occasionally subject to transient routing loops, which can cause non-deterministic delays. The main bottleneck for the DACK protocol is the dissemination delay G_D. DIP, the DIssemination

Protocol, described in [30], was able to disseminate 64 data items to 80 motes in 86 seconds. In experiments discussed in chapter 7, the Drip protocol [51] was able to disseminate one to three DACK packets to 14 motes in under 10 seconds. In the simulations, dissemination on a network of the same size could take as long as a few minutes.

Due to limitations of the current implementation, it is necessary to tune the dissemination interval I_D to be greater than or equal to the longest report interval of any mote in the network. In the experiments and simulations discussed in the following chapters, every mote in the network has the same report interval I_R, and the base station has a dissemination interval I_D equal to the report interval. In many cases setting $I_D = I_R$ may be inefficient, but it makes analysis of the operations of the protocol easier to illustrate. Section 8.2.1 dis-

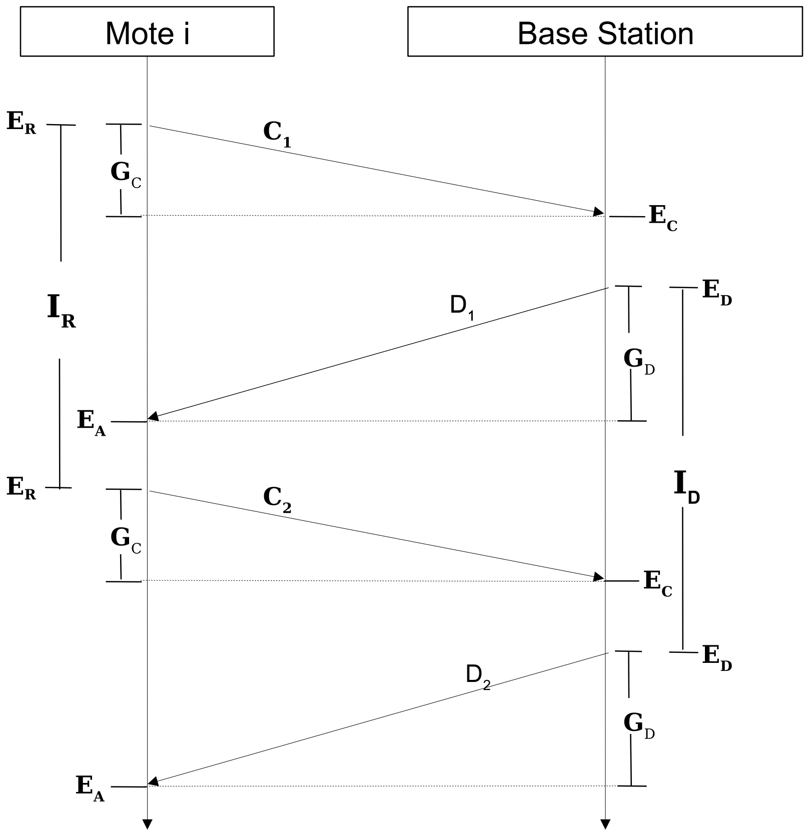


Figure 4.4: Mote i triggers event report E_R at the end of each report interval I_R, and sends data samples in collection packets C to the base station. After a delay of G_C, the message arrives at the base station, and triggers the collection event E_C. Later, the base station triggers event E_D at the end of the dissemination interval I_D, and sends an ack message to mote i. After a delay of G_D, the ack message arrives at the mote and triggers event E_A.

cusses possible methods to have the base station automatically adjust I_D to meet the needs of the network.

4.2.1 Establishing a Connection

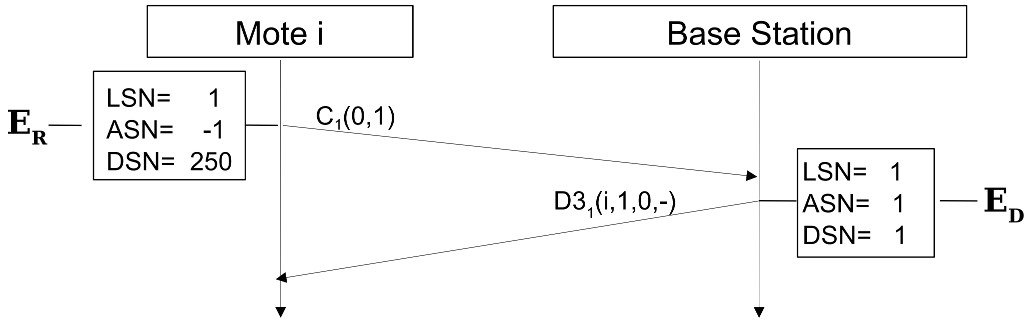


Figure 4.5: The above timeline illustrates how end to end connections are established in the DACK protocol. At the first report event E_R mote i sends collection packet C_1 to the base station containing samples 0 and 1. Because this is the first time the base station heard from mote i, the base station responds at the next dissemination event E_D with a $D3Msg$ with an empty B vector, and the ASN set to 1.

When the base station receives a message from a new mote for the first time, the base station will start acknowleging all data samples starting from the SN of the last data sample received RSN. All data samples before RSN will be ignored. The protocol behaves similarly when receiving a storage overflow message from a mote. (This scenario is described in the scenario in Section 4.2.6.) Figure 4.5 shows a simple scenario, in which mote i only sends one collection message containing two samples every report event, and the first collection packet sent by the mote is received by the base station.

4.2.2 Recovering Dropped Samples

Sometimes collection packets $C(MID, DSN, ASN, LSN, S[s])$ become lost in transmission from a mote to the base station. If at least one collection packet in a report successfully makes it to the base station, then the base station will be able to deduce how many data samples were lost from the LSN value. Figure 4.6 illustrates this sample recovery process. If

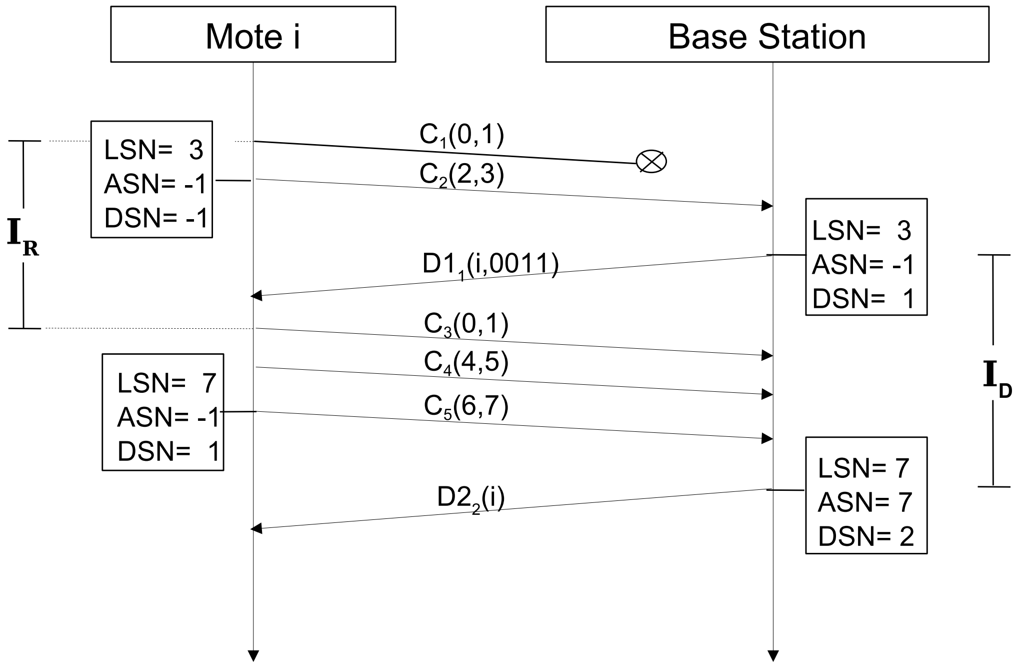


Figure 4.6: The above timeline illustrates when a $D1$ packet is used in the DACK protocol. Mote i sends samples 0 and 1 in collection packet C_1, and samples 2 and 3 in collection packet C_2. Collection packet C_1 gets lost in transmission, and the base station only recieves C_2. Because the collection packet includes the ASN and the LSN for mote i inside each collection packet, the base station is able to determine that samples 0 and 1 are dropped. The base station then disseminates a $D1$ packet containing an ACK for samples 2 and 3, as well as a negative acknowledgment of samples 0 and 1 at the beginning of the next dissemination interval I_D. The next report interval I_R on mote i begins after $D1_1$ is received, so mote i resends samples 0 and 1 in collection packet C_3 before sending new samples.

no packets in a report arrive at the base station, then the base station will have no knowledge of the dropped data samples unless or until a collection packet from a later report arrives. If the range of dropped data sample sequence numbers (SNs) does not exceed the acknowledgement window size W, then the dropped data samples can be recovered using either a $D1Ack(MID, L, B)$ message, or a $D3Ack(MID, ASN, L, B)$ message. Figure 4.6 illustrates a scenario when a $D1Ack$ is required to recover a lost data sample. Sometimes messaging latency causes an $ASNmismatch$, and a $D3Ack$ is required. The $ASNmismatch$ scenario is discussed further in section 4.2.4.

4.2.3 Full Acknowledgements

If every data sample sent by a mote is received by the base station, then the base station replies with a full acknowledgment. Full acknowledgments are made with the $D2Ack$. As described in section 4.1.2, the $D2Ack$ can take the form of a single mote ID "MID" or range of mote IDs "$MID1\ 250\ MID2$". Figure 4.7 illustrates a scenario in which the bas station sends $D2Ack$ message "i" inside a $D2$ packet to acknowledge that all samples have been received from mote i. The range format allows the $DACK$ protocol to run efficiently in sensor networks that have little data loss. A single $D2Ack$ message can acknowledge every mote in a WSN with the message "$0\ 250\ MID_n$", where MID_n is the largest numeric mote ID in the network.

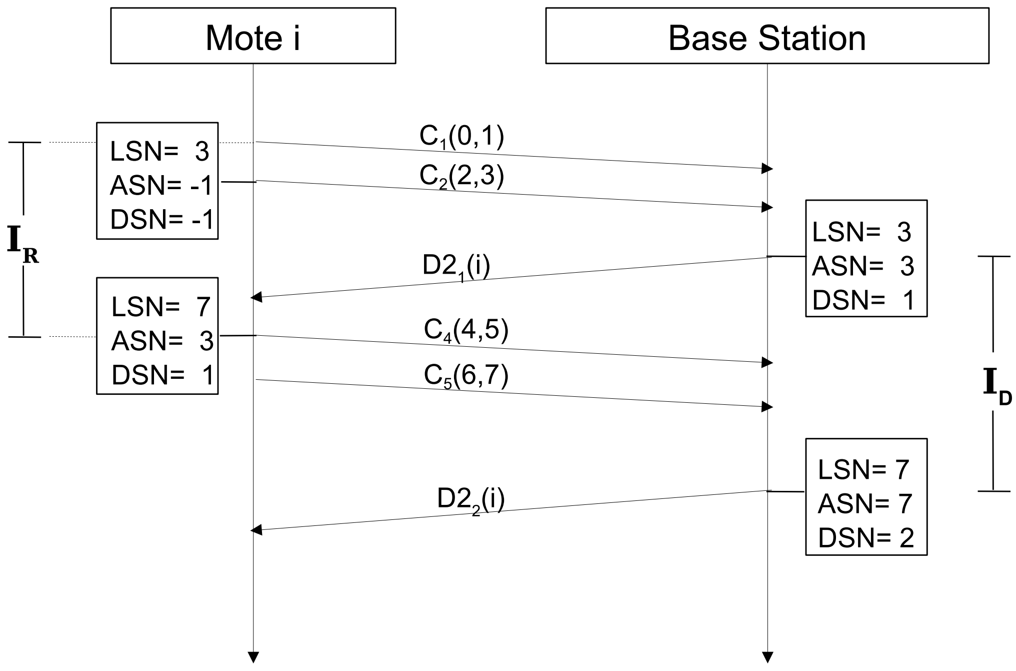


Figure 4.7: The above timeline illustrates when $D2$ packets are used in the DACK protocol. Mote i sends samples 0 and 1 in collection packet C_1, and samples 2 and 3 in collection packet C_2. Both samples are received by the base station. At the next dissemination interval I_D, the base station is able to deduce from the LSN values contained in the collection packet that all packets sent by mote i have been received. The base station then sends a blanket acknowledgement to mote i by including the mote's ID number in the next $D2$ packet.

Sending a $D2Msg$ at all may be considered wasteful in a network with very small data loss. It is possible to modify the protocol to only use $D3Ack$ messages to acknowledge samples when they get lost. This would save us from disseminating any thing at all for successfully received packets. This approach would not be an end-to-end messaging protocol, because motes would have no way of knowing whether the messages were received by the base station.

4.2.4 Overcoming Message Latency

We define $T(E_R)$, $T(E_D)$, $T(E_C)$, and $T(E_A)$ as follows:

$T(E_R)$: the time a report event E_R is invoked on the mote.

$T(E_D)$: the time a dissemination event E_D is invoked on the base station.

$T(E_C)$: the time the first collection packet sent in E_R is received by the base station.

$T(E_A)$: the time that the DACK message sent in E_D arrives at the mote.

Let X be a set of collection packets $C_1, ..., C_x$ sent in a report event E_R where x is the number of collection packets in the set. If any or all collection packets in X become lost enroute to the base station, and if the base station sends a $D2Msg$ to the mote before any of the messages in X arrive at the base station, and if the $D2Msg$ arrives at the mote after the lost collection packets were transmitted by the mote, then the mote will erroneously assume that the $D2Ack$ message is acknowledging the lost data samples, causing an $ASNmismatch$. Figure 4.8 shows the time range relative to a dissemination event E_D when an $ASNMismatch$ can occur.

When the $D2Ack$ message is received by the mote, the mote assumes that the base station is acknowledging all sent data samples, and erroneously updates its ASN value to LSN. The base station does not learn of the $ASNMismatch$ until the revised ASN is received in a subsequent collection packet. In the next dissemination event E_D after the collection packet with the erroneous ASN arrives, the base station notices that the

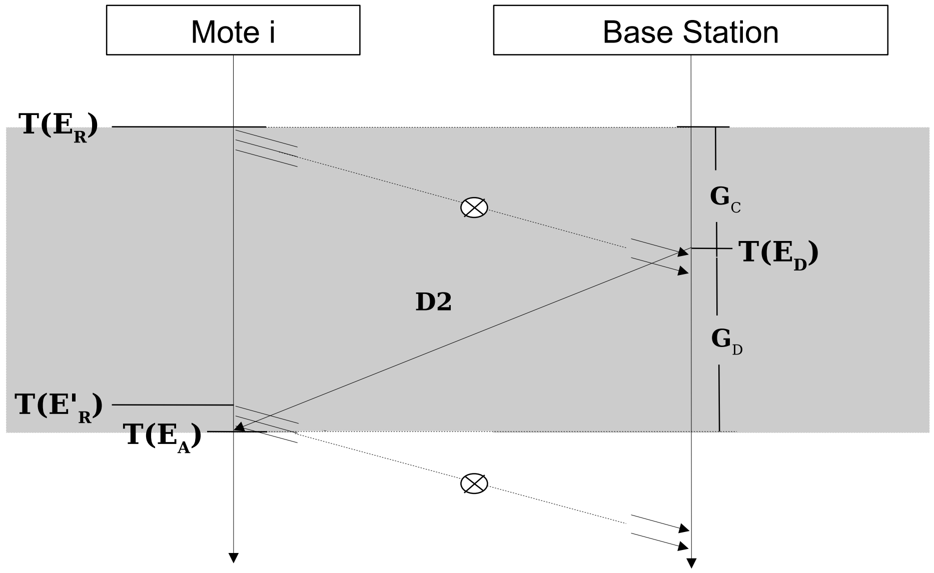


Figure 4.8: In the above timeline the shaded area indicates the time range when an $ASNMismatch$ can occur. For an $ASNMismatch$ to occur, the report event E_R must occur after $T(E_D) - G_C$ where G_C is the time it takes for the first packet in the report to be received by the base station, one or more collection packets sent in E_R must become lost, and the lost collection packets must be sent before $T(E_A) = T(E_D) + G_D$.

ASN value in the collection packet is different than the ASN value in the mote's index object. The base station resolves the $ASNmismatch$ by sending a $D3Ack$ message to the mote. Figure 4.9 illustrates a scenario in which an $ASNmismatch$ occurs, and a $D3Ack$ is required to acknowledge dropped data samples.

To avoid additional $ASNmismatch$ scenarios, the base station should not send a new ack message to a mote until the mote has had enough time to respond to the previous ack message. If $I_D \geq I_R + G_D + G_C$, it will reduce the likelihood of an $ASNMismatch$. G_C and G_D can be highly variable because they depend on network conditions and the routing decisions of the underlying collection and dissemination protocol. If the network operator chooses the minimum $I_D = I_R$, then it is more likely that $I_D \not\geq I_R + G_D + G_C$. To reduce the likelihood of $ASNMismatch$ errors, the base station sends the next ack message at twice the dissemination interval time; i.e. at time $T(E_D) + 2I_D$. For example, let E_{D1} be a dissemination event in which the base station sends a $D2Msg$ in packet $D2_d$ with $DSN = d$ to mote i. Let E_{D2} be the next dissemination event at $T(E_{D2}) = T(E_{D1}) + I_D$. If the base

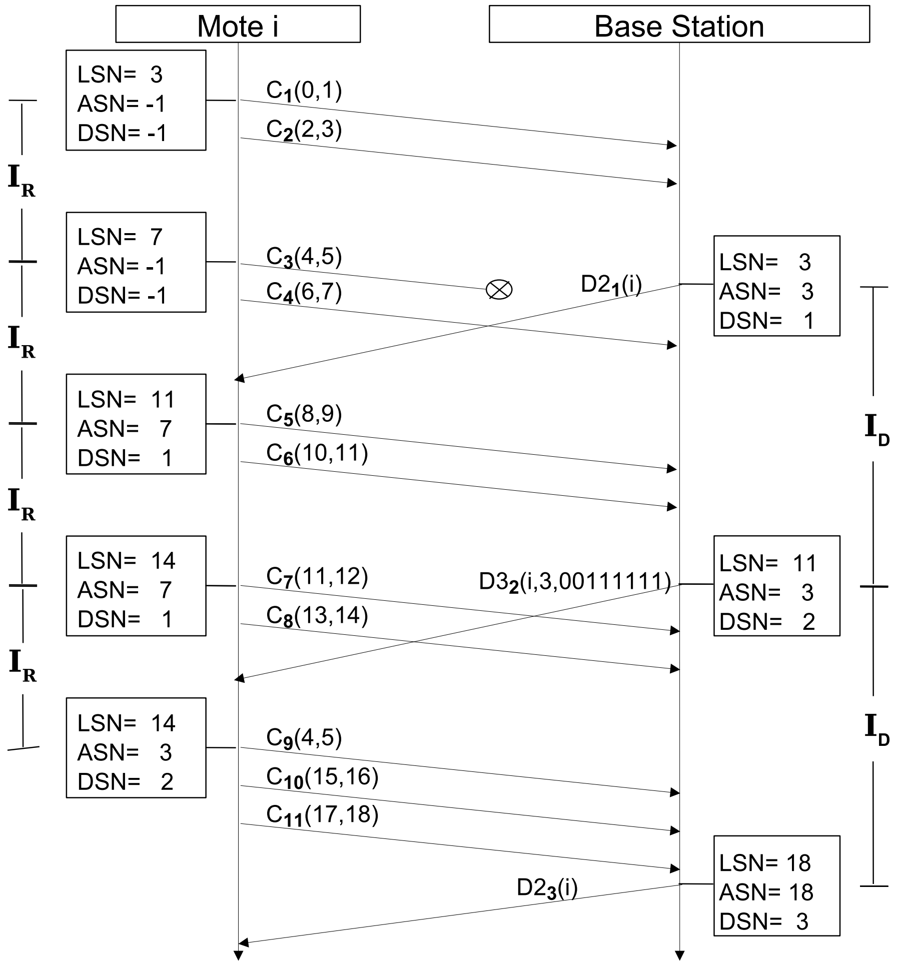


Figure 4.9: The above timeline illustrates when a $D3$ packet is used in the DACK protocol. Samples 0 through 3 are correctly received by the base station from collection packets C_1 and C_2. At the next dissemination interval the base station sends a $D2_1(i)$ to acknowledge that it has received all known samples. Due to routing delays $D2_1(i)$ is not received by mote i until after mote i sends collection packets C_3 and C_4. If both C_3 and C_4 are received by the base station, then everything is fine. If either packet C_3 or C_4 are lost, then this causes mote i to erroneously believe that C_3 and C_4 were acknowledged, and ASN is set to 7 instead of 3 in the next collection packet C_5. When C_5 is received, the base station detects that the ASN value in C_5 does not match the base station ASN value in the mote index object for mote i. At the next dissemination event, the base station sends a $D3$ packet to correct the problem. When mote i receives the $D3$ packet, it corrects its value for ASN, and resends the dropped samples.

station receives a collection packet from mote i containing a $DSN = d$, then receipt of the $D2_d$ packet is acknowledged, and the base station knows it is safe to send a new ack message to mote i in packet $D2_{d+1}$. Otherwise, the base station skips sending an ack message to mote i at E_{D2}, and waits until the next dissemination event at $T(E_{D3}) = T(E_{D2}) + I_D$ to send the next ack message to mote i. At this point the base station sends a new message to mote i regardless of whether or not a collection packet with $DSN = d$ was received.

4.2.5 Accounting for Lost Data on Noisy Channels

If the gap between the sequence number of the last sent data sample LSN, and the sequence number of the last acknowledged data sample ASN becomes greater than W then the DACK protocol will permanently lose all data samples collected before the most recently collected sample. Ignoring old data samples prevents the B vector from growing to the size of F and avoids dissemination packets growing too large or too numerous.

Figure 4.10 shows a scenario in which the DACK protocol is forced to drop samples due to this so-called window overflow. Both the mote and the base station can detect a window overflow. If the mote detects a window overflow, the mote sets its DSN value to the $WO = WINDOWOVERFLOW$ flag. When the base station detects a window overflow, it sets ASN to LSN and sends a $D3Ack$ message to the mote to update the mote's ASN, preventing the mote from resending old samples. The base station counts the number of samples that are lost due to window overflows.

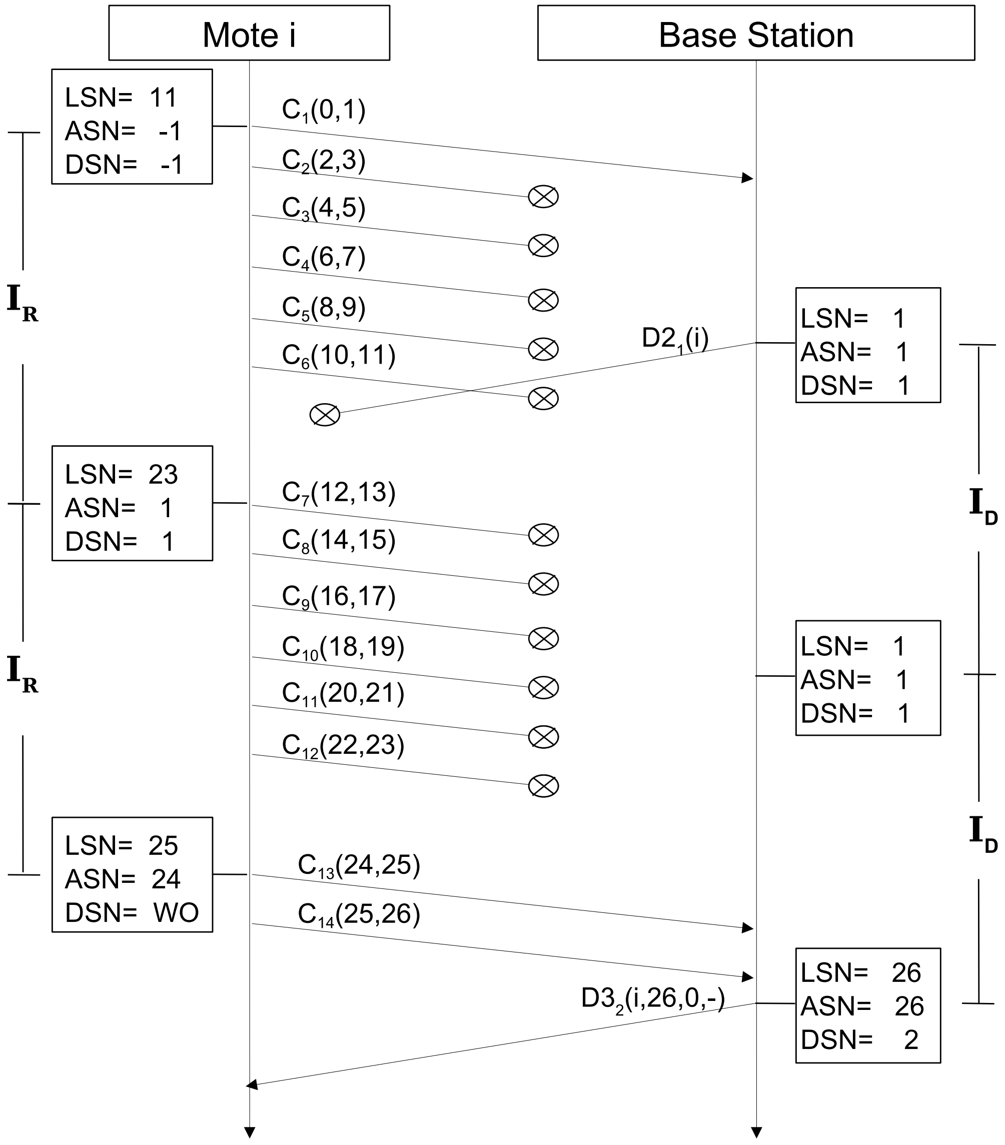


Figure 4.10: The above timeline shows how the DACK protocol handles an acknowledgment window overflow with $W = 20$. In the above case, the combination of a short I_S with respect to I_R, a small value for W, and a noisy link cause several collection packets and a dissemination packet to become lost. Later, when the base station detects a gap between LSN and ASN that is greater than W, it gives up on all the lost samples by updating ASN to LSN, and sends a $D3Ack$ message to inform the mote.

