Wireless Sensor Networks for High Fidelity Sampling

Sukun Kim

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-91
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-91.html

July 20, 2007

Wireless Sensor Networks for High Fidelity Sampling

by

Sukun Kim

B.Eng. (Korea Advanced Institute of Science and Technology) 2002
M.S. (University of California, Berkeley) 2005

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor David E. Culler, Chair
Professor Ion Stoica
Professor Gregory L. Fenves

Fall 2007

The dissertation of Sukun Kim is approved.

Chair Date

Date

Date

University of California, Berkeley

Fall 2007

Wireless Sensor Networks for High Fidelity Sampling

by

Sukun Kim

Abstract

Wireless Sensor Networks for High Fidelity Sampling

by

Sukun Kim

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

Our hypothesis was that Wireless Sensor Networks (WSN) could be used for High Fidelity Sampling (HFS). WSN is constrained by limited resources, and HFS requires data of high quality, like high-accuracy high frequency in a large scale. This dissertation explains how we achieved our goal.

A Wireless Sensor Network (WSN) for Structural Health Monitoring (SHM) is designed, implemented, deployed and tested on the 4200ft long main span and the south tower of the Golden Gate Bridge (GGB). Ambient structural vibrations are reliably measured at a low cost and without interfering with the operation of the bridge. Requirements that SHM imposes on WSN are identified and new solutions to meet these requirements are proposed and implemented. For example, diverse design options for a reliable data transfer are analyzed and an initial solution for a reliable data collection, Straw, is proposed.

In the GGB deployment, 64 nodes are distributed over the main span and the tower, collecting ambient vibrations synchronously at 1kHz rate, with less than 10μs jitter, and with an accuracy of 30μG. The sampled data is collected reliably over

a 46-hop network, with a bandwidth of 441B/s at the 46th hop. The deployment is the largest WSN for SHM. Data from the deployment is analyzed. The collected vibration data agrees with theoretical models and previous studies of the bridge. Interesting behaviors are observed, and pitfalls and lessons are discussed. Especially, to overcome pitfalls of reliable data collection from the deployment, an improvement, Flush, is proposed.

The Flush protocol provides an end-to-end reliability, minimizes transfer time, and adapts to time-varying network conditions. It achieves these properties using end-to-end acknowledgments, implicit snooping of control information, and a rate-control algorithm that operates at each hop along a flow. Using several real network topologies, we show that Flush closely tracks or exceeds the maximum goodput achievable by a hand-tuned, fixed-rate for each hop over a wide range of path lengths.

Professor David E. Culler
Dissertation Committee Chair

To my family: Kuk Kyoung Kim, Okji Lee, and Ah Young Kim

Contents

List of Figures

List of Tables

Acknowledgements

David Culler advised and guided me with enormous insight and passion that allowed me to grow as a researcher. David taught me how to think of research problems critically. He guided me to learn scientific approaches and methods in solving problems. His insight often challenged my assumptions and designs, and led to more compelling and general frameworks. He also dedicated significant passion and patience to advise me and help crystallize my rough ideas. He was willing to accompany me during deployments. When I was initially struggling with the language barrier upon first arriving to Berkeley, he patiently tried to understand what I was saying and helped me move on to a deeper discussion. David is a role model I will always strive to follow in my research and life.

Professors Gregory Fenves, James Demmel and Steven Glaser always guided my research towards the right direction with deep insights and warm encouragements. Their dedicated passion for exploring and building hands-on deployments, is something I admire and wish to keep in my heart. I am fortunate to have received much excellent advice from Professors Ion Stoica, Scott Shenker, and Anthony Joseph. They gave me a wider point of view of wireless sensor networks and systems building in general, which broadened my work and thought.

Shamim Pakzad wisely and patiently motivated and cheered me along throughout this work. His strong intellect was critical in solving problems we encountered. Shamim never lost his sense of humor, even when he was extremely exhausted after climbing the Golden Gate Bridge countless times. He went through all difficulties and excitement with me at the bridge, and those moments will never be forgotten in my memory.

Rodrigo Fonseca has worked together with me in investigating the design space of reliable transfer, and had a critical role in making the Flush happen. Rodrigo

sparked many brilliant ideas, and his fountain of ideas was never depleted. He was very considerate in listening to my concerns confronted in research, and gently gave warm cheers and original comments.

Jaein Jeong always listened to my worries on research and life, and always gave kind advice. Prabal Dutta, Arsalan Tavakoli, and Phil Levis showed bright insights and inspired me all throughout our many projects together. I owe sincere gratitude to them. Jay Taneja generously read and gave great comments on my works.

I enjoyed meeting with people in the TinyOS/NEST/SNA teams as a colleague and as a friend: Alec Woo, Rob Szewczyk, Kamin Whitehouse, Cory Sharp, Joe Polastre, Jonathan Hui, Gilman Tolle, Fred Jiang, Jorge Ortiz, Cheng Tien Ee, and Daekyeong Moon. Our discussions, lunches, and other meetings were a great source of ideas and refinements for our work. I feel very lucky to have shared many wonderful times with them.

In Intel Research at Berkeley, I had an exceptional opportunity to work with great researchers: Kevin Fall, Phil Buonadonna, Wei Hong, and David Gay. I am grateful to Tom Oberheim and Martin Turon for help on numerous occasions in the process of designing, developing, and calibrating an accelerometer board.

I would like to thank the staff and management of the Golden Gate Bridge District, in particular Dennis Mulligan and Jerry Kao, for their close cooperation in every step of the project. I am especially thankful to Jorge Lee. Without his extraordinary help the deployment at the Golden Gate Bridge would not have been possible.

This work was supported by the Defense Advanced Research Projects Agency (grant F33615-01-C-1895), the National Science Foundation (grants EIA-0122599, IIS-033017), the Department of Energy (grant DR-03-01), the Center for Information Technology Research in the Interest of Society (CITRIS), Crossbow Technology Inc., Intel Corporation, and the Hewlett-Packard Company.

Chapter 1

Introduction

1.1 Problem Statement

High Fidelity Sampling (HFS) represents a class of applications which requires highly accurate capturing of physical phenomena. It necessitates high resilience to noise, manufacturing variation and environmental changes. It also requires high sampling frequency with low time uncertainty in the sampling time. In many examples of HFS, highly time-synchronized sampling is required. HFS needs to scale to extreme numbers. Triggering acquisition and the collection of data should be done reliably without any loss of control or data. Structural Health Monitoring (SHM) is a concrete example of HFS, which is used as a research vehicle in this work. However, the result of the work can be readily applicable to other applications in HFS, including earthquake monitoring and machine monitoring.

Structural Health Monitoring (SHM) allows the estimation of the structural state and detection of structural changes that might affect the performance of a structure. Two discriminating factors in SHM are the time-scale of the change (how quickly the state changes) and the severity of the change. These factors represent

two major sources of system change: alarm warnings [87] (e.g. disaster notification for earthquake, explosion, etc.) and continuous health monitoring (e.g. from ambient vibrations, wind, etc.). The general approaches taken to SHM are either direct damage detection (visual inspection, x-ray, etc.) or indirect damage detection (detecting changes in structural properties or system behavior). This dissertation describes a platform for indirect detection of structural state through the measurement and interpretation of ambient vibrations and strong motion. The chosen test bed is the Golden Gate Bridge in San Francisco Bay.

Performing SHM by the use of sensor networks is not a new concept [28, 66]. The traditional approach uses conventional piezoelectric accelerometers hard-wired to data acquisition boards residing in a PC. The drawbacks of such a system include (1) the high cost of installation and disturbance of the normal operation of the structure due to wires having to run all over the structure, (2) the high cost of equipment; and (3) cost of maintenance. Compared to the conventional methods, Wireless Sensor Networks (WSN) provide comparable functionality at a much lower cost, which permits a higher spatial density of sensors. The prototype wireless system presented in this work costs about $600 per node compared to thousands of dollars for a node with the same functionalities in a traditional PC-based wired network. Compared to the wired network, installation and maintenance are easy and inexpensive in a WSN, and disruption of the operation of the structure is minimal.

This work has four main contributions to WSN for SHM and WSN in general:

- It identifies requirements to obtain data of sufficient quality to have real scientific value to civil engineering researchers, and examines how to solve them.
- The system is scalable to a large number of nodes to allow dense sensor coverage of real-world structures. For example, a long-lived 46-hop network was implemented on the Golden Gate Bridge (Figure 1.1 and 1.2).

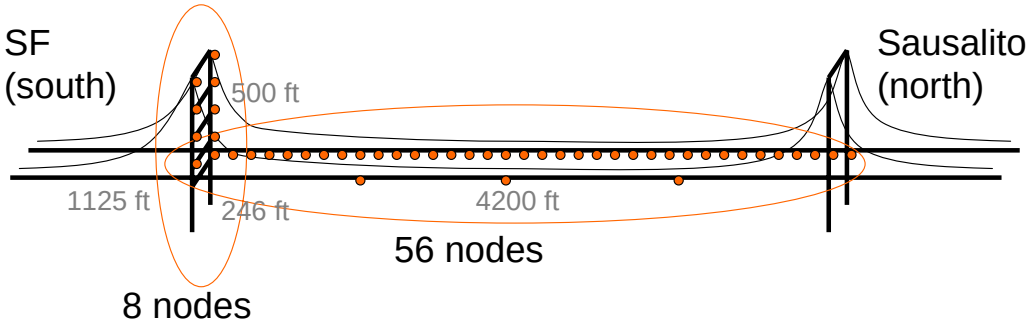

Figure 1.1. The Golden Gate Bridge and layout of nodes on the bridge. To cover this large bridge, a long linear topology needs to be used, bringing challenges to the network.

- It addresses a myriad of problems encountered in a real deployment in difficult conditions, rather than a simulation or laboratory test bed.
- It provides a new reliable data collection protocol, which is verified and improved using lessons from the deployment.

A WSN for SHM was deployed on the Golden Gate Bridge (GGB), see Figure 1.1. The 46-hop system consists of 64 nodes, which measure ambient vibrations with an accuracy of 30μG. The ambient vibrations were sampled at 1kHz with a time jitter less than 10μs. Figure 1.2 illustrates the bandwidth obtained by Straw, a new reliable data collection component written for this installation. The system provided high bandwidth data streaming of 441B/s from the 46th hop to the base station by implementing pipelining. This 46-hop wireless sensor network is the largest number-of-hop installation reported in the literature up to now. Modal properties also match with the simulation model and the previous study.

This dissertation will explain how the system was designed and implemented to achieve this successful deployment on the bridge, and also lessons learned. Especially a reliable data collection will be introduced in detail, then improvements using experiences at the bridge will be presented. The late part of the dissertation will present an analysis of the collected acceleration data recorded along a linearly dense array.

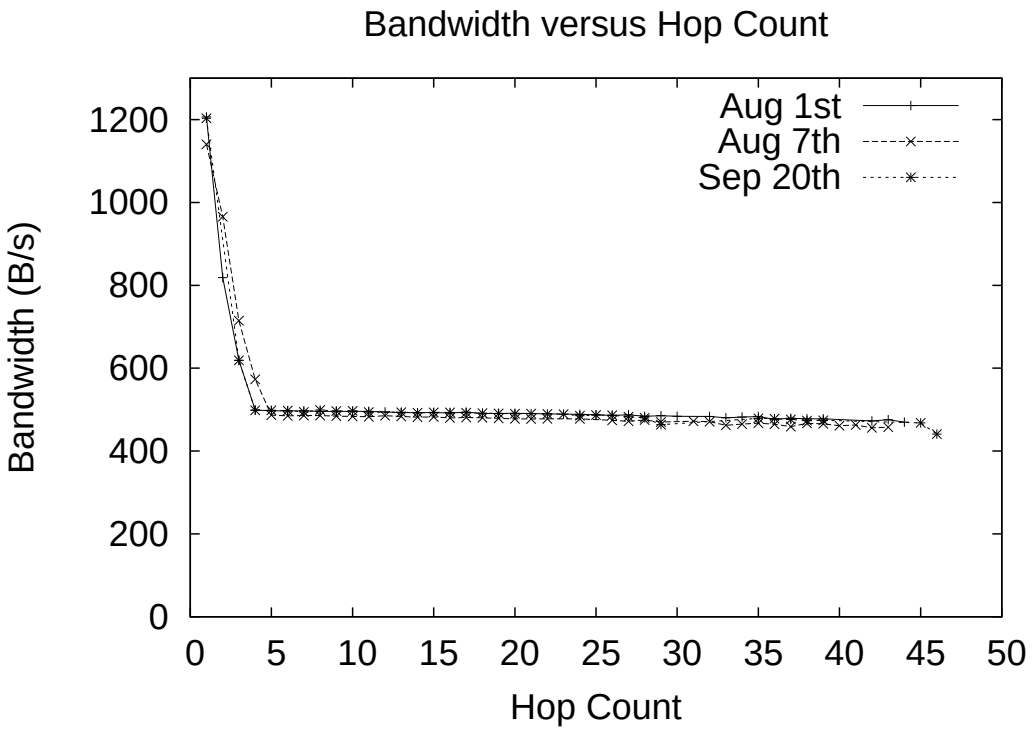

Figure 1.2. Bandwidth of Straw at the Golden Gate Bridge. It works over a 46-hop network. To sustain high bandwidth over a long path, pipelining is used avoiding interference.

1.2 Six Requirements of the Problem

Six major requirements of SHM on WSN are identified here:

1. The data acquisition system had to be able to detect signals with peak amplitudes as low as 500μG [22]. The installation had to minimize sources of distortion such as the noise floor of the system (including accelerometer, amplifier, analog to digital converter, etc.), installation error, and temperature variation. We proposed a solution in Section 2.4.

2. Because of structural interest in local modes of vibration, a sampling rate of 1kHz was required as the target rate. This rate and the need for 16-bit digitization accuracy require low jitter, i.e. low time uncertainty of the sampling intervals. A proposed solution is presented in Section 2.5.

3. Time synchronization in sampling through the bridge is required to perform correlation analysis of the structural vibrations. This was particularly challenging due to the drift of independent clocks at each of the 64 nodes. An earlier reported solution to the time synchronization problem is FTSP [61] with an accuracy of 67μs over 11 hops.

4. The GGB installation required a large-scale multi-hop network due to the great length of the main span and the fact that the aggregator station could only be located in the base of the south tower. One existing solution to the collection network is MintRoute [86].

5. Commands had to be reliably disseminated throughout the entire system so that all parts of the network could start on command, and insure against lost data or a blockage of hopping. Repeated Broadcast [4] can be one solution.

6. Data must be transferred reliably. Vibration data, in this case, is too valuable to

be lost to a communication error. A solution (Straw) is proposed in Section 3.10, and an improvement (Flush) is presented in Chapter 6.

Requirement 4 is connectivity at the network layer, and requirement 6 is reliability at the transport layer. Existing works are used for requirements 3, 4, and 5: FTSP [61] for time synchronization, MintRoute [86] for multi-hop collection routing, and repeated Broadcast [4] for reliable command dissemination. The remaining three requirements are solved in this work: a data acquisition system, high-frequency sampling with low time uncertainty, and reliable data collection. As will be seen in the next section, previous works satisfy some but not all of the requirements.

1.3 Background

This section introduces previous works which provide a background for this work. At first, WSN applications will be presented and categorized. Then previous works in WSN for SHM are introduced, and it is discussed why they fail to meet all of the requirements of SHM, which are introduced in the previous section.

1.3.1 Wireless Sensor Networks Applications

WSN applications can be divided into two categories. The first category is environmental monitoring. It monitors environmental data (e.g. temperature, humidity) over a long period of time. For this category of problem the focus is on networks with a low duty-cycle and low power consumption. Great Duck Island [59] and the Redwood forest deployment [81] are examples of this category. The second category of WSN applications consists of applications that require an identification of a mechanical system through a measured system response. This category requires highly sensitive sensors, high frequency sampling, highly correlated sampling, and so

on. The requirements can be characterized as High Fidelity Sampling (HFS). Health monitoring of mechanical machines [51], condition-based monitoring, volcano monitoring [84], earthquake monitoring [37], and structural health monitoring [68] belong to this category. The focus of this work is to address the requirements of the latter category. As a concrete research vehicle of this abstract category of applications, Structural Health Monitoring (SHM) is used. However, findings from this example application are readily applicable to other applications in the category without much difficulty, because they have very similar requirements.

1.3.2 Wireless Sensor Networks for Structural Health Monitoring

Previous works on using WSN in SHM can be classified mainly into two approaches. The first approach is oriented from the research of structural analysis. The second approach is rooted from the research of computer science.

Examples of the first approach include [27, 32, 52, 72]. MEMS accelerometers are used to sample vibration. They all support highly accurate data acquisition systems and high frequency sampling, satisfying requirements 1 and 2 in Section 1.2. RF radios are used for wireless communication, and microcontrollers drive sensors and radios. However, those early efforts use custom radios, can not scale to multiple nodes over a multi-hop network, and they provide no reliable communication. [46] uses Mica2 and TinyOS. It supports multi-hop network, however there is no provision for time synchronization, which is necessary for time-related sampling of the entire structure. It is unclear whether reliable communication is provided. [57] satisfies requirements 1, 2, 3, 5, and 6. It produces data meaningful for structural analysis. However, the problem is solved in the context of a single hop network and they does not scale to

a long enough multi-hop network needed to cover a large structure. Many previous works have not been implemented and tested in a harsh real-life environment.

As examples of the second approach, Wisden [87] and Tenet [38] satisfy many requirements. They provide reliable data collection over a multi-hop network. However, Wisden can sample only up to 160Hz, and Tenet demonstrated only 50Hz sampling, which is far below the threshold needed for structural health monitoring. Wisden and Tenet have not been analyzed for sampling jitter, which is needed in determining to what degree the resulting data has confidence for analysis in civil engineering. They are tested only in a small-scale indoor test bed. The most critical pitfall of these two systems is that they do not produce time-synchronized data. Wisden has a time stamp on each sample. However, the input for basic modal property analysis is a matrix of time-synchronized samples from multiple nodes [56]. This requirement is not met by Wisden and Tenet, and the data produced by them has no value for a meaningful structural analysis.

1.4 Roadmap

Chapter 2 shows overall architecture of the system: how the components meet requirements and fit together. Then, the data acquisition system is introduced, which is a solution to the first requirement this work proposed. This chapter also explains how high-frequency sampling is achieved with low time uncertainty, which is the second requirement addressed here.

For the last requirement this work handles (reliable data collection), Chapter 3 investigates the design space of reliable transfer, and proposes an initial solution: Straw. Straw uses selective NACKs to provide reliability and relies on rate control to exploit channel capacity.

All solutions are integrated as one system, and the system is deployed on the Golden Gate Bridge (Chapter 4). Challenges and details of the deployment are described in this chapter.

Chapter 5 shows how the network performed on the Golden Gate Bridge. Data is collected reliably and with stability. With pipelining, bandwidth does not drop significantly even when the length of a path exceeds 5 hops.

Based on experiences at the bridge, Chapter 6 presents an improved reliable data collection protocol: Flush. Interference is measured continuously, and the rate is optimized for a given channel capacity dynamically. Compared to a fixed rate used in Straw, Flush achieves the same or better bandwidth, while providing more stability and adaptivity.

Data from a large-scale real-world deployment at the Golden Gate Bridge is presented in Chapter 7, which was not possible in previous works with WSN. The data matches with a simulation model and previous study using a wired system.

In Chapter 8, remaining issues, newly found challenges and their implications are discussed. For example, it is found that heavy traffic disturbs the estimation of a link quality, and there is not a solution to this issue yet.

Chapter 9 concludes revisiting contributions of the work.

Chapter 2

Overall Architecture

In this chapter, our overall system is introduced at a high level. Also, it explains how all software components fit together. Then, our application layer toolkit is introduced. In Section 2.4, a front-end data acquisition system is introduced, which can capture and process a minute signal with very high sensitivity. Our system samples a signal at a high frequency, at the same time satisfying a jitter requirement. Section 2.5 explains how it is achieved. The last section introduces components for time synchronization, multi-hop routing, and reliable data dissemination, all three of which are taken from existing works.

2.1 System Architecture

The wireless network is composed of multiple nodes and a base station, see Figure 2.1. A node consists of a mote and a sensor board. The node measures vibration at two different orders of dynamic bandwidth with the data communicated back to the base station through wireless communication. The base station is a server providing more computational power and larger storage than a mote node, and possibly a

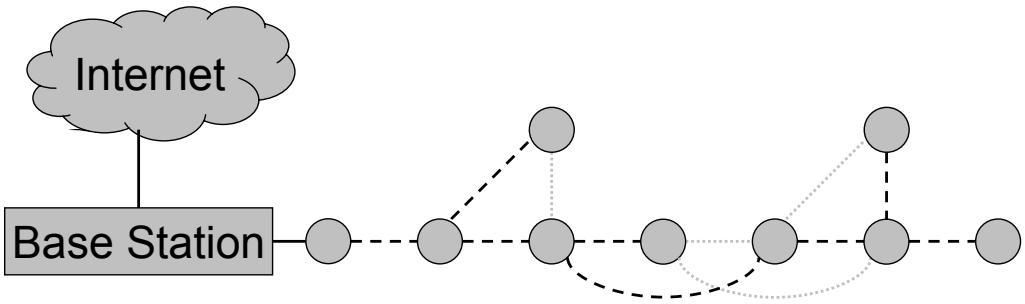

Figure 2.1. High-Level Overview of the System

connection to the Internet. In the GGB deployment a laptop is used as a base station. The software architecture of the GGB nodes uses new components integrated into the TinyOS [41] infrastructure to satisfy the six requirements discussed in Section 1.2.

2.2 Composition of Software Components

Figure 2.2 illustrated the overall software structure. At the bottom, there is a best-effort single-hop communication layer [71]. Above this layer lies a dissemination layer (Broadcast [4]), a collection layer (MintRoute [86]), and a time synchronization (FTSP [61]). All three components will be introduced in Section 2.6. Our new reliable data collection layer, Straw, lies above Broadcast and MintRoute. Low-level Flash is another bottom component. Flash has a high latency of writing. BufferedLog [5] uses a double buffer to overcome the latency, and supports high frequency sampling with light-weight logging. At the top, there is an application layer, Sentri. Sentri combines and drives all the underlying components. Detailed introduction of Sentri follows in the next section.

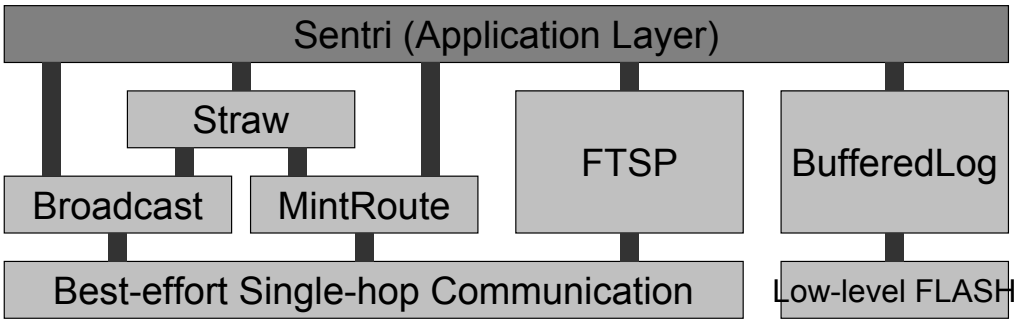

Figure 2.2. Overall Software Architecture

2.3 Sentri: Structural Health Monitoring Toolkit Application

Structural hEalth moNiToRing toolkIt (Sentri) is an application layer program which drives all the components. Instead of a stand-alone program, Sentri is structured like an RPC server: for every operation an operation packet is sent from the base station to a node. In SHM, motes are heavily used and heavy traffic makes network bandwidth the bottleneck of the operation; therefore, additional processing and traffic overhead must be avoided. However, since the project is in the research stage in both computer science and civil engineering and the operation sequences and parameters were changing frequently, the operational model was necessary. It allows us to figure out precisely which parts of the signals are the most valuable, and to fine-tune the system parameters in an interactive process.

This process was first tested and verified through a trial deployment of several nodes on a model steel bridge at the campus and then again on a footbridge over I-80 [68], both of which were used to determine system settings. Sentri provides 16 operations and each operation is contained in a single packet, see Table 2.1. More details of Sentri operations can be found at [2]. Sentri at a server PC-side gives diverse commands to users. One Sentri command is composed of one or more of

Table 2.1. 16 Operations of Sentri

Operation	Explanation
LED_ON	Turn on red LED
LED_OFF	Turn off red LED
PING_NODE	Ping a node
FIND_NODE	Same as PING_NODE, except that it can be specified which nodes should not reply
RESET	Reset all parameters
ERASE_FLASH	Erase Flash
START_SENSING	Ask a node to start sampling at a specified time
READ_PROFILE1	Query the first half of meta data of data in Flash
READ_PROFILE2	Since meta data is large, it fits into two packets
TIMESYNC_INFO	Query information of the time synchronization component
NETWORK_INFO	Query information of the routing layer
FIX_ROUTE	Freeze a routing tree
RELEASE_ROUTE	Thaw a routing tree
TIMESYNC_ON	Turn on time synchronization
TIMESYNC_OFF	Turn off time synchronization
FOR_DEBUG	Internally used for debugging

Sentri operations. In response to one user command, a sequence of Sentri operations are sent in operation packets by the base station. This model provides a great deal of flexibility. Depending on changes in usage, a user command can be reconfigured with a different sequence of operations. [15] shows a manual for Sentri user commands.

2.4 Data Acquisition System (Requirement 1)

Figure 2.3 shows an overview of the hardware as a block diagram. The data acquisition system performs three primary functions: sensing, signal processing and communication. Because of extensive experience with the product, Crossbow MicaZ [20] motes were used for control and communications. The analog signals output by the low-noise accelerometers pass through low-pass antialiasing filters on the way to a 16-bit analog-to-digital converter before the data is logged into the flash of the mote and then wirelessly transmitted.

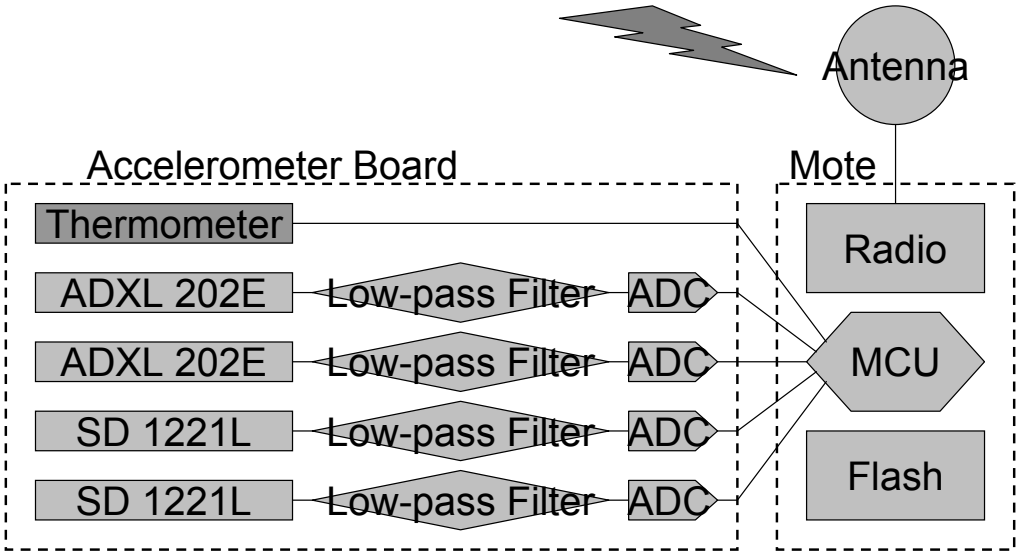

Figure 2.3. Hardware Block Diagram. Details of two accelerometers (ADXL 202E and SD 1221L) are in Table 2.2. A thermometer is used for temperature calibration.