4.2.6 Handling Long Delays

A storage overflow occurs if the break in communication between the mote and the base station continues long enough that the mote is forced to overwrite an unacknowledged data sample in storage. Figure 4.11 shows a timeline scenario in which a storage overflow occurs. Only motes can detect a storage overflow. When a mote does detect a storage overflow, it sets the DSN to the flag $SO = STORAGEOVERFLOW$. This allows the base station to learn about the storage overflow from the DSN value in the collection packets sent by the mote. When the base station learns about the storage overflow, the mote's local ASN is set to the LSN contained in the informing collection packet, and a $D3Ack$ message is sent to the mote to synchronize it with the new ASN. Unlike handling window overflows, when a storage overflow occurs, the base station is unable to detect how many data samples were lost. It would be possible to account for all of these lost samples by counting unacknowledged data samples on the mote, and then having the mote relay this information to the base station. This feature has been left for future work.

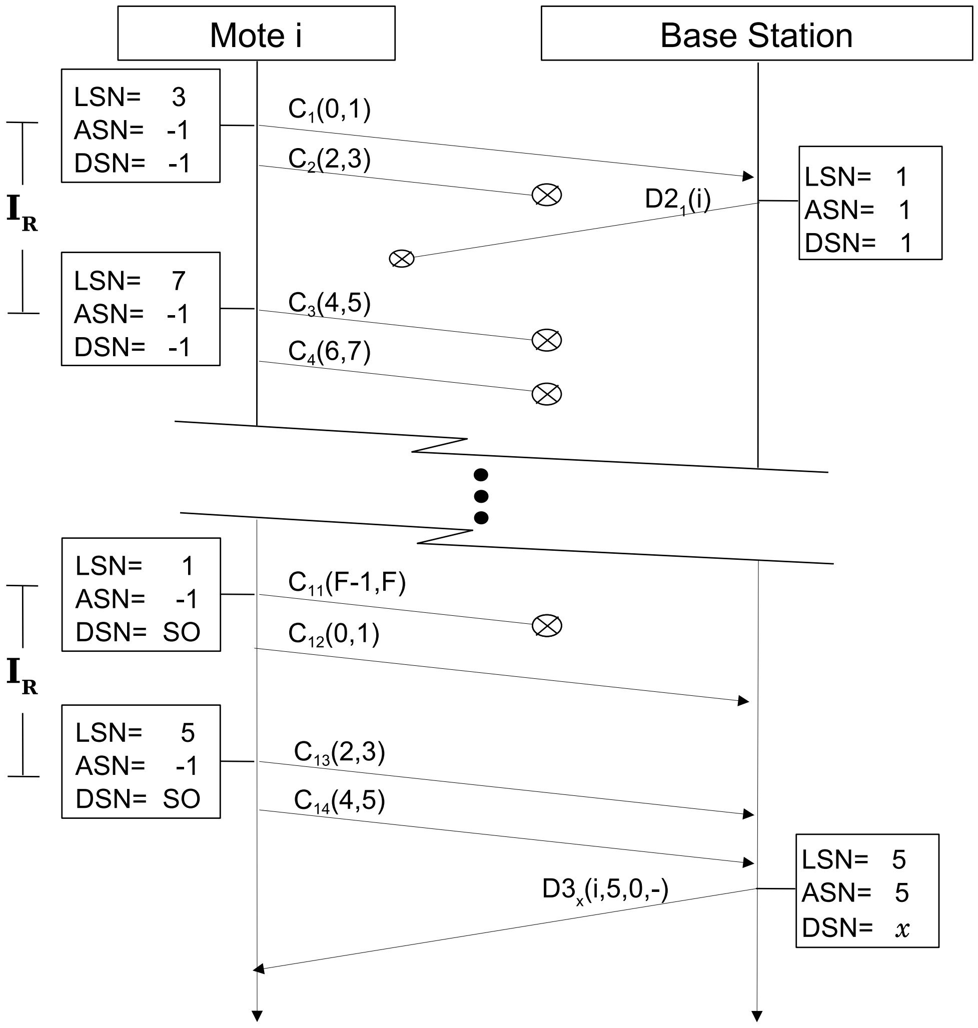


Figure 4.11: The above timeline shows how the DACK protocol handles a storage overflow error. In this case no message is successfully transmitted between mote i and the base station until after the entire sample storage space on mote i has been written to. When mote i detects that it is overwriting the sample stored at ASN, it sets DSN to *STORAGEOVERFLOW*. When the base station sees this DSN in a collection packet, it confirms the update by setting ASN to DSN and then sending a $D3Ack$ message to inform the mote of the new ASN. The DSN value x varies depending on how many DACK packets were disseminated in the interim.

4.2.7 Accounting Errors

Experiments and simulations of the current implementation of the DACK protocol show that in certain cases it can produce false positives and false negatives. False positives are discussed in section Section 4.2.7.1, and false negatives are discussed in Section 4.2.7.1. Potential methods to resolve false positives and false negatives are discussed in Sections 8.2.3 and 8.2.4 respectively.

4.2.7.1 False Positives

As described in algorithm 4, the DACK protocol counts every data sample that is recovered by a $D1Msg$ or $D3Msg$. A *false positive* occurs when the base station assumes that a data sample was recovered by a $D1Msg$ or $D3Msg$, but in fact the data sample would have arrived anyway. In the current implementation, false positives have been observed to occur if data samples arrive during the time interval between when the base station updates dissemination status in algorithm 6 line 28 (see Section 4.1.2) and when the base station forms the dissemination packets. In the simulations and experiments discussed in Chapters 6 and 7, this interval could be as long as 800 milliseconds. In the experiment described in Section 7.3.4, 20 of 719 samples that were recovered over the course of the experiment were actually false positives.

4.2.7.2 False Negatives

A *false negative* occurs when the base station counts a data sample as permanently lost, when in fact the data sample does eventually arrive. False negatives can occur after the base station has already sent a $D3Msg$ to resolve a window overflow as discussed in section 4.2.5, or to resolve a storage overflow as discussed in section 4.2.6. If any data samples arrive at the base station having a timestamp older than the timestamp of the data sample with SN equal to the ASN sent in the $D3Msg$, then the base station will ignore the data samples. These samples were recovered, but the base station counts them as lost, so they are false negatives. In the simulations described in Chapter 6 and the experiments described in Chapter 7, false negatives were very rare.

4.3 Metrics

The efficiency of the DACK protocol is relative to the efficiency of the dissemination and collection protocols it is built on top of. The cost of the DACK protocol is measured as the number of packets disseminated, and the number of collection packets sent for the purpose of resending lost data samples. The benefit of the DACK protocol is measured as the number of data samples it recovers. The effectiveness of the DACK protocol is measured as the ratio of the number of data samples dropped to the number of data samples recovered. The following definitions define the metrics used to determine the DACK protocol benefit and cost:

- nD - 'number of disseminations': the total number of DACK dissemination packets disseminated by the base station during T.
- nC - 'number of collection packets sent': the total number of collection packets sent by motes in the network during T.
- nCR - 'number of collection packets resent': the total number of collection packets sent by motes in the network during T containing data samples that were sent previously.
- nS - 'number of samples': the total number of data samples sent by a mote.
- nA - 'number of acknowledgeable samples': this is the number of samples the base station becomes aware of through the ASN and LSN values in collection packets. This number may be smaller than nS if a mote experiences storage overflows.
- nRX - 'number of received samples': the total number of data samples received at the base station.
- nd - 'number dropped': the number of data samples sent by motes to the base station, but not received by the time the base station sends a $D1Msg$ or $D3Msg$ to recover the data sample. This can also be thought of as the number of data samples dropped by the collection protocol.

- nr - 'number recovered': the number of data samples that were recovered by the base station after sending a $D1Ack$ or $D3Ack$ message requesting a mote resend dropped samples.
- nl - 'number lost': the total number of data samples that the base station gives up on trying to acknowledge as described in Sections 4.2.5 and 4.2.6.
- no - 'number outstanding': this is the number of samples the base station sent a *D1Msg* or *D3Msg* for that has not yet been recovered or lost.
- nso - 'number of storage overflows': the number of storage overflows that occurred during an experiment (see section 4.2.6).
- nwo - 'number of window overflows': the number of window overflows that occurred during an experiment (see section 4.2.5).
- fp - 'false positives': the number of data samples in nr that were counted as recovered after a *D1Msg* or *D3Msg*, but would have been recovered anyway, as described in section 4.2.7.1.
- fn - 'false negatives': the number of data samples in nd the DACK protocol counts as dropped, but were actually received, as described in section 4.2.7.2.

The dissemination cost of using the DACK protocol is nD. The collection cost for using the DACK protocol is nCR. The benefit of the DACK protocol is the number of recovered data samples minus the false positives; i.e. $nr - fp$. The recover ratio rr is a measure of the effectiveness of the DACK protocol at recovering dropped samples. rr is calculated by dividing recovered data samples nr by the number of samples dropped samples nd minus the samples still outstanding no; i.e,

$$rr = nr/(nd - no) \tag{4.1}$$

Chapter 5

Implementation

The DACK protocol has been implemented into a prototype application called *SimpleNetwork*. The *SimpleNetwork* application was developed for TinyOS 2.1. Targeting for TinyOS allowed us to use the same source code for experiments using TelosB motes, and simulations using the TinyOS simulator TOSSIM. The DACK protocol has not yet been modularized, such that it can be used as a library by other applications. The current implementation provides a prototype application as a proof of concept. Section 8.2.5 discusses potential methods to modularize the implementation for use in other applications.

The *SimpleNetwork* application was first developed targeting TOSSIM, and later migrated to run on TelosB motes. The same code is used for both the simulations in Chapter 6 and the experiments in Chapter 7.

5.1 Code Overview

The *SimpleNetwork* application consists of a mote application written in nesC, a base station application written in Java, simulation configuration files, and a make file. Additionally, a Python script is used to parse the *SimpleNetwork* applications log file. Detailed listings of the code can be found in appendices A.1 to A.8. The source code repository is maintained by the sensor web language group at UNB.

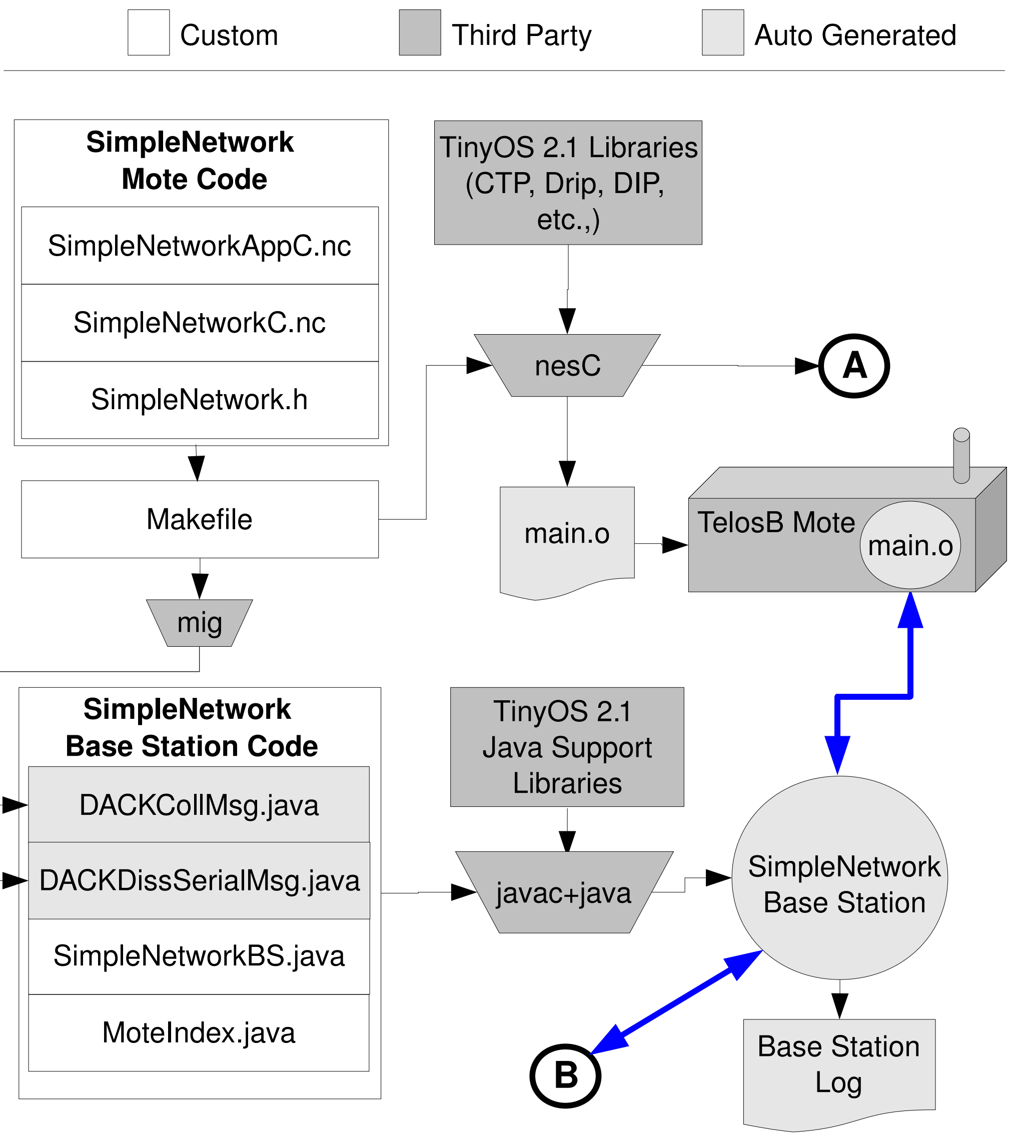


Figure 5.1: Overview of the files in the DACK prototype application *SimpleNetwork* used to build a network on TolosB Mote hardware. Rectangles represent programs, torn pages represent text or binary files, trapezoids represent compilers, and circles represent services. Directional arrows represent input from files and output from compilers and services. Bidirectional arrows represent communication between services. Circles marked A and B mark the connection points to the files used to build a *SimpleNetwork* simulation shown in figure 5.2.

Figure 5.1 shows the various files used in the implementation of the *SimpleNetwork* application, and how they relate to each other. The mote application for *SimpleNetwork* was written in nesC for TinyOS 2.1, and consists of the following three files:

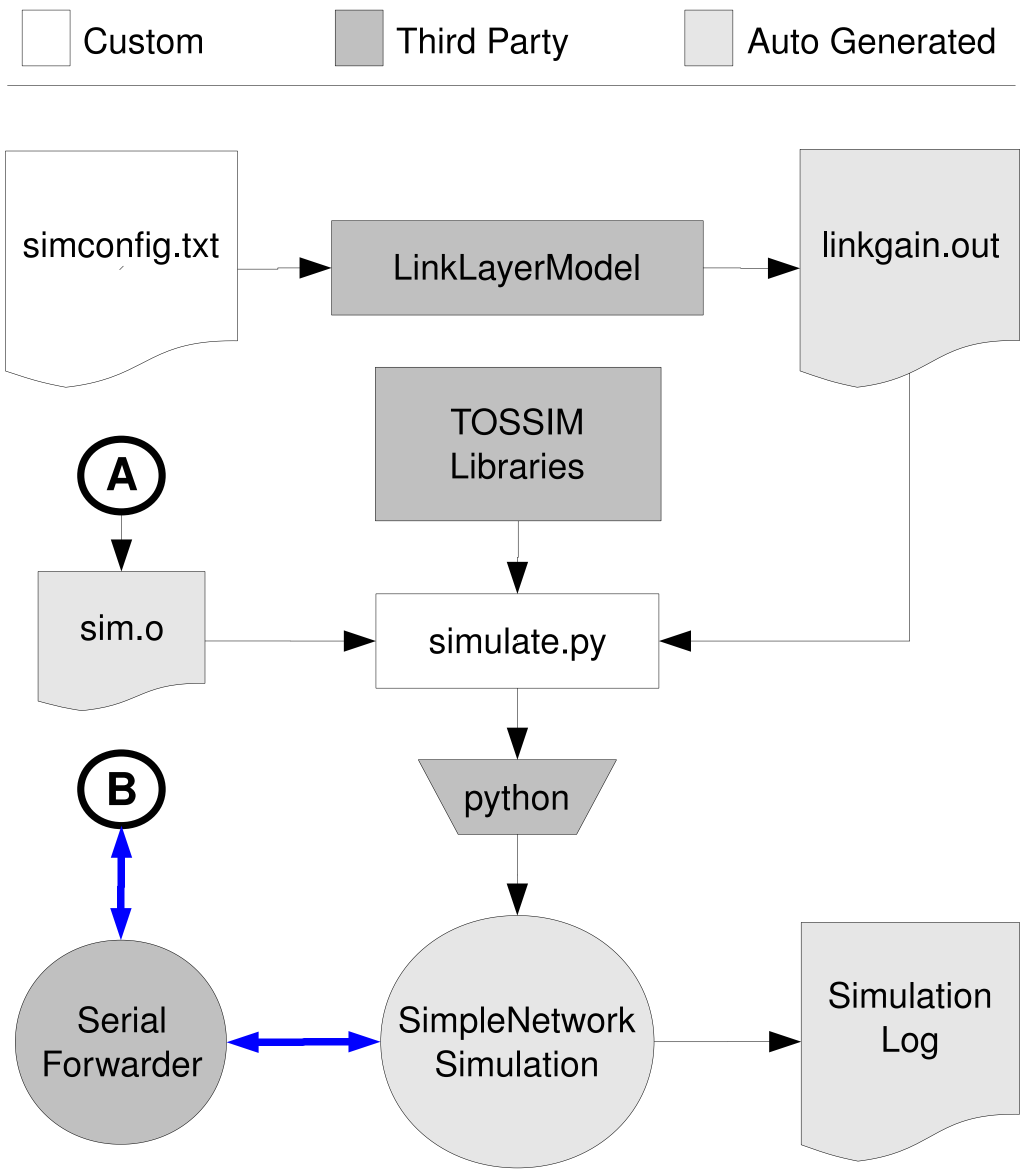


Figure 5.2: Overview of the files required for simulation of the DACK prototype application *SimpleNetwork*. Circles marked A and B mark the connection points to the files shown in figure 5.1. Communication with TOSSIM by the SerialForwarder requires the TOSSIM-Live extension discussed in chapter 3.

- *SimpleNetwork.h*: This is a header file containing packet structures and configuration parameters for the mote application. A complete listing of the source code can be found in appendix A.2.

- *SimpleNetworkC.nc*: This is the core implementation component of the *SimpleNetwork* mote application, and contains implementations of the mote side algorithms discussed in section 4.1. Excerpts from the source code, along with detailed descriptions, can be found in appendix A.6.

- *SimpleNetworkAppC.nc*: This is the top-level configuration file for the mote application. It links the implementation in *SimpleNetworkC.nc* with other components in the TinyOS library. A complete listing of the source code can be found in appendix A.5.

The base station application for *SimpleNetwork* was written in Java 1.5, and takes advantage of the TinyOS 2.1 Java support libraries for communicating with motes. To coordinate message structures with the mote application, the TinyOS Message Interface Generator (mig) is used to generate convenient Java object definitions for the messages structures defined in the SimpleNetwork.h header file. *DACKDissSerialMsg.java* is generated to define dissemination messages, and *DACKCollMsg.java* is generated to define collection messages. The base station application code consists of the following files:

- *SimpleNetworkBS.java*: This is the main base station application, and contains implementations of the base station algorithms discussed in section 4.1. Excerpts from the source code, along with detailed descriptions can be found in appendix A.6.
- *MoteIndex.java*: This is a definition of the *MoteIndex* object described in section 4.1.1. A complete listing of the source code can be found in appendix A.8.

The following two files were defined to run the *SimpleNetwork* application in the TinyOS Simulator:

- $simconfig.txt$: This file is used to define a radio signal model for motes in the wireless sensor network. Parameters can be set to represent mote locations and other network conditions for use in the simulation. This file is parsed by the program *LinkLayerModel* in the TinyOS 2.1 Java support library to produce *linkgain.out*, a text file containing the *LinkLayerModel*'s estimations for the link gain between each mote in the network. An example *simconfig.txt* used for one of the simulations can be found in Appendix A.3.
- $simulate.py$: This is a Python script that controls the TOSSIM simulation. This script parses the *linkgain.out* file, and uses this info to boot and run emulated motes in the simulation. An example *simulate.py* used for one of the simulations can be found in appendix A.4.

Finally, the following Pyhon script is used to parse the log files produced by the simulation and the experiment.

- $tabbs.py$: This python script is used to compile the results of an entire simulation or experiment. Because the results for each mote get reset on the base station after each storage overflow, *tabbs.py* sums the results of each instance of communication between the base station and the mote for the final tally. An instance of communication refers to the time interval between when the base station initializes the *MoteIndex* for a mote i, and the time a storage overflow is detected on Mote i. The results are parsed into tables used in the results in Chapters 6 and 7.

As shown in Figures 5.1 and 5.2 the *Makefile* invokes the nesC compiler to compile the *SimpleNetwork* mote code for either the simulation (*sim.o*), or an actual mote (*main.o*). *main.o* can be installed on one or several motes, and *sim.o* can be used by TOSSIM to simulate one or several motes. The *Makefile* also invokes *mig* to generate the java object definitions for message structures as described above. The *SimpleNetwork* base station code is compiled with *javac*, and then the *SimpleNetwork* base station can be run as a service using java. The base station communicates with an actual mote via a USB cable, or with a simulated gateway mote running inside TOSSIM with the *SerialForwarder* application.

5.2 Packet Structures

In Chapter 4 packets were defined abstractly. This chapter lists the actual DACK packet structures used in the implementation of the *SimpleNetwork* application.

Figure 5.3 shows the 9 byte data sample structure used for data samples $S(SN, SID, R, T)$. Sequence numbers SN are 2 bytes. The Sensor ID (SID) is only one byte, which is sufficient to map a unique integer to the TelosB's 8 ADC inputs, 2 digital inputs, and 5 internal sensors. Sensor reading R is 2 bytes as the TelosB ADC precision is 12 bits. The timestamp T is 4 bytes because it uses the TinyOS *LocalTimeMilliC* component, which returns a 32 bit timestamp of the number of binary milliseconds the mote has been running. Note that there are 1024 binary milliseconds per second [9].

Figure 5.4 shows the 25 byte packet structure for a basic DACK collection packet used in the *SimpleNetwork* application. Two bytes for the mote id MID reflects the 16-bit data type used for MIDs in TinyOS. The last consecutively acknowledged data sample ASN

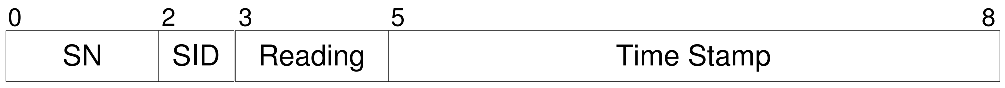


Figure 5.3: Structure of a Data Sample with a sequence number SN, sensor ID SID, a reading R, and a timestamp T. The SID is a sensor identifier which maps the reading to the sensor channel number, type, and data acquisition parameters (see section 3.5).

and the last sent data sample LSN are both 2 bytes, reflecting the size of sample sequence numbers SNs described above. The dissemination sequence number DSN is one byte. The nso metric is 2 bytes, allowing a maximum of 65536 storage overflows to be counted. These values are followed by two "slots" that can each hold one data sample.

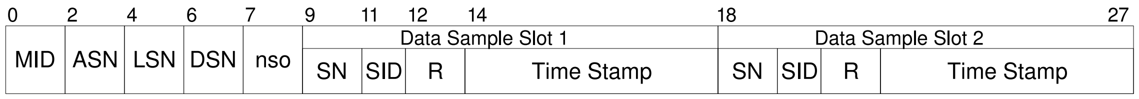


Figure 5.4: Structure of a DACK collection packet. This structure is embedded in a CTP (or other collection protocol) packet before being sent to the base station. Figures on top are in bytes. Each data sample is placed in one of two "slots".

To recover an accurate metric of the collection cost in experiments using SimpleNetwork, the DACK collection packet is supplemented with the two 4 byte metrics nC and nCR, as shown in Figure 5.5. These are not crucial to the functioning of the DACK protocol, but are included to recover accurate metrics in experiments. It is not necessary to include these values in simulations, as they can be recovered from parsing the simulation log file.

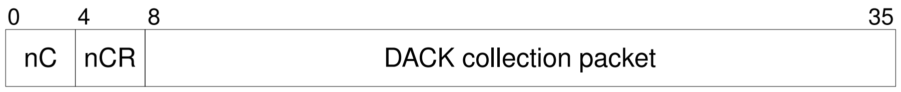


Figure 5.5: A DACK collection packet with the two additional 4 byte metrics nC and nRC.

As shown in Figures 5.6, 5.7, and 5.8, each of the three types of dissemination packets contain the 8-bit DSN as the last byte of the packet. The length of the dissemination packet limits the number of acknowledgment massages that can fit in the packet. In the current implementation, dissemination length is set to 48 bytes. DACK messages in $D1$ and $D3$

packets can vary in length depending on the length L of the acknowledgment vector A. If there are too many pending $D1$, $D2$, or $D3$ messages to fit in a $D1$, $D2$, or $D3$ packet in a dissemination interval, then a second $D1$, $D2$, or $D3$ can be generated. Each dissemination packet sent in parallel, requires a unique buffer on the motes. Memory is limited on motes, so the current application only buffers room for two $D1$ packets, two $D2$ packets, and two $D3$ packets.

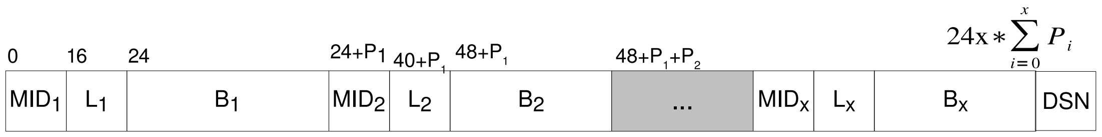


Figure 5.6: Bit structure of a $D1$ type DACK dissemination packet containing x acknowledgments consisting of a sequence of (MID, L, B) triplets.

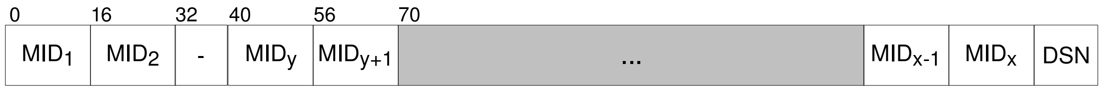


Figure 5.7: Bit structure of a $D2$ type DACK dissemination packet containing x acknowledgments consisting of a sorted sequence of MIDs and '-' characters. A $MID_i - MID_k$ pattern indicates all MIDs between and including MID_i and MID_k are acknowledged.

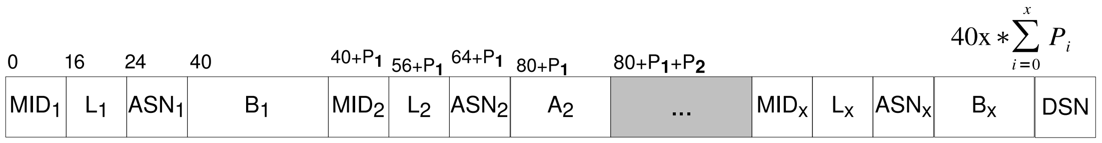


Figure 5.8: Bit structure of a $D3$ type DACK dissemination packet containing x acknowledgments consisting of a sequence of (MID, ASN, L, B) 4-tuples.

In $D1$, and $D3$ messages, the length of B is expanded to the nearest byte boundary. For example, if the length L_i of the ack vector B_i is 6 bits, then a full byte of packet space will be used to store B_i. We defined P to be the number of bytes required to store L bits, such that $P \leftarrow \lceil L/8 \rceil$. If $L_i = 0$, then $P_i = 0$ and B_i is can be skipped in the packet. In the current implementation, the $D1$, $D2$, and $D3$ dissemination packets were each capped at 48 bytes. This means that, if $L_i <= 8$ for all acknowledgment vectors B_i, then at most 15 motes can be acknowledged in one $D1$ packet, and 7 motes can be acknowledged in one $D3$ packet. The code for populating packets is discussed in further detain in Appendix A.7.

5.3 Base Station Data Structure Implementation

As discussed in chapter 4, the base station creates a MoteIndex object for each mote in the sensor network to keep track of each mote's status. The current implementation of MoteIndex object uses four arrays of size F to implement the $Storage$ structure (a circular buffer) described in Section 4.1.1. As in $Storage$, each element of the array corresponds to a possible value of SN. These four arrays are defined as follows:

- $ackvector$: contains the ack vector B stored at positions $[(ASN+1)\ mod\ F\ ..\ LSN]$ if $ASN < LSN$, or positions $[(ASN+1)\ mod\ F\ ..\ F-1]$ and $[0\ ..\ LSN]$ if ASN $> LSN$. All elements of $ackvector$ outside of these ranges are set to 0.
- $droppedackvector$: is used to count the number of dropped data samples. When a data sample with the sequence number SN is dropped, the base station sets $droppedackvector[SN]$ to 1. When the data sample becomes either recovered or lost (see section 4.1.1), the base station sets $droppedackvector[SN]$ back to 0.
- $acktimevector$: contains timestamps in binary ms since startup for each data sample. $acktimevector[SN]$ implements the $Storeage[SN].T$ structure described in section 4.1.1. Newer timestamps overwrite older timestampes.
- $ackcountvector$: contains a count of the number of data samples received or lost with a given sequence number. This is equivalent to $Storage[SN].Count$ described in section 4.1.1.

The MoteIndex object contains several functions to utilize these arrays, and to perform the necessary wrapping logic at the end of the array. The algorithms in Chapter 4 ignored this aspect of the implementation. For instance, Algorithm 6 contains the loop:

```
foreach $S$ in $MI.Storage$
        from $MI.Storage[(ASN+1)$ mod $F]$
        to $MI.Storage[LSN]$ do
    ...
end
```

The above loop presumes that the compiler knows that it may need to wrap around the storage array while traversing from ASN to LSN. Since the Java compiler has no such construct, to accomplish wrapping in Java, the following pattern is used:

```
if (LSN > ASN)
{       for (int i=ASN+1; i<=LSN; i++)
        { ...
        }
}
else
{       for (int i=ASN+1; i<F; i++)
        { ...
        }
        for (int i=0; i<=LSN; i++)
        { ...
        }
}
```

Variations of the above array wrapping pattern can be found in the source code for the MoteIndex object in Appendix A.8. Functions implemented for MoteIndex involving operations that wrap around the four arrays implementing the Storage Structure are:

- $getAckLength()$: returns the Length of the ACK vector B as a calculation on the values of ASN, LSN, and F.
- $getAckString()$: returns the ack vector B as a String, to be printed for debugging purposes.
- $getAckByteArray()$: returns the ack vector B as a byte array to be used in a $D1$ or $D3$ type dissemination message.
- $getOutstanding()$: returns the metric no, the number of outstanding data samples, by counting the number of 0 elements in the ACK vector.
- $isInRange(int\ i)$: returns *true* only if the element i is between $(ASN+1)\ mod\ F$ and LSN.
- $cleanAckVector()$: sets all the elements in the ackvector array that are not currently part of the ACK vector B to 0.
- $recordDroppedSamples()$: counts data samples that have been dropped using the $droppedackvector$.
- $allclear()$: returns *true* only if each element i is between $(ASN+1)\ mod\ F$ and LSN in the ackvector array is set to 1, indicating that every data sample has been acknowledged.
- $findASN()$: returns the value for ASN by analyzing the $ackcountvector$ array.
- $giveup()$: is called whenever a window overflow occurs. This function is responsible for cleaning up all four ackvector arrays, and accounting for lost data samples.

5.4 Protocol Verification

Each metric described in Section 4.3 is counted in a fashion that avoids using other metrics in the implementation. This method allows us to check for errors by making sure that related metrics add up as expected, and gives us some assurance that the protocol is working as expected. Two checks are used in the final implementation. The first check compares the number of dropped data samples nd to the sum of the number of recovered data samples nr, lost data samples nl, and outstanding data samples no, i.e.

$$check1 \leftarrow nd - (nr + nl + no) \tag{5.1}$$

If $check1$ is zero, the methods for counting samples agree. If $check1$ is not zero, the protocol is not behaving as expected. The second check compares the number of acknowledgeable data samples nA with the number of received data samples nRC minus the number of recovered data samples nr (so that they are not double counted), plus the number of dropped data samples, i.e.

$$check2 \leftarrow nA - ((nRC - nr) + nd) \tag{5.2}$$

As with $check1$, $check2 = 0$ indicates that the metrics agree.

The number of dropped samples nd is counted in the dissemination event E_D by calling the $recordDroppedSamples()$ function described in section 5.3. The number of recovered samples nr is counted in the collection packet received event E_C by checking if the $ackdroppedvector$ contains a value of 1 at element SN. The number of lost data samples nl is counted in the $giveup()$ function described in section 5.3. The number of outstanding samples no is counted by the $getOutstanding()$ function described in section 5.3.

The number of acknowledgable data samples nA for each mote is calculated by the function:

$$nA \leftarrow ackcountvector[RSN] * F + ackcountvector[RSN] \tag{5.3}$$

where RSN is the sequence number of the data sample with the largest timestamp from the mote. The number of collection packets resending data samples, nRC, is counted in the collection packet received event E_C, by checking the SN of the incoming data samples against the corresponding value in the $ackvector$ array. Since each of the metrics used in $check1$ and $check2$ are counted using different methods, a nonzero value in either equation (5.1) or (5.2) points to an error in the implementation of the DACK protocol. These checks were used extensively to debug the DACK protocol. The implementation was considered correct, when both equations reported zero for the entire run of a simulation or experiment.

Chapter 6

Simulation

Simulations of the *SimpleNetwork* application are made using the TinyOS Simulator [24] with the TOSSIM-Live extension [31]. The TOSSIM-Live extension allows emulation of serial communication between a simulated gateway mote and the base station application. TOSSIM and TOSSIM-Live are discussed in more detail in chapter 3.

Simulations of *SimpleNetwork* were run to debug the DACK protocol, and test the DACK protocol under various conditions. Many bugs in the implementation only emerged under rare conditions and required long simulations to reproduce. The DACK protocol itself appears to operate as it was designed, with two shortcomings being the false positives and false negatives discussed in Chapter 5. False positives and false negatives can be eliminated using methods discussed in Sections 8.2.3 and 8.2.4, but they are still present in the results of the current simulations and experiments.

6.1 Design

Sample storage was implemented by using an allocated segment of RAM. The limited amount of memory on motes requires a small value for F. Using a small value used for F and W causes both window overflow and storage overflow scenarios to occur frequently within the simulations, helping to accelerate the debugging.

To build our simulations, we used the link layer model discussed in [59] and provided by TOSSIM in the TinyOS 2.1 distribution. Using this method, the wireless channel for motes is modeled using the log-normal path loss model. The path loss model accepts four parameters: a path loss exponent (decay rate of the signal), a reference distance, a signal decay over the reference distance, and a standard deviation value for multi-path effects. For simulations of the *SimpleNetwork* application, these parameters were set to to the football field scenario detailed in the online TOSSIM network topology tutorial [58]. The link layer model also models network topologies, link asymmetries, and noise floor. Simulations of *SimpleNetwork* were done using a grid topology of evenly spaced nodes with symmetric links, and a noise floor of $-105dB$.

Two simulations are described in this chapter. Figures 6.1 and 6.2 show the grid topology with 8 meter spacing used for the two simulations. The gateway sensor node is always mote 0. All parameters were identical for both simulations, except for F and W. The aim of these simulations is to determining whether a larger window size F could improve the reliability of the $DACK$ protocol. We simulated 16 motes on a 4 by 4 grid topology, with motes spaced 8 meters apart. Earlier simulations showed that using 7 meters or less in the simulation causes the collection protocol (CTP) to perform so reliably that too few data samples were dropped to adequately test the DACK protocol. Spacing motes at 9 meters, on the other hand, caused the network to perform so poorly that no communications could be established with most of the motes in the sensor network. With an 8 meter spacing, connections between motes and the base station was good enough to establish a connection to each mote, and volatile enough to produce several storage overflows, and window overflows, such that the full range of the protocol could be tested and verified.

Each mote in the simulations has a sample interval I_S of 10 minutes, and a report interval I_R of 30 minutes. This value for I_S is chosen to give the network enough time to disseminate the dissemination messages to each mote. Dissemination is observed to occur much slower in simulations using TOSSIM than in experiments using real motes. For experiments, dissemination occurred much faster, allowing a sample interval I_S of 10 seconds and a report interval I_R of 30 seconds to be used. In experiments and the simulations, the base station has a dissemination interval I_D of 30 seconds. Simulations are run as fast as the CPU can run them. It takes the approximately 30 seconds for simulator to run a simulation of 16 motes running for 30 minutes, conveniently corresponds to a factor of 60. This allowed the base station code to use the same dissemination interval I_D for both simulations and experiments.

6.2 Measurement Process

The process for running a simulation is shown in Figure 5.2. First a simulation configuration file is generated as specified by [58]. An example of a simulation configure file from our simulations can be found in Appendix A.3. The simulation configuration file is then fed into the *LinkLayerModel* program provided by TOSSIM. The *LinkLayerModel* generates the *linkgain.out* file which lists the gain in decibels between each mote in the network. The *simulate.py* program configures the TOSSIM to run 16 instances of the SimpleNetwork mote application, in which each mote has the gain to other motes defined in the linkgain.out file. The simulated motes are able to communicate with the base station through the Serial-Forwarder application using the TOSSIM-Live extension.

Both TOSSIM and the base station write status information to stdout. This information is collected using the *script* application. After the simulation is run, theses files can be over a hundred megabytes. A python program, *tabbs.py* was developed to tabulate the information after the simulations have completed.

6.3 Results

Dozens of simulations were run in the course of debugging the DACK protocol before the checks described in Section 5.4 were satisfied. The final two simulations discussed here ran for a simulated 48 days, without errors being detected. The simulation was stopped at 48 days, because this is when the mote's 32 bit binary millisecond timer overflows, and handling for the timer overflow has not yet been implemented.

This section shows the results from two simulations with two different values for for F and W. All parameters in both simulations are identical to those described in Section 6.1 except F and W. The aim is to test if increasing the size of W would increase the reliability of the DACK protocol. The first simulation described in Section 6.3.1 uses $F = 50$ and $W = 24$, and the second simulation in Section 6.3.2 uses $F = 200$ and $W = 100$. The results show that the DACK protocol was able to recover approximately 4 times as many lost samples with the larger values of F and W..

6.3.1 Simulation One

In this simulation, F is set to 50 and W is set to 24. The results are shown in two tables. The metrics for each mote at the end of the simulation are shown in Table 6.1. A count of dissemination messages and collection messages, as well as a count of detected errors is recorded in Table 6.2. Figure 6.1 shows the topology used in both simulations. Simulation one ran for a simulated timespan of 48 days and 3 hours. This took approximately 12 hours in real time on a 3.00GHz Intel(R) Pentium(R) 4 CPU with 1 GB of RAM.

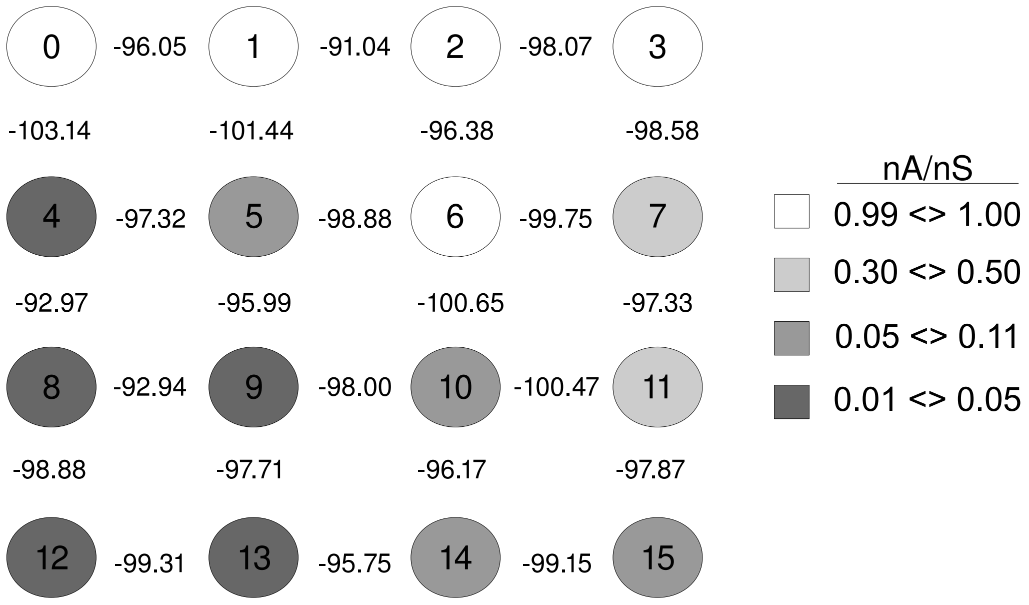


Figure 6.1: The figure above shows the topology of the 16 node grid simulations described in Section 6.3.1. The link gain value generated by the LinkLayerModel is shown between adjacent motes. As shown in the legend, the shade of gray indicates the number of acknowledgeable data samples nA divided by the total number of data samples collected in the experiment nS. Table 6.1 shows the nA/nS ratio for each mote.

As shown in Table 6.1, the DACK protocol recovered an average of 28% acknowledgeable dropped data samples. The benefit of 1290 is small, as network connectivity in this experiment is so poor that most of the data samples were not acknowledgeable.

The number of data samples sent, nS, is derived from the largest data sample timestamp. The discrepancies in the nS column reflect when each mote was last heard from. For example, the value 4609 for mote 8, means that the most recent data sample received from

Table 6.1: Metrics from simulation results with F=50, W=24.

MID	nS	nA	nA/nS	nRX	nd	nr	rr	nl	no	B
0	6932	6930	1.00	6930	0	0	0.00	0	0	0
1	6932	6930	1.00	6930	1	1	1.00	0	0	1
2	6932	6930	1.00	6930	0	0	0.00	0	0	0
3	6932	6930	1.00	6921	47	38	0.81	9	0	37
4	6848	113	0.02	60	59	6	0.10	53	0	6
5	6577	660	0.10	129	558	27	0.05	531	0	27
6	6932	6930	1.00	6922	11	3	0.27	8	0	1
7	6815	3242	0.48	2045	1683	486	0.29	1197	0	471
8	4609	72	0.02	55	25	8	0.32	17	0	8
9	6697	103	0.02	55	54	6	0.11	47	1	6
10	5927	333	0.06	88	260	15	0.06	245	0	15
11	6677	2050	0.31	1267	1196	413	0.35	783	0	406
12	6467	72	0.01	71	37	36	0.97	1	0	31
13	5756	218	0.04	65	163	10	0.06	153	0	10
14	6803	328	0.05	123	260	55	0.23	184	21	55
15	6824	782	0.11	573	430	221	0.51	209	0	216
Totals:	104660	42623	0.41	39164	4784	1325	0.28	3437	22	1290

mote 8 by the base station was the 4609th data sample that mote 8 sent.

The number of acknowledgeable data samples, nA, is the total number of data samples the base station is made aware of through received collection packets. For motes that have a solid connection to the base station, nA should be almost the same size as nS, since each mote was booted after the base station started listening. The difference between the nS and nA in motes with a solid connection to the base station is caused by the message connection protocol, as discussed in section 4.2.1.

The number of acknowledgeable samples divided by the number of known samples sent, nA/nS, reflects the overall reliability of the connection between the each mote and the base station as maintained by the collection protocol. Connection quality varied considerably, with most motes either gravitating towards very good quality, or very poor quality. Figure 6.1 shows how the connection quality varied over the simulated topology as reflected by nA/nS. The total number of received data samples, nRX, shows the number of data samples that were received by the collection protocol alone, plus the data samples that were recovered by the DACK protocol.

The recovery ratio $nr/(nd - no)$ is calculated as the number of acknowledgeable samples that were recovered, nr, divided by the number of data samples that were dropped nd. This ratio also varied widely from mote to mote. The no column shows the data samples that the base station was trying to recover when the simulation was terminated. The remainder of the data samples are lost, as indicated in the nl column. The benefit B of using the DACK protocol, is equal to the number of recovered data samples nr, minus the number of false positives shown fp in Table 6.2.

Table 6.2: Simulation messaging and error results fro F=50, W=24

MID	nD	nD1	nD2	nD3	nC	nRC	nso	nwo	fn	fp
0	2118	0	2069	49	4620	0	0	0	0	0
2	2079	0	2027	52	4620	0	0	0	0	0
3	1665	7	1289	369	4654	34	0	7	0	1
5	1105	14	4	1087	5049	781	169	17	0	0
6	2079	21	2002	56	4645	25	0	3	0	2
7	1234	141	146	947	5903	1413	116	48	0	15
8	1087	5	4	1078	3914	953	169	0	0	0
9	1091	4	6	1081	4993	596	135	1	0	0
10	1096	15	6	1075	4577	702	6	6	0	0
11	1165	114	64	987	6999	2636	149	29	0	7
12	1091	7	1	1083	4277	89	176	0	0	5
13	1093	3	7	1083	3790	142	241	3	0	0
14	1102	19	5	1078	4791	436	41	13	0	0
15	1123	62	19	1042	4864	425	175	14	0	5
Totals:	4863	272	2103	2488	76927	8501	1417	142	0	35

Table 6.2 shows the number of dissemination packets, collection packets, and observed errors during the simulation. The nD, $nD1$, $nD2$, and $nD3$ columns show the count of dissemination packets sent containing acknowledgment messages for each mote. The total number of packets sent to all motes (bottom row), is much smaller than the sum of each mote's count, as multiple dissemination messages are sent per packet as described in section 5.2. Motes that had very poor connectivity mostly required $D3$ type packets, and motes with a solid connection mostly required $D2$ type packets. $D1$ packets were used far less frequently than $D3$ packets. This suggests that dissemination packets were not getting acknowledged in time, forcing the base station to fall back on $D3$ type packets to stay in

sync. The number of $D3$s could be reduced significantly if the base station performed an exponential backoff. In the current implementation the base station continues to send a $D3$ packet every other dissemination regardless of whether it hears from the mote again or not.

The number of collection packets nC includes the number nRC of collection packets resending samples shown in the adjacent column. In this simulation, 11% of all collection packets were used for resending samples.

The final four columns of Table 6.2 show the number of storage overflows nso, window overflows nwo, false negatives fn, and false positives fp. The high number of storage overflows nso reflects the long periods of disconnectivity with motes. The number of window overflows, as well as the number of $D1$ packets, indicate that connectivity was lossy even when some communication could be established. There were no false negatives fn detected, and few false positives fp.

6.3.2 Simulation Two

In this simulation, F is set to 200 and W is set to 100. As with the previous simulation, the results are shown in two tables. The metrics for each mote at the end of the simulation are shown in Table 6.3, and a count of messages and errors is shown in Table 6.4. Figure 6.2 shows the topology and the ratio of acknowledgeable samples nA/nS for the simulation. Simulation two ran for a simulated timespan of 47 days and 22 hours.

As shown in Table 6.3, the DACK protocol recovered approximately 23% of acknowledgeable dropped data samples. This is a poorer recovery ratio than in simulation one. The benefit, on the other hand, seemed to increase significantly from 1290 in simulation one, to 4218 in simulation two. The larger window size appears to have increased the number of acknowledgeable data samples marginally, while significantly boosting the number dropped nd and the number recovered nr proportionally.