2.4.1 Accelerometer Sensor Board

A new accelerometer board [20], shown in Figure 2.4, was designed for SHM applications. The board has four independent accelerometer channels monitoring two directions (vertical and transverse), and a thermometer to measure accelerometer temperature for compensation purposes. Low-amplitude ambient vibrations, due to wind loading and traffic, are resolved by a two-dimensional SiliconDesigns 1221L accelerometer. A low-cost ADXL202E two-dimensional accelerometer was used to monitor stronger shaking as might be expected from earthquake excitation. Because input battery power can vary between 6V and 12V, the sensor board contains a voltage regulator to provide a constant 3V output for the mote and a constant 5V output for the ratiometric accelerometers. Table 2.2 presents the characteristics of the two accelerometers used, and associated analog circuits. Two simple filters are used on the board. One is a hardware-implemented single-pole 6db low-pass filter with a cutoff frequency of 25Hz. Since the on-board ADC quantizes much faster than the target sampling frequency, this extra capacity allows on-the-fly digital filtering after

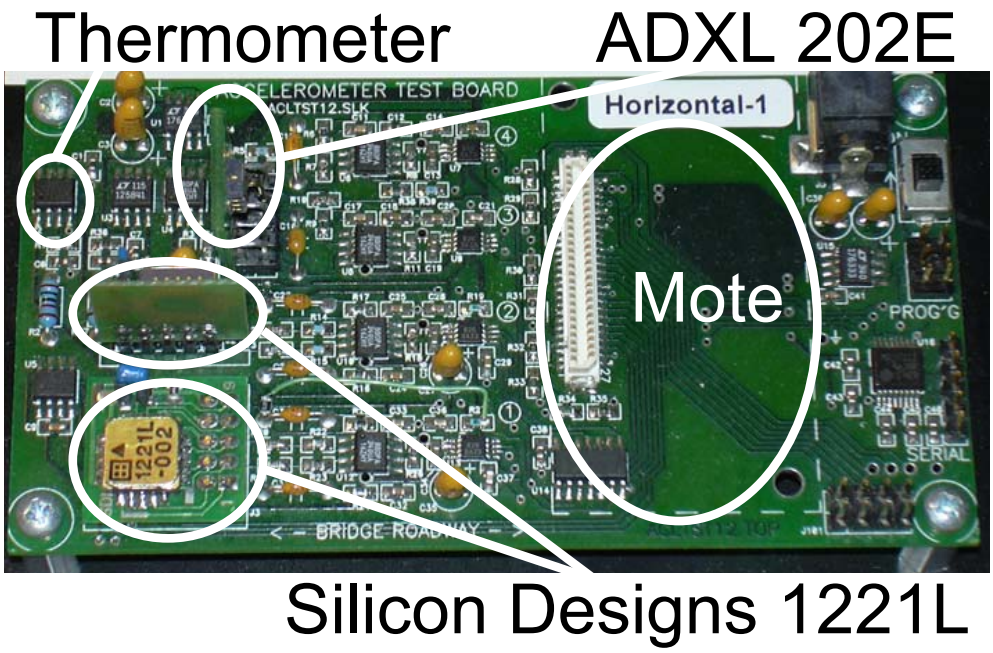

Figure 2.4. Accelerometer Board. ADXL 202E is a two axis accelerometer in a single chip. Either Mica2 or MicaZ can be used as a mote.

Table 2.2. Comparison of the Two Accelerometers. G means the acceleration of gravity.

	ADXL 202E	Silicon Designs 1221L
Type	MEMS	MEMS
Range of System	-2G to 2G	-0.1G to 0.1G
System noise floor	$200(\mu G/\sqrt{Hz})$	$32(\mu G/\sqrt{Hz})$
Price	$10	$150

a factor of $S_{over} = 10$ oversampling, and then averages the samples before logging. Assuming a Gaussian distribution for the noise, oversampling by a factor of $S_{over} = 10$ reduces the noise level by a factor of $\sqrt{S_{over}} \simeq 3.16$.

Another key hardware consideration in WSN is power consumption. The high duty-cycle required by vibration SHM produces data sets that are between two to four orders of magnitude larger than that of an environmental monitoring application. In contrast to environmental monitoring, this application requires the radio to operate most of time due to a large traffic volume. The gain from duty-cycling does

Table 2.3. Power Consumption in Various Operational Situations (9V input voltage). Idle is when both the sensor board and the mote are turned on, but are not performing any operation.

Situation	Consumption (mW)
Board Only	240.3
Mote Only	117.9
Idle	358.2
One LED On	383.4
Erasing Flash	672.3
Sampling	358.2
Transferring Data	388.8

not provide compelling savings compared to its complexity and overhead, so duty-cycling is not used. The higher data volume requires either sophisticated on-board computation with a distributed system identification algorithm (which is expensive in terms of energy), or transmitting all the data to a base station for further processing (which is even more power-expensive). In the Golden Gate Bridge deployment, the latter option is chosen. The use of batteries or other renewable sources of energy is justified for quick and temporary applications, or where a more permanent power source cannot be provided. An analysis of the power consumption of the boards was performed to determine the size of the batteries. In the deployment at the Golden Gate Bridge, 4 lantern batteries are used for each node. Table 2.3 shows the actual power consumption profile of a complete sensor unit, from which it is seen that the sensor board by itself consumes about twice the energy of the mote. The board design had a single power path for the mote, sensors, and ADC. Significantly lower energy consumption could be realized if only the mote is directly connected to the battery, so that all other components can be turned off when the unit is not collecting data.

2.4.2 Calibration

The static noise floor of our accelerometer devices was quantified in the Berkeley Seismological Laboratory underground seismometer calibration vault. Testing showed that the SiliconDesigns 1221L devices have a noise floor of $32\mu\text{G}/\sqrt{Hz}$, which is small enough to allow resolution of the ambient vibrations of most structural systems. Examples of similar measurement systems in civil infrastructures can be found in [22]. Shaking table tests with patterns ranging from 0.5Hz to 8Hz were performed to study the dynamic behavior of the accelerometers. The accelerometers perform well within the expected dynamic range [68]. Each accelerometer channel was range-calibrated using a tilt test process. The boards were attached to a tilting machine [21], which has a rotational accuracy of 0.001 degree, and the digital output correlated to each angle tested. All four channels showed linear response, and the testing provided offset and scale factors. Prototype accelerometers were also tested in an oven to study the response of the devices to temperature. The tests showed that not only were the accelerometer chips sensitive to temperature, but also they are sensitive to the rate of temperature change as well demonstrating hysteretic response to external temperature fluctuations [47]. Calibration of each channel with respect to temperature is necessary for accurate results, especially when the temperature varies throughout the network.

2.5 High-frequency Sampling with Low Time Uncertainty (Requirement 2)

The fundamental frequencies of most civil structures lie below 10Hz. Since the noise level is usually high in uncontrolled structural environments, over-sampling is generally performed to improve the signal-to-noise ratio by reducing the relative

noise energy. A sampling rate of 200Hz was chosen as the target logged sampling rate for this study [68]. In order to allow on-the-fly digital filtering (smoothing), it was decided to digitize by a factor of five faster to a target-sampling rate of 1kHz. At this relatively high sampling rate, it is essential to cap the time uncertainty – jitter – in order to guarantee time synchronization in the node and across the network.

There are two primary sources for jitter: temporal jitter and spatial jitter (see Figure 2.5). Temporal jitter takes place inside a node because the software system cannot keep up with aggressive sampling and logging. Spatial jitter occurs between different nodes because of variation in mote oscillator crystals and imperfect time synchronization; internal clocks of different nodes in the network remain slightly untuned with each other even after the software time-sync component declares them to be in sync. For a target-sampling rate of 200Hz, a total jitter of 250μs or 5% of the sampling interval was selected as the cap to total jitter. A study of the time synchronization component FTSP showed that it caps jitter at 67μs over a fifty-nine node eleven-hop network [61], so spatial jitter in this case is within the tolerance range. Temporal jitter can become larger than spatial jitter during periods of high-speed data collection, so this was studied in detail. In particular, we explored and modeled temporal jitter, and show that our model matches measured data.

2.5.1 Temporal Jitter Analysis

A statistical model may not catch every minute detail of the temporal jitter process, but it will provide understanding of the distribution of temporal jitter. The timer event for sampling ticks at uniform intervals is graphically presented in the upper portion of Figure 2.6. When the timer event fires, the CPU can be in the middle of servicing other tasks, such as writing data from RAM to flash. When the CPU

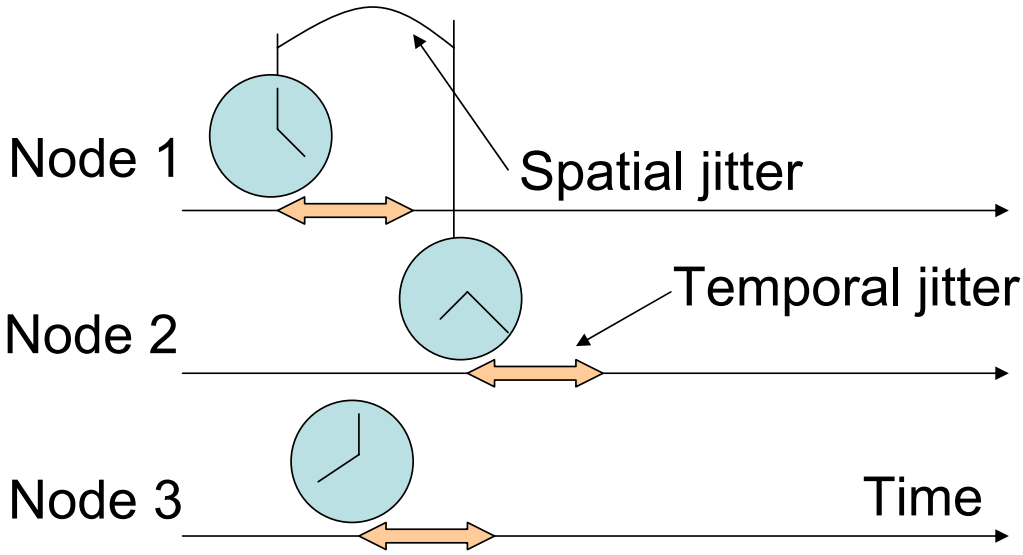

Figure 2.5. Sources of Jitter. Temporal jitter takes place inside a node because the software system cannot keep up with aggressive sampling and logging. Spatial jitter occurs between different nodes because of variation in mote oscillator crystals and imperfect time synchronization. Both temporal jitter and spatial jitter should be within a threshold for the data to have scientific value.

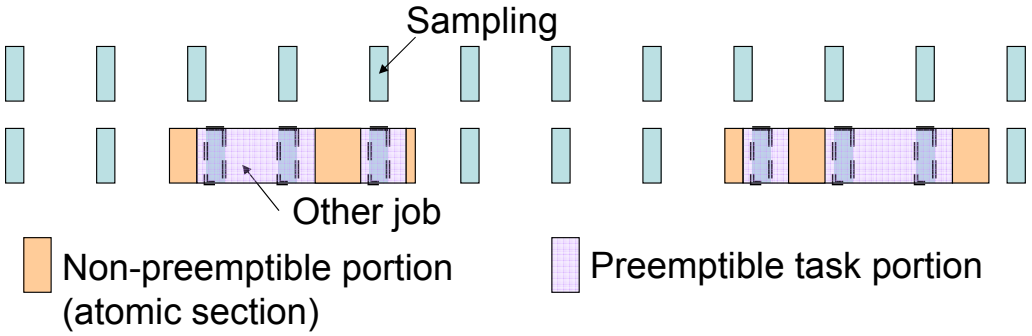

Figure 2.6. Causes of Temporal Jitter. The atomic section blocks and delays the task of sampling.

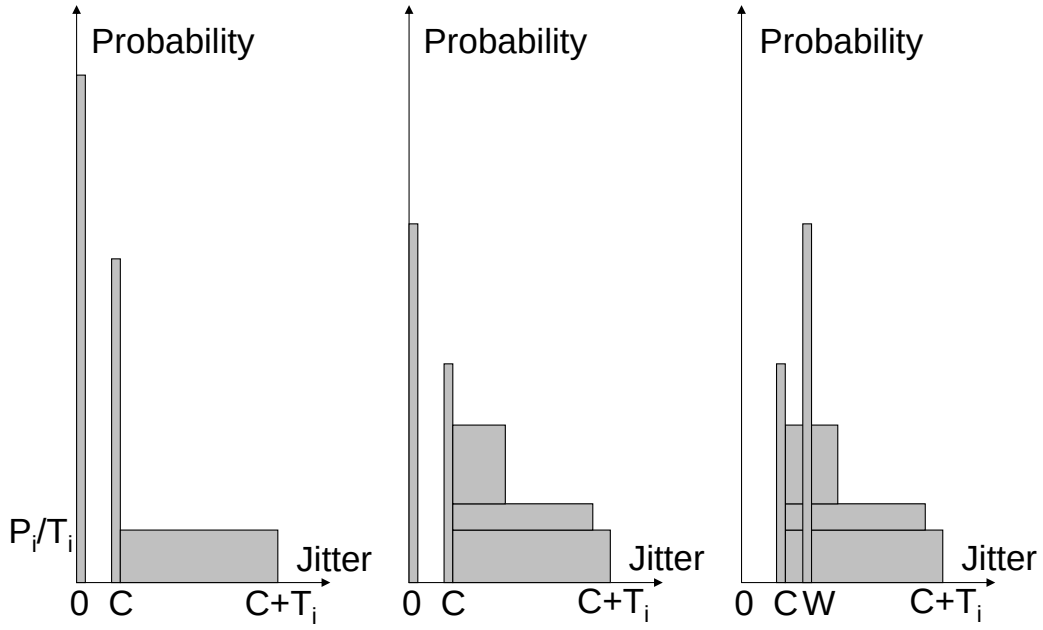

Figure 2.7. (Left) One Atomic Section, (Middle) Multiple Atomic Sections, (Right) Multiple Atomic Sections with CPU Sleep

is servicing an atomic section, the timer event is delayed, shown in the lower part of Figure 2.6.

Let N be the number of atomic sections and C be the context switch time when a timer event occurs while the CPU is executing a preemptible section. For modeling purposes, it is assumed that C is constant regardless of the code running. Furthermore let T_i be the length of atomic section i, P_i be the probability of atomic section i running on the CPU when a timer event occurs and $X(i)$ be a random variable which is the remaining execution time of atomic section i running on the CPU when a timer event happens. It is assumed that $X(i)$ is uniformly distributed in $[0,\ T_i]$. First, assume $N = 1$. The left graph in Figure 2.7 shows the distribution of jitter. The first peak at 0 indicates the case where no job is running on the CPU when a timer event occurs. The peak at C belongs to the case where preemptible code is running when the timer event occurs. The constant portion above C is the case where an atomic section i is running on the CPU when a timer event occurs. The middle graph

of Figure 2.7 shows the general case where $N > 1$. The right graph of Figure 2.7 incorporates the effect of CPU sleep; the CPU goes into sleep mode when no job is running. Let W be the wakeup time; then the peak at 0 moves to W. In fact the entire graph can be moved to the left by C, because consistent jitter of C can be removed.

2.5.2 Temporal Jitter Control

For high-frequency timer events, MicroTimer [1] is used instead of the Timer component [6]. The timer component of TinyOS can only trigger at 200Hz. While MicroTimer supports only a single trigger, it can trigger at the rate of at least 10kHz. The BufferedLog component [5] is used for light-weight flash writing at high frequency. It is clear from the jitter analysis in the previous subsection that the worst case of jitter is determined by the longest atomic section which can run on the CPU when the timer event occurs. This implies that the best way to reduce temporal jitter is to eliminate any chance that an unnecessary component's atomic section is running on the CPU by turning off every component except the flash during sampling.

A jitter test was performed by turning off all unnecessary components on the CPU. Figure 2.8 shows the time series of the jitter test. A 5μs jitter means the data is sampled 5μs later than it should be. Two sections are evident in these time series: a section where there is a significant variation in when the data is written (noisy) followed by a section when there is little variation (quiet). The noisy section is a result of writing the buffer to flash memory as a background task. The quiet sections are when sampling occurs without the interference of writing to the flash memory. The same test was performed for sampling rates of 1kHz, 2kHz, and 6.67kHz, with the jitter making up a higher proportion of the read cycle. At a 6.67kHz sampling rate, flash memory write affected most of the sampling period; this can be explained

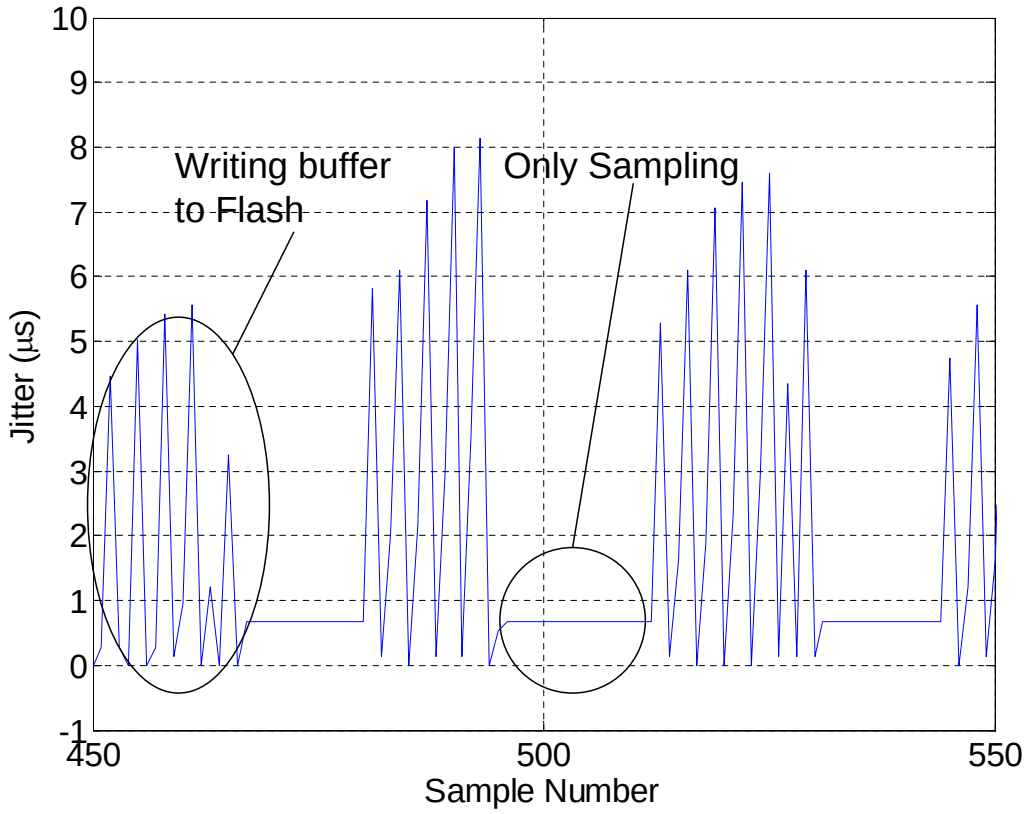

Figure 2.8. Time Series of Jitter at 5kHz Sampling Rate. Writing to flash interferes with the sampling.

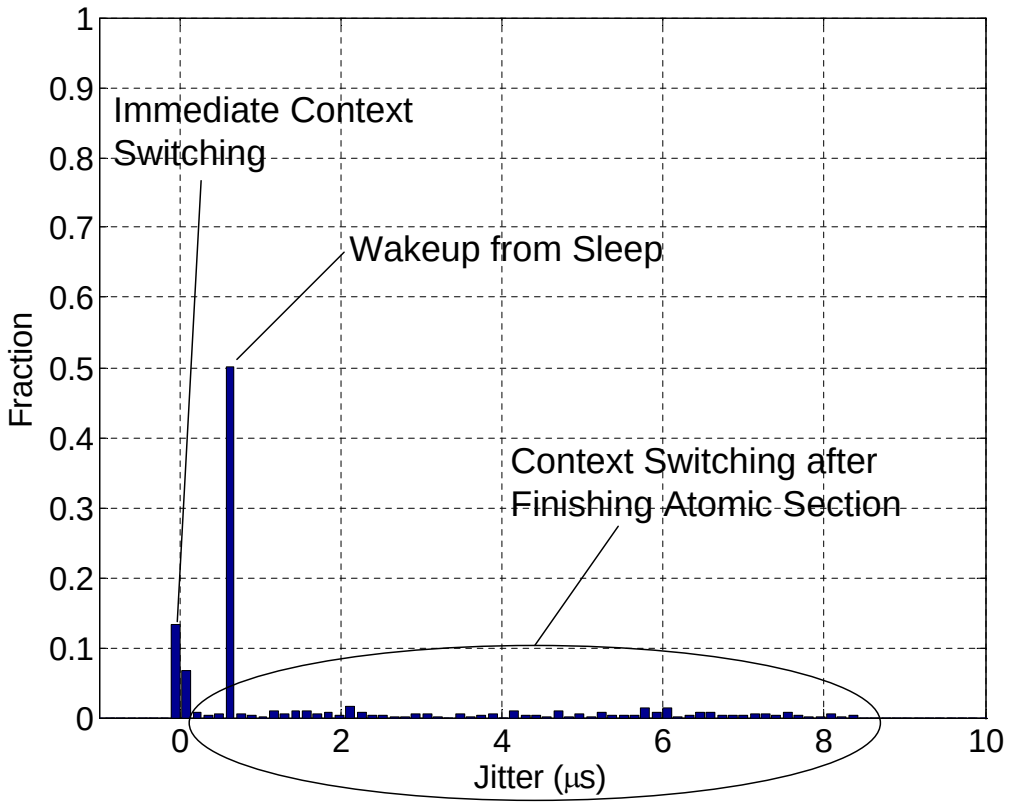

Figure 2.9. Histogram of Jitter at 5kHz Sampling Rate. Long and thin tail indicates that most samples experience small jitter, however a small number of samples still experience long jitter.

by the overhead of sampling itself. During the quiet sampling sections of the time-history plot, there is a constant delay for every sample. This delay is the time required to wake up the CPU between samples. When the CPU is idle, it enters a sleeping mode, and it takes five clock cycles to recover, including a function call to record the time, which adds to 625ns for the Mica2 and MicaZ. Figure 2.9 shows a histogram of sampling time uncertainty. There is an error peak at 625ns, which is wakeup time W, and another near 0s due to immediate context switching. The long but small tail shows that some sampling has a large time uncertainty. This result from the real experiment agrees with the theoretical model of the previous subsection, with temporal jitter values limited to about 10μs, which is smaller than the target value of 250μs.

2.6 Software Components (Requirements 3, 4, and 5)

As discussed in Section 1.2, three out of six requirements are handled by existing works. In this section, those existing software components will be introduced.

For time synchronization, there are a few existing works (e.g. RBS [31], TPSN [34], FTSP [61]). In this work, the flooding time synchronization protocol (FTSP) [61] is used, because it provides the smallest error and is proven in a real application [76]. In FTSP, each node sends a packet with a time stamp. A receiver looks at the time stamp and adjusts its own clock to be in sync with the received time stamp. At a sender, a time stamp is put on a packet at a MAC layer after it obtains access to a channel. This reduces uncertainty in the time of flight by removing MAC delay. The packet exchange process repeats continuously to overcome clock drift due to variations of crystals. However, a receiver adjusts its clock only when the received packet has a smaller root ID and a larger sequence number. This guarantees all nodes eventually synchronize to a single synchronization root, and keeps the clock updated continuously. In multi-hop time synchronization, multiple packets from multiple senders may not match exactly, so a linear regression is used.

There are diverse multi-hop routing protocols (e.g. BVR [33], VRR [26], CLDP [49], MintRoute [86]). Command dissemination and data collection uses a routing protocol. Any-to-any routing provides an efficient command dissemination but has a higher overall cost. When collection routing is used, which can only deliver a packet from nodes to a single sink, data collection overhead is small. However command dissemination has to rely on flooding. Since command dissemination is not a frequent event (which is proven to be true by the data from the real deployment, see Section 5.2), a collection routing layer can reduce the total cost, and is used in

this work. MintRoute [86] is used because it is the most widely used and the most extensively proven in the field. MintRoute forms a tree rooted at a sink. To form a tree, each node broadcasts the expected cost to deliver a packet to the sink. At the same time, each node listens for candidates for a parent, and chooses a node with the least cost, which is the sum of the cost from the candidate and the cost of a link to that candidate. This process repeats continuously to adapt to a changing environment. Packets are forwarded along a branch in a tree toward a root, which is the sink.

Deluge [43] and Drip [80] are examples of reliable data dissemination. Deluge is designed for disseminating a large object, and therefore is not efficient for disseminating a small command packet. A low latency dissemination service is required so that the commands are delivered in a timely manner, therefore Broadcast [4] was used in place of Drip [80]. Drip provides dissemination service with eventual reliability but has long latency. Even though Broadcast provides unreliable dissemination service, with repeated broadcast 100% eventual reliability can be achieved in practice.

Chapter 3

Design Space of Reliable Transfer (Requirement 6)

Maintaining reliable communications over an extended path (e.g. 46 hops in the Golden Gate Bridge) is in itself a challenging problem due to the large round trip time and a high loss rate. In SHM applications, an added imperative is that no data can be lost in the system since these events happen rarely and cannot be duplicated for unique events such as an earthquake. The goal is to have reliable and lossless communications over a large network with minimal overhead for other network components. The two principal aspects of such a protocol are channel capacity and scalability over a large-scale multi-hop network. It is also important to minimize usage of network resources, because wireless sensor nodes are limited in computational power, memory space, and energy.

In this chapter, design space of reliable transfer will be examined, and design options will be evaluated. At the end, our initial solution to this problem, Straw, is presented. Experiences with Straw at the Golden Gate Bridge show that a pessimistic

static rate underutilizes channel capacity (Chapter 5). To overcome this pitfall, in Chapter 6, we will present a reliable data collection service, Flush.

Many applications in Wireless Sensor Networks, including structure monitoring, require collecting all data without loss from nodes. End-to-end retransmissions, such as TCP/IP uses in the Internet for reliable transport, can become very inefficient in Wireless Sensor Networks, since wireless communication and constrained resources pose new challenges including interference and a limited buffer space. We look at factors affecting reliability and search for efficient combinations of the possible options. Information redundancy, like retransmission and erasure code, can be used. Route fixing (dynamic rerouting), which tries an alternative next hop after some failures, also reduces packet loss. We implemented and evaluated these options on a real testbed of Berkeley Mica2Dot motes. Our experimental results show that each option overcomes different kinds of failures. Link-level retransmission is efficient but limited in achieving reliability. Erasure code enables very high reliability by tolerating packet losses at a prepaid static cost. Route fixing responds to link failures quickly. Previous work had found it difficult to increase reliability past a certain threshold. We show that the right combination of primitives can yield more than 99% reliability with low overhead.

3.1 Three Options for Reliable Transfer

Challenges to achieving reliability on Wireless Sensor Networks can be divided into three main categories. The first are problems related to wireless communication [86, 89]. The asymmetry of links makes link quality estimation hard. Correlated losses, due to obstacles, interference, and highly correlated traffic patterns can lead to consecutive losses, decreasing the effectiveness of erasure code as we will see later.

Weak correlation between quality and distance, hidden terminal problems, and dynamic change of connectivity complicates the situation further [69].

The second sort of problems comes from the constrained resources of Wireless Sensor Network motes. A mote is often battery powered, so it has a limited power source. Not to deplete energy too quickly, an algorithm needs to be computationally inexpensive. A mote also has small computational power and memory space. Again, a computationally expensive algorithm cannot be run on a mote. If an algorithm requires large memory space, the memory footprint will not fit into a mote. Furthermore, its communication bandwidth is narrow. Therefore the algorithms running on motes should limit traffic generation.

Finally, from a software engineering standpoint, diverse routing layers (e.g. BVR [33], VRR [26], CLDP [49], MintRout [86]) add more challenges. Since motes are resource constrained, applications tend to make heavy use of customization and cross-layer optimizations [53]. Therefore, there are different routing layers customized for specific purposes: even if we can use a general purpose, point-to-point routing for dissemination of information or collection of data, this approach is very inefficient for some specific cases. For collecting data (convergence routing), each node only needs to keep track of which nodes are candidates for its parent. This reduces the burden of keeping additional information to support routing to any node. Dissemination of information, such as code image distribution, is similar to multicast (divergence routing). In this case, we can benefit from the broadcast nature of wireless communication. By injecting one packet into the channel, all neighboring nodes can hear the packet. Compared to sending packets to each single receiver, this can save a huge effort.

There are three main routing categories: *point-to-point routing*, *convergence routing*, and *divergence routing*. One transport layer may not work for all three cases well.

But it is not a good idea to keep three separate versions of reliable transfer either. At least it will be desirable to share some components if possible, regardless of location in the network stack. Ultimately, we seek to find common reliability primitives or principles that can be used even in different routing layers. In this chapter, we examine diverse options for improving reliability over multiple hops, focusing mainly on point-to-point routing. However, the study should be applicable to other routing patterns without significant difficulty. One possible drawback for collection routing is that redundancy can decrease bandwidth. Since the root is a bottleneck in collection routing, when a channel at the root is fully utilized, added redundancy will reduce effective goodput. First of all, it is worthwhile looking at fundamental factors that determine reliability. Then we look at possible options which improve each factor. Let us simplistically look at the following equation

$$\text{number of packets received} = P_{success} \times \text{number of packets sent}$$

The primary goal is to increase 'number of packets received' sufficiently so that we can get all data, at the same time making the delivered set as close as what we want as the secondary goal. Even though it is also important which packets are received, as we will see later, the basic limitation is delivering a sufficient amount of packets. This in turn amounts to increasing either 'number of packets sent' or increasing the probability to get through '$P_{success}$'. Let us take a look at diverse options to increase reliability: end-to-end retransmission, link-level retransmission, erasure code, thick paths, and alternative routes, a subset of which are further examined later. Let us first see options where adding redundancy to information increases the *number of packets sent*.