Table 6.4 shows that the cost of recovering 4 times as many packets lead to an 11 fold increase in the collection cost nCR, and a 13% increase in the dissemination cost. The

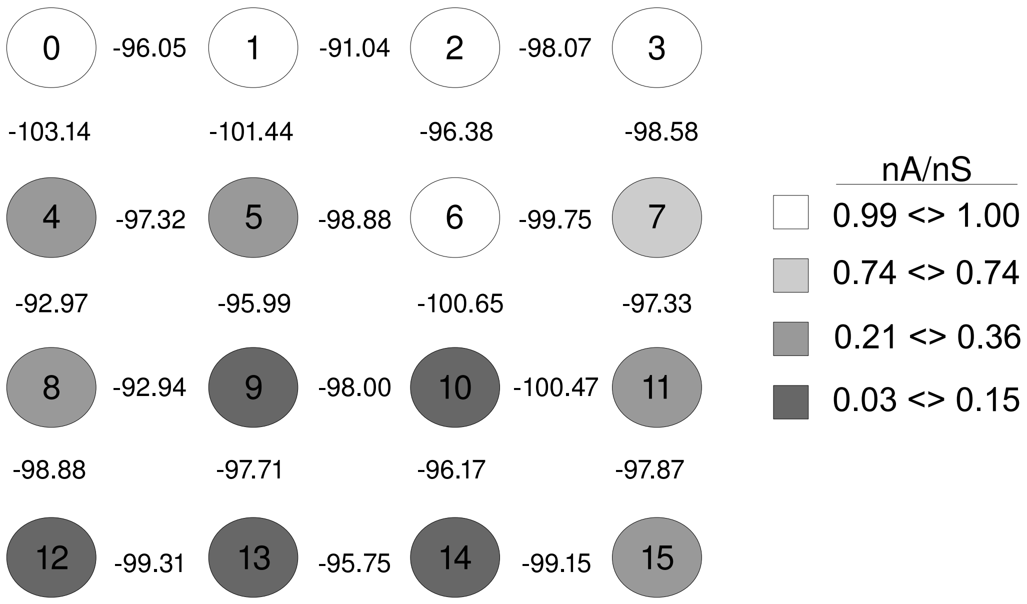


Figure 6.2: The figure above shows the topology of the 16 node grid for simulation 2. Note that the nA/nS ratio has changed significantly from Figure 6.1. The link gain values are the same as those in Figure 6.1. The actual results can be seen in Table 6.3.

Table 6.3: Metrics from simulation results with F=200, W=100

MID	nS	nA	nA/nS	nRX	nd	nr	rr	nl	no	B
0	6896	6894	1.00	6894	0	0	0.00	0	0	0
1	6896	6894	1.00	6894	4	4	1.00	0	0	4
2	6896	6894	1.00	6894	5	5	1.00	0	0	5
3	6839	6835	1.00	6266	2888	2319	0.80	569	0	2305
4	5972	2164	0.36	219	2073	128	0.06	1945	0	128
5	6584	2386	0.36	309	2302	225	0.10	2077	0	223
6	6896	6894	1.00	6894	15	15	1.00	0	0	13
7	6835	5055	0.74	1185	4666	796	0.17	3869	1	782
8	6530	1657	0.25	188	1611	142	0.09	1469	0	142
9	6746	999	0.15	140	964	105	0.11	859	0	105
10	6569	219	0.03	146	187	114	0.61	73	0	114
11	6689	1597	0.24	295	1433	131	0.09	1292	10	129
12	6182	620	0.10	105	592	77	0.13	515	0	77
13	6503	259	0.04	36	238	15	0.06	223	0	15
14	5614	147	0.03	41	127	21	0.17	106	0	21
15	6689	1400	0.21	315	1240	155	0.12	1085	0	155
Totals:	105336	50914	0.48	36821	18345	4252	0.23	14082	11	4218

sharp increase in the collection cost nCR is caused by the mote repeatedly trying to resend samples until an acknowledgment is received or a window overflow occurs.

Table 6.4: Simulation messaging and error results for F=200, W=100

MID	nD	nD1	nD2	nD3	nC	nRC	nso	nwo	fn	fp
0	2114	0	2109	5	4596	0	0	0	0	0
2	2067	3	2054	10	4599	3	0	0	0	0
3	1337	94	652	591	11068	6510	1	10	0	14
5	1082	65	2	1015	20109	15751	49	14	0	2
6	2065	9	2046	10	4606	10	0	0	0	2
7	1135	110	30	995	24650	20114	28	78	0	14
8	1090	26	2	1062	17058	12766	79	45	0	0
9	1117	30	2	1085	12453	7973	87	90	0	0
10	1071	22	1	1048	6824	2492	64	0	0	0
11	1190	36	11	1143	11728	7417	78	230	0	2
12	1066	9	0	1057	6087	2003	52	4	2	0
13	1065	10	1	1054	4826	550	86	2	0	0
14	1067	9	2	1056	6174	2481	76	0	0	0
15	1084	30	10	1044	9179	4658	82	18	0	0
Totals:	5473	353	2112	3008	166552	96779	730	615	2	34

6.3.3 Comparing Simulation Results

Table 6.5 shows the network totals for both simulations side by side. Increasing the storage and window size by a factor of 4 resulted in a proportional increase in the number of known dropped samples nd and the number of recovered data samples nr. nd increased slightly more than nr, resulting in a lower overall effectiveness. The cost of recovering more dropped data samples was mainly an 11 fold increase in the number of collection packets resending data samples.

The larger window size reduced the number of storage overflows from 1417 in simulation one to 730 in simulation two, and increased the number of window overflows from 142 in simulation one to 615 in simulation two. An increased number of $D3$ packets from 2488 to 3008 is also a consequence of the larger window size, as larger window sizes require longer acknowledgment vectors. The number of false positives and false negatives in both simulations is observed to be very small, accounting for less than 0.1% of the acknowledgeable data samples.

Table 6.5: Comparing results from both simulations.

	F=50	F=200
nS	104660	105336
nA	42623	50914
nA/nS	0.41	0.48
nRX	39164	36821
nd	4784	18345
nr	1325	4252
rr	0.28	0.23
nl	3437	14082
no	22	11
B	1290	4218
E	0.28	0.23
nD	4863	5473
nD1	272	353
nD2	2103	2112
nD3	2488	3008
nC	76427	166552
nRC	8501	96779
nso	1417	730
nwo	142	615
fn	0	2
fp	35	34

Chapter 7

Experiment

7.1 Design

The *SimpleNetwork* application was tested on a network of 14 TelosB motes. TelosB motes are discussed in Section 3.1. Figure 7.1 illustrates the approximate arrangement of motes in the ceiling panels of the Information Technology Center on the University of New Brunswick campus. Gateway mote 0 is connected to a PC acting as a base station in the data communications lab. All other motes were placed above ceiling panels in the hallways. Mote 7 was placed in the hallway outside of the door to the lab. Motes 3, 15, and 8 were placed at the end of the hall near the stair case to help with communication between floors. Motes 11, 6, and 10, were placed at the opposite end of the hall near the indoor balcony, also helping with communication between floors.

Figure 7.1 also shows the routing paths followed by collection packets. These routing paths were generated using the *MViz* application povided in the TinyOS 2.x toolchain, and setting the RF power of each mote in the network to 5. According to the CC2420 data sheet [7], this corresponds to a -20 dBm signal strength and a 9.2 mA current consumption. *MViz* is a simple network visualization tool that is included with the TinyOS 2.x distribution. By default, *MViz* uses the same collection protocol library that is used by the *SimpleNetwork*

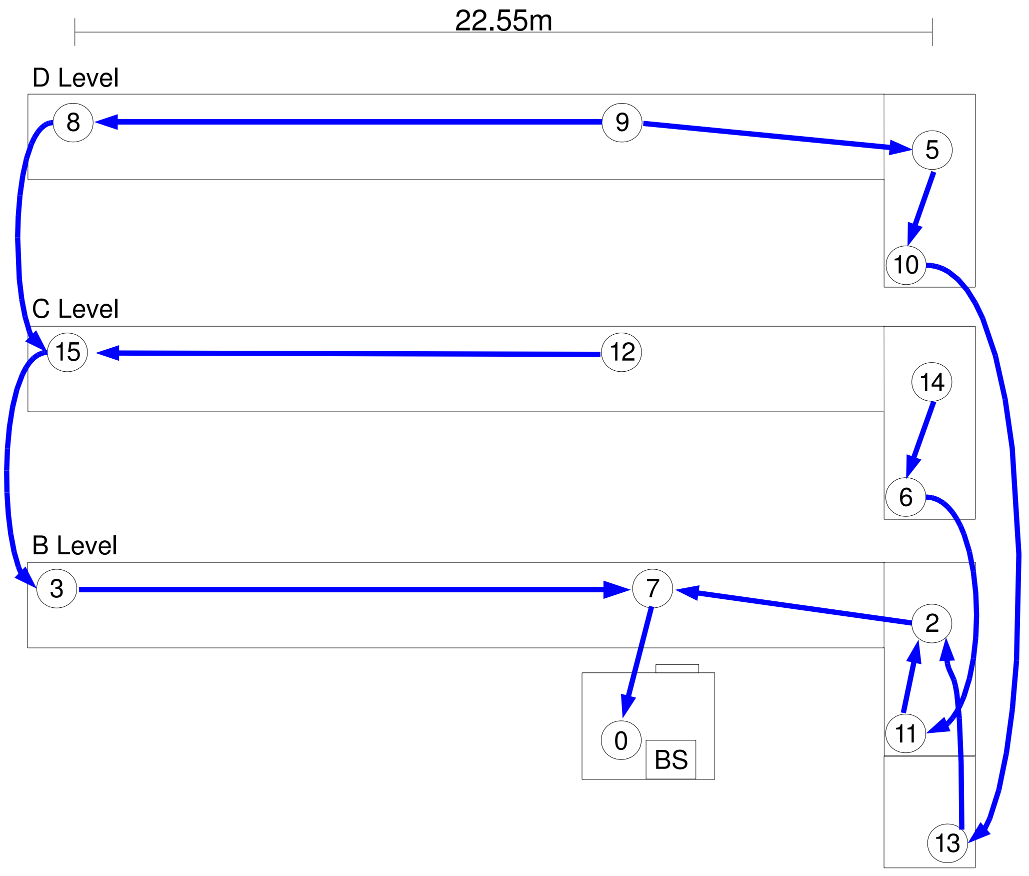


Figure 7.1: Deployment of TelosB motes for an experiment done in the Information Technology Centre on the University of New Brunswick campus. Fourteen motes were deployed, spread out over 3 floors as shown in the above diagram. Floors are vertically separated by 4.07 m. Arrows indicate routing paths traveled by collection packets, as negotiated by the Collection Tree Protocol.

application. This collection protocol is the Collection Tree Protocol (CTP) discussed in Section 3.4.1. In Figure 7.1 there are two arrows exiting mote 9, one pointed at mote 8, and one pointed at mote 5. This indicates that while *MViz* was running on the base station, it received collection packets that arrived after traveling via both routes. The routing paths also seem to indicate that the motes are not able to communicate very well through the floor at this power level.

We tried using both the DIP dissemination protocol and the Drip dissemination protocol. In the simulations, both the DIP and the Drip protocols worked fine. We were unable to get the DIP protocol to function in the *SimpleNetwork* application using the TelosB motes.

The Drip protocol, on the other hand, worked very effectively, disseminating messages out to each mote within seconds. For this reason, the Drip protocol was used for all experiments, as well as simulations.

The Collection Tree Protocol was also observed to work very effectively on the network. In a lab experiment we tested the effectiveness of the collection tree protocol using one mote and one base station at very low RF power levels. The results were similar to what we found in the Simulation. The Collection Tree Protocol works very well so long as a solid connection can be maintained. Varying energy levels and distances, the CTP achieves high reliability until the reception becomes so poor that the base station can not communicate with the mote at all. For example, in our experiments we tested effectiveness of the DACK protocol in our deployment at power levels 9, 7, 6 and 5 (estimated to be $-12dBm$, $-15dBm$ $-17.5dBm$, and $-20dBm$ respectivly.). At power level 6, the network was able to maintain a solid connection throughout the lifetime provided by two double a batteries powering each mote. At 5, estimated to be $-20dBm$ the connection became very poor after a day of operations in one attempt, and in our second attempt, we were unable to estable a connection at all at power level 5. At power level 4, no communications could be established.

As both the Drip and CTP protocol worked very effectively, we chose short intervals for sampling, reporting, and dissemination. In each experiment we set the sample interval I_S to be 10 seconds, and the report interval I_R and dissemination interval I_D both to be 30 seconds.

7.2 Measurement Process

As in the simulation, the *SimpleNetwork* base station application writes a log of all incoming collection messages, outgoing DACK messages, and the contents of its internal data structures. At the end of the experiment, the final results are tabulated using the *tabbs.py* script. Results were verified using the tests described in Section 5.4.

In the simulations, sensor readings all returned the hexadecimal value $0xBEEF$. For the experiment, the TelosB motes sample their internal battery voltage. This allowed us to monitor the energy level of the motes for the lifetime of the experiments.

7.3 Results

This chapter shows the results from 4 experiments. The first experiment, discussed in Section 7.3.1 was done using an older version of the source code that contained several bugs. For this reason only the sample reception rate is discussed. The other three experiments discussed in Sections 7.3.2, 7.3.3, and 7.3.4 were run using the same updated version of the source code that was used in the simulations discussed in Chapter 6.

7.3.1 Experiment One: Power Level 5

In early experiments, we tested the network at various power levels to find out the lowest common power level we could use such that every mote in the network could still communicate with the base station. Initially we observed this power level to be 5, which according to the $CC2420$ datasheet [7] corresponds to a $-20dBm$ and current consumption of $9.2mA$.

Our first experiment with power level 5 was done with an early version of the source code. The final results from this experiment have been omitted due to the observation of several errors that were detected by applying the checks described in Section 5.4. However, it is still interesting to note the connectivity of the network at this power level in this experiment.

Figure 7.2 shows the sample reception for each mote. Every mote reported fine for the first day. After the first day reception became very poor, with several motes loosing communication entirely, except for a few small spurts of activity.

We tried to run the experiment at RF level 5 again, after the source code was improved

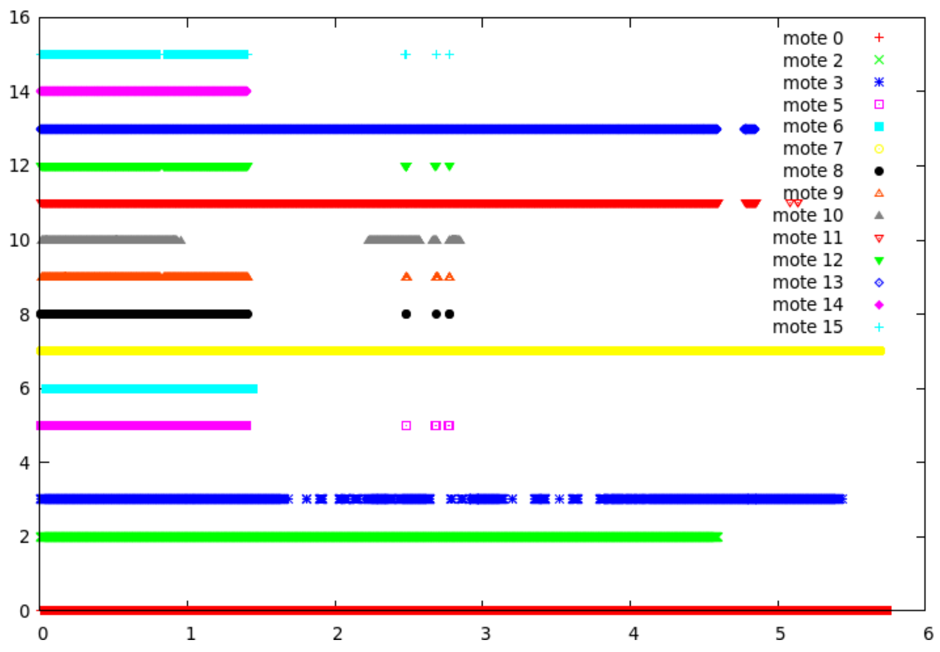


Figure 7.2: Sample reception observed with mote IDs on the y-axis, and day of operation on the x-axis. Motes are at power level 5 ($-20dBm$ and $9.2mA$). Blank spaces indicate no data was received by the base station for this mote.

and debugged. However, we were unable to get the network to communicate at all. This forced us to increase the power level to 6 and try again. The results of the 6 RF experiment are shown in Section 7.3.4.

7.3.2 Experiment Two: Power Level 9

This experiment began on Monday December 7th, 2009. Unfortunately, due to a human error, logging did not begin until Thursday December 10th. The motes continued running until Monday December 14th, when the batteries had drained below the TelosB minimum voltage requirement. According to the CC2420 datasheet, an RF power level of 9 corresponds to $-12dBm$ and a current draw of $10.5mA$.

Figure 7.3 shows the battery energy reading on the motes for the last three days of the experiment. Each mote reached the end of it's battery life after 6 days, and lost communication with the base station around the same time.

The results of the experiment are shown in Table 7.1. The nA/nS ratio reflects that

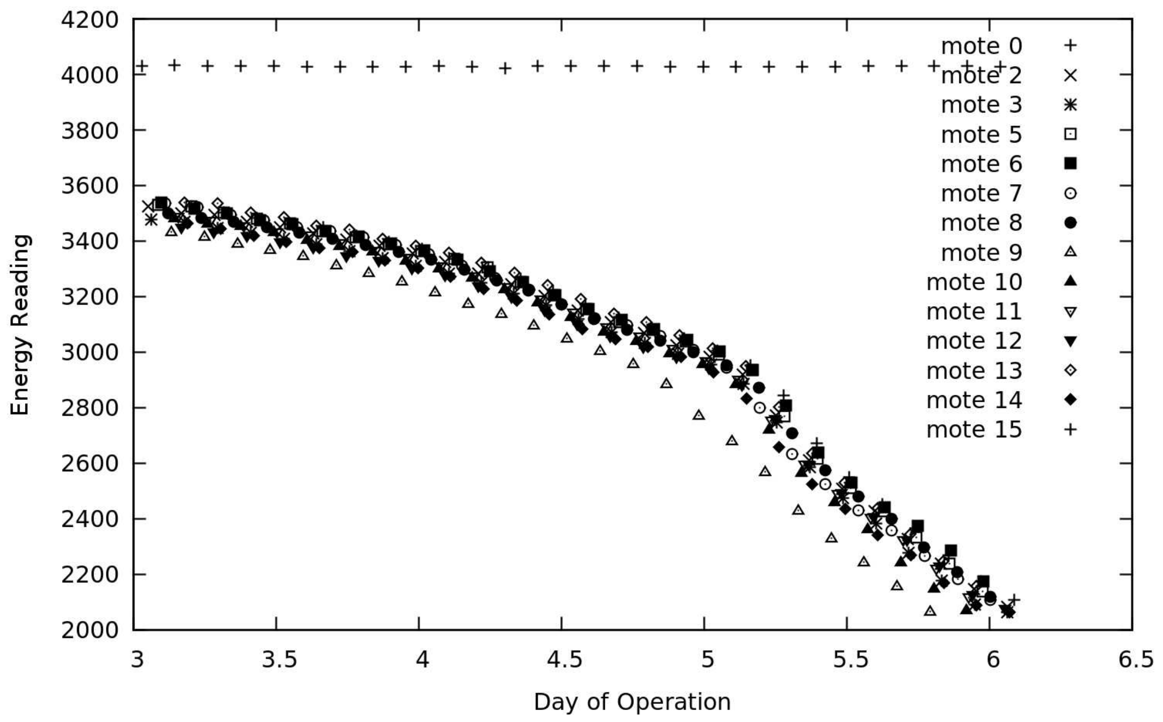


Figure 7.3: *Energy Reading* vs *Day of Operation* of a 14 mote sensor network experiment running the DACK protocol at RF power level 9: $-12dBm$ and $10.5mA$.

the base station did not begin acknowledging samples until halfway through the networks lifetime. The number of data samples that were recovered 154 is very small (0.04%) compared to the total number of samples collected 367514. The perfect recovery ratio rr shows that the DACK protocol managed to recover every data sample that was lost nl while the base station was listening. Though no samples were lost, there were several outstanding no packets. This indicates that communication became very poor towards the tail end of the mote battery life.

Table 7.2 shows the cost of using the DACK protocol in this experiment. It took 479 extra collection packets (nRC) and 9519 dissemination packets to maintain end to end communications and recover the 154 dropped samples shown in Table 7.1. No window overflows nwo, storage overflows nso, or false negatives occurred fn. There were only 6 false positives fp.

Table 7.1: Metrics from simulation results with RF=9.

MID	nS	nA	nA/nS	nRX	nd	nr	rr	nl	no	B
0	53002	26827	0.51	26827	3	3	1.00	0	0	3
2	53003	26830	0.51	26830	7	7	1.00	0	0	7
3	52738	26565	0.50	26542	32	9	1.00	0	23	8
5	52607	26434	0.50	26434	23	23	1.00	0	0	22
6	52606	26433	0.50	26433	11	11	1.00	0	0	9
7	52517	26344	0.50	26344	3	3	1.00	0	0	3
8	52654	26481	0.50	26436	58	13	1.00	0	45	12
9	50663	24490	0.48	24490	7	7	1.00	0	0	7
10	51832	25660	0.50	25659	11	10	1.00	0	1	10
11	51890	25718	0.50	25718	3	3	1.00	0	0	3
12	52604	26438	0.50	26438	7	7	1.00	0	0	7
13	52561	26395	0.50	26377	44	26	1.00	0	18	26
14	52580	26413	0.50	26413	14	14	1.00	0	0	14
15	52658	26486	0.50	26441	63	18	1.00	0	45	17
Totals:	733913	367514	0.50	367382	286	154	1.00	0	132	148

Table 7.2: Experiment messaging and error results for RF=9.

MID	nD	nD1	nD2	nD3	nC	nRC	nso	nwo	fn	fp
0	8794	3	8790	1	35336	4	0	0	0	0
2	8792	5	8785	2	32861	7	0	0	0	0
3	8689	6	8623	60	32749	10	0	0	0	1
5	8229	11	8147	71	32970	295	0	0	0	1
6	8241	8	8164	69	32694	13	0	0	0	2
7	8713	1	8632	80	32605	3	0	0	0	0
8	8226	7	8147	72	32661	20	0	0	0	1
9	7901	4	7512	385	31289	13	0	0	0	0
10	8021	6	7809	206	32246	16	0	0	0	0
11	8147	0	7958	189	32197	8	0	0	0	0
12	8341	4	8268	69	32560	9	0	0	0	0
13	8299	6	8210	83	32666	48	0	0	0	0
14	8308	7	8227	74	32554	13	0	0	0	0
15	8336	10	8256	70	32690	20	0	0	0	1
Totals:	9519	68	8794	657	458078	479	0	0	0	6

7.3.3 Experiment Three: Power Level 7

The experiment begun on Monday December 14, 2009 was interrupted by a power outage in the lab on Friday December 18, 2009. According to the CC2420 datasheet, an RF power level of 7 corresponds to $-15dBm$ and a current draw of $9.9mA$.

Figure 7.4 shows the battery energy readings recorded for this experiment. The results

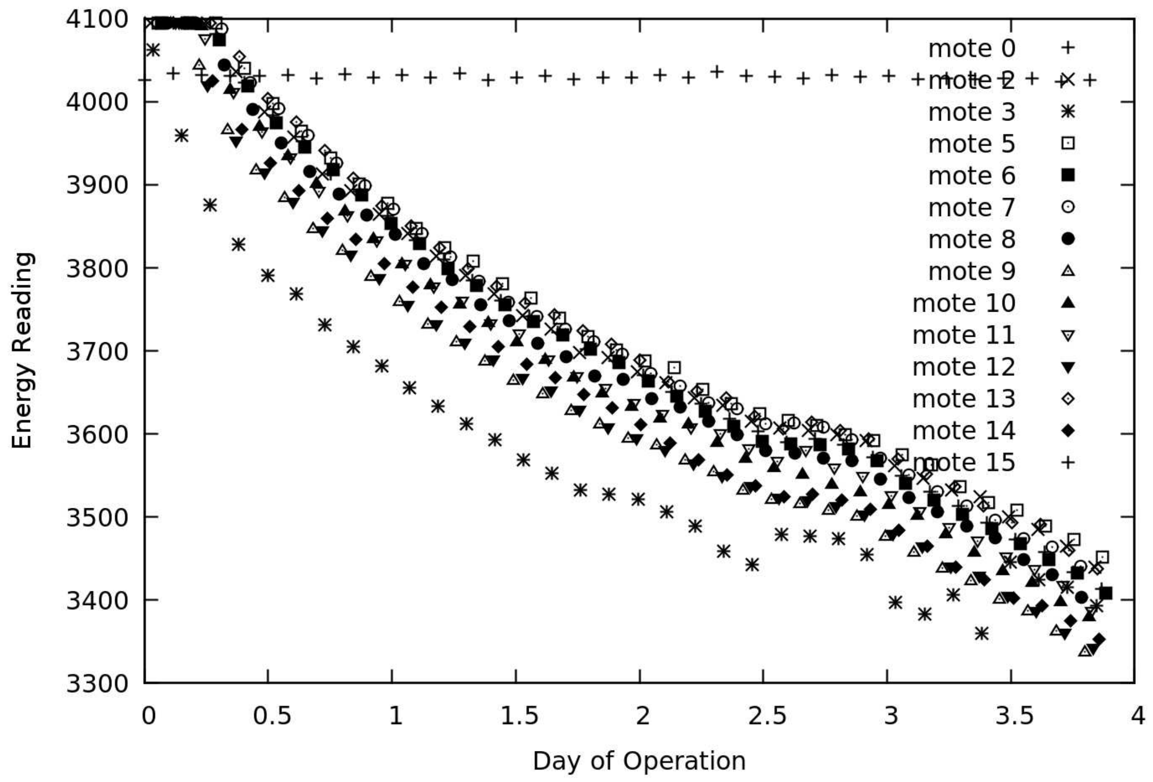


Figure 7.4: *Energy Reading* vs *Day of Operation* of a 14 mote sensor network experiment running the DACK protocol at RF power level 7: $-15dBm$ and $9.9mA$.

for the experiment are shown in Table 7.3. The nA/nS ratio is very close to one this time, because the base station was listening from the moment the motes became operational. With the exception of mote 11, the results for this experiment are very similar to the results for the experiment at power level 9 described in Sections 7.3.2 and power level 6 described in 7.3.4. Very few samples were lost, and all those that were lost were recovered by the DACK protocol.

The samples lost on mote 11 were lost due to the effect of a malformed data sample that broke the functioning of the $DACK$ protocol. The data sample was malformed such that it had a very large timestamp, larger than the lifetime of the entire experiment. The error can be found on line 1269991 of the base station log file explog-dec-14-200F-7RF. The large timestamp caused the base station to ignore incoming data samples with a lower timestamp, until finally a storage overflow occurred. After the storage overflow, communication with mote 11 resumed.

Table 7.3: Metrics from simulation results with RF=7

MID	nS	nA	nA/nS	nRX	nd	nr	rr	nl	no	B
0	33658	33657	1.00	33657	10	10	1.00	0	0	10
2	33661	33658	1.00	33658	35	35	1.00	0	0	34
3	33659	33658	1.00	33658	10	10	1.00	0	0	9
5	33658	33657	1.00	33657	30	30	1.00	0	0	30
6	33658	33657	1.00	33657	19	19	1.00	0	0	19
7	33658	33657	1.00	33657	3	3	1.00	0	0	3
8	33659	33658	1.00	33658	35	35	1.00	0	0	34
9	33658	33657	1.00	33657	15	15	1.00	0	0	15
10	33658	33657	1.00	33657	26	26	1.00	0	0	26
11	33659	33625	1.00	33629	223	36	0.16	163	0	34
12	33659	33658	1.00	33658	10	10	1.00	0	0	10
13	33659	33658	1.00	33658	21	21	1.00	0	0	19
14	33662	33658	1.00	33658	11	11	1.00	0	0	11
15	33662	33658	1.00	33658	16	16	1.00	0	0	16
Totals:	471226	471173	1.00	471177	464	277	0.60	163	0	270

Table 7.4 shows the cost of running the DACK protocol in this experiment. It took 247 extra collection packets (nRC) and 11412 dissemination packets (nD) to maintain end to end communications and recover the 270 dropped samples shown in 7.3. The storage overflow that occurred on Mote 11 is shown in the nso column. There were very few false positives. The 7 window overflows and 1072 false negatives on mote 11 were also the result of the malformed data sample discussed previously.

7.3.4 Experiment Four: Power Level 6

This experiment began on Monday December 21st, and concluded on Monday December 28th. We were able to log the output of the entire lifetime of the batteries with no interruptions. Figure 7.4 shows the battery energy readings recorded for this experiment. According to the CC2420 datasheet, RF power level 6 corresponds to $-17dBm$ and a current draw of $9.5mA$. At this power level, the network was able to maintain a solid connection with the base station using the collection tree protocol. As described in Section 7.3.1, setting the power level to 5 resulted in very poor to no connectivity.

The results for the experiment are shown in Table 7.5. The small difference between

Table 7.4: Experiment messaging and error results for RF=7

MID	nD	nD1	nD2	nD3	nC	nRC	nso	nwo	fn	fp
0	11035	6	11027	2	22444	7	0	0	0	0
2	10133	24	10050	59	20833	28	0	0	0	1
3	10114	7	10023	84	20928	13	0	0	0	1
5	9748	16	9670	62	20900	32	0	0	0	0
6	9824	13	9749	62	20934	15	0	0	0	0
7	11031	3	11025	3	20929	3	0	0	0	0
8	9807	18	9711	78	20917	33	0	0	0	1
9	9843	10	9776	57	20833	12	0	0	0	0
10	9778	14	9701	63	20948	24	0	0	0	0
11	9980	17	9901	62	20954	27	1	7	1072	2
12	10082	6	10026	50	20788	9	0	0	0	0
13	9948	14	9880	54	20994	18	0	0	0	2
14	9943	9	9879	55	20868	10	0	0	0	0
15	10041	12	9975	54	20927	16	0	0	0	0
Totals:	11412	160	11034	218	294197	247	1	7	1072	7

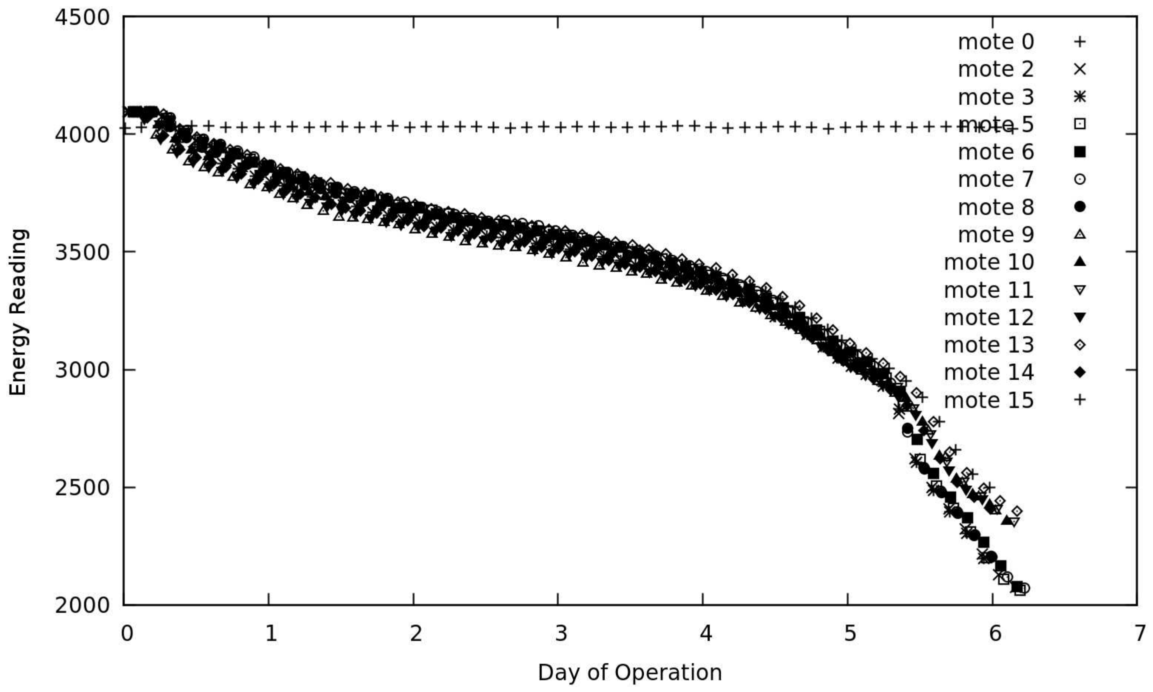


Figure 7.5: *Energy Reading* vs *Day of Operation* of a 14 mote sensor network experiment running the DACK protocol at RF power level 6: $-17dBm$ and $9.5mA$.

nS and nA reflect that the base station did not start until a few minutes after the motes had been booted. In this experiment the DACK protocol was able to achieve a perfect recovery ratio rr and recover 719 data samples.

Table 7.5: Metrics from simulation results with RF=6.

MID	nS	nA	nA/nS	nRX	nd	nr	rr	nl	no	B
0	53737	53671	1.00	53671	13	13	1.00	0	0	13
2	53612	53546	1.00	53546	20	20	1.00	0	0	20
3	53612	53546	1.00	53536	54	44	1.00	0	10	43
5	53612	53546	1.00	53546	69	69	1.00	0	0	67
6	53612	53543	1.00	53543	21	21	1.00	0	0	21
7	53741	53672	1.00	53672	6	6	1.00	0	0	6
8	53615	53545	1.00	53545	117	117	1.00	0	0	116
9	53615	53545	1.00	53545	120	120	1.00	0	0	114
10	53612	53546	1.00	53546	31	31	1.00	0	0	30
11	53612	53545	1.00	53545	21	21	1.00	0	0	19
12	53612	53546	1.00	53546	106	106	1.00	0	0	103
13	53611	53544	1.00	53544	14	14	1.00	0	0	14
14	53612	53546	1.00	53546	74	74	1.00	0	0	71
15	53615	53563	1.00	53563	63	63	1.00	0	0	62
Totals:	750828	749904	1.00	749894	729	719	1.00	0	10	699

Table 7.6 shows the cost of running the DACK protocol in this experiment. It took 720 extra collection packets (nRC) and 18733 dissemination packets (nD) to maintain end-to-end communication and recover the 719 dropped samples shown in 7.5.

Table 7.6: Experiment messaging and error results for RF=6

MID	nD	nD1	nD2	nD3	nC	nRC	nso	nwo	fn	fp
0	17581	10	17569	2	35833	10	0	0	0	0
2	16454	12	16352	90	33313	17	0	0	0	0
3	15330	27	14911	392	33102	61	0	0	0	1
5	15411	37	15213	161	33254	65	0	0	0	2
6	16024	15	15900	109	33351	17	0	0	0	0
7	17579	6	17571	2	33384	20	0	0	0	0
8	15255	59	14964	232	33348	91	0	0	0	1
9	15451	56	15239	156	33215	104	0	0	0	6
10	15589	21	15423	145	33302	58	0	0	0	1
11	16303	17	16184	102	33306	18	0	0	0	2
12	15721	62	15510	149	33213	82	0	0	0	3
13	16158	11	16043	104	33239	14	0	0	0	0
14	15827	40	15638	149	33255	55	0	0	0	3
15	15836	29	15642	165	33383	108	0	0	0	1
Totals:	18733	389	17583	761	468498	720	0	0	0	20

Chapter 8

Conclusions and Future Work

8.1 Conclusions

An end-to-end dissemination acknowledgment protocol (DACK) for wireless sensor networks was presented. The protocol works well when underlying network collection and dissemination protocols (e.g. CTP and Drip) are operating reliably. Our six-day experiment with 14 motes sending 749,904 acknowledgeable data samples showed a 100% recovery of the 719 data samples dropped if sent via the collection (CTP) protocol only. This increasing reliability comes at a cost of resending 720 collection packets plus 18,733 dissemination packets. This cost is reasonable if data reliability is an important consideration in the domain where the sensor network is deployed.

8.2 Future Work

The current implementation is a proof-of-concept. Several possible improvements could be implemented to reduce the energy cost of using the protocol, make it more robust, and to make it more practical for integration with other applications.

8.2.1 Rate Control

The DACK protocol could be improved by using various techniques to limit the rate of disseminations on the base station and sample resending on the motes. In the current implementation, if the base station receives a few messages from mote i and then never hears from mote i again, the base station will continue to disseminate messages to mote i every second dissemination interval until the base station program is terminated. Similarly, if a mote receives a request to resend an old sample and then never hears from the base station again, the mote will resend the sample until a window overflow occurs. In both cases, an exponential backoff for repeated disseminations and data samples may significantly reduce the overall cost without significantly effecting overall reliability.

In the current implementation the dissemination interval is controlled by the network operator. In reality, it would be better to expect the network operator only to have to set the sample and report intervals on motes. The dissemination interval could be dynamically optimized on the base station to minimize cost while meeting the needs of a potentially changing network. One method would be to have the base station set a different dissemination interval for each mote that was equal to motes report interval. Combined with the exponential backoff described above, these two rules could provide adaptive rate control for many wireless sensor network scenarios.

8.2.2 Alternative dissemination methods

The dissemination method described here involved putting DACK messages D1Ack, D2Ack, and D3Ack into D1, D2, and D3 type packets (respectively) for dissemination. This method is dangerous as there is no guarantee that the previous $D1$ message is disseminated to the entire network when the base station next decides to send a different $D1$ message. On large enough time scales, this should be okay. In our experiments using Drip, disseminating a single packet to 14 motes worked reliably in a few seconds.

One alternative would be to take advantage of the DIP protocols method for disseminating small messages. Instead of sending large 48 byte packets, we could disseminate shorter messages, such that there is one small buffer for each mote in the network. This way one mote's dissemination path will not interfere with another motes dissemination path. The downside of this method is it would require each mote in the network to allocate space for every other mote in the network, which would have difficulty scaling to large networks.

8.2.3 Resolving False Positives

False positives can occur as described in Section 4.2.7.1. Results from the simulations in Chapter 6 and the experiments in Chapter 7 show that false positives occur infrequently relative to the number of recovered data samples. The number of false positives could be reduced by having the base station only respond to data samples that are older than the length of the mote's report interval. This will have consequences to the timeliness of the protocol, and may result in larger B vectors, requiring more dissemination space. It may be possible to incorporate some predictive rules to help reduce false positives without suffering the same cost to timeliness.

8.2.4 Resolving False Negatives

False negatives can occur as described in Section 4.2.7.1. False negatives can be resolved by decoupling the DACK protocol from the base station sample storage program. If an incoming data sample is ignored by the DACK protocol, the DACK protocol can still relay the message to the storage layer, and the storage layer could determine that if data sample is new, and store it in the proper location.

8.2.5 Modularising DACK

How difficult would it be to decouple the DACK protocol from the prototype application, such that other network administrators could plug it in to their wireless collection networks? The DACK protocol describes end-to-end acknowledgments for data samples, and not packets. The current implementation relies on the data samples to have consecutive sequence numbers and timestamps. This means that a modularized version of the DACK protocol would have to interface with both the communication stack, and the storage stack.

If there is sufficient space on the mote, a modularized version of the DACK protocol could buffer arbitrary messages as data samples in a storage space specifically allocated for the DACK protocol. The prototype application of the DACK protocol presented in this thesis shows that the DACK protocol is functional using a small storage space that requires less than a few kilobytes of RAM. This is significant, as many motes have only 4 to 16 kilobytes of RAM total, but it is small enough to fit on many modern mote platforms.

8.2.6 Scalability Analysis

The scalability of the current implementation of the DACK protocol is unknown. An analysis of the scalability of the DACK protocol could be achieved though a series of simulations and experiments. It would also be useful to model the DACK protocol mathematically to predict how much storage space on motes would be required for networks of a given size with a set of given sample intervals and report intervals.

References

[1] R. Beckwith, D. Teibel, and P. Bowen. Unwired wine: sensor networks in vineyards. *Sensors, 2004. Proceedings of IEEE*, pages 561–564 vol.2, Oct. 2004.

[2] P.O. Bobbie, C. Deosthale, and W. Thain. Telemedicine: A mote-based data acquisition system for real time health monitoring. In *Telehealth 2006*. ACTA Press, 2006.

[3] T. Bokareva, W. Hu, S. Kanhere, B. Ristic, N. Gordon, T. Bessell, M. Rutten, and S. Jha. Wireless sensor networks for battlefield surveillance. 2006.

[4] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: ultra-low power data gathering in sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 450–459, New York, NY, USA, 2007. ACM.

[5] V. Cerf, Y. Dalal, and C. Sunshine. Specification of Internet Transmission Control Program. RFC 675, December 1974.

[6] Chipcon. *cc1000 Single Chip Very Low Power RF Transeiver*. Chipcon, April 2004. 50 pages.

[7] Chipcon. *cc2420: 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Tranceiver*, June 2004. Priliminary Datasheet (rev 1.2).

[8] Atmel Corporation. Atmel atmega 128 microcontroller datasheet. available from: <http://www.atmel.com/atmel/acrobat/doc2467.pdf>.

[9] David Gay Cory Sharp, Martin Turon. Tep 102: Timers, 2007.

[10] D. D. Couto, D. Aguayo, B. Chambers, and R. Morris. Performance of multihop wirless. In *First Workshop on Hot Topics in Networks)*, 2002.

[11] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[12] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, Sukun Kim, Philip Levis, and Alec Woo. Tep 123: The collection tree protocol (ctp), 2006.

[13] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation (PLD), San Diego, California, USA*, pages 1–11, June 9-11, 2003.

[14] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, and Philip Levis. Ctp: Robust and efficient collection through control and data plane integration. Technical Report SING-08-02, Stanford Information Networks Group, 2008.

[15] Luciano Gonda and Carlos Eduardo Cugnasca. A proposal of greenhouse control using wireless sensor networks. In *Computers in Agriculture and Natural Resources, Proceedings of the 4th World Congress Conference*, Orlando, Florida, USA, July 2006.

[16] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[17] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[18] IEEE. *Wireless medium access control and physical layer specifications for low-rate wireless personal area networks. IEEE Standard, 802.15.4-2003*, May 2003.

[19] Crossbow Technology Inc. Micaz datasheet.

[20] ISO. *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO, February 1995. ISO/IEC 7498-1:1994.

[21] Sukun Kim. *Wireless Sensor Networks for High Fidelity Sampling*. PhD thesis, EECS Department, University of California, Berkeley, Jul 2007.

[22] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. Flush: a reliable bulk transport protocol for multihop wireless networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 351–365, New York, NY, USA, 2007. ACM.

[23] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[24] Phil Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *ACM SensSys 2003*, November 2003.

[25] Philip Levis. Tep structure and keywords. Technical Report 1, TinyOS 2.0 Core Working Group, 2006.

[26] Philip Levis, Eric Brewer, David Culler, David Gay, Samuel Madden, Neil Patel, Joeseph Polastre, Scott Shenker, Robert Szewczyk, and Alec Woo. The emergence of a networking primitive in wireless sensor networks. *Commun. ACM*, 51(7):99–106, 2008.

[27] Philip Levis and David Culler. Mate : a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, December 2002.

[28] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[29] ARM Limited. System-on-chip platform os processor, 2000.

[30] Kaisen Lin and Philip Levis. Data discovery and dissemination with dip. In *IPSN '08: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 433–444, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Chad Stephen Metcalf. *TOSSIM Live: Towards a Testbed in a Thread*. PhD thesis, Colorado School of Mines, 2007. available from: <http://toilers.mines.edu/pub/Toilers/ChadMetcalf/cmetcalf-thesis.pdf >.

[32] Geoff Mulligan. The 6lowpan architecture. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 78–82, New York, NY, USA, 2007. ACM.

[33] Razvan Musaloiu-E., Chieh-Jan Mike Liang, and Andreas Terzis. Koala: Ultra-low power data retrieval in wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 421–432, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Bradford G. Nickerson, Zhongwei Sun, and John-Paul Arp. A sensor web language for mesh architectures. In *CNSR*, pages 269–274, 2005.

[35] J. Hill P. Buonadonna and D. Culler. Active message communication for tiny networked sensors. In *In Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, Anchorage, Alaska, USA, 2001. IEEE.

[36] Paulo Rogrio Pereira, Antnio Manuel Raminhos Cordeiro Grilo, Francisco Rocha, Mrio Serafim Nunes, Augusto Casaca, Claude Chaudet, Peter Almstrom, and Mikael Johansson. End-to-end reliability in wireless sensor networks: Survey and research challenges. pages 67–74, December 2007.

[37] Joseph Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California, Berkeley, CA, May 23, 2003. available from: <http://www.cs.berkeley.edu/ polastre/pubs.html>.

[38] Joseph Polastre. *A Unifying Link Abstraction for Wireless Sensor Networks*. PhD thesis, University of California, Berkeley, CA, 2005. available from: <http://www.cs.berkeley.edu/ polastre/pubs.html >.

[39] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM Press.

[40] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, pages 364–369, Los Angeles, California, April 2005.

[41] Technology Review. 10 emerging technologies that will change the world. *MIT Technology Review*, 2003.

[42] Inc RF Monolithics. *RFM TR1000 Datasheet*. RF Monolithics, Inc, 1999. available from: <http://www.rfm.com/products/data/tr1000.pdf>.

[43] Inderjit Singh. Real-time object tracking with wireless sensor networks. Master's thesis, Lule University of Technology, 2007. available from: http://epubl.ltu.se/1653-0187/2007/059/index-en.html.

[44] J. Slipp, Changning Ma, N. Polu, J. Nicholson, M. Murillo, and S. Hussain. Winter: Architecture and applications of a wireless industrial sensor network testbed for radio-harsh environments. *Communication Networks and Services Research Conference, 2008. CNSR 2008. 6th Annual*, pages 422–431, May 2008.

[45] Inc. Sun Microsystems. Sun small programmable object technology (sun spot) theory of operation, 2007.

[46] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM.

[47] Wei Tan, Qianping Wang, Hai Huang, Yongling Guo, and Guoxia Zhang. Mine fire detection system based on wireless sensor network. In *Proceedings of the 2007 International Conference on Information Acquisition*, pages 148–151, 2007.

[48] Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, Sukun Kim, Jorge Ortiz, David Culler, Phillip Levis, and Scott Shenker. A modular sensornet architecture: past, present, and future directions. *SIGBED Rev.*, 4(3):49–54, 2007.

[49] Arsalan Tavakoli, Prabal Dutta, Jaein Jeong, Sukun Kim, Jorge Ortiz, David Culler, Phillip Levis, and Scott Shenker. A modular sensornet architecture: past, present, and future directions. *SIGBED Rev.*, 4(3):49–54, 2007.

[50] John Heideman Deborah Estrin Karen Weeks Thanos Stathopoulos, Lewis Girod. Centralized routing for resource-constrained wireless sensor networks (sys 5), 2006.

[51] Gilman Tolle and David Culler. Design of an application-cooperative management system for wireless sensor networks. *Second European Workshop on Wireless Sensor Networks (EWSN)*, January 2005.

[52] Gilman Tolle, Joseph Polastre, Robert Szewczyk, Neil Turner, Kevin Tu, Phil Buonadonna, Stephen Burgess, David Gay, Wei Hong, Todd Dawson, and David Culler. A macroscope in the redwoods. *Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2005.

[53] J. Wall, G. Platt, G. James, and P. Valencia. Wireless sensor networks as agents for intelligent control of distributed energy resources. *Wireless Pervasive Computing, 2007. ISWPC '07. 2nd International Symposium on*, pages –, Feb. 2007.

[54] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, March 2006.

[55] Alec Woo and David E. Culler. A transmission control scheme for media access in sensor networks. In *Mobile Computing and Networking*, pages 221–235, 2001.

[56] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM Press.

[57] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the 21st Int. Annual Joint Conf. of the IEEE Computer and Communication Societies (INFOCOM), vol. 3*, pages 1567–1576, New York, NY, U.S.A., June 2002.

[58] Marco Zuniga. Building a network topology for tossim. available from: <http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/usc-topologies.html>.

[59] Marco Zuniga, Bhaskar Krishnamachari, and Rahul Urgaonkar. Realistic wireless link quality model and generator v. 1.1. Technical report, December, 2005. available from: <http://anrg.usc.edu/www/downloads/LinkModellingTutorial.pdf>.