One option is retransmission. End-to-end retransmission is used in TCP on the Internet [45]. The cost of this option is that when a packet is lost, it is retransmitted from end to end. The retransmission increases the latency of a data transfer. This

option overcomes random losses and a transient link failure. However, when the link failure is prolonged, this option cannot recover losses efficiently.

A link-level retransmission may be used on intermediate links where loss rate is high in wireless communication. Cost of link-level retransmission is very low; it is the minimum of an effort required to recover a loss. However, this option can survive only random losses. For example, a transient link failure, if longer than the retransmission trial period, cannot be overcome by this option.

Adding redundant data is also an option. Sending an additional parity packet for some number of previous packets is a good example. An erasure code can be thought as a generalization of a parity code. Rather than sending one additional packet, an erasure code can send multiple additional packets. In a parity packet case, any M out of $M + 1$ packets will reconstruct original M data. Likewise, an erasure code enables a reconstruction of M original data packets if any M out of $M + R$ packets are received. In Figure 3.1, $S1$ is sending data with an erasure code. As a cost, $\frac{M+R}{M}$ times more packets need be sent. Random losses and transient link failures can be survived, since this option can recover from several packet losses. However, long correlated losses are problematic, as will be discussed later.

Alternatively we can also exploit spatial redundancy along the path. As $S2$ in Figure 3.1, a 'thick path' can be used as in [60]. Every node within the nearby area along the path will participate in transferring data. This method adds in-network data redundancy. The number of messages injected to a network increases dramatically by a factor of the width of a path. However, the wide range of a failure, even a permanent link failure, can be survived without increasing the latency unless there is a network outage over a wide area.

In general, we can provide redundancy where losses occur. Increasing the probability of successful delivery and changing the loss distribution can alleviate problems

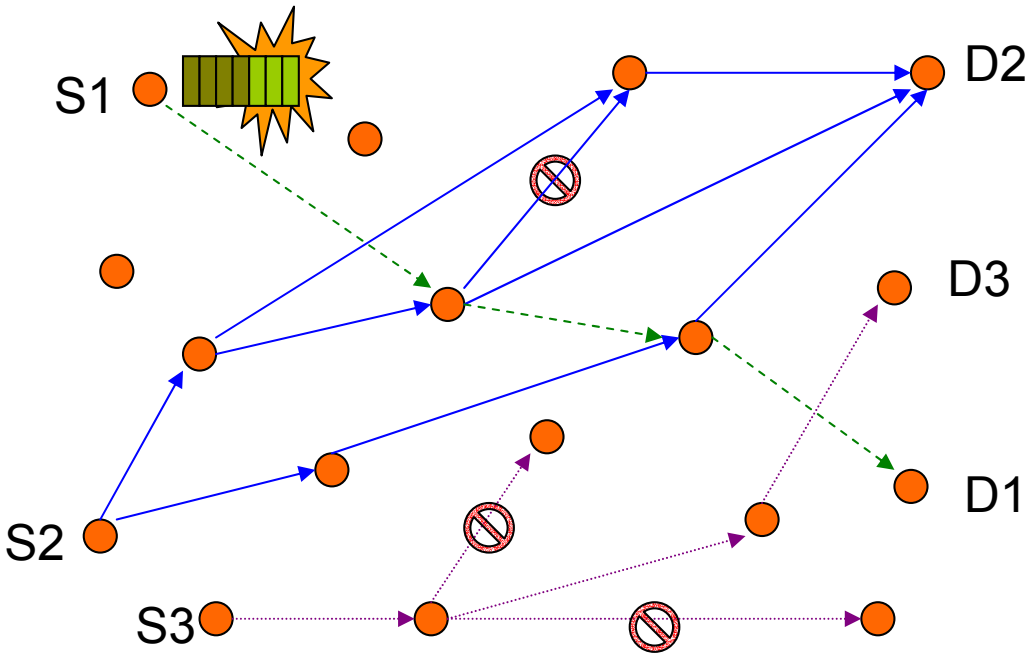

Figure 3.1. Possible options to achieve reliability. S1 uses erasure code producing additional code words, S2 uses thick (multiple) path which is not examined in this work, and S3 does route fixing, finding alternative for the next hop when stuck.

which are hard to overcome by redundancy (*number of packets sent*) alone. For example, link failures in a certain region cannot be handled by redundancy in a short time frame, if all redundant packets go through that area. Let us assume $P_{success}$ is not randomly distributed, which is also what we are trying to change. An erasure code can survive up to R losses. When consecutive $R+1$ or more packets are lost, an erasure code is unable to reconstruct the original data. This phenomenon happens in wireless communication. For example, after a link failure, it takes time for the routing table to be updated. Until then, all packets sent to that link will fail, introducing consecutive failures.

In the situation above, we can quickly try an alternative next hop. This is shown as $S3$ in Figure 3.1. The cost is low since packets are sent to an alternative next hop only when needed. However, alternative next hops need be identified and continuously updated. Correlated losses and link failures can be overcome by this option.

In this chapter, after an introduction of related works in Section 3.2, we look at link-level retransmissions (Section 3.3), erasure code (Section 3.4, 3.5), and alternative routes (Section 3.6). Other possible options like thick path and end-to-end retransmission remain as future work. We examine several options on a real-world testbed. We provide results in Section 3.7. After a discussion in Section 3.8, we then see which options and which combinations thereof are good choices in Section 3.9. Based on lessons, an initial reliable data collection protocol, Straw, is presented in Section 3.10.

3.2 Related Work

There are many algorithms proposed and implemented for multi-hop communications in sensor networks (e.g. [33, 35, 43, 49, 54, 58, 86, 88]) , and as noted these can be broadly divided into convergence, divergence, and point-to-point. Our work is largely orthogonal to these routing implementations, as we examine techniques that can be employed to varying degrees in most multihop routing schemes. However, we show that it is a good feature of a routing algorithm to provide alternative next hops towards a destination.

Previous work has been done in reliable transport for sensor networks. PSFQ [83] examines the problem of retasking a sensor network (an example of divergence) reliably, and make use of hop by hop recovery with caching at intermediate nodes, as opposed to end-to-end recovery. RMST [78] investigates through simulation the tradeoff between having reliability implemented at the MAC, transport, and application layers. Both works conclude that hop by hop recovery is very important for achieving reliability and that end-to-end recovery is not adequate. They only consider different retransmission/repair options and use simulated data. Our contribution to their findings is the addition of the very effective options of erasure coding and al-

ternate routes for providing reliability, as well as examining the interaction of these different mechanisms. We also use real implementation of the options on a testbed of wireless motes, which allows us to see the effect of the radio environment on the reliability.

There exist diverse algorithms for erasure coding which can be implemented in either software or hardware [24, 74]. [74] exploits diverse optimizations such as extension fields, systematic code and an operation table, from which this work gained many hints. It is an excellent introduction to Reed-Solomon codes, but focuses on the implementation in desktop computers. We leverage many of its optimizations, carefully choosing parameters suitable for very resource constrained WSNs. Rateless codes [25, 62] is a class of erasure code in which an arbitrary number of code words can be produced, and is optimized for delivery of very large amounts of data over high-bandwidth, high-latency Internet links. These works are not optimized for systems with low capability: not much attention was paid to cases with extreme space limitation. Work in this chapter focuses on optimization for nodes with very limited resources.

3.3 Link-level Retransmissions (Option 1)

The loss rate on wireless links is much higher than that of wired links, and this effect accumulates quickly as the number of hops increases. For example, let us assume that loss rate is 10% per hop and uniformly distributed. After 15 hops loss rate becomes 80%! If a message is lost at the n^{th} hop, all previous $n-1$ transfers become wasted effort. To deliver the packet to n^{th} hop again, we need $n-1$ additional transfers if all $n-1$ transfers succeed. With link-level retransmission, just one retransmission can bring a packet to the same point. For efficient use of the wireless channel, link-level retransmissions are a very attractive choice.

There are drawbacks to link-level retransmissions when used in some specific contexts. When retransmission is implemented with link-level acknowledgments, there is a decrease in channel utilization. This has been measured to be as high as 20% in TinyOS.[1] This overhead can, however, be mitigated in some contexts by using techniques such as passive acknowledgments, in which the next hop transmission is interpreted as an acknowledgment. Another minor downside is that the intermediate node needs to hold the packets in a buffer until it receives acknowledgment from the next hop. Lastly, the delivery time depends on the number of retransmissions along the route, so the end-to-end round trip time (RTT) can vary significantly. A large variation in RTT makes an end-to-end retransmission inefficient. If there can be up to N link-level retransmissions, RTT can be N times larger. To overcome the variation in RTT an (overestimated) upper bound needs to be used. The sender holds its buffer for a longer time than necessary. Holding memory space for a long time is not desirable in resource-constrained Wireless Sensor Networks.

3.4 Erasure Code (Option 2)

Another important mechanism we employ is erasure coding. It is a scheme with which we can reconstruct m original messages by receiving any m out of n code words ($n > m$). If n is sufficiently large compared to the loss rate, we can achieve high reliability without retransmission. Figure 3.2 shows the high-level mechanism of erasure code. We use a particular erasure coding algorithm, Reed-Solomon coding. Before we explain the Reed-Solomon code, we first introduce linear codes and Vandermonde matrices.

[1]The MAC layer in TinyOS waits 20% of a packet transmission time before considering the transmission successful.

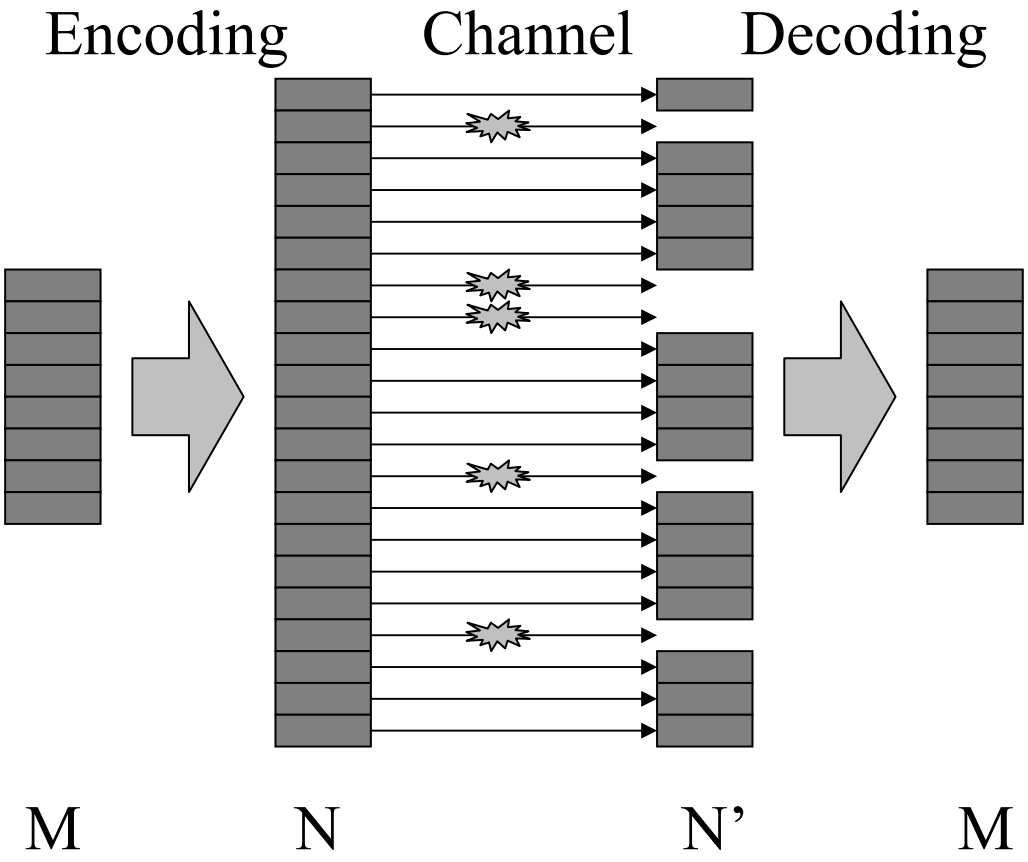

Figure 3.2. Mechanism of Erasure Code

3.4.1 Linear Code

For the encoding process, an encoding function $C(X)$ is used, where X is a vector of m messages. $C(X)$ produces a vector of n code words ($n > m$). If the code has the property that $C(X)+C(Y) = C(X+Y)$, then it is called a linear code. Linear codes can be represented by a matrix A, and encoding can be represented by a matrix-vector multiplication: the code word vector Z for message vector X is simply AX. Decoding entails finding X such that $AX = Z$, for a received code word vector Z, *i.e.* finding the solution to the linear equation $AX = Z$. We can see that A should have m linearly independent rows so that the linear equation has a unique solution, which represents a unique message vector.

This code is very useful in practice, since encoding and decoding are computationally inexpensive, and this is especially so in resource-constrained Wireless Sensor Networks.

3.4.2 Vandermonde Matrix

There is one more thing we need to look at before describing Reed-Solomon codes. A Vandermonde matrix is a matrix with elements $A(i, j) = x_i^{j-1}$, where each x_i is nonzero and distinct from each other, as shown in Figure 3.3. For an n *by* m Vandermonde matrix ($n > m$), any set of m rows forms a non-singular matrix. For whatever set with m rows we may choose, rows in the set are linearly independent. Let's define for future reference this property as *Property V*.

Definition (*Property V*): For an n *by* m ($n > m$) matrix A, if any set S of m rows of A form non-singular matrix such that all rows in S are linearly independent, then A is said to have *Property V*.

Vandermonde matrices have *Property V*. So we can see that in a linear equation

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{m-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{m-1} \\ \vdots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{m-1} \\ 1 & x_n & x_n^2 & \cdots & x_n^{m-1} \end{pmatrix}$$

Figure 3.3. Vandermonde Matrix

$AX = Z$, where A is a Vandermonde matrix, any m rows and corresponding m elements of Z form an m *by* m square matrix and a vector of size m, where the matrix is non-singular. Then we can uniquely determine X. This is a key property used in our implementation of the Reed-Solomon code.

3.4.3 Reed-Solomon Code

The basic idea of Reed-Solomon code is to produce n equations with m unknown variables ($n > m$) such that with any m out of n equations, we can find those m unknowns.

For some given data, let us break it down into m messages $w_0, w_1, w_2, \ldots, w_{m-1}$, and construct the polynomial $P(X)$ using these messages as coefficients, such that

$$P(X) = \sum_{i=0}^{m-1} w_i x^i$$

We then evaluate this polynomial $P(X)$ at n different points $x_1, x_2, , x_n$. $P(x_1), P(x_2), ..., P(x_n)$ can be represented as multiplication of a matrix and a vector, as shown in Figure 3.4.

$$\begin{pmatrix} 1 & x_1 & \cdots & x_1^{m-1} \\ 1 & x_2 & \cdots & x_2^{m-1} \\ 1 & x_3 & \cdots & x_3^{m-1} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ 1 & x_{n-1} & \cdots & x_{n-1}^{m-1} \\ 1 & x_n & \cdots & x_n^{m-1} \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{m-1} \end{pmatrix} = \begin{pmatrix} p(x_1) \\ p(x_2) \\ p(x_3) \\ \vdots \\ \vdots \\ p(x_{n-1}) \\ p(x_n) \end{pmatrix}$$

Figure 3.4. High level diagram showing how Reed-Solomon code works

Here we can see that matrix A is a Vandermonde matrix, W is a vector of messages, and code words are contained in a vector AW. If we have any m rows of A and their corresponding $P(X)$ values, we can obtain the vector W which contains coefficients of the polynomial, which is again the original messages. Reed-Solomon codes can also be used to correct errors. However, in the current implementation of TinyOS, each packet has a CRC to detect bit errors. We can assume that there will be no bit errors in packets containing code words, as these are dropped by the lower layers. Therefore, error correction is not used in the implementation.

3.5 Modifications for Wireless Sensor Networks (Option 2)

There are modifications needed to bring erasure codes to a real world implementation, especially in resource-constrained Wireless Sensor Networks (WSN). Several methods used to improve efficiency in motes are heavily borrowed from [74]. We need an efficient representation of the data and efficient and precise operations, including vector arithmetic and matrix inversions. Fortunately, these can be made very efficient

with modular operations on finite fields and clever lookup tables, which we discuss next.

3.5.1 Extension Fields

To make efficient use of bits in the packet, maintain precision, and reduce computational effort, we do all calculations in an *extension field* with base 2. We briefly introduce fields, and prime fields. A field [10] is any set of elements with two operations, addition and multiplication, that satisfies the *field axioms* – commutativity, associativity, distributivity, identity, and inverses – for both operations. Every nonzero element has an inverse.

A field with a finite number of members is known as a finite field or Galois field. For a given Galois field of size q, if $q-1$ powers of an element x $(x^1, x^2, \ldots, x^{q-1})$ produce all non-zero elements, that element x is called a generator of the given Galois field.

A *prime field* is a Galois field whose elements are integers in $[0, p-1]$, where p is prime. Addition and multiplication are normal integer addition and multiplication with a modulo operation at the end. A prime field always have a generator. The size of a prime field is p, and we need $\lceil \log_2(p) \rceil$ bits to represent all elements. Since p is not a power of 2, there is waste in bit usage. For example, to represent a prime field with prime 11, we need 4 bits with which 16 numbers can be represented.

An *extension field* is a Galois field whose elements are integers in $[0, p^r - 1]$. Extension fields can be though as polynomials on prime field(p). Operations follow the rules of polynomial operation with modulo operation at the end. A primitive polynomial is the generator of extension field. Interestingly, this set with polynomial operations stated above still satisfies the properties of fields. Moreover, by setting

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \\ 1 & x_{m+1} & \cdots & x_{m+1}^{m-1} \\ 1 & x_{m+2} & \cdots & x_{m+2}^{m-1} \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \cdots & x_n^{m-1} \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{m-1} \end{pmatrix} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{m-1} \\ p(x_{m+1}) \\ p(x_{m+2}) \\ \vdots \\ p(x_n) \end{pmatrix}$$

Figure 3.5. Systematic code

$p = 2$, we can fully utilize bits in the message. *Property V* of Vandermonde matrices still holds for prime field, and even for extension fields.

3.5.2 Systematic Code

When coding a message, if part of the encoded message is the original message itself, it is possible to recover the original message without decoding, in the event that this part arrives intact. Codes with this property are called systematic codes. Another good property of Vandermonde matrices is that if any m rows of an n *by* m ($n > m$) Vandermonde matrix are substituted with rows of the m *by* m identity matrix, the new matrix still has *Property V*, even for extension fields. Figure 3.5 shows one possible case. This matrix will clearly produce a systematic code, as m of the code words will be the original message. When we use a systematic code in this way, at the encoding side, we don't need any computation for the portion of code words containing original messages. Systematic codes can give a benefit even when we lose some packets. At the decoding side, the more of the original message part

we have, the closer the decoding matrix is to the identity matrix, and the quicker the decoding process becomes.

3.5.3 Multiple Independent Code Words in a Packet

If one packet carries one code word, each code word will be very large. This makes the implementation intractable since operations on such a large field require huge space and time. One solution would be to use small messages and small code words. Then, however, the payload in a packet gets too small. By putting multiple independent code words into a packet, we can fully utilize payload space of a packet without problems of large code words.

Imagine dividing one big block of data into t small pieces of data. Then each piece of data is again divided into m messages, and encoded into n code words. We have a total of tn code words to send. Pack the i^{th} code words from each independent k data into a single packet. We either get all the i^{th} code words for k data, or we get nothing. Any m packets will provide m code words for all k data, and we can reconstruct the original k data. Since all k data have code words with same sequence set, the decoding process is the same; the same decoding matrix can be used. This further enhances decoding efficiency by amortizing the matrix inversion cost over k data packets. Figure 3.6 shows an example of this. Here data is divided into 6 small data chunks. Each data chunk is divided again to 4 messages. Messages from each chunk are encoded to 7 code words independently. Then code words from all data chunks with same sequence number are packed into the same packet.

The drawback of dividing packets into multiple code words is the constraints on the number of messages and code words. The number of messages can not exceed the number of bits used to represent the message. The number of code words should be smaller than the size of the extension field. For example, if each code word is 8 bits

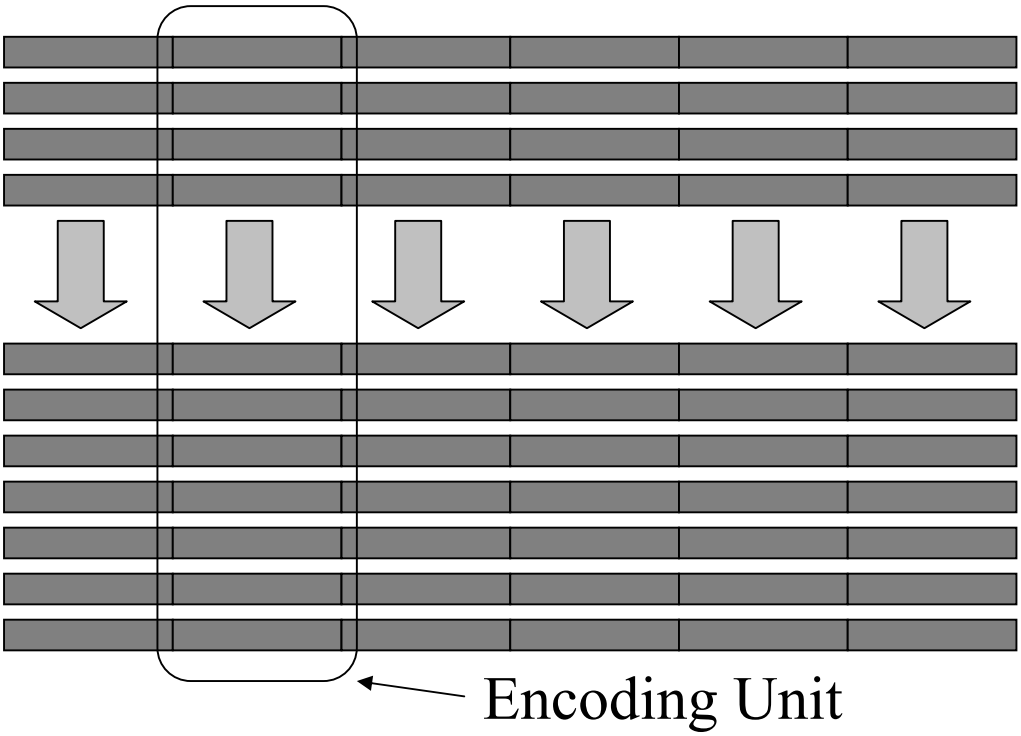

Figure 3.6. Divide packet into multiple independent code words

long, the maximum number of messages is limited to 8, and the maximum number of code words is limited to 255.

3.5.4 Operation Table

Operations on extension fields are not simply addition and multiplication combined with a modulo operation. They are polynomial operations with modulo. Therefore, rather than performing complex computation on the fly, we use lookup tables. Addition is simply the XOR of two numbers, and we don't need a table. For multiplication and division, exponent and log values are computed and stored as tables.

Let the size of the extension field be $q = p^r$, where p is prime. The extension field has generators. Let one of them be α. For any generator α, when we keep multiplying α, we can produce all $q-1$ non-zero elements of the field. Then α is produced again,

restarting the cycle. That means that

$$\alpha^q \ mod \ q = \alpha, \ \alpha^{q-1} \ mod \ q = 1.$$

Let

$$x = \alpha^{k_x} \ mod \ q, \ y = \alpha^{k_y} \ mod \ q.$$

Exponent and log are defined as follows:

$$exp(k_x) = x, \ log(x) = k_x \ where \ x, k_x \in GF(p^r).$$

Then multiplication of xy is

$$xy \ mod \ q = \alpha^{k_x}\alpha^{k_y} \ mod \ q = \alpha^{k_x+k_y} \ mod \ q$$

$$= \alpha^{k_x+k_y \ mod \ (q-1)} \ mod \ q = exp(k_x + k_y \ mod \ (q-1))$$

$$= exp((log(x) + log(y)) \ mod \ (q-1)),$$

and the inverse of x is

$$\frac{1}{x} = \alpha^{-k_x} = \alpha^{q-1-k_x} = exp(q-1-k_x)$$

$$= exp(q-1-log(x)).$$

Therefore, multiplication involves two log table lookups, one addition, one modulo, and one exponent table lookup. Inverse involves one log table lookup, one subtraction, and one exponent table lookup, making these operations quite efficient. The size of the tables is an important parameter when choosing the size of the extension field: it is 2^q. For current sensor networks, this means that extension fields of size 4 or 8 are good choices, but 16 is probably too large, as the lookup tables will require 64K entries.

3.6 Alternative Routes (Option 3)

Adding an alternative route in the case of the failure of a given link is yet another way to increase reliability. When a link between two nodes fails, the messages sent through that link will successively be dropped, until the link estimation component is triggered and selects a new route. This process, if prevalent, can eliminate the benefits gained from erasure coding, since many consecutive losses will very likely be above the tolerance of redundancy added by erasure code. In this case, it should be clear that link-level retransmissions are of high temporal correlation, unless used for a prohibitively long extent. A sensible strategy, then, is to detect the failure as soon as possible, and send the packet to an alternative route, if possible. As a contrast to previous options being oblivious to the source of losses, this approach avoids the source of losses.

This points to the need of special support from the routing layer for enabling alternative paths towards the destination. This flexibility ultimately depends on the routing geometry of the routing algorithm [39]. For example, in the case of aggregation, in which nodes route to a parent in the tree to the root, there may be many nodes within communication range that decrease the distance to the root. In geographic routing, there may also be more than one neighbor that allows progress towards the destination. In our evaluation we use an implementation of Beacon Vector Routing (BVR) [33]. We describe the algorithm in some detail in Section 3.7, but for now it suffices to say that it allows flexibility in the selection of routes. We stress the point that using alternative routes is not particular to BVR, and that our findings in this regard can be reproduced in many other routing disciplines.

Sending packets in alternate routes can be seen as a type of retransmission to a different node, and so in effect it increases the number of packets injected into the network.

3.7 Evaluation

We implemented and evaluated the different reliability options described in previous sections – link level retransmissions, erasure coding, and alternative routes – in the context of Beacon Vector Routing. We briefly introduce our experiment methodology, followed by our results.

3.7.1 Experiment Methodology

As hardware platforms for our evaluation, Mica2 [13] and Mica2Dot [14] are used. In the evaluation of erasure code (Subsection 3.7.2), the Mica2 mote is used. For comparing options (Subsection 3.7.3), the Mica2Dot testbed in Soda hall [16] is used. TinyOS [42] is used as a software platform in all evaluations. TinyOS is a de facto standard in Wireless Sensor Networks.

In our experimental evaluation, we use an implementation of Beacon Vector Routing, a point-to-point routing algorithm for wireless sensor networks. For the purpose of our evaluation, it is not necessary to describe the routing algorithm in much detail, except for its aspects that provide flexibility in selecting routes.

BVR assigns virtual coordinates to nodes, derived solely from the network connectivity information. A subset of r nodes is selected as "beacons", and these beacons flood the network at least once, so that all nodes learn their distance to the set of beacons. The beacons act as reference points for routing. A node p's coordinates are then given by $\mathcal{P}(p) = (B_{1p}, B_{2p}, ..., B_{rp})$, where B_{ip} is the distance between p and B_i. Each node in BVR is required to know its distance to each of the beacons, and the coordinates of its one-hop neighbors.

The basic routing exported by BVR is a route-to-coordinate interface. Routing in BVR is a form of greedy routing, similar to the routing used in geographic routing

algorithms. When given a packet to route to a coordinate, a node selects the neighbor whose coordinates are the closest to the destination's coordinates, by some distance metric. The simplest such metric is given by Equation 3.1 below, and is equal to the sum of the absolute component-wise differences of the two coordinates (a form of an $L1$ metric).

$$\delta(\mathcal{P}(p), \mathcal{P}(q)) = \sum_{i=1}^{r} |B_{ip} - B_{iq}|, \tag{3.1}$$

This greedy-routing procedure may fail when no neighbor makes progress in the coordinate space towards the destination. To get out of these 'local minima', BVR employs a fallback routing mode that ultimately guarantees that the destination will be reached. In fallback mode, the node forwards the packet towards the beacon that is closest to the destination. This beacon is readily determined by the smallest component of the destination's coordinates. The minimum distance reached by the packet is recorded in the packet; this allows each node to resume normal greedy routing when one of the neighbors makes progress. Eventually, a packet may reach the beacon which is closest to the destination. In this situation, normal greedy routing cannot be used without the guarantee of no loops. The beacon then initiates a scoped flood that will reach the destination. The choice of the fallback beacon as the closest to the destination minimizes the flood scope.