Appendix A

Implementation Source Code and Descriptions

A.1 Makefile

Listing A.1: Project Makefile

```
COMPONENT=SimpleNetworkAppC
CFLAGS += -I$(TOSDIR)/lib/net
#CFLAGS += -I$(TOSDIR)/lib/net/dip
#CFLAGS += -I$(TOSDIR)/lib/net/dip/interfaces
CFLAGS += -I$(TOSDIR)/lib/net/drip
CFLAGS += -I$(TOSDIR)/lib/net/4bitle
CFLAGS += -I$(TOSDIR)/lib/net/ctp #-DNO_DEBUG
CFLAGS += -DTOSH_DATA_LENGTH=92
BUILD_EXTRA_DEPS += DACKDiss1SerialMsg.class DACKCollMsg.class
DACKDiss1SerialMsg.class: DACKDiss1SerialMsg.java
    javac DACKDiss1SerialMsg.java
DACKDiss1SerialMsg.java:
    mig java -target=$(PLATFORM) -java-classname=DACKDiss1SerialMsg \
                     $(CFLAGS) SimpleNetwork.h DACKDiss1SerialMsg -o $@
DACKCollMsg.class: DACKCollMsg.java
    javac DACKCollMsg.java
DACKCollMsg.java:
    mig java -target=$(PLATFORM) -java-classname=DACKCollMsg \
                     $(CFLAGS) SimpleNetwork.h DACKCollMsg -o $@
include $(MAKERULES)
```

A.2 SimpleNetwork.h

Listing A.2: SimpleNetwork.h

```
#ifndef SIMPLE_NETWORK_H
#define SIMPLE_NETWORK_H

#include <AM.h>

#ifndef ACTIVE
#define ACTIVE 1
#endif

#ifndef PASSIVE
#define PASSIVE 0
#endif


#ifndef DACK_SETS
#define DACK_SETS 2
#endif

#ifndef DACK_DISS1_LENGTH
#define DACK_DISS1_LENGTH 48
#endif

#ifndef DACK_COLL_LENGTH
#define DACK_COLL_LENGTH 20
#endif

#ifndef DACK_SAMPLE_SIZE
#define DACK_SAMPLE_SIZE 10 // s = number of bytes for one sample.
#endif

#ifndef DACK_COLL_NUM_SAMPLES
#define DACK_COLL_NUM_SAMPLES (DACK_COLL_LENGTH/DACK_SAMPLE_SIZE)
#endif

#ifndef DACK_STORAGE_SIZE
#define DACK_STORAGE_SIZE 50 // F = maximum number of samples.
#endif

enum {
 AM_DACKDISS1SERIALMSG = 0xD1,
 AM_DACKCOLLMSG = 0xC0,
};

typedef nx_struct DACKDiss1Data {
  nx_uint8_t data[DACK_DISS1_LENGTH];
} DACKDiss1Data;

typedef nx_struct DACKDiss1SerialMsg {
  nx_uint8_t type; // dip_msgid_t
  nx_uint16_t key;
  nx_uint16_t version;
  nx_uint8_t size;
  nx_uint8_t data[DACK_DISS1_LENGTH];
} DACKDiss1SerialMsg;


typedef nx_struct DACKSample {
  nx_uint16_t sn;  // Sequence Number
  nx_uint8_t  sid;  // Sensor ID
  nx_uint16_t reading;
  nx_uint32_t timestamp;
} DACKSample;

typedef nx_struct DACKCollMsg {
  nx_uint32_t cn;  // Collection Packet Sequence Number
  nx_uint16_t mid;  // Mote ID
  nx_uint16_t lan;  // Last Acknowledged Sequence Number
  nx_uint16_t lsn;  // Last Sequence Number Sent in a Collection report
  nx_uint8_t dsn;  // Sequence Number of the last processed dissemination packet
  //nx_uint8_t rsn;  // Sequence Number of current Report
  DACKSample  sample[DACK_COLL_NUM_SAMPLES]; // Data Samples
} DACKCollMsg;

#endif
```

A.3 simconfig.txt

Listing A.3: simconfig.txt

```
PATH_LOSS_EXPONENT = 4.7;
SHADOWING_STANDARD_DEVIATION = 3.2;
D0 = 1.0;
PL_D0 = 55.4;

NOISE_FLOOR = -105.0;
S11 = 0;
S22 = 0;
WHITE_GAUSSIAN_NOISE = 4;

TOPOLOGY = 1;
GRID_UNIT = 7.0;
NUMBER_OF_NODES = 64;
```

A.4 simulate.py

Listing A.4: test.py

```
from TOSSIM import *
from tinyos.tossim.TossimApp import *
from random import *
import sys
import random

t = Tossim([])
r = t.radio()

sf = SerialForwarder(9001)
throttle = Throttle(t, 100)

numnodes = 16

f = open("linkgain.out", "r")
lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    if s[0] == "gain":
      r.add(int(s[1]), int(s[2]), float(s[3]))

print "1->0:",r.connected(1,0)
print "2->0:",r.connected(2,0)
```

```
noise = open("linkgain.out", "r")
lines = noise.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    if s[0] == "noise":
      m = t.getNode(int(s[1]));
      for i in range(0, 500):
        m.addNoiseTraceReading( -105 + int(random.random()*float(s[3])) );

for i in range(0, numnodes):
  m = t.getNode(i);
  m.createNoiseModel();
  time = randint(t.ticksPerSecond(), numnodes * t.ticksPerSecond())
  m.bootAtTime(time)
  print "Booting ", i, " at time ", time

sf.process();
throttle.initialize();

print "Starting simulation."

t.addChannel("boot", sys.stdout)
t.addChannel("sending", sys.stdout)
t.addChannel("sendingprep", sys.stdout)
t.addChannel("resending", sys.stdout)
t.addChannel("resendingprep", sys.stdout)
#t.addChannel("reporting", sys.stdout)
#t.addChannel("sampling", sys.stdout)
#t.addChannel("problem", sys.stdout)
t.addChannel("changed", sys.stdout)
#t.addChannel("D1", sys.stdout)
#t.addChannel("D2", sys.stdout)
#t.addChannel("D3", sys.stdout)
#t.addChannel("messaging", sys.stdout)

while (1):
  #throttle.checkThrottle();
  t.runNextEvent();
  sf.process();
```

A.5 SimpleNetworkAppC.nc

Listing A.5: SimpleNetworkAppC.nc

```
/**
 * SimpleNetwork tests the dissemination of DACK packets using DIP.
 *
 * See TEP118: Dissemination, TEP 119: Collection, and TEP 123: The
 * Collection Tree Protocol for details.
 *
 * @author John-Paul Arp
 *
 * Based on EasyDissemination Tutorial
 */

#include "SimpleNetwork.h"

configuration SimpleNetworkAppC {}
implementation {
  components MainC;
```

```
SimpleNetworkC.Boot -> MainC;

components LedsC;
SimpleNetworkC.Leds -> LedsC;

components SimpleNetworkC;

components SerialActiveMessageC;
SimpleNetworkC.SerialControl -> SerialActiveMessageC;

components ActiveMessageC;
SimpleNetworkC.RadioControl -> ActiveMessageC;

// Time Components
components new TimerMilliC() as SampleTimer;
components new TimerMilliC() as ReportTimer;
components new TimerMilliC() as DelayTimer;
components new TimerMilliC() as StaggeredStartTimer;
//BusyWaitMicroC
SimpleNetworkC.SampleTimer -> SampleTimer;
SimpleNetworkC.ReportTimer -> ReportTimer;
SimpleNetworkC.DelayTimer -> DelayTimer;
SimpleNetworkC.StaggeredStartTimer -> StaggeredStartTimer;

components LocalTimeMilliC;
SimpleNetworkC.LocalTime -> LocalTimeMilliC;

// Dissemination Components
components DisseminationC;
SimpleNetworkC.DisseminationControl -> DisseminationC;

components new DisseminatorC(DACKDiss1Data, 0xDE01) as DACKDiss1C;
SimpleNetworkC.ValueD1 -> DACKDiss1C;
SimpleNetworkC.UpdateD1 -> DACKDiss1C;
components new DisseminatorC(DACKDiss1Data, 0xDE02) as DACKDiss2C;
SimpleNetworkC.ValueD2 -> DACKDiss2C;
SimpleNetworkC.UpdateD2 -> DACKDiss2C;
components new DisseminatorC(DACKDiss1Data, 0xDE03) as DACKDiss3C;
SimpleNetworkC.ValueD3 -> DACKDiss3C;
SimpleNetworkC.UpdateD3 -> DACKDiss3C;

components new DisseminatorC(DACKDiss1Data, 0xDE04) as DACKDiss1BC;
SimpleNetworkC.ValueD1B -> DACKDiss1BC;
SimpleNetworkC.UpdateD1B -> DACKDiss1BC;
components new DisseminatorC(DACKDiss1Data, 0xDE05) as DACKDiss2BC;
SimpleNetworkC.ValueD2B -> DACKDiss2BC;
SimpleNetworkC.UpdateD2B -> DACKDiss2C;
components new DisseminatorC(DACKDiss1Data, 0xDE06) as DACKDiss3BC;
SimpleNetworkC.ValueD3B -> DACKDiss3BC;
SimpleNetworkC.UpdateD3B -> DACKDiss3BC;


components new SerialAMReceiverC(AM_DACKDISS1SERIALMSG) as DissSerialReceiver;
SimpleNetworkC.DissSerialReceive -> DissSerialReceiver;

// Collection Components
components CollectionC as Collector;
SimpleNetworkC.RootControl -> Collector;
SimpleNetworkC.RoutingControl -> Collector;
SimpleNetworkC.CollRadioReceive -> Collector.Receive[AM_DACKCOLLMSG];

components new CollectionSenderC(AM_DACKCOLLMSG);
SimpleNetworkC.CollRadioSend -> CollectionSenderC.Send;

components new SerialAMSenderC(AM_DACKCOLLMSG) as CollSerialSender;
SimpleNetworkC.CollSerialSend -> CollSerialSender;

components new PoolC(message_t, 5) as UARTMessagePoolP;
```

```
  SimpleNetworkC.UARTMessagePool -> UARTMessagePoolP;

  components new QueueC(message_t*, 5) as UARTQueueP;
  SimpleNetworkC.UARTQueue -> UARTQueueP;

  // Sensing Components
  components new DemoSensorC() as Sensor;
  SimpleNetworkC.Read -> Sensor;
}
```

A.6 SimpleNetworkC.nc

A.6.1 Processing Sample Events on a mote

Each mote has one report timer, and at least one sample timer. Whenever a sample timer fires, the mote takes a reading from the appropriate sensor. The reading R is then written to storage with the sensor ID number SID, a timestamp T, and a sequence number SN. The $10 - byte$ structure of a data sample is shown in Figure 5.3. The timestamp is a $4 - byte$ value containing the number of binary milliseconds since the mote was first activated. Data Samples are written sequentially to an allocated segment of storage of size $F * 10$ bytes, where F is the maximum number of samples of size 10 that can be stored in the allocated segment. When the allocated segment of storage becomes full, the application loops back to the beginning creating a circular buffer. The sample sequence number SN also rolls over when storage becomes full. This allows the SN to be used as a pointer to the memory location of samples in storage.

The mote keeps an index in RAM to keep track of where samples are stored in flash. This index contains the following values:

- F: the number of samples that can be stored in the mote's storage (e.g. flash), and the maximum value for SN.
- SN: the sequence number of the next data sampled to be sampled.
- LSN: the sequence number of the last data sample to be sent in a collection packet in a report interval. LSN is included in all DACK collection packets (see Figure 5.4), and is the same for every packet sent in a report interval.

- ASN: the sequence number of the last consecutively acknowledged data sample. For example, if 10 samples were sent with SNs from 0 to 9, and later the samples 0 to 5, and 7 to 9 were acknowledged, then the ASN would be equal to 5.

- DSN: the sequence number of the most recently received DACK dissemination packet (DDP) that contained a local acknowledgment. Each DDP contains a sequence of acknowledgments for various Motes in a WSN, and a DSN indicating the order in which it was sent from the base station. DDPs are discused further in section A.7.2.

- B: an L-bit acknowledgment vector, stuffed into the rightmost L bits in a $\lceil L/8 \rceil$)-byte array. Bits in B correspond to the bits in between ASN and SN, such that for each bit b at position i in B, a 0 indicates $SN \leftarrow (ASN + i) \textbf{ mod } F$ is an unacknowledged sample, and a 1 indicates $SN \leftarrow (ASN + i) \textbf{ mod } F$ is an acknowledged sample. In the curent implementation a 64-bit unsigned integer is used to hold B intead of a byte array on the mote.

- L: the number of bits in B

- W: the maximum size of the acknowledgment vector, called the Acknowledgment Window. When B grows larger than W, it causes an Acknowledgment Window Overflow Error, or $WindowError$ for short. Because B is contained in a 64-bit unsigned integer, the current maximum possible size for W is 64.

Initially SN and L are set to 0, and LSN and ASN are set to -1. As samples are collected, only SN is incremented. Algorithm A.6 illustrates how Motes process sample events using the above index values. Figure A.1 illustrates an example index composition with 3 sample data readings.

Listing A.6: Process Sample Event on mote

```
/*! Trigger: Sample Event for sensor SID is signaled by the Sample
 *          Timer
 *  Input: the new sample reading 'data'
 *  Effect: updates SN, and writes the sample to storage at SN
 */
event void Sensor1Read.readDone(error_t result, uint16_t data)
{
  DACKSample ds;

  // Storage Overflow Error detected, Restart.
  if ((SN+1)%DACK_STORAGE_SIZE == ASN)
  {
    dsn = 253;
    dacklength = 0;
    SN = 0;
    ASN = -1;
    LSN = -1;
    resendcursor = 0;
  }

  ds.sn = SN;
  ds.sid = 1;
  ds.reading = data;
  ds.timestamp = call LocalTime.get();
  storage[SN] = ds;
  SN=(SN+1)%DACK_STORAGE_SIZE;
}
```

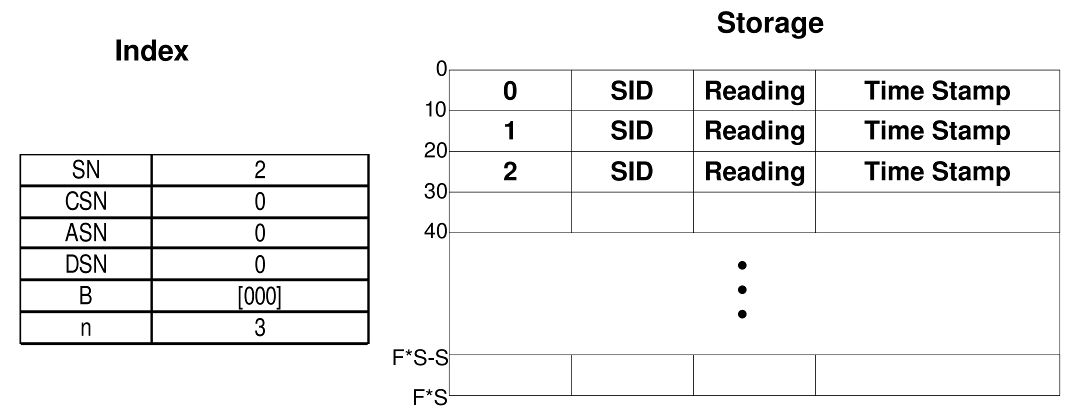


Figure A.1: An example of what the mote index and storage would look like after 3 samples have been collected.

A.6.2 Process Report Event on mote

Whenever the report timer fires, the mote sends a report of all data samples recorded since the last report interval, as well as all unacknowledged samples in the acknowledgment window. Before it sends any packets, the mote checks to see if the B vector has exceeded its maximum size W. If it has, then all previously sent unacknowledged packets are permanently lost by setting $ASN \leftarrow LSN$. Each DACK collection packet contains MID, the id of the mote transmitting the message, the ASN, the LSN, the DSN, and then a series of data samples. Figure 5.4 shows the structure of a DACK collection packet containing 2 data samples. Algorithm A.7 illustrates the logic of processing report events on Motes. Figure A.2 shows how the contents of flash, the mote index, and a DACK collection packet may looks after the first report event.

DACK collection packets are sent to the base station using a best effort collection protocol, such as the Collection Tree Protocol discussed in section 3.4.1. The number of samples that can be stored in a single collection packet varies on different hardware and different MAC protocols. In the current implementation, collection packets each have 2 data samples, but platforms with a larger maximum packet size can hold more samples.

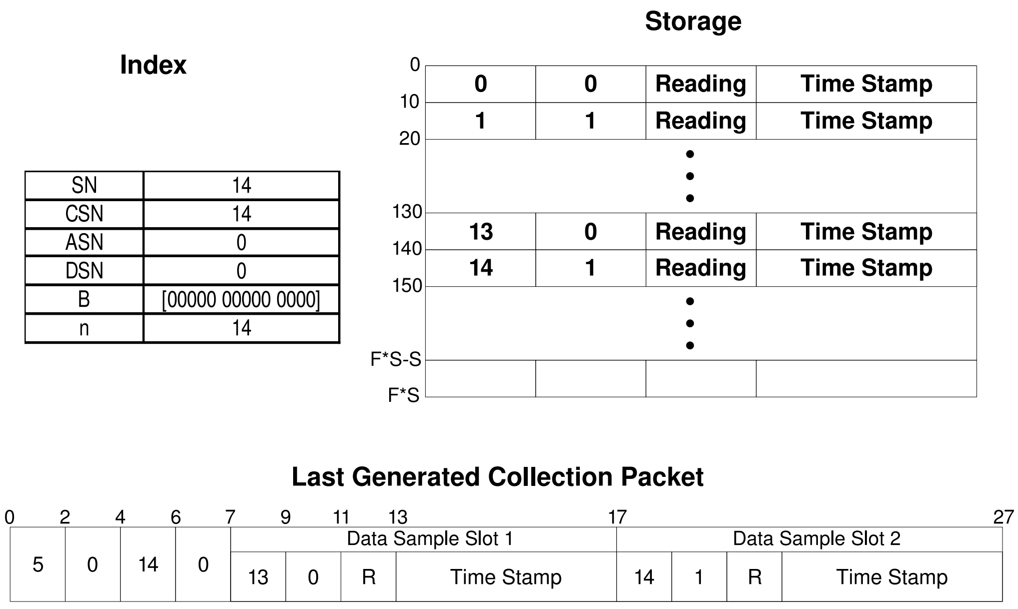


Figure A.2: An example of what the mote index and storage would look like after 14 samples have been collected, and the first Report Event completed from two sensors (one with $SID = 0$ and the other with $SID = 1$) The corresponding last generated collection packet is shown below. All values are in bytes, except the bit vector B.

Listing A.7: Report Timer Fired

```
event void ReportTimer.fired()
{ int i;
  resendcursor = (ASN + 1)%DACK_STORAGE_SIZE;
  tdackvector = dackvector;
  tdacklength = dacklength;
  if (dacklength < W)
  { resending = TRUE;
  }
  if ((dacklength >= W) && (dacklength < DACK_STORAGE_SIZE - 5))
  { dsn = 255;  // Signal that we are going to stop trying to resend old samples
    resending = FALSE;
  }
  post resendCollectionPacket();
}
```

Listing A.8: Resend Lost Samples

```
/*! Contains the logic for resending unacknowledged packets < LSN
 */
task void resendCollectionPacket()
{
  DACKCollMsg *dcm;
  DACKSample *s1;
  DACKSample *s2;
  int i = 0;
  int numsent = 0;
  if ((!collsendbusy) && (tdacklength > 0))
  { dcm = (DACKCollMsg *)call CollRadioSend.getPayload(&collsendbuf, sizeof(DACKCollMsg));
    s1 = &(dcm->sample[0]);
    s2 = &(dcm->sample[1]);
    if (dcm == NULL)
    { fatal_problem();
      return;
    }
    for (i = 1; i <= DACK_COLL_NUM_SAMPLES; i++)
    { while ((tdackvector | 0xFFFFFFFFFFFFFFFE) == 0xFFFFFFFFFFFFFFFF)
      { tdacklength--;
        tdackvector=tdackvector>>1;
        resendcursor=(resendcursor+1)%DACK_STORAGE_SIZE;
      }
      if (tdacklength > 0)
      { if (storage[SN].timestamp == 0) storage[SN].timestamp = 1;
        memcpy(&(dcm->sample[i-1]), &storage[resendcursor],  sizeof(DACKSample));
        tdacklength--;
        tdackvector=tdackvector>>1;
        resendcursor=(resendcursor+1)%DACK_STORAGE_SIZE;
      }
      else
      {  memcpy(&(dcm->sample[i-1]), "\0\0\0\0\0\0\0\0\0",  sizeof(DACKSample));
      }
    }
    dcm->mid = TOS_NODE_ID;
    dcm->lan = ASN;
    if (SN == 0)
    {  dcm->lsn = DACK_STORAGE_SIZE - 1;
    }
    else
    {  dcm->lsn = SN-1;
    }
    dcm->dsn = dsn;
    if (call CollRadioSend.send(&collsendbuf, sizeof(localcollmsg)) == SUCCESS)
    { collsendbusy = TRUE;
    }
    else
    {  report_problem();
    }
  }
  else
  {  post sendCollectionPacket();
  }
}
```

Listing A.9: Send New Samples

```
/*! Contains the logic for resending unacknowledged packets > LSN
 */
task void sendCollectionPacket()
{ DACKCollMsg *dcm;
  int i = 0;
  int numsent = 0;
  if ((!collsendbusy) && (SN != (LSN + 1)%DACK_STORAGE_SIZE ))
  { dcm = (DACKCollMsg *)call CollRadioSend.getPayload(&collsendbuf, sizeof(DACKCollMsg));
    if (dcm == NULL)
    { fatal_problem();
      return;
    }
    for (i = 1; i <= DACK_COLL_NUM_SAMPLES; i++)
    { if ( ((LSN+i)%DACK_STORAGE_SIZE) != SN)
      { memcpy(&(dcm->sample[i-1]), &storage[(LSN+i)%DACK_STORAGE_SIZE],
                sizeof(DACKSample));
        numsent++;
      }
      else
      {  memcpy(&(dcm->sample[i-1]), "\0\0\0\0\0\0\0\0\0",  sizeof(DACKSample));
      }
    }
    dcm->mid = TOS_NODE_ID;
    dcm->lan = ASN;
    if (SN == 0)
    {  dcm->lsn = DACK_STORAGE_SIZE - 1;
    }
    else
    {  dcm->lsn = SN-1;
    }
    dcm->dsn = dsn;
    if (call CollRadioSend.send(&collsendbuf, sizeof(localcollmsg)) == SUCCESS)
    { LSN = (LSN + numsent)%DACK_STORAGE_SIZE;
      collsendbusy = TRUE;
    }
    else
    {  report_problem();
    }
  }
}
```

Listing A.10: sendDone Event

```
event void CollRadioSend.sendDone(message_t* msg, error_t error)
{ post task sendmore();
}
```

Listing A.11: sendmore

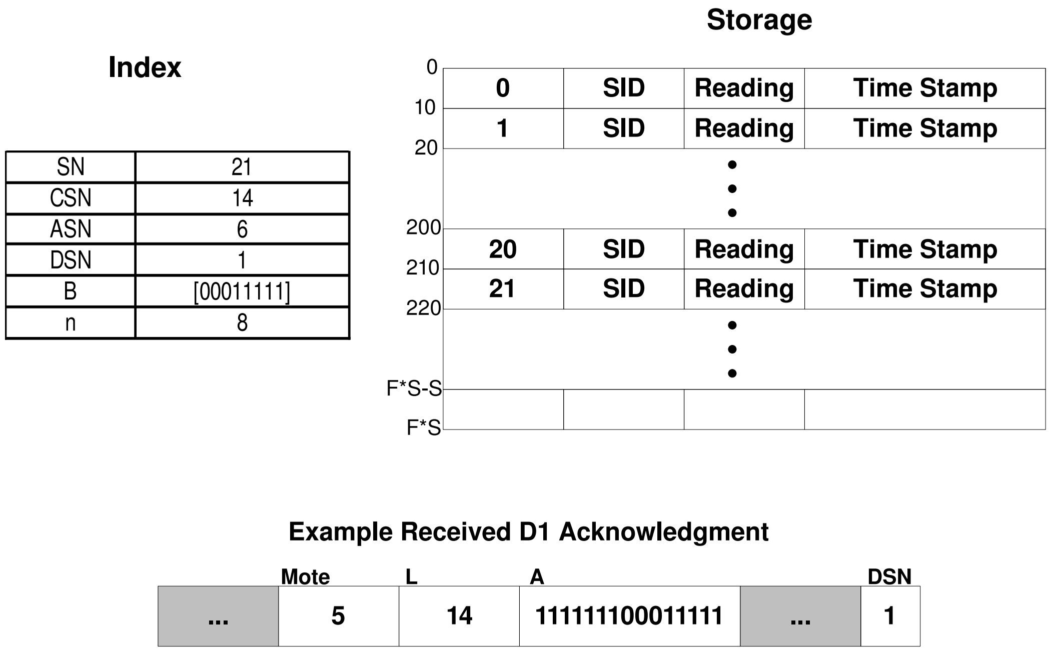
```
task void sendmore()
{ collsendbusy = FALSE;
  if (tdacklength <= 0) resending = FALSE;

  if (resending == FALSE)
  { post sendCollectionPacket();
  }
  else
  { post resendCollectionPacket();
  }
}
```

A.6.3 Processing DACK Dissemination Packets on the mote

As Motes receive dissemination packets, they have to parse each packet to look for their MID. If the mote's MID is not present, the packet is ignored. For each of the three types of dissemination packets, there is a corresponding packet processing function. Code listings A.12, A.13, A.14 illustrates how the dissemination packets are processed for $D1$, $D2$, and $D3$ type packets.

A.6.3.1 Processing D1 type packets

If a D1 packet contains the mote's MID, the next byte is checked for L. If $L = 0$ then, ASN is set to LSN. If $L > 0$, then A is parsed. If the first bit in A is a zero, then ASN will remain unchanged. Otherwise, ASN is updated to reflect the last consecutive 1 in A, to reflect the new last consecutively acknowledged bit. The mote then updates acknowledgment vector B to start from $ASN + 1$ **mod** F, and fills in each bit entry with the corresponding bits found in A. At the next report interval, the mote will then resend



SN	21
CSN	14
ASN	6
DSN	1
B	[00011111]
n	8

Figure A.3: An example of what the mote index and storage would look like after 21 samples have been logged, 14 samples have been sent, and a $D1$ type disseminated acknowledgment is received. In this example SNs 8, 9, and 10 were not received by the base station.

all of the unacknowledged data samples, as well as any new samples that are collected. Algorithm A.12 describes the logic of processing D1 type packets. Figure A.3 illustrates how the flash and mote index might look on mote 5 after a disseminated acknowledgment is received with 11 acknowledged samples and 3 unacknowledged samples.