We can now explain how in BVR one can get flexibility for choosing next hops. At each step of greedy routing, there may be many nodes which make progress in coordinate space to the destination. Also, when doing fallback-mode routing, any node that is reachable and is closer to the desirable root is good to be used. In the BVR implementation, we fix the maximum number of alternative routes to 6, and the routing layer returns these alternatives ordered by progress and link quality. Two critical features of BVR related to this work are a link-level retransmission and

maintaining alternative routes. Other routing layers, which also provides those two features (e.g. MintRoute [86]), would have a similar result.

3.7.2 Erasure Code

To analyze how much gain in reliability can be obtained by an erasure code, a statistical model is used assuming losses are randomly distributed. Given a loss rate, the expected probability to receive a certain number of packets can be calculated statistically. For each case with a certain number of packets received, it can be determined whether an original message can be reconstructed or not. Using the distribution of the number of packets received, the final loss rate of an erasure code is calculated statistically. Figure 3.7 shows how much reliability can be gained from an erasure code. Again, this graph is analytically obtained assuming loss is randomly distributed for different redundancy levels. The X-axis represents a raw end-to-end loss rate provided by a routing layer. Y-axis is the final loss rate after an erasure code is used. The bandwidth consumed by additional code words is not shown: more redundant code words achieve higher reliability, but more bandwidth will be consumed also. When there is a small amount of redundancy, the loss rate is higher than the raw loss rate. This happens because when we can not decode, we lose everything. So receiving 7 packets is effectively the same as receiving 0 packets. Systematic code is good not only for saving computation, but also for increasing reliability. By using a systematic code, even if we receive 7 packets, when 3 packets are codes containing original messages, we get 3 packets.

Figure 3.8 shows the improvement with systematic code. The graph is also obtained analytically with a statistical modeling, just like Figure 3.7. The final loss rate is always smaller than the raw loss rate. All the following tests use systematic code.

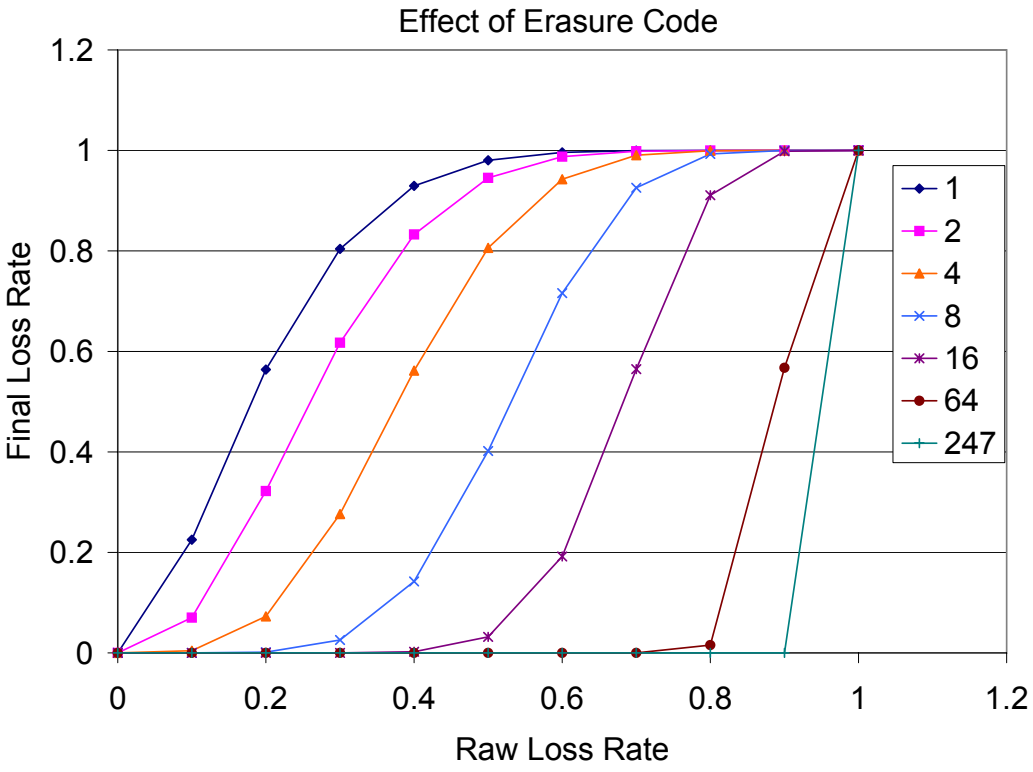

Figure 3.7. Increase or decrease in loss rate by using erasure code. Each line indicates how many redundant erasure code words are added to 8 original messages

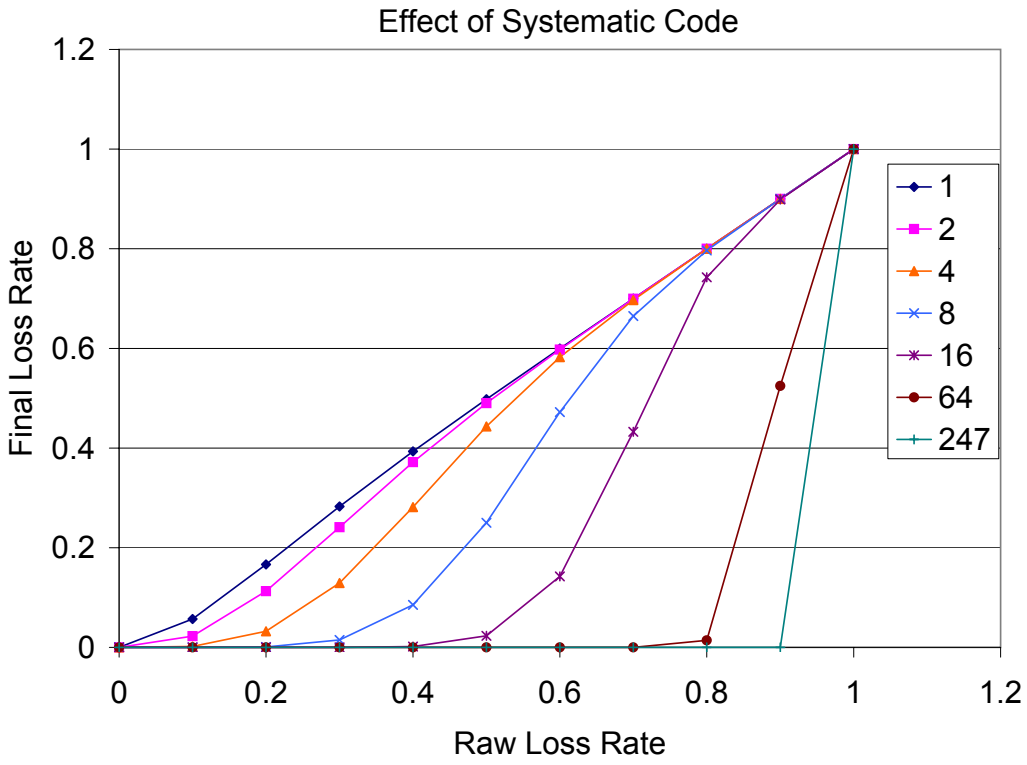

Figure 3.8. Decrease in loss rate by systematic code. Each line indicates how many redundant systematic code words are added to 8 original messages

Later in this section, we examine the trade-off between reliability and bandwidth overhead by varying the number of additional code words.

We measured the encoding and decoding speed of our implementation of an erasure code on Mica2 motes. Messages and code words are 29 bytes long. Message and code words are divided into 8 bit-long units, and there are 8 original messages to send.

Table 3.1 shows the encoding time. In systematic codes, the first 8 code words do not require any computation, they are just memory copies. Each additional code word requires 1.7ms, which is smaller than the transmission time of a packet (20ms) by an order of magnitude. This means that we can encode on the fly.

Table 3.2 shows the decoding time. In systematic code, decoding time depends on the mix of code words. The more code words contain original messages, the quicker

Table 3.1. Encoding time to produce all code words. Left column indicates how many additional code words are produced in addition to 8 original messages

Number of Redundant Code Words	Time (ms)
0	0.780
1	2.539
2	4.298
3	6.057
4	7.816
5	9.575
6	11.334
7	13.093
8	14.852

Table 3.2. Decoding time of all 8 messages given how many code words are not original messages

Number of non-original-message code words	Time (ms)
0	0.427
1	4.027
2	6.876
3	9.820
4	13.713
5	17.119
6	21.059
7	24.604
8	27.065

the decoding becomes (Section 3.5.2 explains it in more detail). Decoding time is roughly linear to the number of non-original message code words.

Table 3.3 shows the expected decoding time calculated from Table 3.2, given the packet loss rate. Decoding time is also roughly proportional to the packet loss rate. Decoding takes less than 30ms even in the worst case, and it takes 160ms to receive the next 8 code words. So each decoding step can be done well before the next decoding occurs, even though there is an issue of buffering that may need to be addressed.

The mix of code words (how many code words are original messages) determines decoding time, but given the number of code words containing original messages,

Table 3.3. Effect of loss rate on time to decode 8 messages

Packet Loss Rate	Time(ms)
0	0.427
0.1	3.143
0.2	5.696
0.3	8.263
0.4	10.928
0.5	13.700
0.6	16.548
0.7	19.416
0.8	22.220
0.9	24.832

the combination of code words does not affect decoding time significantly. This is shown in Figure 3.9. 30 random cases are produced with 4 code words containing original messages with 4 additional code words, in total decoding 8 messages. The average decoding time was 13.44ms, with a 95% confidence interval of 1.52. Standard deviation was 0.74 – less than 10%.

Memory usage depends on the size of the encoding unit, which is the sub code word in code packet as shown in Figure 3.6. Most of the memory requirement comes from operation table and matrix. With 8 bit-long units, 512 bytes are used for the operation table, 64 bytes for the matrix, 232 bytes for 8 packet buffers, and 4 bytes are used for other variables. Packet buffers will be provided and shared by the application, and the operation table can be stored in program memory. The memory usage by the erasure code component is then 68 bytes.

3.7.3 Comparing Options

We compared different combinations of options (link-level retransmissions, erasure code, alternative routes) using experimental data on a real testbed. We ran the case with a maximum of 5 link-level retransmissions, with route fixing which tries up to 6 alternative routes at each hop (also with 5 as the maximum number of retransmissions

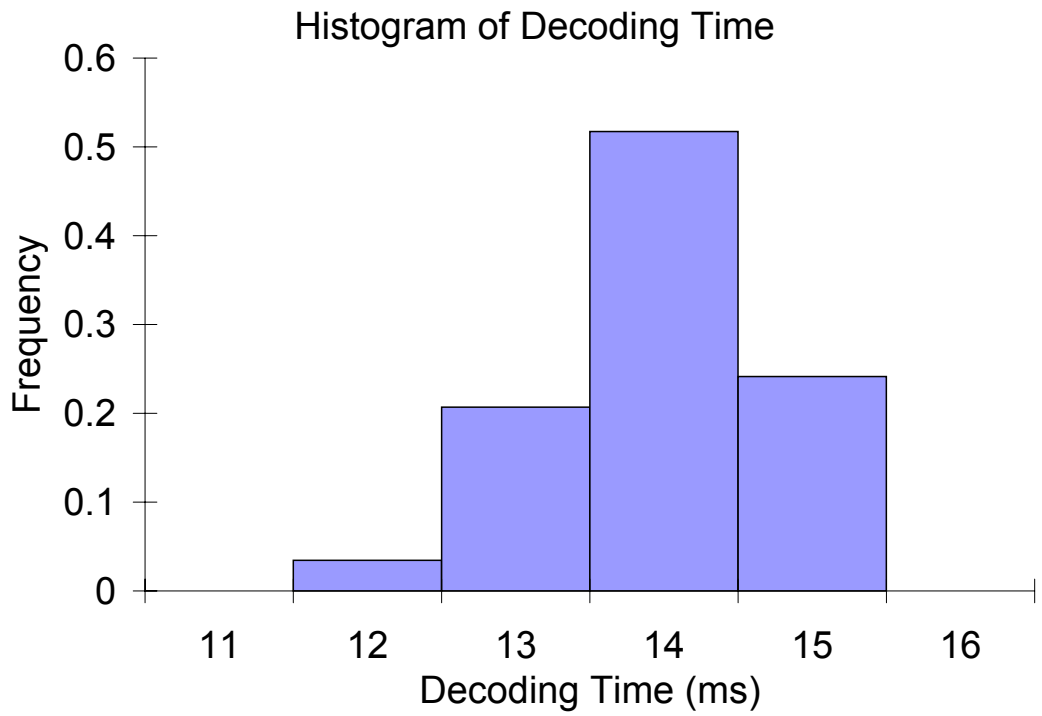

Figure 3.9. Histogram of time to decode 8 messages with 4 code words containing original messages

per each next hop). From this data we simulated and computed the results for other cases: 0 to 5 retransmissions per link without route fixing. We also compute the results as if the stream of packets consisted of erasure coded packets: we label each packet as being an original packet or a redundant packet, and are able to infer the results of the decoding process by labels of the received packets. Summarizing, in our evaluation we vary two dimensions: retransmission and redundancy. We vary the first from 0 to a maximum of 5 link retransmissions with no route fixing, and 5 maximum retransmissions with route fixing. We vary the redundancy from 0 to 8 redundant packets for each 8 packets of data. Route fixing is only for 5 maximum retransmissions.

The testbed we use is deployed on the fourth floor of the Computer Science building – Soda Hall – at the University of California, Berkeley. It consists of 78 Mica2Dot motes deployed in graduate student offices and is depicted in Figure 3.10. Test data

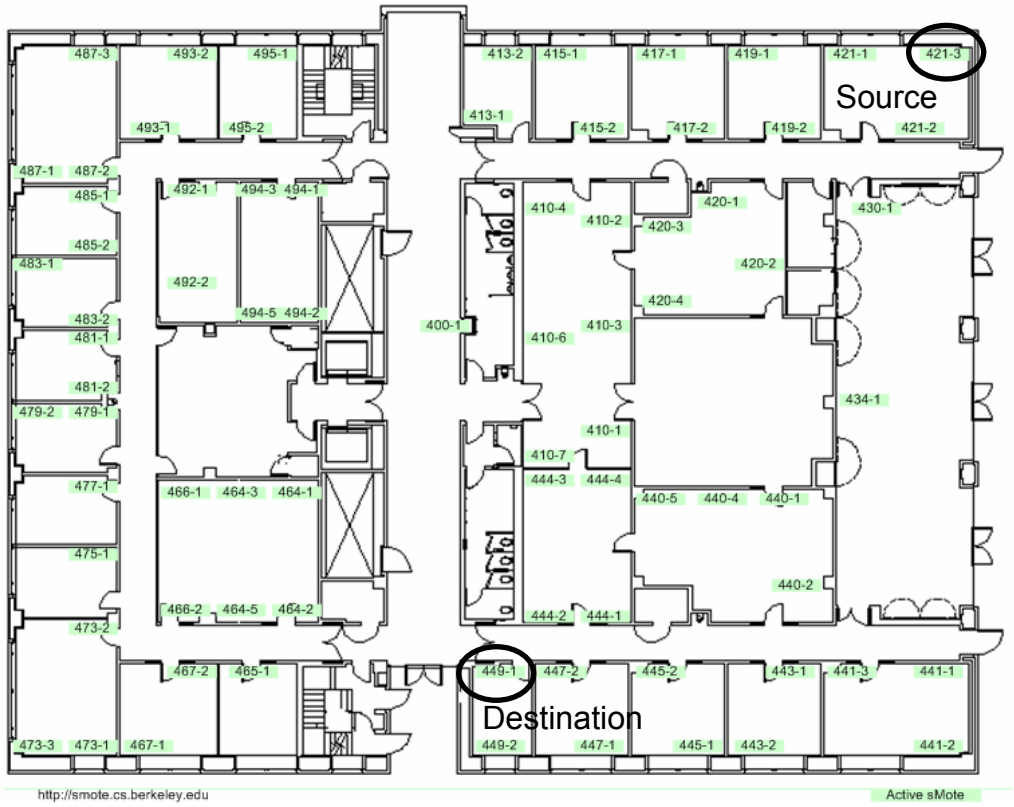

Figure 3.10. Map of Soda hall testbed. Source and destination are also indicated.

shown here was collected in the following way: we let the BVR routing information in all nodes stabilize for 75 minutes, and then had an external program send 300 packets of data from one specific node to another. We chose the nodes so that they would be separated by a significant number of hops. Packets are separated by 1 second, which is long enough to eliminate interference between two consecutive packets. The pair of nodes considered is also shown in Figure 3.10.[2] The path we use presented an average of 5 hops across all packets that were delivered, and the overall loss rate in the network was 26.28%. This takes into account all messages that were sent over all links during the course of the experiment.

The metrics we use to analyze the different options are reliability, cost, and overhead. Reliability is the percentage of original data packets that arrive at the final destination: a goodput. It measures the actual data that two applications at both ends can exchange successfully. Cost is the total number of packets injected into the network per unique packet of data. Cost includes both effect of loss rate, and the average number of hops from source to destination. Since some options may take a more reliable path even though it could be longer, cost is more meaningful than packet loss rate. However, as we shall see, cost alone also does not tell the whole story, because in the presence of loss one may incur cost and not do useful work. We define overhead as the cost per hop per each successfully delivered data packet. It is normalized by dividing by the path length, allowing us to make more meaningful comparisons. The overhead thus measures the amount of work done in the network (per hop) to deliver one data packet end-to-end. Ideally it should be 1, and we should look for options that maximize the reliability with the smallest overhead. One thing to note in cost and overhead is that they do not account for undelivered packets; even if some packet is not delivered, this is still considered in the calculation and negatively affects cost and overhead.

[2]We ran other similar experiments among other pairs of nodes with very similar results.

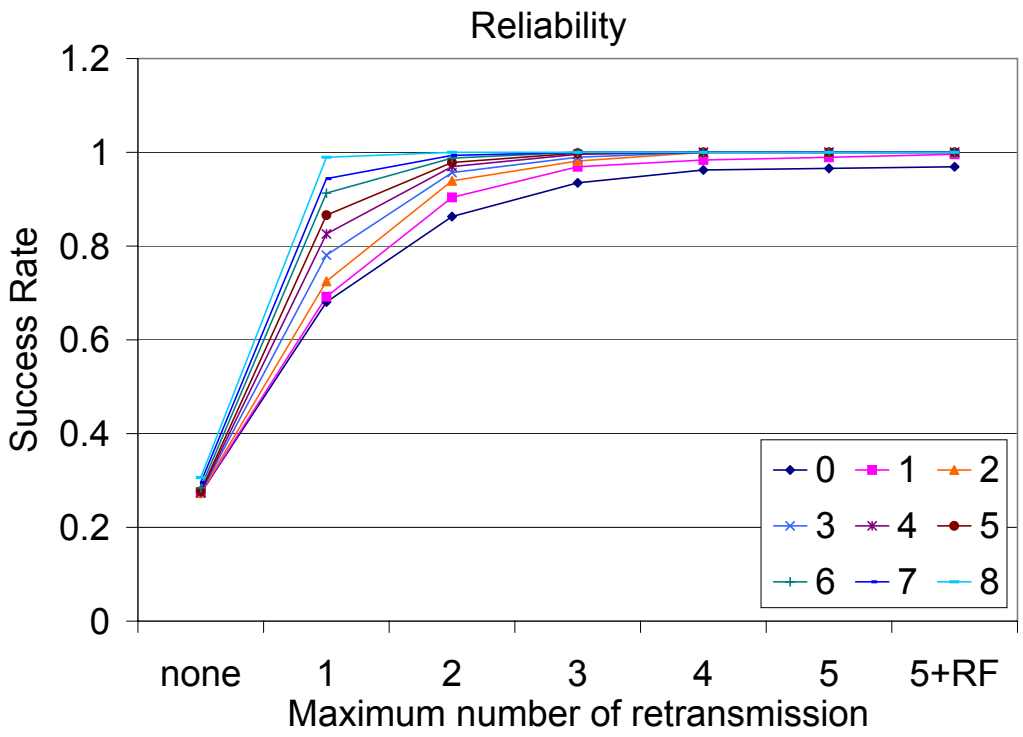

Figure 3.11. End-to-end reliability achieved by options. Each line represents number of redundant code words for 8 original messages. RF means route fixing is used.

Figure 3.11 shows the reliability each option (link-level retransmissions, erasure code, and route fixing) achieves. As described earlier in this subsection, we ran the case with a maximum of 5 link-level retransmissions, with route fixing which tries up to 6 alternative routes at each hop (also with 5 maximum number of retransmissions per each next hop). From this data, other cases are statistically calculated. The x-axis shows the number of retransmissions and whether route fixing is used. Each curve represents how many redundant code words were added to each 8 original messages. Figure 3.12 shows the normalized overhead for each option with the same x-axis and legends.

Our first observation is that link level retransmissions should be used in any case. With no retransmissions, the reliability is so low that the effect of redundancy is negligible (in spite of adding overhead, as in Figure 3.12). The low number (less than 30%) seen in Figure 3.11, is close to the expected value for a five hop transmission over

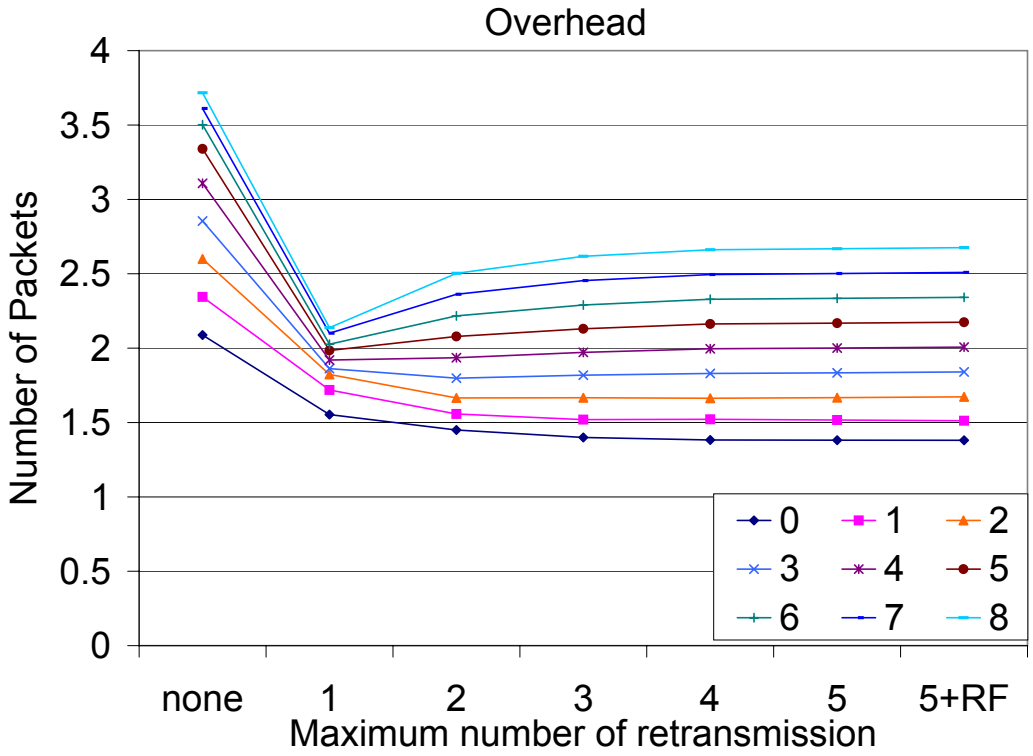

Figure 3.12. Number of packets injected to network per hop per successfully received data. Each line represents number of redundant code words for 8 original messages. RF means route fixing is used.

links with 26.28% loss rate. When using at most 1 per-link retransmission, not only does the success rate go substantially up, but also the effect of adding redundancy increases.

Our second observation from Figure 3.11 is that even with 5 retransmissions and route fixing, the reliability does not reach 100%. The reason for this may come from the nature of the loss process: there can be packets which are dropped even after 5 retransmissions, because a link may have gone down, and this information has not yet reached the routing layer or the link estimation component. In these cases, unless some costly measure is taken by the network that may include holding the packet in buffers for extended periods or backtracking the packet in the reverse path, it may be inevitable to drop the packet. Erasure codes are useful in this scenario exactly because they do not require that all packets be delivered to recover the data, and it is safe to drop some packets that would otherwise be too costly to deliver.

Erasure codes, however, add a fixed overhead, since redundant packets are always sent at a given rate. We can see in Figure 3.12 that for a given retransmission option, the overhead always increases with the number of redundant packets. With little redundancy added, the overhead, as shown in Figure 3.12, decreases as retransmission increases. This is mostly due to the increase in the success rate, and thus the decrease in wasted effort to deliver packets. We can notice, however, that with 4 or more redundant packets per each 8 data packets (50% or more redundancy added), the overhead increases with more retransmissions, with no corresponding gain in reliability. With high redundancy, the destination already gets enough number of code words to reconstruct original data. Additional packets delivered by more retransmissions do not increase the reliability any more. They just add burden to the network. Also, when the maximum number of retransmissions is large and end-to-end reliability is high, erasure code wastes too much bandwidth and overhead gets high.

Table 3.4. Given a threshold reliability requirement, what is the retransmission/redundancy combination that has the smallest overhead?

Threshold	Retransmission	Redundancy	Overhead
90%	5+RF	0	1.381
95%	5+RF	0	1.381
98%	5+RF	1	1.512
99%	5+RF	1	1.512
99.9%	4	2	1.663

Finally, in Figures 3.13 and 3.14, we plot, for the different options, reliability versus overhead. These plots give insight into the tradeoff at hand, and we caution the reader that the axes have different roles than in previous plots. Each curve in the figure corresponds to one retransmission option, and the nine points in each curve correspond to the redundancy with that option. In all curves redundancy increases from left to right. In this graph, we would like to choose points that have overhead close to 1, and reliability close to 1, thus as close to the upper left corner as possible. We can notice that adding on-demand retransmissions increases the reliability without incurring overhead, at least for low redundancy cases. On the other hand, adding redundancy, while always incurring overhead, is needed to get the last few percent of reliability. We see that for a given retransmission option, in order to add reliability one has to add redundancy, but the gains are very different for different maximum numbers of retransmissions, or that is where an end-to-end methodology is needed.

In Table 3.4, we pose the question of how one chooses an option, given these trends in reliability and overhead. As the threshold increases, the sweet spot moves toward more redundancy. And when the number of redundant code words increases, the maximum number of retransmissions drops. When the number of redundant code words increases, more packet losses can be tolerated, so retransmission for additional packet delivery becomes unnecessary.

Causes of failures are shown in Table 3.5. Data is from the case with 5 maximum

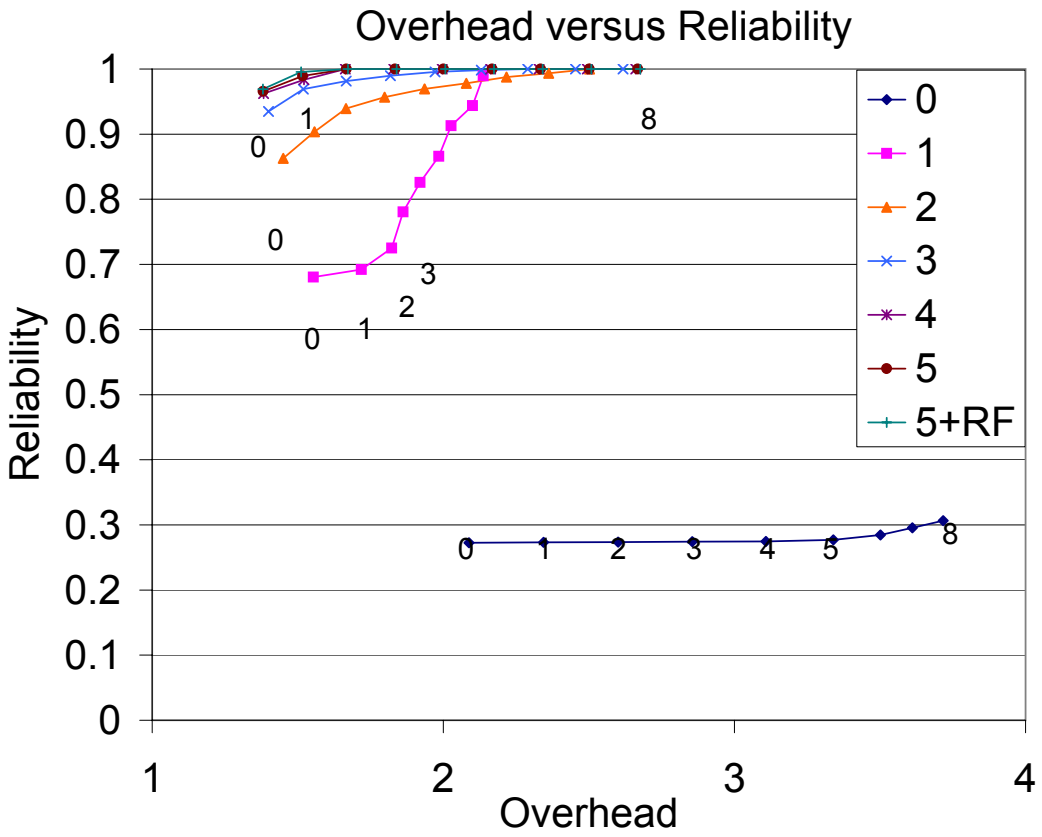

Figure 3.13. Overhead versus reliability for different combinations of retransmission and redundancy options. Overhead is the number of packets injected per hop per received data packet. Points in the same curve have the same retransmission option, and each curve has 9 points (indicated by numbers), corresponding to the number of redundant packets for each 8 packets of data.

Table 3.5. Decomposing causes of failures

Cause	Percentage
Independent Queue Overflow	2.667%
Consecutive Queue Overflow	0%
Independent Reroute	0.333%
Consecutive Reroute	0%
Nowhere to send	0.333%

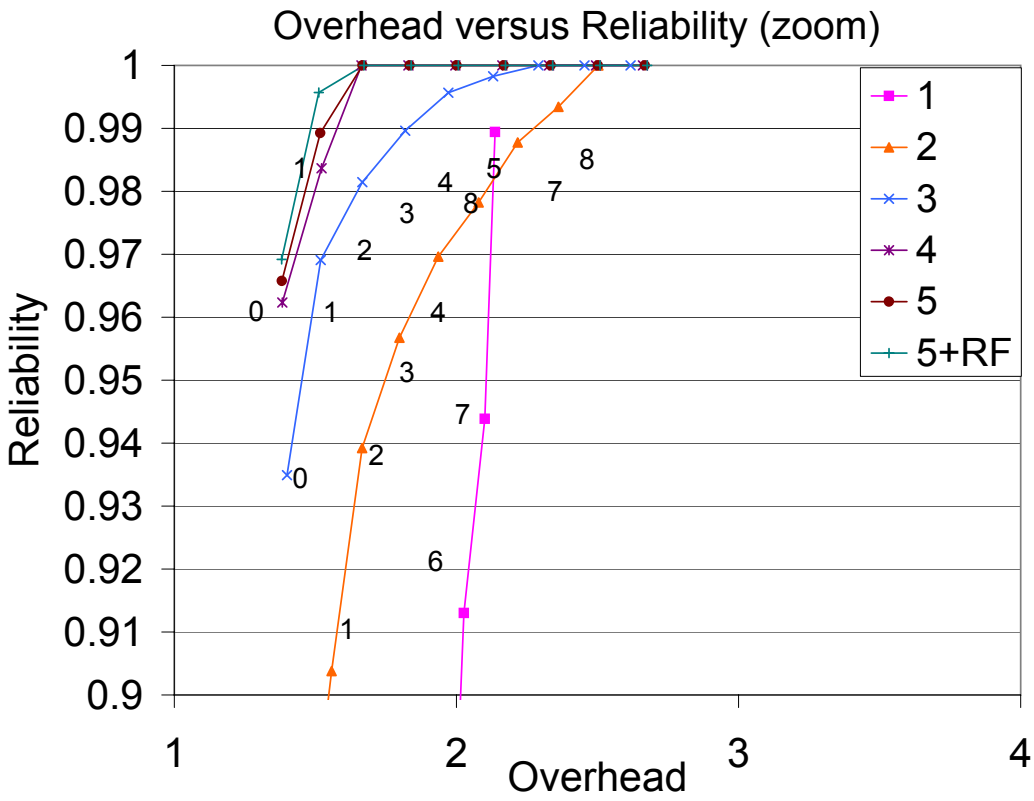

Figure 3.14. Overhead versus reliability for different combinations of retransmission and redundancy options (Zoom).

retransmissions and route fixing, and the table is a simulation for the case with 5 maximum retransmissions without route fixing . It is to show the effectiveness of route fixing. 'Reroute' is a failure without route fixing, but which would succeed with route fixing. 'Nowhere to send' is a failure without route fixing, and also a failure with route fixing: it could not send to any next hop candidates. This failure happens when a packet can be delivered into the node, but cannot be forwarded out. 'Queue Overflow' happens when pending outgoing packets fill up the queue and a new packet arrives. Reroute and queue overflows are divided into independent failures and consecutive failures. A consecutive reroute failure indicates a stale routing table value. Beacon vector routing adapts to link failure quickly, and we did not shoot packets too quickly in the test, so there are no stale routing table problems. Queue overflow constitutes 80% of failures, indicating a need for congestion control. When link-level retransmission and route fixing are used, packets tend to reside in queue longer until they are successfully delivered to the next hop. Then it increases the chance of queue overflow. Therefore, those options may not always increase reliability.

3.8 Discussion

This work presents an initial evaluation of several options for achieving reliability. We leave as subsequent work further exploration of the design space. Route fixing is tested only with 5 maximum retransmissions, which already provides high reliability. Even though we see marginal improvement at this point, it surely not only improves reliability but also decreases the overhead. It will be interesting to see a case with route fixing and a small maximum number of retransmissions, and compare its cost per reliability to a case with no route fixing and a large maximum number of retransmissions.

A direct comparison with end-to-end retransmission is missing. For very high

reliability end-to-end retransmission would be an attractive solution, even though it will increase delay.

Thick path, in which messages are forwarded simultaneously by several nodes that make progress towards the destination, is another possible option. It achieves reliability only through information redundancy, and can survive link failure. Moreover it has low delay to deliver packets. The downside is that it injects a large number of packets: it is the product of path length and path thickness. Since traffic is correlated locally, channel contention will not significantly affect the whole network. However, in terms of energy consumption this would be a bad choice. It will be interesting to see a trade-off of success rate, overhead, delay, and energy consumption.

Some form of congestion control is needed. Large chunk transfer and admission control would be a good candidate solution. This enables back pressure working as congestion control without much overhead.

Initially it appeared that our implementation of erasure codes worked as long as $M + N < 2^r - 1$. In experiments, when $M > r$ it worked in most cases, but not always. Mathematical reasoning of this phenomenon is also left for future work. Also, if we can avoid these cases without expensive operations, it would be helpful. If we can have a larger M, tolerating any 6 packet losses provides more robustness than tolerating 3 packet losses from each of two transfers.

3.9 Lessons

In this chapter, diverse options for achieving reliable transfer in wireless sensor networks are discussed, implemented, and tested in a real testbed. Link-level retransmissions, erasure code, and route fixing are implemented and evaluated. Link-level retransmissions handle transient link failure and contention very efficiently. Erasure

code introduces static overhead, however its use loosens the burden of delivering the last few packets (99.99% versus 99%), which are very expensive and inefficient using other methods. Route fixing solves the stale routing table problem, providing quick adaptation to link failure if the routing layer provides flexibility in route selection. In turn, route fixing reduces consecutive losses, increasing the usefulness of erasure code, which does not work well with successive losses. Link-level retransmissions happen on demand: packets are retransmitted only when necessary. Route fixing is also on-demand: only when packet cannot be forwarded to the next hop. Those local and on-demand options are very efficient approaches (cost per reliability). Erasure code allows some flexibility in the losses, and route fixing provides flexibility in selecting the next hop. Some options address some problems efficiently but not all failures, which can be effectively cured by some other options. Our results show that combining options would provide a sweet spot.

3.10 Initial Reliable Data Collection Protocol: Straw

Reflecting studies in this chapter, we designed and implemented the Straw (Scalable Thin and Rapid Amassment Without loss) component, a reliable, highly scalable, data collection service. Straw works over a multi-hop routing layer like MintRoute [86], with transfer initiated by the receiver. Since it is a collection protocol, the receiver is always a PC, and the sender a node. At a high-level, selective NACKs are used. In response to the request of the receiver, the sender sends the entire data once, the receiver identifies missing packets, and then sends a list of those packets (selective NACK) back to the sender. The size of a selective NACK is a single packet. Only a single NACK can be sent before data packets from that NACK

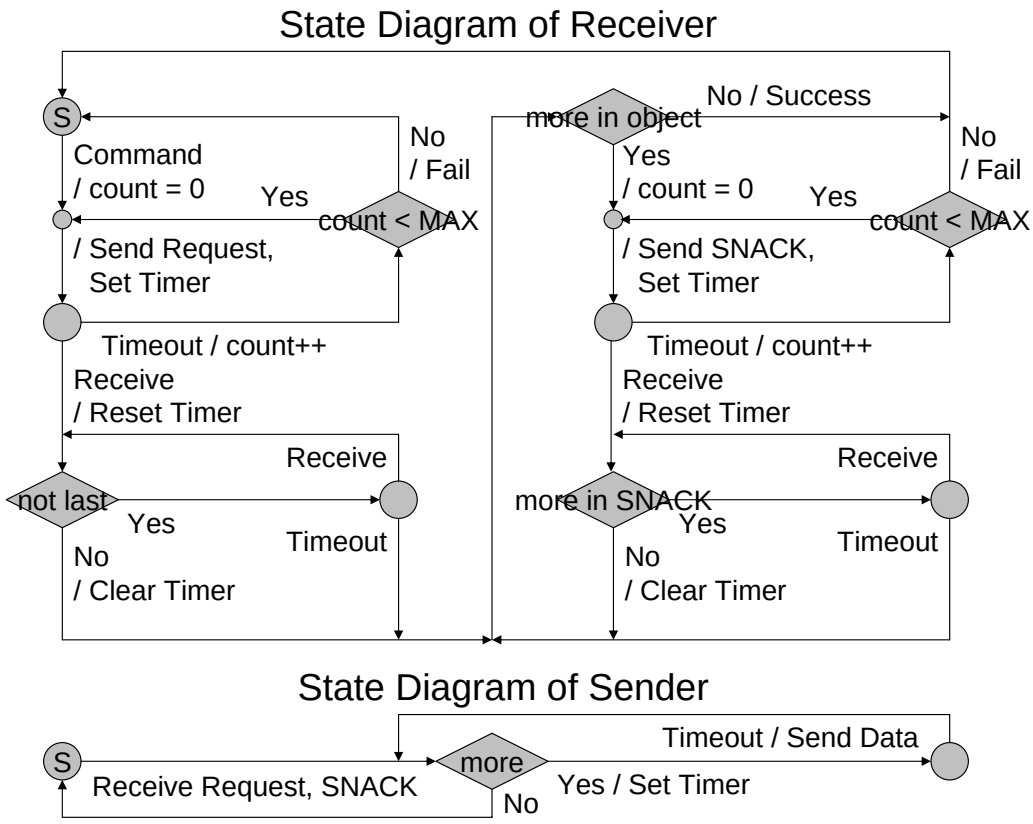

Figure 3.15. Finite State Diagram of Straw Protocol

are received. The sender resends those missing packets. At this point, the selective NACK is a single packet, so if there are missing packets some of them may not be reported. The receiver may send a selective NACK again, and this process repeats until all the packets are successfully received. Figure 3.15 shows finite state diagrams of a receiver and a sender of Straw protocol. The receiver initiates and drives the transfer, so the complexity is confined to the powerful receiver (base station), and the sender (mote) task is kept simple and light weight.

The sender always decides at what interval to send consecutive packets, which is the timeout value of a sender in Figure 3.15. For WSN, there can be interference between two adjacent transfers, so the inter-packet interval should be large enough to prevent this from occurring. One possibility for the length of the inter-packet interval is the time taken for a transmitted packet to reach the receiver. This approach can

work for a small network. However, as the number of hops in the path increases, this time interval becomes too large. In this case, a packet sent toward the receiver in the past can be far enough down the line that the next packet can be sent without interfering with the first. To maximize channel usage in a long multi-hop path, pipelining is used. For nearby nodes, the sender chooses the interval by looking at its depth in the communication tree – the needed delay interval is the time for a packet to arrive at the receiver. For nodes many hops from the receiver, the interval is forced to be at most five times the one-hop packet transfer time. The five hop threshold is empirically chosen from extensive field testing.

Now let us look at the detailed controls and packet formats of Straw. A receiver sends 4 types of commands to a sender. STRAW_NETWORK_INFO requests information of the routing layer at a receiver. This information is used to tune round trip time (RTT) and timeout value at the receiver. STRAW_TRANSFER_DATA triggers a transfer of the entire data at the sender. STRAW_RANDOM_READ is a selective NACK containing sequence numbers of missing packets. STRAW_ERR_CHK asks for a checksum of data at a sender side. A receiver compares checksums at both sides to determine whether the receiver data is corrupted. A sender has 3 types of replies. STRAW_NETWORK_INFO_REPLY is a reply to STRAW_NETWORK_INFO. STRAW_DATA_REPLY is a reply to both STRAW_TRANSFER_DATA and STRAW_RANDOM_READ. It is not necessary to distinguish to which command a reply is responding. A STRAW_DATA_REPLY packet contains a sequence number and data. STRAW_ERR_CHK_REPLY is a reply to STRAW_ERR_CHK, and contains a checksum of data at a sender. To save space in a packet, sequence numbers start from 10 instead of 0. If the first 16 bit number in a packet is equal to or larger than 10, it actually is a sequence number and implies the type of packet is STRAW_RANDOM_READ or STRAW_DATA_REPLY depending on whether it is from a receiver or a sender. All other types are represented by numbers smaller

than 10. With the default packet size of the MicaZ platform, the data payload of STRAW_DATA_REPLY is 20 bytes, and STRAW_RANDOM_READ (NACK) contains 12 sequence numbers. Detailed packet formats can be found at [3].

Lessons from analyzing options to achieve reliability influenced the design of Straw. In Straw, link-level retransmissions are used due to their efficiency. To provide end-to-end reliability, end-to-end retransmissions are also used. Erasure code is not used in a deployment. With link-level retransmissions, end-to-end reliability was expected to be high-90% in many cases. This expectation was based on preliminary field tests, and was actually true as will be seen in Section 5.1. In this situation, the cost per reliability of erasure code is too high. Moreover, since end-to-end reliability provides 100% reliability, trying to achieve high reliability through costly erasure code is not meaningful. Alternative routes are not used, either. Due to link-level retransmissions, the RTT already has quite large variations. Adding more variation with an alternative route option is not desirable for maintaining a tight timeout of end-to-end retransmissions.

In this and the previous chapters, the overall system architecture and major components in the system are explained. The following chapter introduces how the system is actually deployed on the Golden Gate Bridge.

Chapter 4

Deployment at the Golden Gate Bridge

In previous chapters, solutions are proposed to unsolved challenges introduced in Chapter 1. Components handling challenges, either by existing works or this work, are put together into a complete system as shown in Chapter 2. Then, it is deployed to the target site: the Golden Gate Bridge. In this chapter, let us take a close look at how the system is deployed in the real world.

The Golden Gate Bridge at the entrance to the San Francisco Bay is a compelling test bed for proving the usefulness of WSN for actual, difficult SHM installations. The cable-supported bridge was designed and constructed in the 1930s and opened to traffic in 1937. With a tower height of 746ft (227m) above sea level, and a 4200ft (1280m) long main span (see Figure 1.1), it was the longest suspension bridge in the world when it was completed. The extreme loading events for the bridge are expected to be from wind and earthquakes. The goal was to determine the response of the structure to both ambient and extreme conditions and compare actual behavior to design predictions. The network measured ambient structural accelerations from wind

load at closely spaced locations, as well as strong shaking from a possible earthquake, all at low cost and without interfering with the operation of the bridge. For this deployment, 64 nodes were deployed over the main span and southern tower (see Figure 1.1), creating the largest wireless sensor network ever installed for structural health monitoring purposes.

4.1 Putting Components Together

As described in Section 2.1, the system is composed of a base station and multiple nodes. A node actually senses the vibration of a structure, and sends the data to a base station. Hardware for a node includes a MicaZ [20] mote and an accelerometer board as shown in Figure 2.3. They are contained in a water-proof box [17]. A node also includes an external bi-directional patch antenna [19] to extend range, and batteries [18] to provide power. A node runs TinyOS software. A deployed node is shown in Figure 4.1. A base station stores the delivered data for further analysis. It has much more computational power and storage than a node. A laptop PC is used in this deployment. It runs software to interact with nodes. Due to limited access to the Internet at the bridge, real-time access to the deployed WSN is not provided in the deployment. A deployed base station is shown in Figure 4.2.

4.2 Environmental Challenges

The bridge is located in a difficult environment; gusty wind, strong fog, and rain present serious engineering challenges for the deployment and maintenance of an electronic system. The combination of sea fog and strong wind results in quick condensation of salty water and fast oxidation of metallic components. An example of the rust accumulated at a bridge connection over a short duration is shown in

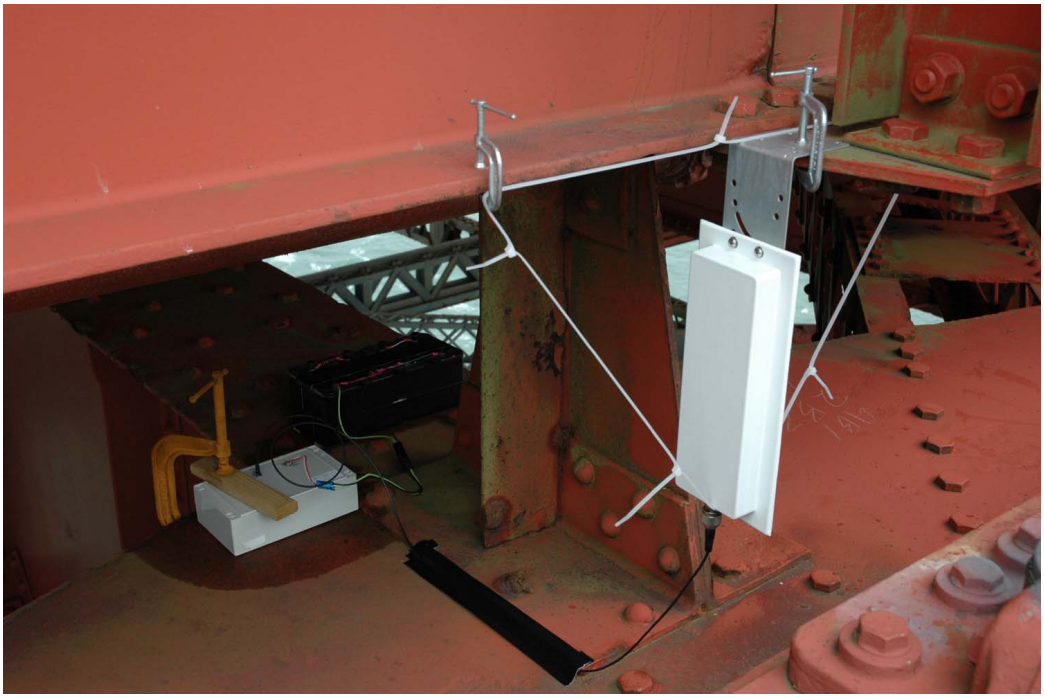

Figure 4.1. Board enclosure, antenna, and battery installed on the main span. The zip tie had to be put around the antenna to control wind vibration. Poor link quality was experienced with a vibrating antenna under strong wind. Corrosion of the C-clamp can be observed in the figure.

Figure 4.2. A laptop PC is used as a base station. It is located inside of the south tower.

Figure 4.3. Severity of rusting of the bridge can be seen. Rusting not only threatens the bridge, but also was a hazard to the monitoring system, see Figure 4.1

Figure 4.3. C-clamps, metal supporting structures on the bridge tower, and even electrical connectors quickly gather rust. The enclosure for the boards is a waterproof plastic box that performed very well during the deployment, as shown in Figure 4.1. Due to strong wind, major items had to be secured to the bridge with C-clamps, and cables had to be tied or attached to the steel structure. When the wind was strong, the bidirectional antenna vibrated back and forth fiercely, resulting in poor link quality, so zip ties were used to fasten all antennas in order to reduce the vibration. Since the bridge has a linear geometry, the radio signal had only to be bi-directional; therefore an external bidirectional patch antenna was used for communication, adding signal splitters when necessary to change direction. There is a very narrow passage along the side of the bridge which provides limited line of sight for the bidirectional antennas. This space is, of course, surrounded by steel components and reinforced concrete slabs, and at some places it is obstructed by tools and materials belonging to the maintenance crew. The range of the radio in that harsh environment is severely

limited, with the functional range of the Crossbow MicaZ Mote used in this project being 50ft (15.24m) to 100ft (30.48m).

4.3 Deployment Plan

The bridge has suspension cables tying the stiffening longitudinal trusses to the main cables every 50ft (15.24m). The node mounting plates are attached to the gusset plate (or in a few cases to the top flange) on top of the plate-girders connecting the top flanges of the stiffening trusses. The deployment plan was initially designed based on radio tests on the bridge. MicaZ motes attached to the bidirectional antennas were deployed and the signal strength was measured. The tests showed that the signal weakens sharply after 175ft (53m), so 150ft (45.72m) was selected as the modular distance between the boards, hence twenty-nine boards were needed to cover each side of the main span. That nicely matched the distance between the suspension cables and floor beams as well. The actual deployment, however, required some readjustments since the second batch of MicaZ motes, purchased at a later time, proved to have weaker radio strength, by up to 7.5dBm, than the prototype devices. The new motes only yielded a reliable transmission range of 100ft (30.48m), and in some cases the inter-mote interval had to be reduced to 50ft (15.24m). Based on the adjusted deployment plan, a total of 64 sensor nodes were deployed: 53 on the west side of the main span, 3 on the east side of the main span, and 8 on both sides of the south tower. Figure 1.1 shows the overall layout of the nodes in the deployment. Nodes are deployed on the east side of the main span to provide information necessary for distinguishing between vertical and torsional modes of vibrations. A base station is located inside of the south tower. A small control room is located near the entrance from the west side of the span, see Figure 4.2.

The high-level operation is executed as follows. At the trigger signal from the

base station, every node starts sampling the vibration data. The sampling period is usually set to fill up the 512KB of flash memory on the MicaZ. Then sampled data is reliably collected from every node one by one. These constituted one cycle, and one cycle takes about 12 hours.

Chapter 5

Network Analysis of the Golden Gate Bridge

Network data from the Golden Gate Bridge and its analysis are provided in this chapter. Control traffic of a routing layer (MintRoute) shows a collision with data traffic of a transport layer (Straw). The data and experience with Straw indicates problems with a static rate, and motivates an improved reliable data collection with rate control: Flush.

5.1 Routing Layer

Figure 5.1 shows an average end-to-end loss rate provided by a routing layer. In the routing layer, link-level retransmission is enabled. The data is taken when Straw transfers an entire data. Only successful Straw transfers are used, because only successful transfers have a log for end-to-end reliability at the routing layer. As will be seen in Section 7.2, unsuccessful Straw transfers due to network problem are not common, so inclusion of those cases will not change the trend of a result.

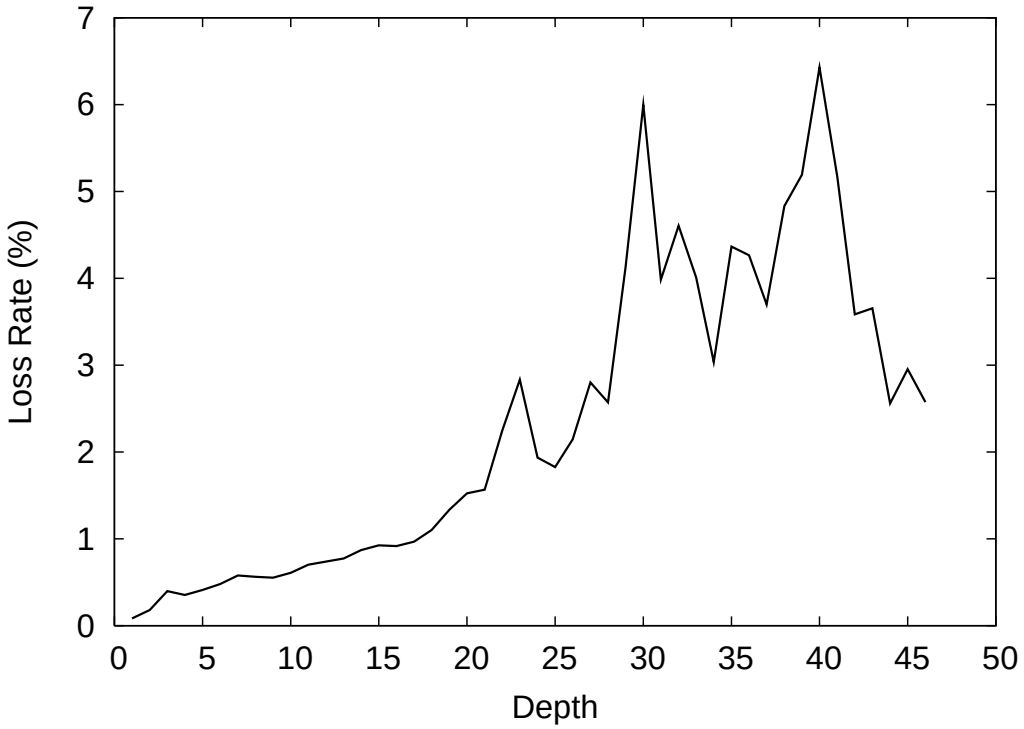

Figure 5.1. Average end-to-end loss rate over different depths in a routing tree. In many cases, the loss rate is below 5%, and never exceeds 7%.

2,613 Straw transfers are taken, and depending on the depth of a source, they are grouped and averaged. In many cases, the loss rate is below 5% and never exceeds 7%. This data supports our design decision of Straw based on an assumption that an end-to-end success rate will be high-90% in many cases.

Let us look at the routing tree formed at the GGB. Figure 5.2 shows one example of the routing tree. Data is taken from one case on September 21st and there are 56 nodes in the tree. The leftmost node is the basestation. The tree is skewed and the rightmost node is 45 hops away from the basestation. Figure 5.3 shows how many children each node has. As we can expect from the skewed tree, majority of nodes have one child.

In link quality data, one interesting phenomenon is observed: estimation for link quality drops suddenly after data collection. Link quality indicates the expected number of packets to send before one packet is successfully transferred through that

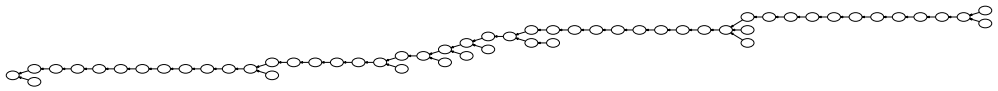

Figure 5.2. An example of the routing tree formed at the GGB. There are 56 nodes in the tree. The leftmost node is the basestation. The rightmost node is 45 hops away from the basestation. The tree is skewed.

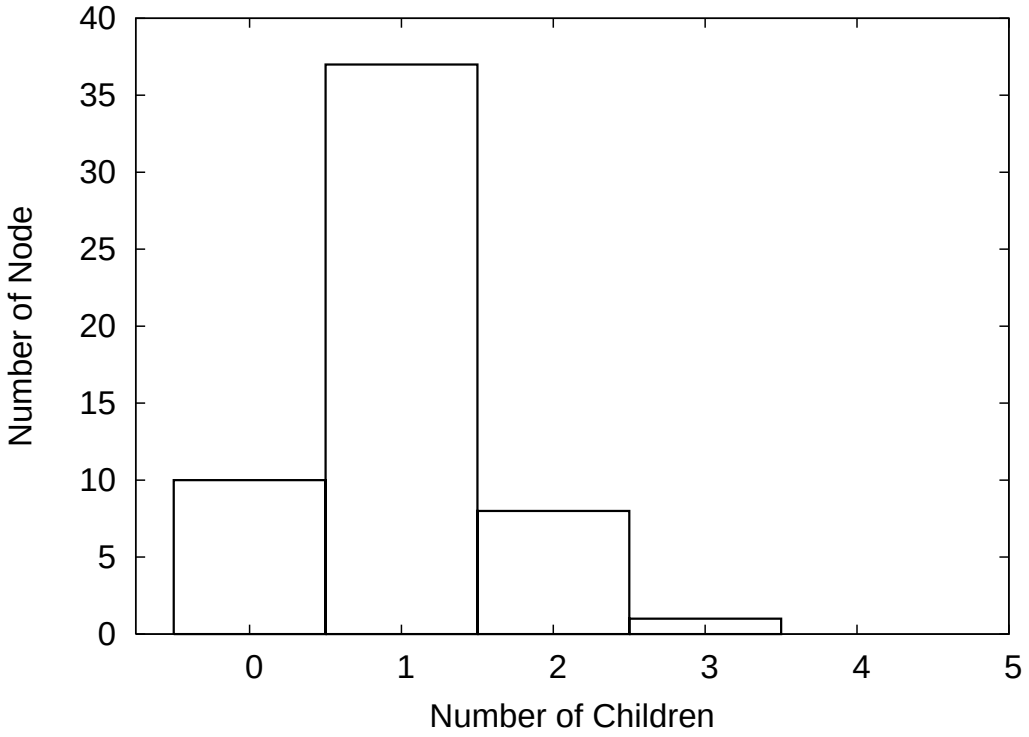

Figure 5.3. Distribution of the number of children of 56 nodes in a routing tree shown in Figure 5.2. Majority of nodes have one child.