Listing A.12: Process D1 Packet on the mote

```
/*! Trigger: D1 Packet Recieved
 *  Input: D1 Packet
 *  Effect: If D3 packet contains TOS_NODE_ID then
 *          the index is updated with the incoming ack vector
 */
task void processD1()
{ int ds = 0; int i=0; int j=0; int ac=0;
  uint16_t cmid = 0; uint8_t length = 0; uint8_t bytelength = 0;
  if (resending) return;
  while (i+2 < sizeof(d1data[ds]))
  { cmid = d1data[ds][i];
    cmid = (cmid<<8) | d1data[ds][i+1];
    if (cmid == 0xFFFF) break;
    length = d1data[ds][i+2];
    bytelength = ceil(((float)length+1.0)/8.0);
    if (length == 0) break;
    i = i + 3;
    if (bytelength+i < sizeof(d1data[ds]))
    { if (cmid == TOS_NODE_ID)
      { dsn = d1data[ds][sizeof(d1data[ds])-1];
        dackvector = 0;
        dacklength = length;
        resending = TRUE;
        for (j = 0; j < bytelength; j++)
        { dackvector = (dackvector << 8) | d1data[ds][i+j];
        }
        while ((dackvector | 0xFFFFFFFFFFFFFFFE) == 0xFFFFFFFFFFFFFFFF)
        { ASN=(ASN+1)%DACK_STORAGE_SIZE;
          dackvector = dackvector >> 1;
          dacklength--;
        }
        break;
      }
      i = i + bytelength;
    } else { break; }
  }
  return;
}
```

A.6.3.2 Processing D2 type packets

If a $D2$ type packet contains a reference to the receiving mote's MID, then all DACK data samples have been acknowledged. The mote then clears the B vector, and ASN is set to LSN. Algorithm A.13 describes the logic of processing $D2$ type packets. Figure A.4 illustrates how the mote's index values and storage might look on mote 5 after a disseminated acknowledgment is received inside a $D2$ type packet.

Listing A.13: Process D2 Packet on the mote

```
/*! Trigger: D2 Packet Recieved
 *  Input: D2 Packet
 *  Effect: If D2 packet contains TOS_NODE_ID, then all data samples
 *          up to LSN is acknowledged
 */
task void processD2()
{ int acked = FALSE; int ds = 0; int i;
  uint8_t inchar; int16_t cmid = -1; int16_t pmid = -1;
  for (i = 0; i < sizeof(d2data[ds]); i=i+2)
  { inchar = d2data[ds][i];
    if (inchar == '-')
    { if (pmid != -1)
      { i = i + 1;
        inchar = d2data[ds][i];
        cmid = inchar;
        cmid = cmid << 8;
        cmid = cmid | d2data[ds][i+1];
        if ((cmid >= TOS_NODE_ID) && (pmid <= TOS_NODE_ID))
        { dsn = d2data[ds][sizeof(d2data[ds])-1];
          ASN = LSN;
          dackvector = 0;
          dacklength = 0;
          return;
        }
      } else { return; }
    }
    else
    { cmid = inchar;
      cmid = cmid << 8;
      cmid = cmid | d2data[ds][i+1];
      pmid = cmid;
      if (cmid == TOS_NODE_ID)
      { dsn = d2data[ds][sizeof(d2data[ds])-1];
        ASN = LSN;
        dackvector = 0;
        dacklength = 0;
        return;
      }
    }
  }
  return;
}
```

A.6.3.3 Processing D3 type packets

An acknowledgment in a $D3$ type will tell the mote to update its ASN, B, and n, directly. Algorithm A.14 describes the logic of processing $D3$ type packets. Figure A.5 illustrates how the flash and mote index might look on mote 5 in the scenario that a $D3$ type packet is received to correct an $ASNmismatch$ cause by the $D2$ packet in Figure A.4.

Listing A.14: Process D3 Packet on the mote

```
/*! Trigger: D3 Packet Recieved
 *  Input: D3 Packet
 *  Effect: If D3 packet contains TOS_NODE_ID
 *          index is updated with new ASN and ack vector
 */
task void processD3()
{ int ds = 0; int i=0; int j=0; int ac=0;
  uint16_t cmid = 0;   uint16_t fixedASN = 0;
  uint8_t length = 0;  uint8_t bytelength = 0;
  if (resending) return;
  while (i+2 < sizeof(d3data[ds]))
  { cmid = d3data[ds][i];
    cmid = (cmid<<8) | d3data[ds][i+1];
    if (cmid == 0xFFFF) break;
    length = d3data[ds][i+2];
    bytelength = ceil(((float)length+1.0)/8.0);
    if (length == 0) break;
    fixedASN = d3data[ds][i+3];
    fixedASN = (fixedASN<<8) | d3data[ds][i+4];
    i = i + 5;
    if (bytelength+i < sizeof(d3data[ds]))
    { if (cmid == TOS_NODE_ID)
      { dsn = d3data[ds][sizeof(d3data[ds])-1];
        ASN = fixedASN; resending = TRUE;
        dackvector = 0; dacklength = length;
        for (j = 0; j < bytelength; j++)
        { dackvector = (dackvector << 8) | d3data[ds][i+j];
        }
        while ((dackvector | 0xFFFFFFFFFFFFFFFE) == 0xFFFFFFFFFFFFFFFF)
        { ASN=(ASN+1)%DACK_STORAGE_SIZE;
          dackvector = dackvector >> 1; dacklength--;
        }
        break;
      }
      i = i + bytelength;
    } else { break; }
  }
  return;
}
```

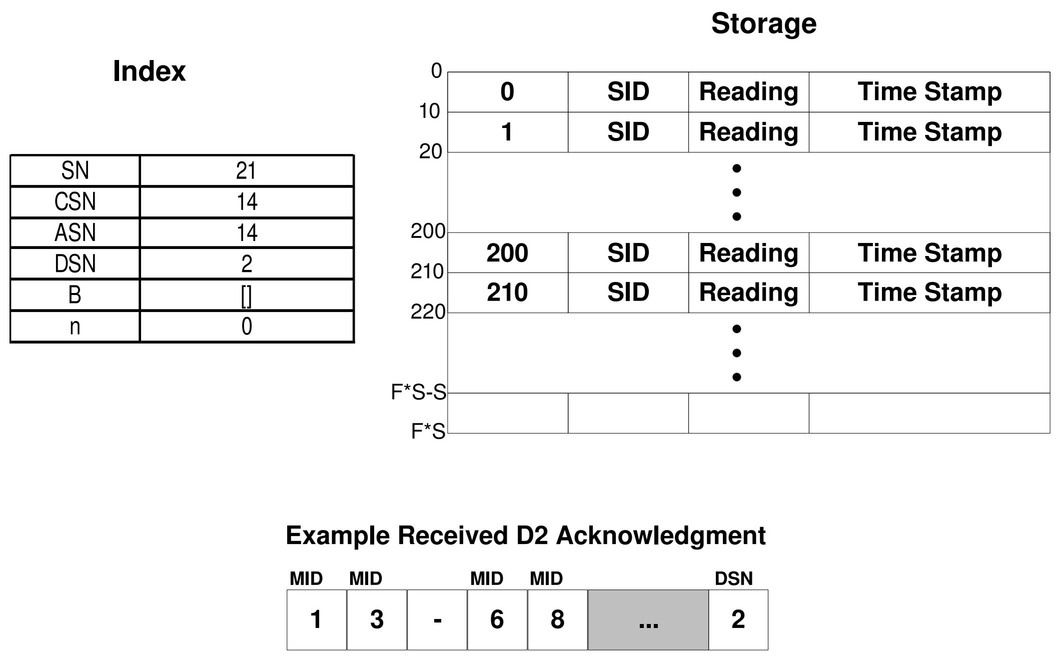


Figure A.4: An example of what the mote index and storage would look likeon mote 5 after 21 samples have been logged, 14 samples have been sent, and a $D2$ type disseminated acknowledgment is received. The dissemination packet contains full acknowledgments for

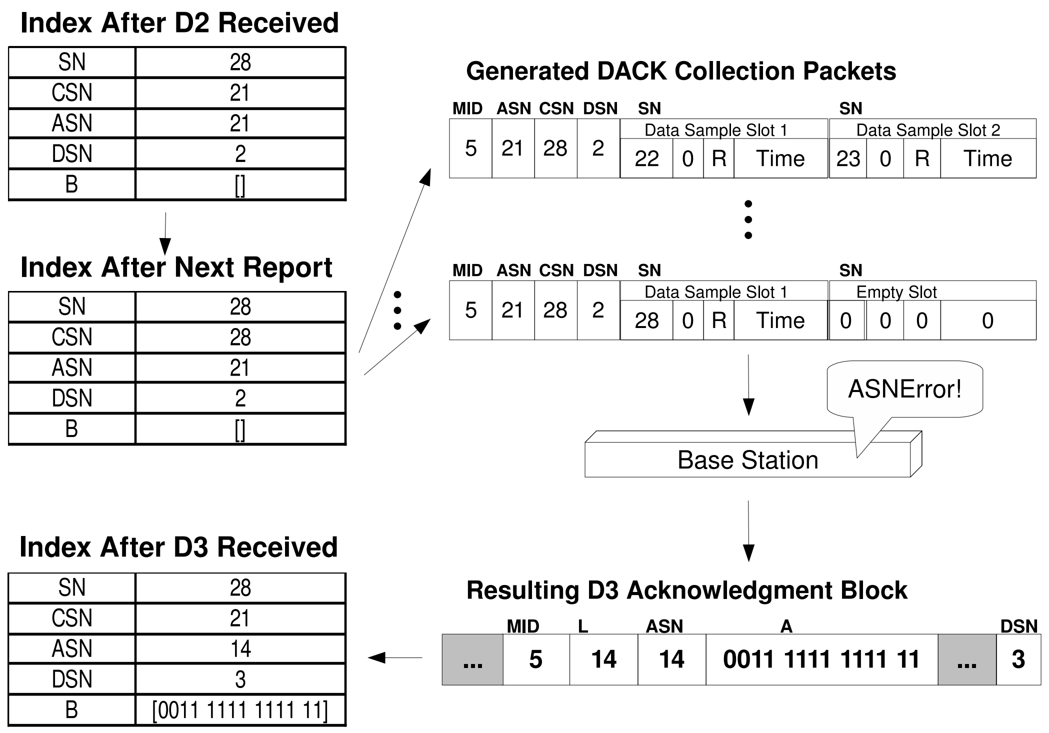


Figure A.5: This Figure show a scenario in which a $D3$ packet is required. Suppose the $D2$ acknowledgment sent in Figure A.4 was received by mote 5 after it had sent data samples 15 to 21, but before those samples were recieved by the base station. Suppose further that the packet containing SN 15 and 16 were lost in transmission. The mote would assume that the base station was acknowledging those samples as well, and set its ASN to 21, not knowing that SN 15 and 16 were lost. The base station will detect the problem when the mote sends the next report, because it will expect the incoming collection packets to have an $ASN = 14$, but discover that they have an $ASN = 21$ instead. The base station identifies this as a $ASNmismatch$, and responds by embedding an acknowledgment to mote 5 inside a $D3$ type packet.

A.7 SimpleNetworkBS.java

A.7.1 Processing Samples on the Base Station

As the base station receives collection packets, it stores the samples in a database, and keeps a record of acknowledgment information for each mote. The base station creates a $MoteIndex$ object for each unique mote to keep track of incoming data samples. $MoteIndex$ objects are stored in a hashtable called the $MoteTable$, using MID as the key. Each $MoteIndex$ object contains the following data items:

- MID: The mote ID of the mote this MoteIndex object is keeping track of.
- ASN: MID's last consecutive acknowledged data sample SN.
- LSN: The last data sample SN sent in a report interval from MID.
- DSN: The sequence number of the last dissemination packet sent containing an acknowledgment for MID.
- $numsent$: The total number of unique data samples sent by MID.
- $numdropped$: The total number of unique data samples from MID that the DACK protocol detects as missing prior to generating any acknowledgements.
- $numrecovered$: The total number of data samples on MID that were recovered by acknowledgments disseminated in $D1$ or $D3$ type packets. This is discussed further in section 4.3.
- $numlost$: The total number of data samples on MID that were lost due to a Acknowledgment Window Overflow Error.
- $ASNmismatch$: Set to $TRUE$ when the $MoteIndex$ object and incoming collection packets from the corresponding mote disagree on the value of ASN. This is discussed further in sections A.7.2 and A.6.3.3.
- $WindowError$: Set to $TRUE$ for MID when the DSN value inside an incoming collection packet from MID is set to 255. This indicates that the mote has detected an Acknowledgment Window Overflow Error, and is alerting the base station.
- $ackwaiting$: A boolean value indicating whether or not a base station is waiting to receive an acknowledgment of a disseminated acknowlegment. It is set to $TRUE$ before an acknowledgment with a unique DSN is disseminated to MID, and set to $FALSE$ when a DACK collection packet is received by the base station containing the same DSN. This is used so that the base station will not send any new acknowledgments, until the previous acknowledgments are themselves acknowledged.
- $ackvector$: An integer vector of size F, in which a value of 1 at position i indicates the a data sample with $SN = i$ has been received; 0 indicates the data sample has not been received. This vector only keeps track of SNs between $(ASN+1) \bmod F$ and LSN. Elements in the vector outside of $(ASN + 1) \bmod F$ and LSN are set to 0.
- $ackcountvector$: A vector of size F, in which each element is initially set to 0, and each element at position i is incremented whenever a data sample with $SN = i$ is received.
- $acktimevector$: A vector of size F, in which each position i contains the latest timestamp from a data sample with $SN = i$

- $lostackvector$: a vector of size F, that acts as the inverse of the $ackvector$.
- A: an L-bit acknowledgment vector, stuffed into the rightmost L bits in a $\lceil L/8 \rceil$-byte array. Bits in A correspond to the bits in between ASN and LSN, such that for each bit a at position i in A, a 0 indicates $SN = (ASN + i) \textbf{ mod } F$ is an unacknowledged sample, and a 1 indicates $SN = (ASN + i) \textbf{ mod } F$ is an acknowledged sample. A is a compressed format for the $ackvector$ to be packed into communication packets. A is the mirror of the B vector on the mote.
- L: the size of the vector A.

In addition to the above data items, the MoteIndex also contains the following methods:

- $getAckInBytes()$: generates and returns A.
- $cleanAckVector()$: sets all bits in the $ackvector$ outside ASN and LSN to 0.
- $countLostSamples()$: updates the $lostackvector$, and increments $numlost$ for each known missing data sample.
- $giveup()$: this method tells the base station to give up on requesting a retransmission for any data samples before a given SN. This method is called when the base station recieves a $WindowError$ flag in the DSN value of incoming collection packets. This method increments $numlost$ for every data sample that will consequently never be acknowledged.
- $allclear()$: this method returns $TRUE$ if and only if there are no unacknowledged data samples in the $ackvector$.
- $checkASNmismatch()$: this method returns true if the ASN in the incoming DACK-CollectionPacket is in disagreement with the $MoteIndex$ object's value for ASN.
- $findASN()$: this searches the $ackcountvector$ to find the correct value for ASN.

The DACK protocol uses three types of packets, with different levels of detail. By using these three types of packets, the base station can reduce the number of bytes disseminated per mote, by packaging it in the most appropriate dissemination packet. The three types of dissemination packets are:

- The $D1$ type packet (shown in Figure 5.6) contains a sequence of (MID, L, A), for Motes for which some data samples were not received at the base station.

- The $D2$ type packet (shown in Figure 5.7) contains a sequence of (MID), for Motes that are not missing any data samples. MID for $D2$ are sorted by MID in ascending order. If the MID values are contiguous, then only the first and last contiguous MID are put into the packet with a '-' charachtor placed in between. $D2$ packets are the most compressed dissemination packet type.

- The $D3$ type packet (shown in Figure 5.8), contains a sequence of (MID, ASN, L, A), for Motes for which some data samples were not received at the base station, and for wchich an $ASNmismatch$ was detected.

It is possible to build a functioning dissemination-based acknowledgment protocol using only $D3$ packets, but it would be less efficient. $D1$ packets, and $D2$ packets by themselves are not sufficient, because the transmission delay for both the collection and dissemination protocols can cause DACK messages to be received out of sync. For example, a mote may send a report just after the base station has disseminated a $D2$ type acknowledgment for a previous report, but before that acknowledgment packet reaches the mote. When the acknowledgment does reach the mote, the mote will believe it was for both reports, and not just the previous one. One solution would be to include a report IDs inside collection and dissemination packets. This would require additional byte for every mote you want to acknowledge in a dissemination packet, severely reducing the compression of the $D2$ type packet. $D3$ packets present a more economical solution. After the mote makes another report, the base station will be able to detect that the ASN on the incoming collection packet does not match the local value for the mote's ASN. This is called an "$ASNmismatch$". When the base station detects an $ASNmismatch$, it will send a detailed $D3$ type acknowledgment containing the corrected ASN, L, and A vector required to properly acknowledge the mote.

As shown in Figures 5.6, 5.7, and 5.8, each of the three types of dissemination packets contain the 8-bit DSN as the last byte of the packet. The dissemination protocol limits the number of acknowledgments that can fit in a packet.

Code listing A.15 illustrates processing incoming collection packets on the base station. The base station first looks up the MID contained in the incoming DACK Collection packet DCP in the $MoteTable$ hashtable. If the $MoteTable$ does not contain a $MoteIndex$ object for MID, then a new $MoteIndex$ object is created and put into the $MoteTable$ using the $MoteTable$'s $put()$ method. Otherwise, the $MoteIndex$ is retrieved from the $MoteTable$ using the $get()$ method. The base station then checks the DSN to see if the mote is replying to the latest ACK disseminated to the mote. If the $DSN = 255$, then a Acknowledgment Window Overflow Error has been detected and the $WindowError$ flag is set to $TRUE$. Next, the base station checks to see if the ASN contained in the collection packet matches the value for ASN in the $MoteIndex$. If not, an $ASNmismatch$ is detected, then the $ASNmismatch$ flag is set to $TRUE$. The base station then checks to see if the incoming data samples are known to be missing samples, and if so, it increments the $numrecovered$ value. Finally, the reading is stored in a database, and the local $ackvector$s are updated.

Listing A.15: Process Collection Packet on the base station

```
// Trigger:Collection Packet Recieved Event
// Input: Collection Packet
// Effect: Samples are processed and the MoteIndex is updated
public void messageReceived(int to, Message message)
{ DACKCollMsg dcm = (DACKCollMsg)message;
  MoteIndex mi;
  if( motetable.containsKey( dcm.get_mid() ) )
  { mi = motetable.get( dcm.get_mid() );
    for (int i = 0; i < dcm.numDataSamples(); i++)
    { if (dcm.get_timestamp(i) > mi.RSNTimestamp)
      { mi.RSNTimestamp = dcm.get_timestamp(i);
        mi.RSN = dcm.get_sn(i); mi.LSN = dcm.get_lsn();
    } }
    if  ( mi.DSN == dcm.get_dsn() )
    { mi.ackwaiting = 0;
      if (!mi.ASNerror)
      { mi.ASN = dcm.get_lan();
        if (mi.ASN == 65535) mi.ASN = -1;
      }
      if (mi.DSNtype == 0xD3)
      { mi.ASNerror = false;
  } } } else
  { int ASN = dcm.get_lan(); int LSN = dcm.get_lsn();
    if (ASN == 65535) ASN = -1; if (LSN == 65535) LSN = -1;
    mi = new MoteIndex(dcm.get_mid(), LSN, ASN);
    for (int i = 0; i < dcm.numDataSamples(); i++)
    { if (dcm.get_timestamp(i) > mi.RSNTimestamp)
      { mi.RSNTimestamp = dcm.get_timestamp(i);
        mi.RSN = dcm.get_sn(i);
  } } }
  if ((dcm.get_dsn() == 253) && (mi.DSNmote != 253))
  { int ASN = dcm.get_lan(); int LSN = dcm.get_lsn();
    if (ASN == 65535) ASN = -1; if (LSN == 65535) LSN = -1;
    mi = new MoteIndex(dcm.get_mid(), LSN, ASN);
    for (int i = 0; i < dcm.numDataSamples(); i++)
    { if (dcm.get_timestamp(i) > mi.RSNTimestamp)
      { mi.RSNTimestamp = dcm.get_timestamp(i);
        mi.RSN = dcm.get_sn(i);
  } } }
  mi.DSNmote = dcm.get_dsn();
  for (int i = 0; i < dcm.numDataSamples(); i++)
  { if(   !((dcm.get_sn(i)==0)&&(dcm.get_sid(i)==0)&&
          (dcm.get_reading(i)==0)&&(dcm.get_timestamp(i)==0))&&
          ( mi.isInRange( dcm.get_sn(i) ) ) )
    { if (mi.lostackvector[dcm.get_sn(i)] == 1)
      { mi.lostackvector[dcm.get_sn(i)] = 0;
        mi.numfixed++;
      }
      mi.ackvector[dcm.get_sn(i)] = 1;
      if (mi.acktimevector[dcm.get_sn(i)] != dcm.get_timestamp(i))
      { mi.acktimevector[dcm.get_sn(i)] = dcm.get_timestamp(i);
        (mi.ackcountvector[dcm.get_sn(i)])++;
  } } }
  motetable.put(mi.id, mi);
}
```

A.7.2 Disseminating Acknowledgments from the base station

To construct the dissemination packet, the base station uses the information contained in each mote's $MoteIndex$ object.

The current implementation used for simulation and experiment uses the DIP protocol (see section 3.4.2) with 48 byte dissemination packets. In DIP, data values d that are to updated via dissemination each have a unique key k and a version number v. A message is disseminated by having one mote change the value for d and then incriment d. The message disseminates as each mote in the WSN overhears a new version v for data item d with key k is available. Because we might want to disseminate a $D1$ packet to acknowledge one subset of Motes, and a $D2$ or $D3$ packet to acknowledge another subset of Motes, we will need to be able to disseminate $D1$, $D2$, and $D3$ packets at the same time. To do this via DIP, each dissemination type is given its own DIP key. To disseminate a $D1$ type packet through the network, the base station sends the dissemination message to the gateway with the appropriate value for k. When the gateway receives the message, it changes its internal value for $D1$, and increments the version number, initializing the dissemination process. Withen a matter of seconds, the dissemination process relays the message to the entire network

Listing A.16: Checking for ASN-Mismatch and Ack-Window Overflow

```
Enumeration ea = motetable.keys();
while( ea.hasMoreElements() )
{ Integer moteIDObj = (Integer)ea.nextElement();
  int moteID = moteIDObj.intValue();
  MoteIndex mi = (MoteIndex)motetable.get(moteIDObj);
  mi.ackwaiting--;
  if (mi.checkASNError())
  { mi.ASNerror = true;
    int newASN = mi.findASN();
    mi.ASN = newASN;
  }
  if (mi.checkWindowError() == true)
  { mi.giveup();
    mi.ackwaiting = 0;
    mi.ASNerror = true;
  }
  mi.cleanAckVector();
  mi.recordLostSamples();
}
```

Listing A.17: Preparing D1 type packets

```
Enumeration e = motetable.keys();
while( e.hasMoreElements() && ((bytecursorD1[j] + 3) < dataD1[j].length) )
{ Integer moteIDObj = (Integer)e.nextElement();
  int moteID = moteIDObj.intValue();
  MoteIndex mi = (MoteIndex)motetable.get(moteIDObj);
  if (!mi.allclear() && !mi.ASNerror && (mi.ackwaiting<=0))
  { byte[] ackbytes = mi.getAckByteArray();
    if (bytecursorD1[j] + ackbytes.length + 4 >= dataD1[j].length) break;
    mi.ackwaiting = 2;
    mi.DSN = D1Seqno[j];
    mi.DSNtype = 0xD1;
    dataD1[j][bytecursorD1[j]++] = (byte)(moteID >> 8 & 0xff);
    dataD1[j][bytecursorD1[j]++] = (byte)(moteID & 0xff);
    dataD1[j][bytecursorD1[j]++] = (byte)(mi.getAckLength());
    for (int i = ackbytes.length-1; i >= 0; i--)
    {  dataD1[j][bytecursorD1[j]++] = ackbytes[i];
} } }
```