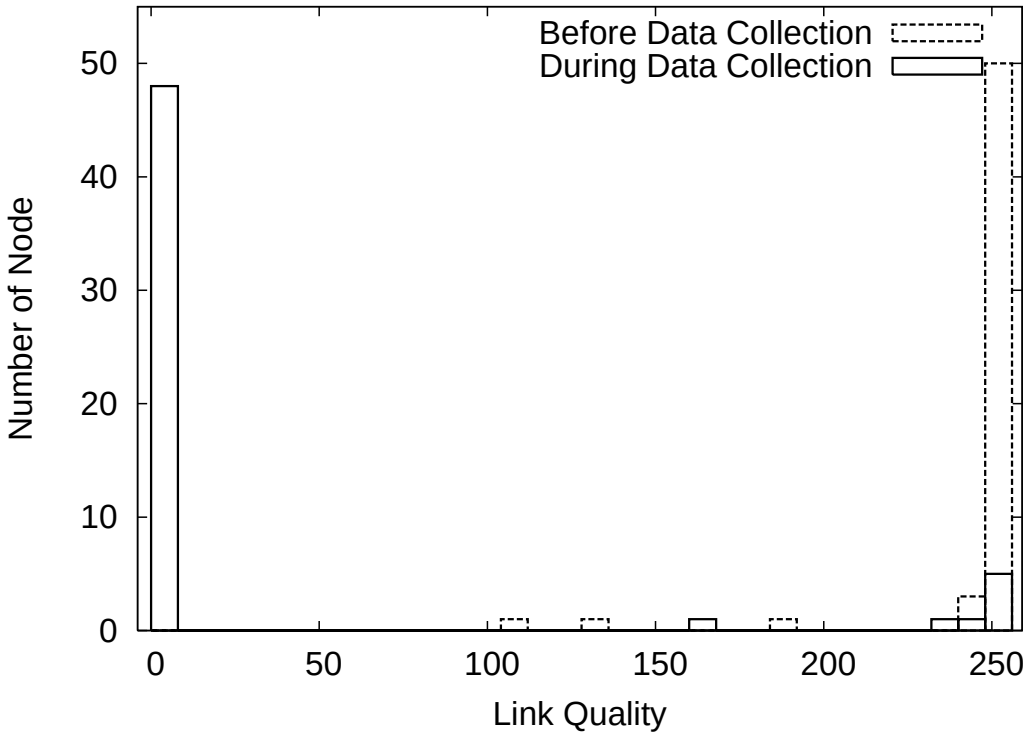

Figure 5.4. Distribution of link quality measurements of 56 nodes before and after data collection. After data collection, measured quality drops dramatically, even though actual link quality does not change much.

link. Figure 5.4 shows a distribution of link quality before and after data collection. Data is taken from the same case used in Figure 5.2 and 5.3 However, all data from other cases show the same characteristics. Before data collection, link quality to a successor is very good. The histogram shows that most of the 56 nodes are close to the best quality, 255. However, after the data collection, link quality moves to the worst quality, 0. We suspect this is not from an actual change of radio connectivity or environment, because such a dramatic and consistent change in an environment after data collection is not likely to repeat all the time. It should be instead from a strange behavior of a link estimator at MintRoute [86]. This issue will be discussed later in Chapter 8.

5.2 Transport Layer

Straw is an initial solution to reliable data collection, and was introduced in Section 3.10. This transport layer is deployed on the Golden Gate Bridge (GGB), and the resulting bandwidth from the GGB installation are presented in Figure 1.2. The first few hops show sharp decreases in bandwidth avoiding direct interference. However, after the fourth hop, pipelining begins, and the packet interval stays constant; the transmission bandwidth decreases very slightly from the fourth hop down to the 46th hop.

During 2630 successful Straw transfers 36,926 packets are sent from the basestation and 41,567,165 packets are received at the basestation. About 1126 times more packets are sent than received. This data supports our assumption that command dissemination is not a frequent event and collection routing is more efficient than any-to-any routing even if command dissemination has to rely on flooding, see Section 2.6.

To enable pipelining, Straw forces an inter-packet interval to be limited by a threshold. To determine the threshold, field tests were performed at the Richmond Field Station (RFS) and GGB. In those tests, depending on a topology and time 3 or 4 packet transfer time yielded the highest bandwidth. If an inter-packet interval is smaller than what a channel capacity can provide, it leads to cascaded losses and unstable performance. Therefore, the threshold is pessimistically set to be higher than a test result as 5 packet transfer times, in order to survive variations in topologies and environment over time. Straw operated on the bridge with high stability and without any significant problems in bandwidth, as can be seen in Figure 1.2. However, in favor of stability, channel capacity was underutilized.

A less empirical method for determining the inter-packet interval for a given topology with a dynamically changing link quality and an interference range is needed to

exploit channel capacity and to avoid a tedious manual tuning for each environment and topology. The following chapter introduces Flush, which provides a solution to this problem.

Chapter 6

Reliable Data Collection (Requirement 6)

Previous chapter explained experiences with Straw in a real deployment at the Golden Gate Bridge. An inter-packet interval or a rate should be statically set, so a pessimistic value is used yielding an underutilization of a channel capacity. To fully exploit channel capacity but still survive variations in environment and topology, a rate needs to be dynamically adjusted.

Based on experience at the GGB with an initial protocol, Straw, we present *Flush*, a reliable, high goodput transport protocol for wireless sensornets. Flush is a useful network primitive for many sensor network applications which require transmitting bulk data across multiple wireless hops, like structural health monitoring, volcanic activity monitoring, or protocol evaluation. Indeed, we collected the performance data presented in this chapter using Flush itself. The Flush protocol provides end-to-end reliability, minimizes transfer time, and adapts to time-varying network conditions. It achieves these properties using end-to-end acknowledgments, implicit snooping of control information, and a rate-control algorithm that operates at each hop along

a flow. Using several real network topologies, we show that Flush closely tracks or exceeds the maximum goodput achievable by a hand-tuned, fixed-rate for each hop over a wide range of path lengths: in one experiment, Flush efficiently transfers a 26,600-byte dataset over a 48-hop wireless network.

6.1 Revisiting the Problem

This chapter presents the design and implementation of Flush, a reliable, high goodput, bulk data transport protocol for wireless sensor networks. Flush has been evaluated on indoor and outdoor testbeds with networks of up to 48 hops. Target applications for Flush include structural health monitoring, volcanic activity monitoring, and bulk data collection [48, 67, 84]. Some of these applications cover large physical extents, measured in kilometers, and have network depths that range from a few to over forty hops. Power concerns and challenging radio environments can make using smaller diameter networks built from higher-power radios unappealing. While delivery of bulk data to the network edge may sound simple, the vagaries of wireless communication make efficient and reliable delivery a challenge in multihop networks: efficiency and reliability are hampered by lossy links [82], intra-path interference is hard to avoid [85], inter-path interference is hard to cope with [44, 73], and transient rate mismatches can overflow queues. These challenges make naïve or greedy approaches to multihop wireless transport difficult.

Intra-path interference occurs when transmissions of the same packet by successor nodes prevent the reception of the following packet from a predecessor node. If the packet sending rate is set too high, persistent congestion occurs and goodput suffers – a condition that is potentially undetectable by single-hop medium access control algorithms because it stems from a hidden terminal effect. If the packet sending rate is set too low, then channel utilization suffers and data transfers take longer than

necessary. The optimal rate also depends on the route length and link quality. Since in a dynamic environment, some of these factors may change during the course of a transfer, we show that no static rate is optimal at all hops and over all links. This suggests that a dynamic rate control algorithm is needed to estimate and track the optimal transmission rate.

Inter-path interference occurs when two or more flows interfere with each other. Designing multihop wireless transport protocols that are both interference-aware and have congestion control mechanisms is difficult in the general case. In this work, we greatly simplify the problem by allowing only a single flow at a time. Collecting data sequentially from nodes, rather than in parallel, does not pose a problem for collecting bulk datasets if the overall completion time is the critical metric, like in our target applications. Ignoring inter-path interference allows us to focus on maximizing bandwidth through optimal use of pipelining.

To be viable in the sensornet regime, a protocol must have a small memory and code footprint. As of early 2007, typical motes have 4KB to 10KB of RAM, 48KB to 128KB of program memory, and 512KB to 1MB of non-volatile memory. This limited amount of memory must be shared between the application and system software, limiting the amount available for message buffers and network protocol state.

In summary, our *goal* is to develop a protocol that delivers data *(1) reliably to the edge (2) with minimal transfer time* and has *(3) a small memory and code footprint.* The initial protocol, Straw, satisfied the first goal. However, given a limitation of a static rate, the rate is set at a low value to overcome variations in an environment and a topology, missing the second goal. Our solution, Flush, considers a single flow at a time, and uses rate-based hop-by-hop flow control to match the available bandwidth. The rate is controlled to avoid intra-path interference, not inter-path congestion, which is already removed by a single flow limitation. Flush finds the available band-

width along a flow using a combination of local measurements and a novel interference estimation algorithm. Flush efficiently communicates this rate to every node between the bottleneck and source, allowing the system to find and maintain the maximum rate that avoids intra-path interference. On long paths, Flush pipelines packets over multiple hops, maximizing spatial reuse. Flush was implemented in TinyOS, the *de facto* operating system for sensor networks [41] and evaluated using the 100-node Mirage testbed [11] as well as an ad hoc, 79-node outdoor network at the Richmond Field Station (RFS) [12]. The results show that Flush's rate control algorithm closely tracks or exceeds the maximum effective bandwidth sustainable using a fixed rate optimized for each location, *even over a 48-hop wireless network*. On the MicaZ platform, our implementation of Flush requires just 629 bytes of RAM and 6,058 bytes of ROM.

Section 6.2 places Flush in the context of the large prior literature on transport reliability, rate optimization, congestion control, and flow control. Section 6.3 introduces different connectivity models and provides background for interference and rate control. Section 6.4 analyzes the performance gain of pipelining. Section 6.5 describes the Flush protocol and Section 6.6 describes our implementation. Section 6.7 explains experimental setup used in an evaluation. Section 6.8 evaluates Flush in comparison to standard routing protocols as well as static rate algorithms and distinguishes the contributions that layer 3 and layer 4 congestion control have on goodput. Section 6.9 presents, and attempts to resolve, some open concerns. Section 6.10 presents our concluding thoughts.

6.2 Related Work

Our work is heavily influenced by earlier work in congestion mitigation, congestion control, and reliable transfer in wired, wireless, and sensor networks. Architecturally,

our work was influenced by Mishra's hop-by-hop rate control [63], which established analytically that a hop-by-hop scheme reacts faster to changes in the traffic intensity and thus, utilizes resources at the bottleneck better and loses fewer packets than an end-to-end scheme. However, this work is focused on a model of switches in the Internet. Kung et al's work on credit-based flow control for ATM networks [23] also influenced our work. Their approach of using flow-controlled virtual circuits (FCVC) with guaranteed per-hop buffer space is similar to our design. We adapted ideas of in-network processing in high-speed wired networks to wireless networks in which transmissions interfere due to the nature of the broadcast medium. Li et al. [55] studied the theoretical capacity of a chain of nodes limited by interference using 802.11, which is related to our work in finding a capacity and rate. Indeed, our results generally appear to agree with Li's models but our work also demonstrates how real-world factors can cause significant variance from the ideal performance.

ATP [79] and W-TCP [77], two wireless transport protocols that use rate-based transmission, have also influenced our work. In ATP, each node in a path keeps track of its local delay and inserts this value into the data packet. Intermediate nodes inspect the delay information embedded in the packet, and compare it with its own delay, and then insert the larger of the two. This way, the receiver learns the largest delay experienced by a node on the path. The receiver reports this delay in each epoch, and the sender uses this delay to set its sending rate. W-TCP uses purely end-to-end mechanisms. In particular, it uses the ratio of the inter-packet separation at the receiver and the inter-packet separation at the sender as the primary metric for rate control. As a result, ATP and W-TCP reduce the effect of non-congestion related packet losses on the computation of transmission rate. The goal of both works is a congestion control of multiple flows. Queueing delay is used as an inter-packet delay. The goal of Flush, in contrast, is avoiding intra-path interference of a single flow. So, Flush uses the time to escape an interference range, as an inter-packet delay.

Flush computes fine-grained estimates of the bottleneck capacity in real-time and communicates this during a transfer, allowing our approach to react as packet losses occur. Since the capacity of a link depends on an interference range of neighboring nodes, link capacity differs among different nodes. A link along a path with the lowest capacity determines the bottleneck capacity. Flush also applies rate control to the problem of optimizing pipelining and interference, neither of which is addressed in ATP and W-TCP. For this purpose, Flush actually measures interference along a path, which is not done in ATP nor W-TCP.

A number of protocols have been proposed in the sensor network space which investigate aspects of this problem. Fusion [44], IFRC [73], and the work in [30] address the problems of rate and congestion control for collection, but are focused on a fair allocation of bandwidth among several competing senders, rather than efficient and reliable end-to-end delivery. Fusion [44] uses only buffer occupancy to measure congestion and does not try to directly estimate forward path interference. IFRC estimates the set of interferers on a collection tree with multiple senders, and searches for a fair rate among these with an AIMD scheme. It does not focus on reliability, and we conjecture that the sawtooth pattern of rate fluctuations makes for less overall efficiency than Flush's more stable rate estimates. Event-to-Sink Reliable Transport (ESRT) [75] defines reliability as "the number of data packets required for reliable event detection" collectively received from all nodes experiencing an event and without identifying individual nodes. This does not satisfy our more stringent definition of reliability. PSFQ [83] is a transport protocol for sensor networks aimed at node reprogramming. This is a dissemination problem that is different problem from our collection problem, as the data moves from the basestation to a large number of nodes.

Fetch [84] is a reliable bulk-transfer protocol used to collect volcanic activity monitoring data. Fetch's unit of transfer is a 256-byte block, which fits in 8 packets.

Similar to Flush, Fetch requests a block, and then issues a repair request for missing packets in a block. Fetch was used to collect data from a six hop network in an extremely hazardous environment. In collecting 52,736 bytes of data, the median bandwidth for the 1st hop was 561 bytes per second and the median bandwidth for 6th hop was 129 bytes per second.

Wisden [67], like Flush, is a reliable data collection protocol. Nodes send data concurrently at a static rate over a collection tree and use local repair and end-to-end negative acknowledgments. Before a comparison, let us revisit performance metrics. The first goal is end-to-end reliability. 100% reliability is provided by both Fetch and Wisden. The second goal is minimal transfer time, which is analogous to a maximal overall bandwidth. The paper [67] reports on data collected from 14 nodes in a tree with a maximum depth of 4 hops. Of the entire dataset, 41.3% was transferred over a single hop. Overall bandwidth was about 782 bytes per second. To compare with Flush, we assume the same distribution of path lengths. Based on the data from our experiments, it would take Flush 465 seconds for an equivalent transfer. We ran a microbenchmark in which we collected 51,680 bytes using a packet size of 80 bytes (the same as Wisden) and 68 byte payload. This experiment, repeated four times, shows that Flush achieved 2,226 bytes per second from a single hop compared with Wisden's 782 bytes per second. This difference can be explained by the static rate at every node in Wisden. Incorrectly tuned rates or network dynamics can cause buffer overflows and congestion collapse at one extreme and poor utilization at the other extreme. Since in Wisden, nodes are sending without avoidance and adjustment to interference, cascading losses can occur, leading to inefficiency.

Finally, we attempted to compare Flush with Tenet's reliable stream transport service with end-to-end retransmission [38]. Since comparable performance for Tenet is not available, we attempted to use and characterize its performance on the Mirage testbed. Unfortunately, despite many bug fixes and weeks of assistance from its

authors, Tenet could not be instrumented to run on the Mirage testbed we used for the majority of our experiments before the writing of this dissertation.

6.3 Models of Connectivity

For a better understanding of interference, connectivity models are introduced. This section provides a background for rate control in Section 6.5.

6.3.1 Unit Disk Model

The unit disk model is the simplest connectivity model [9]. A node has a circular region. It has perfect connectivity to a node within this region. For a node out of this region, there is absolutely no connectivity. Connectivities can be drawn as a graph.

By changing transmission power, connectivities can be changed. When transmission power is increased, a hop count between two arbitrary nodes will decrease. A shorter path will lead to a high throughput. However, it will also increase the number of connectivities among nodes, and the number of edges in a connectivity graph. This increase, in turn, leads to increased interference among nodes. Increased interference will decrease the chance of concurrent multiple transfers.

6.3.2 Multi Disk Model

In a radio chip, to successfully decode a received packet, received signal strength should be at least some signal to noise ratio (SNR) threshold higher than the noise floor. A node can interfere with transmissions beyond its own packet delivery range. Therefore, the delivery range (DR) is smaller than the interference range (IR) [40, 64].

Compared to the unit disk model, the multi disk model represents the real world more closely [55].

In addition to IR and DR, let us define the forwarding range (FR). A routing layer (e.g. [86]) can choose a subset of nodes within DR as candidates to which packets will be forwarded. Let us define the forwarding to delivery ratio (FDR) as a ratio of FR to DR. Higher FDR implies an optimistic approach, sending packets to edge nodes within DR.

Higher FDR will decrease hop counts and lead to shorter paths and higher throughput. However, received signal strength at a target node will not be much higher than the SNR threshold plus a noise floor. As will be discussed in Subsection 6.5.4, in this case a transmission will be more vulnerable to jammers, nodes beyond DR but within IR.

6.3.3 Multi Cloud Model

In a real world, communication range is not likely to be circular due to environmental effects [50]. The packet delivery rate does not change from 0% to 100% at some threshold. Rather it is closer to a continuous probability distribution over space, and all these factors change over time. A multi cloud model fits better here. Thickness and shape of a cloud change over time, and they determine probability distributions of packet reception rate and received signal strength.

A multi cloud model is accompanied by more side effects when transmission power and FDR are changed. When transmission power increases, a routing layer has more alternative choices as candidates for the next hop. When FDR is increased, it also gives more alternative choices for the next hop. However, links chosen with a higher FDR are likely to have higher loss rates, leading to a higher end-to-end loss rate.

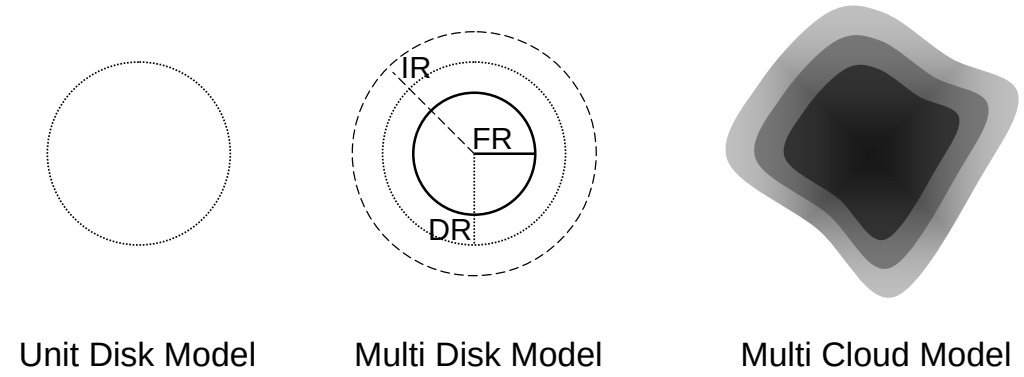

Figure 6.1. Three different connectivity models. The unit disk model is the simplest. The multi cloud model is the most complex and the closest to a real world.

In the deployment at the Golden Gate Bridge, there is only a single flow at a time, so it is less important to worry about multiple concurrent flows. It is more desirable to increase bandwidth, so increasing transmission power is reasonable. Actually maximum power is used, and an external antenna is used to increase transmission power further. An even larger antenna could have worked better. MintRoute [86] is cautious in selecting a forwarding range, because their experiments showed nodes near DR tended to have unstable links. Lower FDR alleviated a problem with a jammer in our proposed solution to reliable data collection: Flush. In Subsection 6.5.4, we will talk about this issue with more detail.

Changing the density of a deployment is another option to changing the transmission power. Increasing the density of a deployment by adding more nodes provides more alternative choices for the next hop. However, deploying more nodes incurs a higher cost.

Figure 6.1 visualizes three models.

6.4 Analysis of Pipelining

In this Section, let us take a look at how much improvement can be gained through pipelining. We will also see the effect of diverse parameters on bandwidth. In this model, a slightly generalized version of the reliability protocol of Straw is used, which is similar to that of Flush.

6.4.1 Modeling of Pipelining

Let us first define terms. An object is something we are trying to transfer. Let us refer to the size of an object as o. Then let us define a chunk as a unit of transfer. This means that a chunk is a unit of reliable data transfer. The protocol makes sure all data in one chunk is delivered reliably, then it begins to send the next chunk. Let us refer to the size of a chunk as c. In Straw and Flush, a chunk size is set to be an object size ($c = o$). However, let us consider a general case in this model. A packet is a unit of transfer of a radio. Let us refer to the size of a packet as p. Let us define n to be the number of sequences that fit into a single NACK packet. Let us refer to the bandwidth as b, the hop count of a data source from a base station as h, and the depth of pipelining as d. The depth of pipelining is a distance between two adjacent packets in a pipeline. For example, a pipelining depth of 5 means that the difference between starts of two adjacent packet transfers is 5 packet transfer times. A routing layer provides, but does not guarantee, reliability. Let us refer to a single-hop packet success rate at this routing layer as s. Many radios have an error checking scheme at a packet granularity. However, a physical radio sometimes suffers multiple errors, and an error checking scheme can fail to detect these multiple errors in a packet. Let us refer to this rate of false positives as e. To provide integrity of data, a redundant error check is facilitated at a chunk level. When one or more packets in a chunk suffer false positives, the entire chunk has to be retransmitted. Again, in Straw and Flush,

Table 6.1. Terms used in the modeling of pipelining.

Term	Meaning
o	object size
c	chunk size
p	packet size
n	number of sequences in a NACK packet
b	bandwidth
h	hop count of the source
d	depth of pipelining
s	single-hop packet success rate at a routing layer
e	rate of false positives of packet error check

a checksum is used for an object, because a chunk contains one complete object. Table 6.1 summarizes all terms used.

Now that we have all terms defined, let us take a look at how long it takes to transfer an object. For simplicity, let us assume that a wired link between a base station mote and a powerful base station PC is not a bottleneck.

An object is divided into multiple chunks and individual chunks are transferred independently. Therefore the expected time to transfer an object can be factored into two components: the expected number of chunks to transfer and the expected time to transfer a chunk.

$$\begin{aligned} &(expected\ time\ to\ transfer\ an\ object) \\ =\ &(expected\ number\ of\ chunks\ to\ transfer) \\ &\times(expected\ time\ to\ transfer\ a\ chunk) \end{aligned}$$

At first, let us take a look at how many chunks are expected to be transferred. If there is no undetectable error in a packet, the number of chunks to deliver is deterministic as the number of chunks in an object. However, due to errors which can only be detected by a checksum of a chunk, more chunks have to be delivered including end-to-end retransmissions of chunks. This increase is determined by a

chunk error rate as follows.

$$\begin{aligned} & (expected\ number\ of\ chunks\ to\ transfer) \\ = \ & (number\ of\ chunks\ in\ an\ object) \times \frac{1}{1-(chunk\ error\ rate)} \end{aligned}$$

Let us enumerate the denominator of the latter term.

$$\begin{aligned} & 1-(chunk\ error\ rate) \\ = \ & (correct\ chunk\ rate) \\ = \ & (end-to-end\ correct\ packet\ rate)^{\frac{c}{p}} \\ = \ & \{(one-hop\ correct\ packet\ rate)^{h}\}^{\frac{c}{p}} \\ = \ & (1-e)^{\frac{hc}{p}} \end{aligned}$$

Then, the expected number of chunks to transfer becomes

$$\begin{aligned} & (expected\ number\ of\ chunks\ to\ transfer) \\ = \ & \frac{o}{c} \times \frac{1}{(1-e)^{\frac{hc}{p}}} = \frac{o}{c} \times (1-e)^{-\frac{hc}{p}} \end{aligned}$$

Now let us see how much time it takes to transfer a single chunk. A transfer of a chunk has two main phases: a data transfer phase and an acknowledgment phase. During the data transfer phase, the source sends the entire chunk once to the sink. When the data transfer stage completes, the acknowledgment phase begins. The sink sends selective NACKs to the source. As in Section 3.10, only one selective NACK can be sent before every piece of data from that NACK is received.

$$\begin{aligned} & (expected\ time\ to\ transfer\ a\ chunk) \\ = \ & (expected\ time\ in\ data\ transfer\ phase) \\ & +(expected\ time\ in\ acknowledgment\ phase) \end{aligned}$$

In the data transfer phase, a request is sent from the sink to the source, then data moves from the source to the sink. After the first data packet arrives, the remaining

packets arrive separated by some interval; k. When the source is close to the sink, k is equal to the hop count of the source. However, when the source is is far away, k is equal to the depth of pipelining. In short, when $h < d$, $k = h$. Otherwise, $k = d$.

$$
\begin{aligned}
& (expected\ time\ in\ data\ transfer\ phase) \\
= \ & (time\ to\ send\ a\ request) + (time\ to\ receive\ the\ first\ data) \\
& \quad + (time\ to\ receive\ remaining\ data) \\
= \ & \frac{p}{b} \times h + \frac{p}{b} \times h + \frac{c-p}{b} \times k \\
= \ & \frac{1}{b} \times \{2ph + (c-p)k\}
\end{aligned}
$$

Since each NACK is a single packet, there are multiple NACKs. Each NACK is independent from others. The expected time in the acknowledgment phase can be factored into the expected number of NACKs and the expected time for a single NACK, as follows.

$$
\begin{aligned}
& (expected\ time\ in\ acknowledgment\ phase) \\
= \ & (expected\ number\ of\ NACKs) \\
& \quad \times (expected\ time\ for\ a\ single\ NACK)
\end{aligned}
$$

For each NACK, a NACK packet is sent from the sink to the source, then data moves from the source to the sink. After the first data packet arrives, remaining packets arrive separated by k, just like the data transfer phase. The operation of a single NACK is similar to the data transfer phase, except that the size of the data is determined by the number of sequences in a NACK packet not by the size of a chunk.

$$
\begin{aligned}
& (expected\ time\ for\ a\ single\ NACK) \\
= \ & (time\ to\ send\ a\ NACK) + (time\ to\ receive\ the\ first\ data) \\
& \quad + (time\ to\ receive\ remaining\ data) \\
= \ & \frac{p}{b} \times h + \frac{p}{b} \times h + \frac{(n-1) \times p}{b} \times k \\
= \ & \frac{p}{b} \times \{2h + (n-1)k\}
\end{aligned}
$$

The expected time in the acknowledgment phase becomes

$$\begin{aligned} & (\textit{expected time in acknowledgment phase}) \\ = & \frac{\frac{c}{p} \times (1 - s^h)}{n} \times \left[\frac{p}{b} \times \{2h + (n-1)k\} \right] \end{aligned}$$

However, there can be losses in the acknowledgment phase. To compensate for losses, we can multiply by a factor $\frac{1}{s^h}$.

$$\begin{aligned} & (\textit{expected time in acknowledgment phase}) \\ = & \frac{\frac{c}{p} \times (1 - s^h)}{n} \times \left[\frac{p}{b} \times \{2h + (n-1)k\} \right] \times \frac{1}{s^h} \\ = & \frac{1}{b} \times \left[\{2h + (n-1)k\} \times \frac{c}{n} \times \frac{1 - s^h}{s^h} \right] \end{aligned}$$

Then, the total expected time to transfer a chunk becomes,

$$\begin{aligned} & (\textit{expected time to transfer a chunk}) \\ = & \frac{1}{b} \times \{2ph + (c-p)k\} \\ & + \frac{1}{b} \times \left[\{2h + (n-1)k\} \times \frac{c}{n} \times \frac{1 - s^h}{s^h} \right] \\ = & \frac{1}{b} \times \left[2ph + (c-p)k + \{2h + (n-1)k\} \times \frac{c}{n} \times \frac{1 - s^h}{s^h} \right] \end{aligned}$$

At last, the expected time to transfer an object is

$$\begin{aligned} & (\textit{expected time to transfer an object}) \\ = & \frac{o}{c} \times (1 - e)^{-\frac{hc}{p}} \times \frac{1}{b} \times \left[2ph + (c-p)k + \{2h + (n-1)k\} \times \frac{c}{n} \times \frac{1 - s^h}{s^h} \right] \end{aligned}$$

6.4.2 Analysis of Model

Using a model proposed before, let us take a look at the effect of each term in Table 6.1 on a transfer time. Let us refer to the expected time to transfer an object as T.