Listing A.18: Preparing D2 type packets

```
// Trigger: Dissemination Timer
// Input: MoteTable
// Effect: Prepare D2 type packets for sending
Vector atv = new Vector(motetable.keySet());
Collections.sort(atv); int mti = 0;
while (mti < atv.size())
{ if (bytecursorD2[j] < dataD2[j].length)
  { Integer moteIDObj = (Integer)atv.get(mti);
    int moteID = moteIDObj.intValue();
    MoteIndex mi = (MoteIndex)motetable.get(moteIDObj);
    if (mi.allclear() && !mi.ASNerror && (mi.ackwaiting<=0))
    { if (bytecursorD2[j] + 3 >= dataD2[j].length) break;
      mi.ackwaiting = 2; mi.DSN = D2Seqno[j]; mi.DSNtype = 0xD2;
      dataD2[j][bytecursorD2[j]++] = (byte)(moteID >> 8 & 0xff);
      dataD2[j][bytecursorD2[j]++] = (byte)(moteID & 0xff);
      if (bytecursorD2[j]+4 >= dataD2[j].length) break;
      if (mti+2 < atv.size())
      { Integer moteIDObj2 = (Integer)atv.get(mti+1);
        int moteID2 = moteIDObj2.intValue();
        MoteIndex mi2 = (MoteIndex)motetable.get(moteIDObj2);
        Integer moteIDObj3 = (Integer)atv.get(mti+2);
        int moteID3 = moteIDObj3.intValue();
        MoteIndex mi3 = (MoteIndex)motetable.get(moteIDObj3);
        if (mi2.allclear() && !mi2.ASNerror && (mi2.ackwaiting<=0) &&
            mi3.allclear() && !mi3.ASNerror && (mi3.ackwaiting<=0))
        { mi2.ackwaiting = 2; mi2.DSN = D2Seqno[j]; mi2.DSNtype = 0xD2;
          mi3.ackwaiting = 2; mi3.DSN = D2Seqno[j]; mi3.DSNtype = 0xD2;
          int l = 2; boolean donext = true;
          while (donext)
          { l = l + 1;
            if (mti+l<atv.size())
            { moteIDObj3 = (Integer)atv.get(mti+l);
              mi3 = (MoteIndex)motetable.get(moteIDObj3);
              if (mi3.allclear() && !mi3.ASNerror && (mi3.ackwaiting<=0))
              { moteID3 = moteIDObj3.intValue();
                mi3.ackwaiting = 2; mi3.DSN = D2Seqno[j]; mi3.DSNtype = 0xD2;
              } else {  donext = false; }
            } else {  donext = false; }
          }
          dataD2[j][bytecursorD2[j]++] = '-';
          dataD2[j][bytecursorD2[j]++] = (byte)(moteID3 >> 8 & 0xff);
          dataD2[j][bytecursorD2[j]++] = (byte)(moteID3 & 0xff);
          mti=mti+l;
      } }
      else if (mti+1 < atv.size())
      { Integer moteIDObj2 = (Integer)atv.get(mti+1);
        int moteID2 = moteIDObj2.intValue();
        MoteIndex mi2 = (MoteIndex)motetable.get(moteIDObj2);
        if (mi2.allclear() && !mi2.ASNerror && (mi2.ackwaiting<=0))
        { mi2.ackwaiting = 2; mi2.DSN = D2Seqno[j]; mi2.DSNtype = 0xD2;
          dataD2[j][bytecursorD2[j]++] = (byte)(moteID2 >> 8 & 0xff);
          dataD2[j][bytecursorD2[j]++] = (byte)(moteID2 & 0xff);
  } } } }
  mti++;
}
```

Listing A.19: Preparing D3 type packets

```
Enumeration e3 = motetable.keys();
while( e3.hasMoreElements() && (bytecursorD3[j] < dataD3[j].length) )
{ Integer moteIDObj = (Integer)e3.nextElement();
  int moteID = moteIDObj.intValue();
  MoteIndex mi = (MoteIndex)motetable.get(moteIDObj);
  if (mi.ASNerror && (mi.ackwaiting<=0))
  { byte[] ackbytes = mi.getAckByteArray();
```

```
    if (bytecursorD3[j] + ackbytes.length + 6 >= dataD3[j].length) break;
    mi.ackwaiting = 2;
    mi.DSN = D3Seqno[j];
    mi.DSNtype = 0xD3;
    dataD3[j][bytecursorD3[j]++] = (byte)(moteID >> 8 & 0xff);
    dataD3[j][bytecursorD3[j]++] = (byte)(moteID & 0xff);
    dataD3[j][bytecursorD3[j]++] = (byte)(mi.getAckLength());
    dataD3[j][bytecursorD3[j]++] = (byte)(mi.ASN >> 8 & 0xff );
    dataD3[j][bytecursorD3[j]++] = (byte)(mi.ASN & 0xff );
    for (int i = ackbytes.length-1; i >= 0; i--)
    {  dataD3[j][bytecursorD3[j]++] = ackbytes[i];
} } }
```

Listing A.20: Disseminating Messages

```
try
{ for (int j = 0; j < dacksets; j++)
  { dataD1[j][dataD1[j].length-1] = D1Seqno[j];
    dataD2[j][dataD2[j].length-1] = D2Seqno[j];
    dataD3[j][dataD3[j].length-1] = D3Seqno[j];
    int dkey = -1;
    if (dataD1[j][0] != -1)
    { d1sm[j].set_data(dataD1[j]);
      dkey = 0xDE01 +(j*3);
      d1sm[j].set_key(dkey);
      moteIF.send(0, d1sm[j]);
      Thread.sleep(350);
    }
    if (dataD2[j][0] != -1)
    { d2sm[j].set_data(dataD2[j]);
      dkey = 0xDE02 +(j*3);
      d2sm[j].set_key(dkey);
      moteIF.send(0, d2sm[j]);
      Thread.sleep(350);
    }
    if (dataD3[j][0] != -1)
    { d3sm[j].set_data(dataD3[j]);
      dkey = 0xDE03 +(j*3);
      d3sm[j].set_key(dkey);
      moteIF.send(0, d3sm[j]);
      Thread.sleep(350);
} } } catch (Exception exception) { exception.printStackTrace(); }
```

Listing A.21: Calculate Network Totals

```
while( e4.hasMoreElements())
{
  Integer moteIDObj = (Integer)e4.nextElement();
  int moteID = moteIDObj.intValue();
  MoteIndex mi = (MoteIndex)motetable.get(moteIDObj);

  mi.samplessent = mi.getSamplesSent();
  mi.check1 = mi.samplessent - (mi.samplesreceived - mi.samplesrecovered + mi.samplesdropped);
  mi.check2 = mi.samplesdropped - (mi.samplesrecovered + mi.sampleslost + mi.getOutstanding());
  if ((mi.check1) != 0 || (mi.check2 != 0))
  {  mi.msg = "Error";
  }
  else
  {  mi.msg = "";
  }
  System.out.println("Mote "+mi.id+"\t\t\t"
      +mi.samplessent+"\t"
      +mi.samplesreceived+"\t\t"
      +((int)(((double)mi.samplesreceived/Math.max((double)mi.samplessent,1.0))*100.0))+"%"+"\t|\t"
      +mi.samplesdropped+"\t"
      +mi.samplesrecovered+"\t\t"
      +((int)(((double)mi.samplesrecovered/Math.max((double)mi.samplesdropped,1.0))*100))+"% \t"
      +mi.sampleslost+"\t"
      +mi.getOutstanding()+"\t|\t"
      +mi.totDisseminations+"\t"
      +mi.totD1+"\t"
      +mi.totD2+"\t"
      +mi.totD3+"\t|\t"
      +(mi.CN-mi.initCN)+"\t"
      +mi.CN+"\t|\t"
      +mi.overflows+"\t"
      +mi.resends+"\t|\t"
      +mi.check1+"\t"
      +mi.check2+"\t"
      +mi.msg);

  totalsent = totalsent + mi.samplessent;
  totalreceived = totalreceived + mi.samplesreceived;
  totaldropped = totaldropped + mi.samplesdropped;
  totaloutstanding = totaloutstanding + mi.getOutstanding();
  totalfixed = totalfixed + mi.samplesrecovered;
  totallost = totallost + mi.sampleslost;
  totalCN = totalCN + mi.CN - mi.initCN;
  totalCN2 = totalCN2 + mi.CN;
  totaloverflows = totaloverflows + mi.overflows;
  totalresends = totalresends + mi.resends;
}
```

A.8 MoteIndex.java

Listing A.22: MoteIndex Class Discrition

```
import java.lang.*;

public class MoteIndex
{
  public int samplecapacity = 200;
  public long W = 100;  // Window size for acknowledgments

  public int id;      // ID of the mote this object is describing
  public int samplessent;    // Number of samples known to be sent by this mote
  public int samplesreceived=0;    // Total number of samles recieved by the mote
  public int samplesrecovered = 0;
  public int sampleslost = 0;
  public int samplesdropped = 0;
  public int samplesignored = 0;
  public int RSN;      // Recieved SN
  public long RSNTimestamp;
  public int LSN;
  public int ASN;
  public long CN;
  public long initCN;    // Value for CN when the mote is first contacted
  public int ackvector[];
  public int ackcountvector[];

  public long totDisseminations = 0;
  public long totD1 = 0;
  public long totD2 = 0;
  public long totD3 = 0;
  public long acktimevector[];
  public long droppedackvector[];

  public boolean ASNerror = false;
  public int ackwaiting = 0;
  public int backoff = 0;


  public short DSN = 0;
  public short DSNmote = 0;
  public int   DSNtype = 0xD3;
  public boolean   DSNacked = false;

  public int   overflows = 0;
  public long   resends = 0;

  public long ignorebefore = 0;

  public boolean initlock1 = true;
  public boolean initlock2 = true;


  public int check1 = 0;
  public int check2 = 0;
  public String msg = "";

  public MoteIndex ()
  { id = 0;
    RSNTimestamp = -1;
    RSN = 0;
    LSN = 0;
    ASN = -1;

    ackvector = new int[samplecapacity];
    ackcountvector = new int[samplecapacity];
    acktimevector = new long[samplecapacity];
    droppedackvector = new long[samplecapacity];

    for (int a = 0; a < ackvector.length; a++)
    { ackvector[a] = 0;
      ackcountvector[a] = 0;
      acktimevector[a] = 0;
      droppedackvector[a] = 0;
    }
  }

  public MoteIndex (int i, int csn, int asn)
  { id = i;
    RSNTimestamp = -1;
    RSN = 0;
    LSN = csn;
    ASN = asn;

    ackvector = new int[samplecapacity];
    ackcountvector = new int[samplecapacity];
    acktimevector = new long[samplecapacity];
    droppedackvector = new long[samplecapacity];

    for (int a = 0; a < ackvector.length; a++)
    { ackvector[a] = 0;
      ackcountvector[a] = 0;
      acktimevector[a] = 0;
      droppedackvector[a] = 0;
    }
  }

  public void initialize()
  {
    ASN = RSN;
    LSN = RSN;

    samplesignored = RSN+1;
    initCN = CN;

    for (int a = 0; a < ackvector.length; a++)
    { ackvector[a] = 0;
      ackcountvector[a] = 0;
      acktimevector[a] = 0;
      droppedackvector[a] = 0;
    }

    for (int i = 0; i <= ASN; i++)
    { acktimevector[i] = RSNTimestamp;
      ackcountvector[i] = 1;
    }

    ASNerror = true;
    initlock1 = true;
    initlock2 = true;
    backoff = 0;

    System.out.println("initializing");
  }

  public int getAckLength()
  {
    if (LSN == ASN) return 0;

    if(LSN >= ASN)
    {  return (LSN - ASN);
    }
    else
    {  return ((samplecapacity-ASN)+LSN);
    }
  }

  public int getSamplesSent()
  {
    if (LSN >= RSN)
    {  return Math.max((ackcountvector[RSN]-1),0)*samplecapacity+LSN+1-samplesignored;
    }
    else
    {  return Math.max((ackcountvector[RSN]),0)*samplecapacity+LSN+1-samplesignored;
    }
  }

  public byte[] getAckByteArray()
  {
    byte[] bytes;

    if (LSN == ASN)
    {   bytes = new byte[0];
      return bytes;
    }

    if (LSN > ASN)
    {  bytes = new byte[(LSN - ASN)/8+1];
      for (int i = 0; i < bytes.length; i++)
      {  bytes[i]=0;
      }
      for (int i=LSN; i>ASN; i--)
      {

        // shift the bits in the byte array to the left by 1 bit
        for (int b = bytes.length-1; b > 0; b--)
        {   bytes[b] = (byte)(bytes[b] << 0x01);
        byte checkbyte = (byte)(bytes[b-1] | 0x7F);

        //if (checkbyte == 0xFF)
        if (checkbyte == -1)
        {  bytes[b] = (byte)(bytes[b] | 0x01);
        }
      }
      bytes[0] = (byte)(bytes[0] << 0x01);

      // Set the ack bit
      if (ackvector[i]==1)
      {  bytes[0] = (byte)(bytes[0] | 0x01);
      }
    }
  }
  else
  { bytes = new byte[((samplecapacity-ASN)+LSN)/8+1];
    for (int i = 0; i < bytes.length; i++)
    {  bytes[i]=0;
    }
    for (int i=LSN; i>=0; i--)
    {
      // shift the bits in the byte array to the left by 1 bit
      for (int b = bytes.length-1; b > 0; b--)
      {   bytes[b] = (byte)(bytes[b] << 0x01);

        byte checkbyte = (byte)(bytes[b-1] | 0x7F);

        //if (checkbyte == 0xFF)
        if (checkbyte == -1)
        {  bytes[b] = (byte)(bytes[b] | 0x01);
        }
      }
      bytes[0] = (byte)(bytes[0] << 0x01);

      // Set the ack bit
      if (ackvector[i]==1)
      {  bytes[0] = (byte)(bytes[0] | 0x01);
      }
    }
    for (int i=samplecapacity-1; i>ASN; i--)
    {
      // shift the bits in the byte array to the left by 1 bit
      for (int b = bytes.length-1; b > 0; b--)
      {   bytes[b] = (byte)(bytes[b] << 0x01);

        byte checkbyte = (byte)(bytes[b-1] | 0x7F);

        //if (checkbyte == 0xFF)
        if (checkbyte == -1)
        {  bytes[b] = (byte)(bytes[b] | 0x01);
        }
      }
      bytes[0] = (byte)(bytes[0] << 0x01);

      // Set the ack bit
      if (ackvector[i]==1)
      {  bytes[0] = (byte)(bytes[0] | 0x01);
      }
    }
  }
  return bytes;
}

public void cleanAckVector()
{
  if (LSN == ASN)
  { for (int i = 0; i < samplecapacity; i++)
    {  ackvector[i] = 0;
    }
  }

  if (LSN > ASN)
  { for (int i = 0; i < samplecapacity; i++)
    { if ((i > LSN) || (i <= ASN))
      {  ackvector[i] = 0;
      }
    }
  }
  else
  { for (int i = 0; i < samplecapacity; i++)
    { if ((i > LSN) && (i <= ASN))
      {  ackvector[i] = 0;
      }
    }
  }
}

public void recordDroppedSamples()
{
  if (LSN == ASN) return;

  if (LSN > ASN)
  { for (int i=ASN+1; i<=LSN; i++)
    {   if ((ackvector[i] == 0) && (droppedackvector[i] == 0))
        {   droppedackvector[i] = 1;
            samplesdropped++;
        }
    }
  }
  else
  {
    for (int i=ASN+1; i<samplecapacity; i++)
    {  if ((ackvector[i] == 0) && (droppedackvector[i] == 0))
       {   droppedackvector[i] = 1;
           samplesdropped++;
       }
    }
    for (int i=0; i<=LSN; i++)
    {  if ((ackvector[i] == 0) && (droppedackvector[i] == 0))
       {   droppedackvector[i] = 1;
           samplesdropped++;
       }
    }

  }
}

public String getAckString()
{
  if (LSN == ASN) return "{}";

  String result = "{";

  if (LSN > ASN)
  { for (int i=ASN+1; i<=LSN; i++)
    { result = result + "" + ackvector[i];
    }
  }
  else
  {
    for (int i=ASN+1; i<samplecapacity; i++)
    { result = result + "" + ackvector[i];
    }
              for (int i=0; i<=LSN; i++)
    { result = result + "" + ackvector[i];
    }

  }
  result = result +"}";

  return result;
}

public int getOutstanding()
{
  int count = 0;

  if (LSN == ASN) return count;

  if (LSN > ASN)
  { for (int i=ASN+1; i<=LSN; i++)
    { if (ackvector[i] == 0) count++;
    }
  }
  else
  {
    for (int i=ASN+1; i<samplecapacity; i++)
    { if (ackvector[i] == 0) count++;
    }
    for (int i=0; i<=LSN; i++)
    { if (ackvector[i] == 0) count++;
    }

  }

  return count;
}

public String getSampleAckString()
{
  String result = "{";
  for (int i=0; i<samplecapacity; i++)
  {  result = result + "," +ackvector[i];
  }
  result = result + "}";

  return result;
}

public String getSampleLostString()
{
  String result = "{";
  for (int i=0; i<samplecapacity; i++)
  {  result = result + "," +droppedackvector[i];
  }
  result = result + "}";

  return result;
}

public String getSampleCountString()
{
  String result = "{";
  for (int i=0; i<samplecapacity; i++)
  {  result = result + "," +ackcountvector[i];
  }
  result = result + "}";

  return result;
}

public boolean isInRange(int i)
```

```
{
  boolean result = false;

  if ((i < 0) || (i > samplecapacity)) return false;

  if (LSN == ASN) return false;

  if (LSN > ASN)
  { if ((LSN >= i) && (i > ASN))
    {  result = true;
    }
  }
  else
  { if ( ((samplecapacity > i) && (i > ASN)) ||
         ((LSN >= i) && (i >= 0)) )
    {  result = true;
    }
  }

  return result;
}

public boolean checkWindowError()
{
  if (getAckLength() > W)
  {  return true;
  }
  else
  {  return false;
  }
}

public void giveup()
{
  int csampleslost = 0;
  int acvcopy[] = new int[samplecapacity];

  for (int i = 0; i < ackvector.length; i++)
  {  acvcopy[i] = ackcountvector[i];
  }

  ignorebefore = RSNTimestamp;

  System.out.println("mote " + id + ": " + "ASN = " + ASN + " RSN = "
                           + RSN + " ignorebefore = " + ignorebefore);
  System.out.println("mote " + id + ": " + getSampleCountString());
  System.out.println("mote " + id + ": " + getSampleAckString());
  System.out.println("mote " + id + ": " + getSampleLostString());

  if (RSN > ASN)
  { for (int i=ASN+1; i<=RSN; i++)
    {   if(acvcopy[i] < ackcountvector[RSN])
      {  ackcountvector[i] = ackcountvector[i] + 1;
        if (droppedackvector[i] == 0)  samplesdropped++;
        droppedackvector[i] = 0;
        csampleslost++;
        System.out.print(i + ", ");
      }
    }
  }
  else
  {
    for (int i=RSN; i>=0; i--)
    {   if(acvcopy[i] < ackcountvector[RSN])
        {  ackcountvector[i] = ackcountvector[i] + 1;
        if (droppedackvector[i] == 0)  samplesdropped++;
        droppedackvector[i] = 0;
        csampleslost++;
        System.out.print(i + ", ");
        }
    }
    for (int i=samplecapacity-1; i>LSN; i--)
    {   if(acvcopy[i] < ackcountvector[RSN]-1)
        {  ackcountvector[i] = ackcountvector[i] + 1;
        if (droppedackvector[i] == 0)  samplesdropped++;
        droppedackvector[i] = 0;
        csampleslost++;
        System.out.print(i + ", ");
        }
    }
  }
  System.out.println();

  if (RSN < LSN)
  {  for (int i = RSN +1; i <= LSN; i++)
    {  if (droppedackvector[i] == 0)  samplesdropped++;
      droppedackvector[i] = 1;
    }
  }
  if (RSN > LSN)
  {  for (int i = RSN +1; i < samplecapacity; i++)
    {   if (droppedackvector[i] == 0)  samplesdropped++;
      droppedackvector[i] = 1;
    }
    for (int i = 0;      i <= LSN;           i++)
    {  if (droppedackvector[i] == 0)  samplesdropped++;
      droppedackvector[i] = 1;
    }
  }

  for (int i = 0; i < samplecapacity; i++)
  {  if (acktimevector[i] < acktimevector[RSN])
    {  ackvector[i] = 0;
    }
  }

  System.out.println("mote " + id + ": " + getSampleCountString()
                             + " " + csampleslost +" samples lost.");
  System.out.println("mote " + id + ": " + getSampleAckString());
  System.out.println("mote " + id + ": " + getSampleLostString());
  sampleslost = sampleslost + csampleslost;

  ASN = RSN;
}

// Returns true if every seqno has been acknowledged.
public boolean allclear()
{  boolean result = true;

  if (LSN == ASN) return true;

  if (LSN > ASN)
  {  for (int i=ASN+1; i<=LSN; i++)
    { if(ackvector[i] == 0) result = false;
    }
  }
  else
  {
    for (int i=LSN; i>=0; i--)
    { if(ackvector[i] == 0) result = false;
    }
    for (int i=samplecapacity-1; i>ASN; i--)
    { if(ackvector[i] == 0) result = false;
    }
  }
  return result;
```

```
}

public boolean checkASNError()
{ if (ASN == findASN())
  {  return false;
  }
  else
  {  return true;
  }
}

public int findASN2()
{ int result = ASN;
  int min;
  boolean found = false;

  if (ASN == -1) return -1;

  min = ackcountvector[0];
  for (int i = 0; i < samplecapacity; i++)
  {  if (ackcountvector[i] < min)
     {   min = ackcountvector[i];
         result = i;
         found = true;
     }
  }

  if (found == true)
  {   if (result > 0)
      {   return result - 1;
      }
      else
      {   return samplecapacity - 1;
      }
  }
  else
  {   return ASN;
  }
}

public int findASN()
{ int result = ASN;
  int min;
  boolean found = false;

  if (ASN == -1) return -1;
  if (ASN == LSN) return ASN;


  if (ASN < LSN)
  { min = ackcountvector[RSN];
    for (int i = ASN; i >= 0; i--)
    {
       if (ackcountvector[i] < min)
       {   result = i; found = true;
           System.out.println("a \t"+min +"\t"+ackcountvector[i]+"\t"+i);
       }
    }
    min = ackcountvector[RSN]-1;
    for (int i = samplecapacity-1; i > LSN; i--)
    {  if (ackcountvector[i] < min)
       {   result = i; found = true;
           System.out.println("b \t"+min +"\t"+ackcountvector[i]+"\t"+i);
       }
    }
  }
  else
  { if (RSN < ASN )
```

```
      {  min = ackcountvector[RSN] - 1;
      }
      else
      {  min = ackcountvector[RSN];
      }
      for (int i = ASN; i > LSN; i--)
      {  if (ackcountvector[i] < min)
         {   result = i; found = true;
             System.out.println("c \t"+min +"\t"+ackcountvector[i]+"\t"+i);

         }
      }
    }

    if (found == true)
    {   if (result > 0)
        {   return result - 1;
        }
        else
        {   return samplecapacity - 1;
        }
    }
    else
    {   return ASN;
    }
  }

  public boolean reasonablepacket(DACKCollMsg dcm)
  {
    if ( (dcm.get_lan() < 0) ||
         ((dcm.get_lan() > samplecapacity) && (dcm.get_lan() != 65535)) ||
         (dcm.get_lsn() < 0) || (dcm.get_lsn() > samplecapacity) )
     {  return false;
     }

     for (int i = 0; i < dcm.numElements_sample_sn(); i++)
     {
        if ((dcm.getElement_sample_sn(i) < 0) || (dcm.getElement_sample_sn(i) > samplecapacity))
        {   return false;
        }
     }

     return true;
  }


  public static String shortToHexString(short s){
    int i = s & 0xFF;
    return Integer.toHexString(i);
  }

  public static String byteToHexString(byte b){
    int i = b & 0xFF;
    return Integer.toHexString(i);
  }

  public static String byteToBinaryString(byte b){
    int i = b & 0xFF;
    return Integer.toBinaryString(i);
  }
}
```

Vita

Candidate's full name: John-Paul Arp
Place & date of birth: Fredericton, New Brunswick, September 11 1978

University attended:

September 2005 - January 2010
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

Bachelor of Computer Science
September 2002 - May 2005
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

Enrolled in Bachelor of Arts and Computer Science Program
September 1996 - May 2001
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

Publications:
Arp, John-Paul and Nickerson, B.G. ”A User Friendly Toolkit for Building Robust Environmental Sensor Networks”, Proc. of the Communication Networks and Services Research Conference, CNSR 2007, May 14-17, 2007, Fredericton, N.B., Canada, pp. 76-81.

Nickerson, Bradford G., Sun, Zhongwei and Arp, John-Paul ”A Sensor Web Language for Mesh Architectures”, Communication Networks and Services Research Conference, CNSR 2005, May 16-18, 2005, Halifax, Nova Scotia, Canada, pp. 269-274.

Posters:
Arp, John-Paul and Nickerson, Bradford G. ”The Disseminated ACKnowledgement (DACK)

Protocol for Data Collection in Wireless Sensor Networks”, Poster at the 11th Annual GEOIDE Scientific Conference, Vancouver, B.C., May 27-29, 2009.

Arp, John-Paul and Nickerson, Bradford G. ”Water Level Monitoring by Image Observation of Bridge Piers”, Poster at the 10th Annual GEOIDE Scientific Conference, Niagara Falls, Ontario, May 28-30, 2008, available at
http://www.cs.unb.ca/ bgn/talks-/geoide2008JPA_BGNposterV2.ppt

Arp, John-Paul and Nickerson, Bradford G. ”A User Friendly Toolkit for Building Robust Environmental Sensor Networks”, Proceedings of the GEOIDE 2007 Scientific Conference, Halifax, N.S., June 7-8, 2007, poster presentation.

Arp, John-Paul and Nickerson, B.G. ”Reliable Low-Power Communications for Mobile Ad Hoc Networks”, Communication Networks and Services Research Conference, CNSR 2006, May 23-25, 2006, Moncton, N.B., Canada, ”work in progress” poster presentation.

Other Documents:
Nickerson, Bradford G. and Arp, John-Paul ”Saint John River Image Based Water Level Monitoring System Hardware and Software Guide”, UNB Faculty of Computer Science internal report for New Brunswick Emergency Measures Organization, June, 2009, 62 pages.

Nickerson, Bradford G. and Arp, John-Paul ”Bridge Sensor System Design Report Decision Support for Flood Event Prediction and Monitoring”, Version 2.0, UNB Faculty of Computer Science internal report for New Brunswick Emergency Measures Organization, March 31, 2008, 23 pages.