- T increases linearly with the size of an object (o).
- The relationship between a chunk size (c) and T is not very clear in the model. Instead let us consider a high-level relationship. If c is too small, the relative overhead of sending a request and a NACK increases. Then, T also increases. On the other hand, if c is too large, the chance of a chunk getting an error increases. This will lead to a larger fraction of chunks being retransmitted, leading to a larger T. An optimal value of c will depend on other parameters.
- In $2ph + (c - p)k$, since $c \gg p$, the packet size (p) has a negligible effect on this term. In $(1 - e)^{-\frac{hc}{p}}$, as p increases, this term decreases. Therefore, a larger p leads to a smaller T. p will be eventually limited by the size of RAM in a mote. One subtle side effect that is not represented here is that as p increases, s decreases. That is to say, a longer packet has a higher probability of having an error and getting discarded, even though we have not witnessed a significant increase of s within the limit of p imposed by the RAM size.
- If a larger number of sequences can fit into a NACK packet (larger n), T will decrease. However, there is an upper bound on n, because increasing n requires increasing p.
- T is inversely proportional to the bandwidth at a routing layer (b).
- Regarding the hop count of the source (h), since $0 < 1 - e < 1$ and $0 < s < 1$, a larger h will take more time (larger T).
- In our formula, when $h < d$, a larger d (depth of pipelining) makes T larger. However, there is one subtle issue here that a smaller d leads to a smaller s, which can lead to a larger T. So we cannot make d arbitrarily small. This chapter is about finding an optimal balance between d and s to minimize T.
- A larger single-hop packet success rate at a routing layer (s) makes T smaller.

Table 6.2. Typical values for each term in the deployment at the Golden Gate Bridge.

Term	Value
o	512KB
c	512KB
p	20B
n	10
b	3.2KB/s
h	hop count of the source
d	5
s	0.999
e	4.4e-8

- A higher rate of false positives of packet error check (e) makes T larger.

6.4.3 Use Case

Let us take a concrete use case. In Table 6.2, values for terms are listed in the typical case of a deployment at the Golden Gate Bridge. The expected number of chunks to transfer becomes

$$\begin{aligned} & (\textit{expected number of chunks to transfer}) \\ = & \frac{o}{c} \times (1-e)^{-\frac{hc}{p}} \\ = & \frac{512KB}{512KB} \times (1-4.4\times 10^{-8})^{-\frac{h\times 512KB}{20B}} = (0.999999956)^{-25600h} \\ = & 0.99887^{-h} \end{aligned}$$

The expected time in the data transfer phase will be

$$\begin{aligned} & (\textit{expected time in data transfer phase}) \\ = & \frac{1}{b} \times \{2ph + (c-p)k\} \\ = & \frac{1}{3.2KB/s} \times \{2\times 20B \times h + (512KB - 20B)\times k\}\ (s) \\ = & 0.0125h + 159.99k\ (s) \end{aligned}$$

Then the expected time in the acknowledgment phase is

$$\begin{aligned}
&(\textit{expected time in acknowledgment phase}) \\
= &\quad \frac{1}{b} \times \left[\{2h + (n-1)k\} \times \frac{c}{n} \times \frac{1 - s^h}{s^h} \right] \\
= &\quad \frac{1}{3.2KB/s} \times \left[\{2h + (10-1)k\} \times \frac{512KB}{10} \times \frac{1 - 0.999^h}{0.999^h} \right] \ (s) \\
= &\quad (32h + 144k) \times \frac{1 - 0.999^h}{0.999^h} \ (s)
\end{aligned}$$

Finally the expected time to transfer an object becomes

$$\begin{aligned}
&(\textit{expected time to transfer an object}) \\
= &\quad (\textit{expected number of chunks to transfer}) \\
&\quad \times \{(\textit{expected time in data transfer phase}) \\
&\quad + (\textit{expected time in acknowledgment phase})\} \\
= &\quad 0.99887^{-h} \times \left\{ 0.0125h + 159.99k + (32h + 144k) \times \frac{1 - 0.999^h}{0.999^h} \right\} \ (s)
\end{aligned}$$

Let us take 3 cases, where the source node is 1, 5, and 46 hops away from the sink node (the base station node).

First, let us take a node which is one hop away from the base station. Then, $h = 1$ and $k = 1$.

$$\begin{aligned}
&(\textit{expected time to transfer an object}) \\
= &\quad 0.99887^{-1} \times \left\{ 0.0125 \cdot 1 + 159.99 \cdot 1 + (32 \cdot 1 + 144 \cdot 1) \times \frac{1 - 0.999^1}{0.999^1} \right\} \ (s) \\
= &\quad 1.0011 \times (160.00 + 0.17618) \ (s) \\
= &\quad 160.35 \ (s)
\end{aligned}$$

This yields a bandwidth of 3193 (B/s). It is higher than the bandwidth at the bridge, partly because a wired link between the base station mote and the base station PC limits the bandwidth in reality.

Then, let us take the case where the source is 5 hops away; $h = 5$ and $k = 5$.

$$
\begin{aligned}
& (\textit{expected time to transfer an object}) \\
= \;& 0.99887^{-5} \times \left\{ 0.0125 \cdot 5 + 159.99 \cdot 5 + (32 \cdot 5 + 144 \cdot 5) \times \frac{1 - 0.999^5}{0.999^5} \right\} (s) \\
= \;& 1.0057 \times (800.01 + 4.4132) \; (s) \\
= \;& 809.01 \; (s)
\end{aligned}
$$

The bandwidth becomes 633 (B/s).

Finally, let us take the case where the source is 46 hops away; $h = 46$ and $k = 5$.

$$
\begin{aligned}
& (\textit{expected time to transfer an object}) \\
= \;& 0.99887^{-46} \times \left\{ 0.0125 \cdot 46 + 159.99 \cdot 5 + (32 \cdot 46 + 144 \cdot 5) \times \frac{1 - 0.999^{46}}{0.999^{46}} \right\} (s) \\
= \;& 1.0534 \times (800.53 + 103.24) \; (s) \\
= \;& 952.03 \; (s)
\end{aligned}
$$

The bandwidth becomes 538 (B/s). This is a little bit higher than what is achieved in the bridge.

6.5 Flush

Flush is a receiver-initiated transport protocol for moving bulk data across a multihop wireless sensor network. Flush assumes that only one flow is active for a given sink at a time. The sink requests a large data object, which Flush divides into packets and sends in its entirety using a pipelined transmission scheme. End-to-end selective negative acknowledgments provide reliability: the sink repeatedly requests missing packets from the source until it receives all packets successfully. During a transfer, Flush continually estimates and communicates the bottleneck bandwidth using a dynamic rate control algorithm. To minimize overhead and maximize goodput, the algorithm uses no extra control packets, obtaining necessary information by snooping.

Flush makes five assumptions about the link layer below and the clients above:

- **Isolation:** A receiver has at most one active Flush flow. If there are multiple flows active in the network they do not interfere in any significant way.
- **Snooping:** A node can overhear all single-hop packets destined to other nodes.
- **Acknowledgments:** The link layer provides efficient single-hop acknowledgments.
- **Forward Routing:** Flush assumes it has an underlying best-effort *routing* service that can forward packets toward the data sink.
- **Reverse Delivery:** Flush assumes it has a reasonably reliable *delivery* mechanism that can forward packets from the data sink to the data source.

The reverse delivery service need not route the packets; a simple flood or a data-driven virtual circuit [69] is sufficient. The distinction between forward routing and reverse delivery exists because arbitrary, point-to-point routing in sensornets is uncommon and unnecessary for Flush. Only NACKs move in a reverse direction, which is only infrequent traffic. Moreover, links are not necessarily symmetric, so the cost of maintaining a reverse path is more than just reversing a forward route.

6.5.1 Overview

Flush moves through four phases.

Before initiating a data transfer, Flush first probes the depth of a target source node in a "Topology Query" phase. The request is sent using the underlying delivery protocol. The topology query is needed to tune the RTT and compute a timeout at the receiver.

Then a "Transfer Phase" starts. To initiate a data transfer, the sink sends a request for a data object to the source node in the network. Naming of the data object is outside of the scope of Flush, and is left to an application running above it. During the data transfer phase, the source sends packets to the sink using the maximum rate that does not cause intra-path interference. Over long paths, this rate pipelines packets over multiple hops, spatially reusing the channel. Section 6.5.3 provides intuition on how this works, and describes how Flush actively estimates this rate. The initial request contains conservative estimates for Flush's runtime parameters, such as the transmit rate, learned from field tests. When it receives the request, the data source starts sending the requested data packets, and nodes along the route begin their dynamic rate estimation. On subsequent requests or retransmissions, the sink uses estimated, rather than conservative, parameters.

The sink keeps track of which packets it receives. When a data transfer stage completes, an "Acknowledgment Phase" begins. The sink sends the sequence numbers of packets it did not receive back to the data source. Flush uses selective negative rather than positive acknowledgments because it assumes the end-to-end reception rate exceeds 50% substantially.

This process repeats in the acknowledgment phase until the sink has received the requested data *in toto*. When that occurs, the sink verifies the integrity of the data in an "Integrity Check" phase. If the integrity check fails, the sink discards the data and sends a fresh request for the data. If the check succeeds, the sink can request the next data object, perhaps from another node. Integrity is checked at the level of both packets and data objects.

To minimize control traffic overhead, Flush bases its estimates on local data and snoops on control information in forwarded data packets. The only explicit control packets are those four cases above (e.g. to start a flow and request end-to-end re-

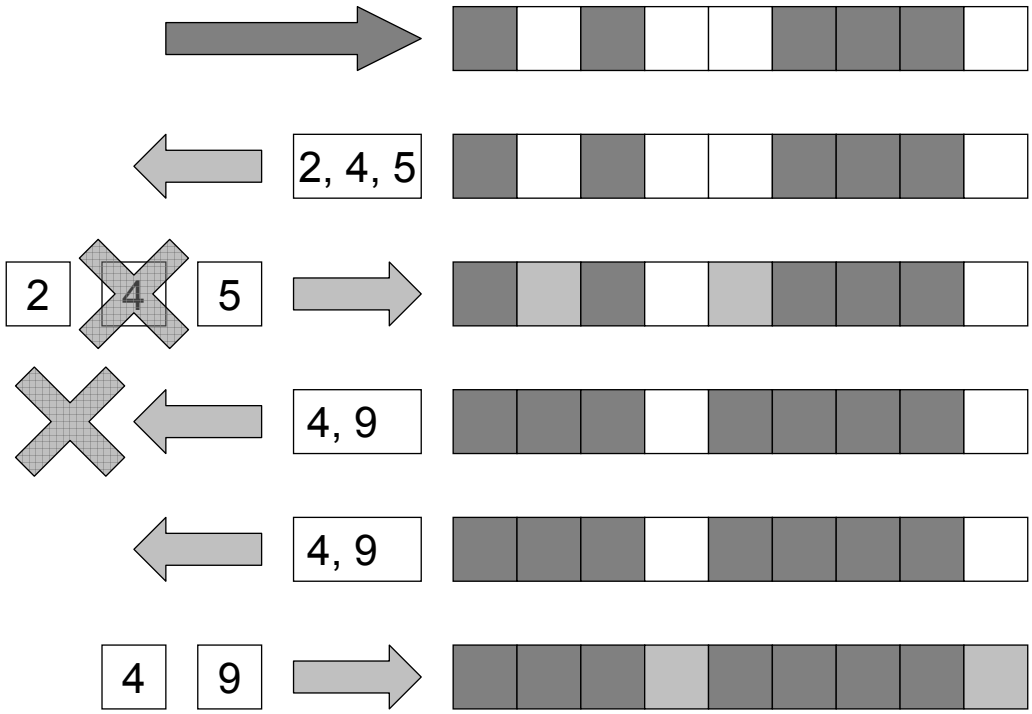

Figure 6.2. NACK transmission example. Flush has at most one NACK packet in flight at once.

transmissions). Flush is miserly with packet headers as well: three 1-byte fields are used for rate control and one field is used for the sequence number. The use of few control packets and small protocol headers helps to maximize data throughput, reducing transfer time. Section 6.6 describes a concrete implementation of the Flush protocol.

6.5.2 Reliability

Flush uses an end-to-end reliability protocol to be robust to node failures. Figure 6.2 shows a conceptual session of the protocol, where the data size is 9 packets, and a NACK packet can accommodate at most 3 sequence numbers. In the data transfer stage, the source sends all of the data packets, of which some are lost (2, 4, 5, and 9 in the example), either due to retransmission failures or queue overflows. The sink keeps track of all received packets. When it believes that the source has finished

sending data, the sink sends a single NACK packet, which can hold up to N sequence numbers, back to the source. This NACK contains the first N (where $N = 3$ in this case) sequence numbers of lost packets, 2, 4, and 5. The source retransmits the requested packets. This process continues until the sink has received every packet. The sink uses an estimate of the round-trip time (RTT) to decide when to send NACK packets in the event that all of the retransmissions are lost.

The sink sends a single NACK packet to simplify the end-to-end protocol. Having a series of NACKs would require signaling the source when the series was complete, to prevent interference along the path. The advantage of a series of NACKs would be that it could increase the transfer rate. In the worst case, using a single NACK means that retransmitting a single data packet can take two round-trip times. However, in practice Flush experiences few end-to-end losses due to its rate control and use of link layer acknowledgments and retransmissions.

In one experiment along a 48-hop path deployed in an outdoor setting, Flush had an end-to-end loss rate of 3.9%. For a 760 packet data object and room for 21 NACKs per retransmission request, this constitutes a cost of two extra round trip times – an acceptable cost given the complexity savings.

6.5.3 Rate Control

The protocol described above achieves Flush's first goal: reliable delivery. Flush's second goal is to minimize transfer time. Sending packets as quickly as the data link layer will allow poses problems in the multihop case. First, nodes forwarding packets cannot receive and send at the same time. Second, retransmissions of the same packet by successive nodes may prevent the reception of the following packets, in what is called *intra-path interference* [82]. One-hop medium access control algorithms will not solve the problem, due to hidden-terminal effects [69]. Third, rate mismatches may

cause queues further along the path to overflow. When a queue overflows, incoming packets get dropped leading to packet losses. Then, energy used for delivering the dropped packet is wasted, and more end-to-end retransmissions are required.

Flush strives to send packets at the maximum rate that will avoid intra-path interference. On long paths, it pipelines packets over multiple hops, allowing spatial reuse of the channel. To better understand the issues involved in pipelining packets, we first present an idealized model with strong simplifying assumptions. We then lift these assumptions as we present how Flush dynamically estimates its maximum sending rate.

A Conceptual Model

In this simplified model, there are N nodes arranged linearly plus a basestation B. Node N sends packets to the basestation through nodes $N-1, ..., 1$. Nodes forward a packet as soon as possible after receiving it. This can be a path through a larger network. Time is divided in slots of length $1 second$ (the unit is irrelevant to a result), and nodes are roughly synchronized. They can send exactly one packet per slot, and cannot both send and receive in the same slot. Nodes can only send and hear packets from neighbors one hop away, and there is no loss. There is however a variable range of interference, I: a node's transmission interferes with the reception of all nodes that are I hops away. As discussed in Section 6.3, a packet reception range is smaller than an interference range. For the simplicity of modeling, these two ranges are in a unit of hops.

We ask the question: *what is the fastest rate at which a node can send packets and not cause collisions*?

Figure 6.3 shows the maximum rate we can achieve in the simplified pipeline model for key values of N and I. If there is only one node, as in Figure 6.3(a), it

can send to the basestation at the maximum rate of 1 *packetspersecond*. There is no contention, as no other nodes transmit. For two nodes (b), the maximum rate falls to 1/2, because node 1 cannot send and receive at the same time, it is simply half duplex. The interference range starts to play a role if $N \geq 3$. In (c), node 3 has to wait for node 2's transmission to finish, and for node 1's, because node 1's transmission prevents node 2 from receiving. This is true for any node beyond 3 if we keep I constant, and the stable maximum rate is 1/3. Finally, in (d) let us assume I is 2 or larger. Any node past node 3 has to wait for its successor to send, and for its successor's *two* successors to send. Thus, the rate becomes 1/4. However, if I is 1, node 4 can send while node 1 is sending. Then the maximum rate is 1/3. Generalizing, the maximum transmission rate in this model for a node N hops away with interference range I is given by

$$r(N, I) = \frac{1}{\min(N, 2 + I)} \tag{6.1}$$

Thus, the maximum rate at which nodes can send depends on the interference range at each node, and on the path length (for short paths). If nodes send faster than this rate, there will be collisions and loss, and the goodput can greatly suffer. If nodes send slower than this rate, throughput will be lower than the maximum possible. The challenge is to efficiently discover and communicate this rate, which will changes with the environment. MintRoute yields a small I value. One open question is whether other routing protocol choose next hops that result in $I \gg 1$.

Dynamic Rate Control

We now describe how Flush dynamically estimates the sending rate that maximizes the pipeline utilization. The algorithm is agile in reacting to increases and decreases in per-hop throughput and interference range, and is stable when link qualities do not vary. The rate control algorithm follows two basic rules:

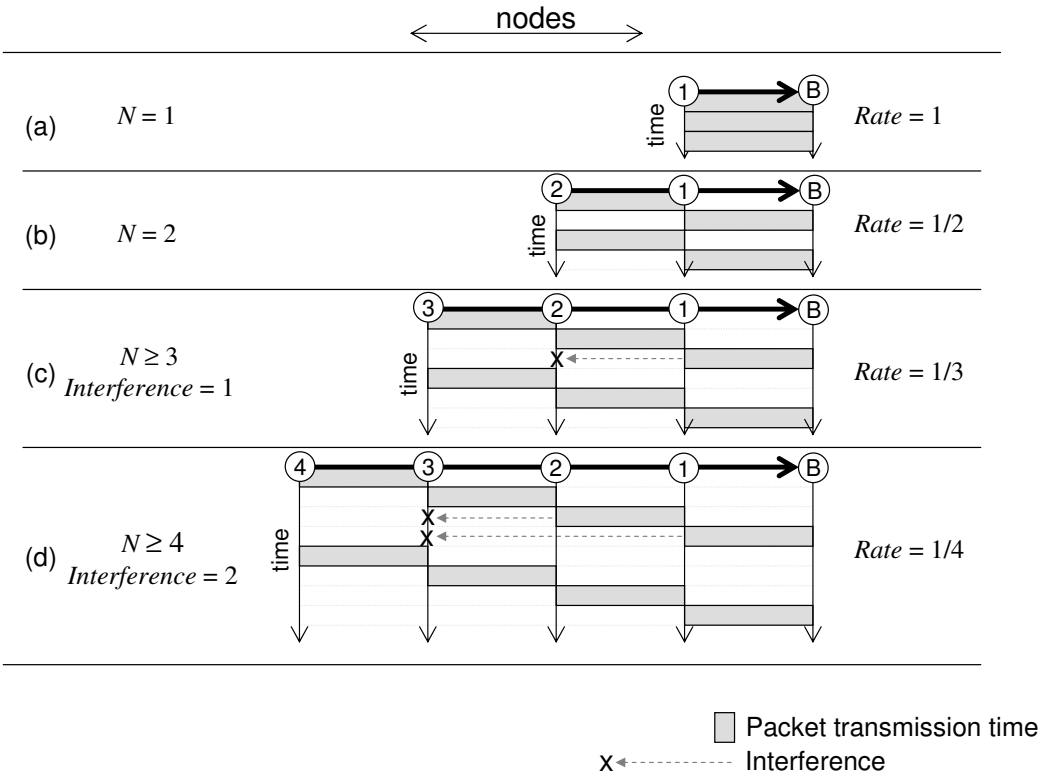

Figure 6.3. Maximum sending rate without collision in the simplified pipelining model, for different numbers of nodes (N) and interference ranges (I).

- **Rule 1:** A node should only transmit when its successor is free from interference.

- **Rule 2:** A node's sending rate cannot exceed the sending rate of its successor.

Rule 1 derives from a generalization of our simple pipelining model: after sending a packet, a node has to wait for (i) its successor to forward the packet, and for (ii) all nodes whose transmissions interfere with the successor's reception to forward the packet. This minimizes intra-path interference. Rule 2 prevents rate mismatches: when applied recursively from the sink to the source, it tells us that the source cannot send faster than the slowest node along the path. This rule minimizes losses due to queue overflows for all nodes. The two rules form a control algorithm and are essentially the goal of rate control. Within these two rules, we want to maximize the sending rate.

Let us consider a more realistic model, where packet reception range is not a single hop. A routing layer would not choose the farthest node as the next hop in favor of a stability of a link. Establishing the best rate requires each node i to determine the smallest safe inter-packet delay d_i (from start to start) that maintains Rule 1. As shown in Figure 6.4, d_i comprises the time node i takes to send a packet, δ_i, plus the time it takes for its successor to be free from interference, H_{i-1}. δ_i is measured between the start of the first attempt at transmitting a packet and the first successfully acknowledged transmission. H_{i-1} is defined for ease of explanation, and has two components: the successor's own transmission time δ_{i-1} and the time f_{i-1} during which its interfering successors are transmitting. We call the set of these interfering nodes I_{i-1}. In summary, for node i, $d_i = \delta_i + (\delta_{i-1} + f_{i-1})$: the minimum delay is the sum of the time it takes a node to transmit a packet, the time it takes the next hop to transmit the packet, and the time it takes that packet to move out of the next hop's interference range.

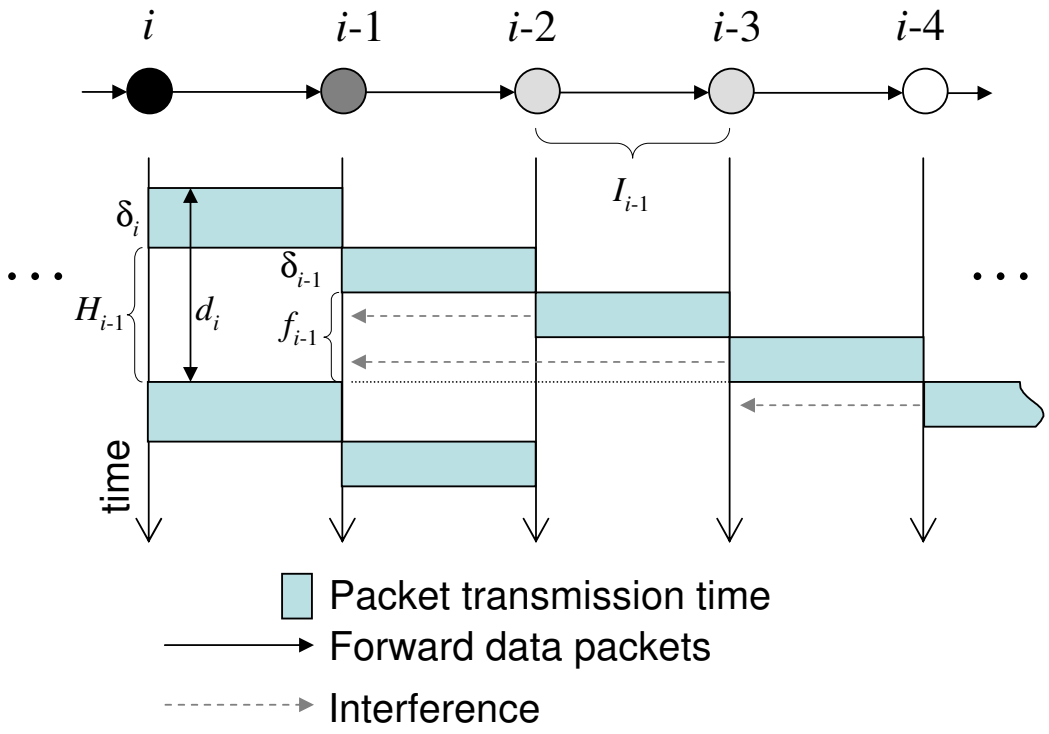

Figure 6.4. A detailed look at pipelining from the perspective of node i, with the quantities relevant to the algorithm shown.

Flush can locally estimate δ_i by measuring the time it takes to send each packet. However, each node needs to obtain δ_{i-1} and f_{i-1} from its successor, because most likely node i cannot detect all nodes that interfere with reception at node $(i-1)$. Instead of separate control packets, Flush relies on snooping to communicate these parameters among neighbors. Every Flush data packet transmitted from node $(i-1)$ contains δ_{i-1} and f_{i-1}. Using these, node i is able to approximate its own f_i as the sum of the δs of all successors that node i can hear. As the values δ_{i-1} and f_{i-1} of its successor may change over time and space due to environmental effects such as path and noise, Flush continually estimates and updates δ_i and f_i.

Let us look at an example. In Figure 6.5, node 7 determines, by overhearing traffic, that the transmissions of node 6 and 5 (but not node 4) can interfere with reception of traffic from node 8. This means that node 7 can not hear a new packet from node 8 until node 5 finishes forwarding the previous packet. Thus, $f_7 = \delta_6 + \delta_5$.

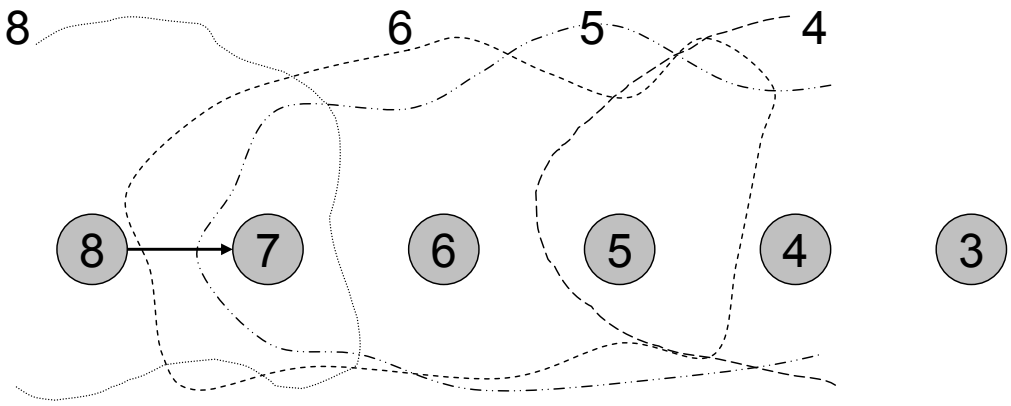

Figure 6.5. Packet transfer from node 8 to node 7 interferes with transfer from node 5 to node 4. However it does not interfere with transfer from node 4 to node 3

Node 7 cannot receive a packet while sending, and $H_7 = \delta_7 + f_7$. Considering node 8's own transmission time, $d_8 = \delta_8 + H_7 = \delta_8 + \delta_7 + f_7 = \delta_8 + \delta_7 + \delta_6 + \delta_5$. So the interval between two packets should be separated by at least that time.

As described above, each node attempts to determine its own fastest sending rate. This, however, is not enough for a node to ensure the optimal sending rate for the path. Rule 2 provides the necessary condition: a node should not send faster than its successor's sending rate.

When applied recursively, Rule 2 leads to the minimum sending interval at node i: $D_i = \max(d_i, D_{i-1})$. Most importantly, this determines the sending interval at the source, which is the maximum d_i over all nodes. This rate is easy to determine at each node: all nodes simply include D_i in their data packets, so that the previous node can learn this value by snooping. To achieve the best rate it is necessary and sufficient that the source send at this rate, but as we show in Section 6.8, it is beneficial to impose a rate limit of D_i for each node i in the path, and not only for the source. Flush takes an advantage of in-network rate control on a hop-by-hop basis. Figure 6.6 presents a concise specification of the rate control algorithm and how it embodies the two simple rules described above.

The Flush rate control algorithm

(1)	δ_i	:	actual transmission time at node i	
(2)	I_i	:	set of forward interferers at node i	
(3)	f_i	=	$\sum_{k \in I_i} \delta_k$	
(4)	d_i	=	$\delta_i + (\delta_{i-1} + f_{i-1})$	(Rule 1)
(5)	D_i	=	$\max(d_i, D_{i-1})$	(Rule 2)

Figure 6.6. The Flush rate control algorithm. D_i determines the smallest sending interval at node i.

Finally, while the above formulation works in a steady state, environmental effects and dynamics as well as simple probability can cause a node's D_i to increase or decrease. Because it takes n packets for a change in a delay estimate to propagate back n hops, for a period of time there will be a rate mismatch between incoming and outgoing rates. In the case an incoming rate is higher than an outgoing rate, queues will begin to fill. In order to allow the queues to drain, a node needs to temporarily tell its previous hop to slow down. We use a simple mechanism to do this, which has proved efficient: while a node's queue occupancy exceeds a specified threshold, it temporarily increases the delay it advertises by doubling δ_i.

6.5.4 Identifying the Interference Set

A node can interfere with transmissions beyond its own packet delivery range, as introduced in Section 6.3. Two nodes which can communicate well when other nodes are quiet, may not hear each other when other nodes are sending packets, even if those packet can not be decoded. The fact that Flush assumes it can hear its interferers raises the question of how common this effect is in the routes it chooses. Packet reception rates follow a curve with respect to the signal-to-noise ratio, but for simplicity's sake we consider the case where it follows a simple threshold not including capture, such that reception is worse than reality.

Consider nodes m_i and node m_{i-1} along a path, where m_i is trying to send a packet to m_{i-1} with received signal strength S_i as measured at m_{i-1}. For another node j to conflict with this transmission, it must have a signal strength of at least $S_i - T$, as measured at the receiver, where T is the SNR threshold. For a node j to be a jammer – a node which can conflict with the transmission but cannot be heard – $S_j > S_i - T$ and $S_j < N_{i-1} + T$, where N_{i-1} is other sources of noise at m_{i-1}. That is, its signal strength must be within T of m_i's received signal strength at m_{i-1} to conflict and also be within T of the noise floor such that it can never be heard. For there to be a jammer, S_i can be at most $2T$ stronger than N_{i-1}.

Using the 100-node Mirage testbed, we examined the topology that Flush's underlying routing algorithm, MintRoute [86], establishes. We measured the noise floor of each node by sampling the CC2420 RSSI register and the signal strength of its predecessor using TinyOS packet metadata. Figure 6.7 shows the results. Fewer than 20% of the links chosen are within the range $[N_i + T, N_i + 2T]$. For there to be a jammer, it must be within T of these links, or approximately 3.5dBm. While Flush's interference estimation is not perfect, its window of inaccuracy is narrow. One open question is that while this holds true for our given hardware (CC2420) at a given power level with a specific routing layer (MintRoute), whether the result remains true for other combinations of hardware, power level, and routing layers.

6.6 Implementation

To evaluate the Flush algorithms, we implemented them in the nesC programming language [36] and the TinyOS [42] operating system for sensor networks. Our implementation runs on the Crossbow MicaZ platform but we believe porting it to other platforms like the Mica2 or Telos would be straightforward.

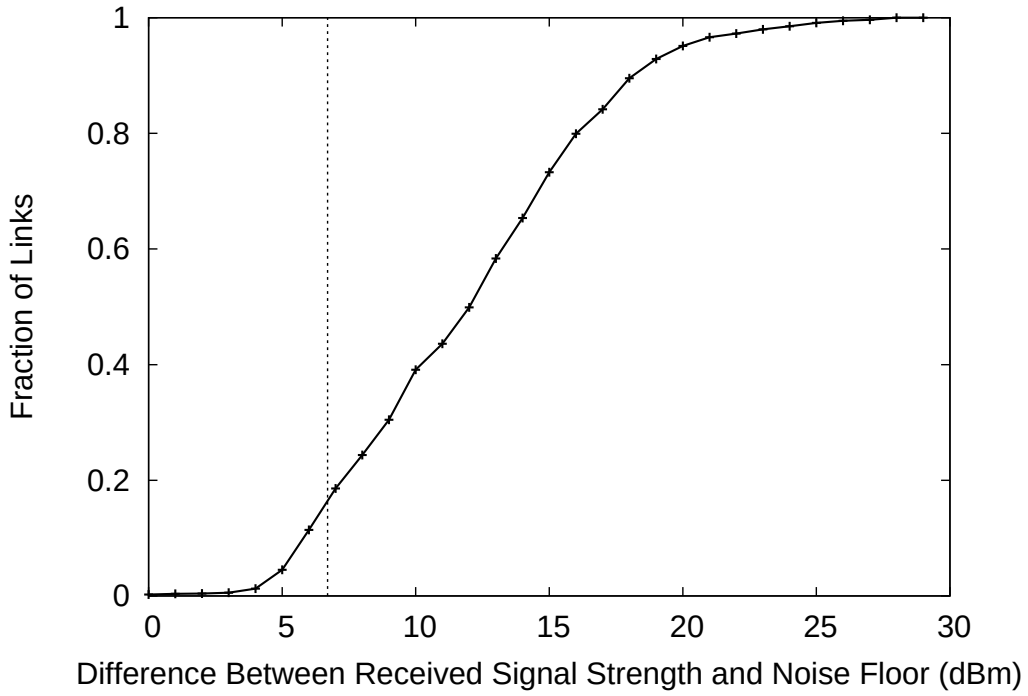

Figure 6.7. CDF of the difference between the received signal strength from a predecessor and the local noise floor. The dotted line indicated twice the SNR threshold. Links with an SNR exceeding this threshold will not be undetectably affected by interferers. A large fraction of interferers are detectable and avoidable.

6.6.1 Protocol Engine

The Flush *protocol engine* implements the reliable block transfer service. This module receives and processes data read requests and selective NACKs from the receiver. The requests are forwarded to the application, which is responsible for returning the requested data. After the requested data have been returned, the protocol engine writes the data and identifying sequence numbers into a packet and then submits the packet to the routing layer at the rate specified by the packet delay estimator, which is discussed below.

Although the Flush interface supports 32-bit offsets, a packet sequence number is a 16-bit number, which limits the size of an object. So our current implementation only supports a limited range of data object sizes: 917,504 bytes (64K x 14 bytes/packet), 1,114,112 bytes (64K x 17 bytes/packet), or 2,293,760 bytes (64K x 35 bytes/packet), depending on the number of bytes available for the data payload in each packet. Again, this restriction comes from the use of 16-bit sequence numbers, which we use in part to conserve data payload and in part because the largest flash memory available on today's sensor nodes is 1 MB, and the size of an object should be smaller than that.

6.6.2 Routing Layer

MintRoute [86] is used to convergecast packets from the source to the sink, which in our case is the root of a collection tree. Flush does not place many restrictions on the path other than it be formed by reasonably stable bidirectional links. Therefore, we believe Flush should work over most multihop routing protocols like CLDP [49], TinyAODV [7], or BVR [33]. However, we do foresee some difficulty using routing protocols that do not support reasonably stable paths. Some routing protocols, for

example, dynamically choose distinct next hops for packets with identical destinations [65]. It is neither obvious that our interference estimation algorithm would work with such protocols nor clear that a high rate could be achieved or sustained because Flush would be unable to coordinate the transmissions of the distinct next hops.

Flush uses the TinyOS flooding protocol, `Bcast`, to send packets from the receiver to the source for both initiating a transfer and sending end-to-end selective NACKs. `Bcast` implements a simple flood: each node rebroadcasts each unique packet exactly once, assuming there is room in the queue to do so. A packet is rebroadcast with a small, random delay. Although we chose a flood, any reasonably reliable delivery protocol could have been used (e.g. a virtual circuit, an epidemic dissemination protocol, or a point-to-point routable protocol), even though this may impact performance.

6.6.3 Packet Delay Estimator

The *packet delay estimator* implements the Flush rate control and interference estimation algorithms. The estimator uses the MintRoute `Snoop` interface to intercept packets sent by a node's successor hops and predecessor hop along the path of a flow, for estimating the set of interferers. The δ, f, and D fields, used by the estimator, are extracted from the next hop's intercepted transmissions.

The Flush estimator extracts the received signal strength indicator (RSSI) of packets received from the predecessor hop and snooped from all successor hops along the routing path. These RSSI values are smoothed using an exponentially-weighted moving average to filter out transients on single-packet timescales. History is weighted more heavily because RSSI is typically quite stable and outliers are rare, so a single outlier should have little influence on the RSSI estimate. A node i considers a successor node $(i - j)$ an interferer of node $i + 1$ at time t if, for any $j > 1$,

$rssi_{i+1}(t) - rssi_{i-j}(t) < 10$ dBm. The threshold of 10 dBm was chosen after consulting the literature [70] and empirically evaluating a range of values.

Since the forwarding time f_i was defined to be the time it takes for a packet transmitted by a node i to no longer interfere with reception at node i, we set f_i accordingly, such that all values j for which the above inequality holds contribute to f_i. We implemented a timeout mechanism under which if no packets are overheard from a successor during an interval spanning 100 consecutive packet receptions, that successor is no longer considered an interferer. However, we left this mechanism turned off so none of the experiments presented in this work use this timeout. Based in part on the preceding information, the estimator computes d_i, the minimum delay between adjacent packet transmissions. The estimator provides the delay information, D_i, to the protocol engine to allow the source to set the sending rate. The estimator also provides the parameters δ_i, f_i, D_i to the queuing component so that it can insert the current values of these variables into a packet immediately prior to transmission.

6.6.4 Queuing

Queues provide buffer space during transient rate mismatches which are typically due to changes in link quality. In Flush, these mismatches can occur over short time scales because rate estimates are based on averaged interval values, so unexpected losses or retransmissions can occur. Note that a retransmission does not follow a schedule and occurs as soon as a CSMA backoff ends. Also, control information can take longer to propagate than data: at a node i along the path of a flow, data packets are forwarded with a rate $\frac{1}{\delta_i}$ while control information propagates in the reverse direction with a rate $\frac{1}{\delta_i + f_i}$. The forwarding interference time f_i is typically two to three times larger than the packet sending delay δ_i, so control information

flows three to four times slower than data. Since it can take some time for the control information to propagate to the source, queues provide buffering during this time.

Our implementation of Flush uses a 16-deep *rate-limited queue*. Our queue is a modified version of `QueuedSend`, the standard TinyOS packet queue. Our version, called `RatedQueuedSend`, implements several functions that are not available in the standard component. First, our version measures the local forwarding delay, δ, and keeps an exponentially-weighted moving average over it. This smoothed version of δ is provided to the packet estimator. Second, `RatedQueuedSend` enforces the queue departure delay D_i specified by the packet delay estimator. Third, when a node becomes congested, it doubles the delay (δ') advertised to its descendants but continues to drain its own queue with the smaller delay until it is no longer congested. We chose a queue depth of 5, about one-third of the queue size, as our congestion threshold. Fourth, the queue inserts the then-current local delay information into a packet immediately preceding transmission. Fifth, `RatedQueuedSend` retransmits a packet up to four times (for a total for five transmissions) before dropping it and attempting to send the next packet. Finally, the maximum queuing delay is bounded, which ensures the queue will be drained eventually, even if a node finds itself neighborless. In addition to six core functions, it also records all parameters together with a globally synchronized timestamp, whenever a packet is submitted to a lower layer including retransmissions. This information is used to construct detailed views of Flush for evaluation. For example, by looking at when each unique packet is submitted to the MAC layer, it is possible to calculate a real sending rate.

Figure 6.8 shows the overall structure of Flush. This figure shows where each component and function is located and how they interact. δ' and D' are used for an advertisement. δ' gets doubled when the queue length grows. D' is computed using δ'.

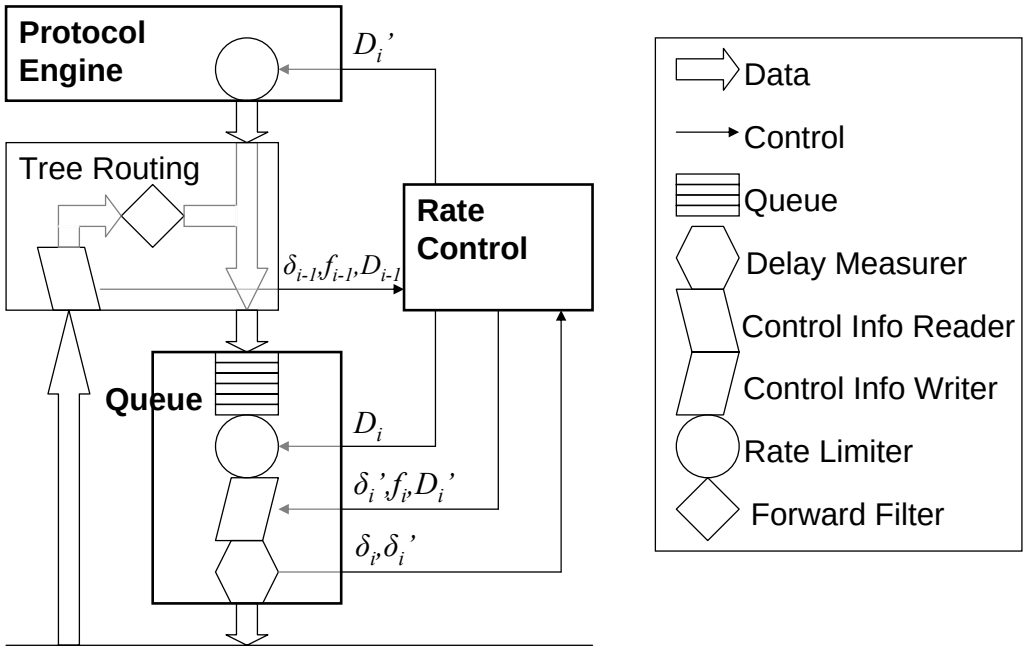

Figure 6.8. Overall structure of Flush

6.6.5 Link Layer

Flush employs link-layer retransmissions at an underlying layer to reduce the number of expensive end-to-end transmissions that are needed. Flush also snoops on the channel to overhear the next hop's delay information and the predecessor hop and successor hops' RSSI values. Unfortunately, these two requirements – hardware-based link layer retransmission and snooping – are at odds with each other on the MicaZ mote. The CC2420 radio used in on the MicaZ does not simultaneously support hardware acknowledgments and snooping, and the default TinyOS distribution does not provide software acknowledgments. Our implementation enables the snooping feature of the CC2420 and disables hardware acknowledgments. We use a modified version of the TinyOS MAC, `CC2420RadioM`, which provides software acknowledgments [73]. We configure the radio to perform a clear channel assessment and employ CSMA/CA for medium access. Since Flush performs rate control at the network layer, and does not

schedule packets at the link layer, CSMA decreases collision due to retransmissions during transient periods.

6.6.6 Protocol Overhead

Our implementation of Flush uses the default TinyOS packet which provides 29 bytes of payload above the link layer. The allocation of these bytes is as follow: MintRoute (7 bytes), sequence numbers (2 bytes), Flush rate control fields (3 bytes), and application payload (17 bytes). Flush shares packet structures and types with Straw. Details of Straw packet and control structures are in Section 3.10. Since in the default implementation, only 17 bytes are available for the application payload, Flush's effective data throughput suffers. During subsequent experiments, we changed the application payload to 35 bytes. Future work might consider an A-law style compressor/expander (compander) [8], used in audio compression, to provide high resolution for expected delay values while allowing small or large outliers to be represented.

6.6.7 Memory and Code Footprint

We round out our implementation of Flush by reviewing its footprint. Flush uses 629 bytes of RAM and 6,058 bytes of code, including the routines used to debug, record performance statistics, and log traces. These constitute 15.4% of RAM and 4.62% of program ROM space on the MicaZ platform. Table 6.3 shows a detailed breakdown of memory footprint and code size. The Protocol Engine accounts for 301 out of the 629 bytes of RAM, or 47.9% of Flush's memory usage. A significant fraction of this memory (180 bytes) is used for message buffers, which are used to hold prefetched data.

Table 6.3. Memory and code footprint for key Flush components compared with the regular TinyOS distribution of these components (where applicable). Flush increases RAM by 629 bytes and ROM by 6,058 bytes.

	Memory Footprint		Code Size	
Component	Regular	Flush	Regular	Flush
Queue	230	265	380	1,320
Routing	754	938	76	2,022
Proto Eng	-	301	-	2,056
Delay Est	-	109	-	1,116
Total	984	1,613	456	6,514
Increase	629		6,058	

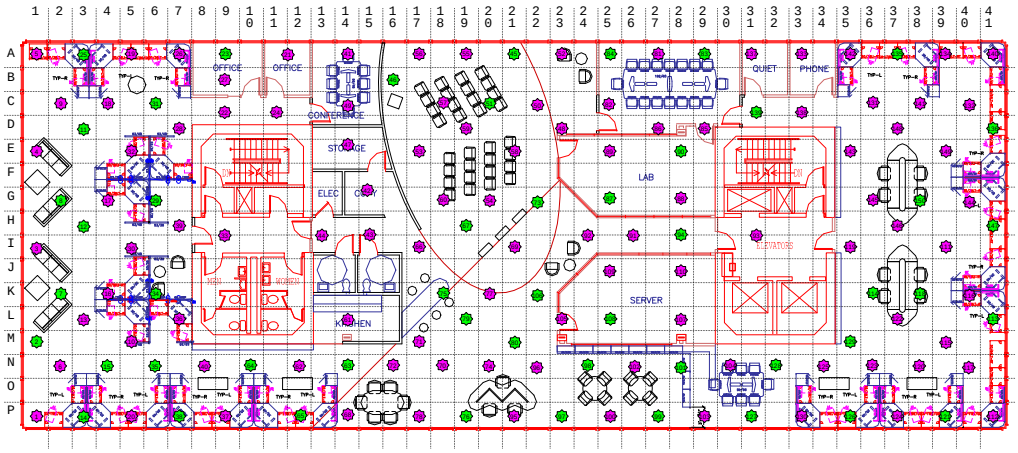

Figure 6.9. Mirage Testbed at Intel Research Berkeley. Purple (darker) stars are 100 MicaZ nodes used in experiments.

6.7 Experimental Methodology

We evaluate the effectiveness of Flush through a series of experiments on the Intel Research Berkeley sensornet testbed, Mirage, as well as a 79-node, ad hoc, outdoor testbed at RFS. The Mirage testbed consists of 100 MicaZ nodes, see Figure 6.9.

We used node 0, in the southwest corner, as the sink or basestation. After setting the MicaZ node's CC2420 radio power level to -11 dBm, the diameter of the resulting network varied between 6 and 7 hops in our experiments. The end-to-end quality of the paths was generally good, but in Section 6.8.4 we present the results of an experiment in which the quality of a link was artificially degraded. The outdoor

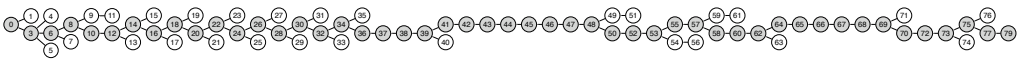

Figure 6.10. The network used for the scalability experiment. Of the 79 total nodes, the 48 nodes shown in gray were on the test path. This test is a demonstration that Flush works over a long path and is *not* limited to a linear topology, as will be shown in other tests.

testbed consisted of 79 nodes deployed on the ground in a linear fashion with 3ft spacing in an open area, creating an approximately 48 hop network. The physical extent of the network spanned 243ft. The radio transmission power was lowered to decrease range, but not so much so that the network would be void of interference. The resulting topology is shown in Figure 6.10, where the rightmost node is 48 hops from the root, which is the leftmost node.

We use the MintRoute [86] protocol to form a collection tree with the sink located at the root. MintRoute uses periodic beacons to update link quality estimates. Prior to starting Flush, we allow MintRoute to form a routing tree, and then freeze this tree for the duration of the Flush transfer. In Section 6.9, we discuss Flush's interactions with other protocols and applications.

In our experiments the basestation issues a request for data from a specific node. All the nodes are time synchronized prior to each trial, and they log to flash memory the following information for each packet sent: the Flush sequence number, timestamp, the values of δ, f, and D, and the instantaneous queue length. After each run we collect the data from all of the nodes *using Flush itself*. We compare Flush with a static algorithm that fixes the sending rate. Then, to evaluate the benefits of using hop-by-hop in-network rate control, we compare Flush with a variation which only adjusts the sending rate at the *source*, even though the intermediate nodes still estimate and propagate the delays as described in Section 6.5.3.

To better appreciate the choice of evaluation metrics presented in this section, we

revisit Flush's design goals. First, Flush requires complete reliability, which the protocol design itself provides (and our experiments validate, across all trials, networks, and data sizes). The remaining goals are to maximize goodput, minimize transfer time, and adapt gracefully to dynamic changes in the network.

6.8 Evaluation

We perform a series of experiments to evaluate Flush's performance. First, we establish a baseline using fixed rates against which we compare Flush's performance. This baseline also allows us to factor out overhead common to all protocols and avoid concerning ourselves with questions like, "why is there a large disparity between the raw radio rate of 250 kbps and Flush?" Next, we compare Flush against the baseline. In the following subsection, we then take a more in-depth look at Flush's performance. We also explore the benefits of hop-by-hop rate control of Flush compared with controlling the rate at only the source. Then, we consider the effects of abrupt link quality changes on Flush's performance and analyze Flush's response to a parent change in the middle of a transfer. The preceding experiments are carried out on the Mirage testbed [11]. Next, we consider Flush's scalability by evaluating its performance over a 48-hop, ad hoc, outdoor wireless network at the Richmond Field Station (RFS) [12]. To the best of our knowledge, this is the longest multihop path used in evaluating a protocol in the wireless literature.

6.8.1 High Level Performance

In this section, we examine the effective packet throughput and bandwidth by comparing Flush with various values of the fixed-rate algorithm. To establish a baseline for comparison, we first consider the packet throughput achieved by the fixed

rate algorithm. For each inter-packet interval of 10, 20, and 40ms, we reliably transfer 17,000 bytes in 850 packets along a fixed path of length ranging from zero to six hops. The smallest inter-packet interval our hardware platform physically sustains is 8ms, which we empirically discover using a one hop goodput test.

We begin by initiating a multihop transfer from a source node six hops away from the sink. After this transfer completes, we perform a different transfer from the 5th hop, and continue this process up to, and including, a transfer in which the source node is only one hop away. The data is also transferred from a basestation mote to a basestation PC. Figure 6.11 shows the results of these trials. The X-axis represents a hop count from the basestation sink. The 0th hop indicates the basestation mote. The Y-axis represents an effective bandwidth in terms of *packets per second*. Each line indicates a different inter-packet interval value among 10ms, 20ms, and 40ms. Each point in the graph is the average of four runs, with the vertical bars indicating the standard deviation.

Each path length has a fixed sending rate which performs best. When transferring over one hop there is no forward interference, and a node can send packets as fast as the hardware itself can handle. As the path length increases, the rate has to be throttled down, as packets further along in the pipeline interfere with subsequent ones. For example, sending with an interval of 20ms provides the best packet throughput over 3, 4, 5, and 6 hop transfers, as there are too many losses due to queue overflows when sending faster. Slower rates do not cause interference, but also do not achieve the full packet throughput as the network is underutilized. From the view point of a conceptual model (Equation 6.1), when hop count (N) is 1 and 2, N determines a rate and an inter-packet interval. 10ms is close to this rate, while 20ms and 40ms is too large and channel capacity is underutilized. As N increases, 20ms gets closer to an optimal inter-packet time than 10ms. 10ms is too small and has a lower throughput than 20ms. Detailed reasons for this phenomenon will be investigated later. Straw is

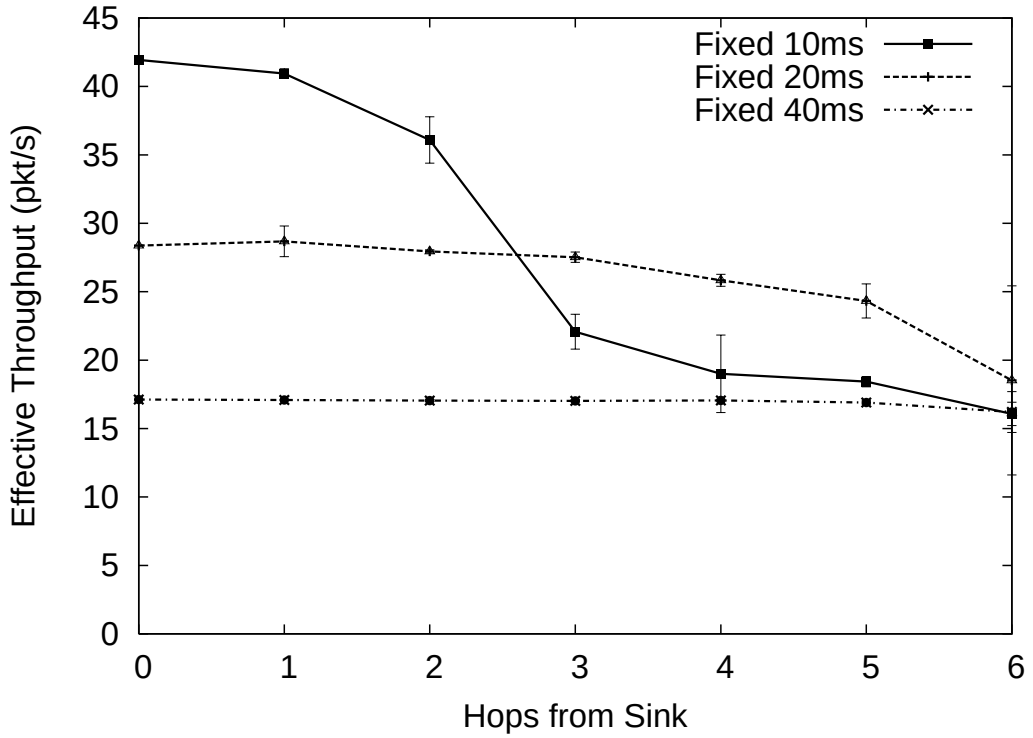

Figure 6.11. Packet throughput of fixed rate streams over different hop counts. The optimal fixed rate depends on the distance from the sink.

the same as a fixed rate, except that it chooses this fixed value by looking at a depth of a node.

Figure 6.12 shows the results of the same experiments with Flush. The circles in the figure show the performance of the best fixed rate at the specific path length. Flush performs very close or better than this envelope, on a packets/second basis. These results suggest that Flush's rate control algorithm is automatically adapting to select the best sustainable sending rate along the path and optimizing for changing link qualities and forward interference.

Figure 6.13 shows the effective data bandwidth on a bytes/second basis. The effective bandwidth of Flush is sometimes lower than the best fixed rate because we adjust for protocol overhead (see Subsection 6.6.6). In this figure, Flush's rate control

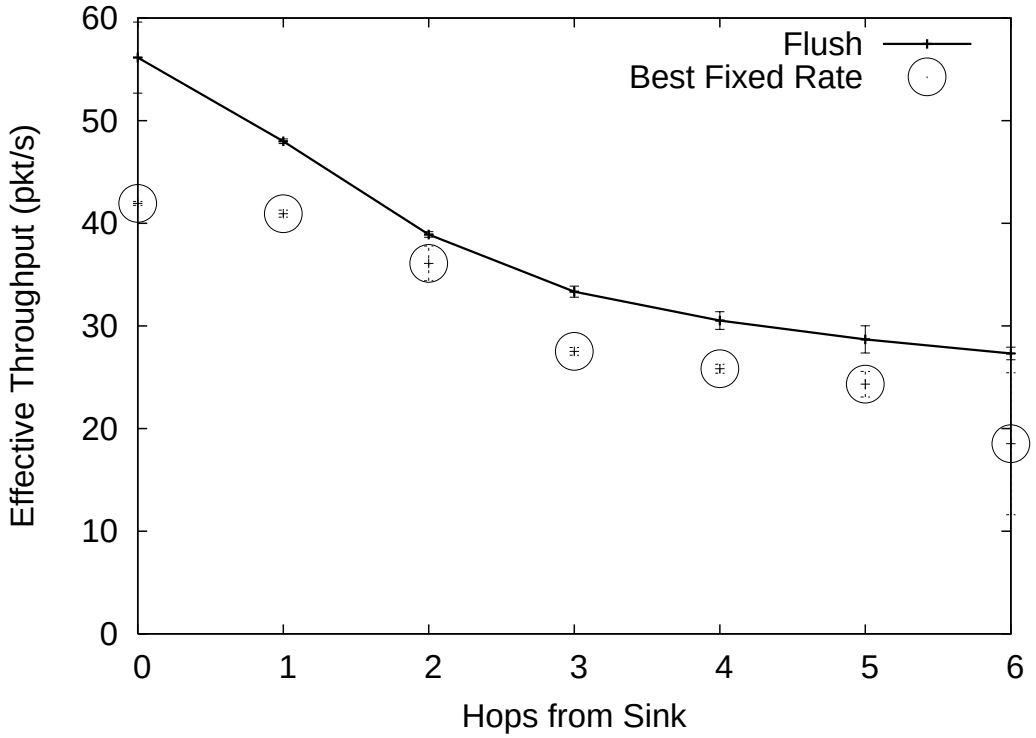

Figure 6.12. Effective packet throughput of Flush compared to the best fixed rate at each hop, taken from Figure 6.11. Flush tracks the best fixed packet rate.

header fields account for 3 bytes (δ, f, and D each require 1 byte), leaving only 17 bytes for the payload – a 15% protocol overhead penalty.

These figures show that fixing a sending interval may work best for a specific environment, but that no single fixed rate performs well across different path lengths, topologies, and link qualities. We could fine tune the rate for a specific deployment and perhaps get slightly better performance than Flush, but that process is cumbersome because it requires tuning the rate for every node and brittle because it does not handle changes in the network topology or variations in link quality gracefully.

The Chipcon CC2420 radio in a MicaZ mote has 250kbps capacity. Bandwidth from a node that is one hop away from a basestation is 6,530bps. Let us take a look at why this difference exists. The TinyOS CSMA MAC layer provides an effective packet throughput of 160 packets per second over a single-hop wireless link. A wired

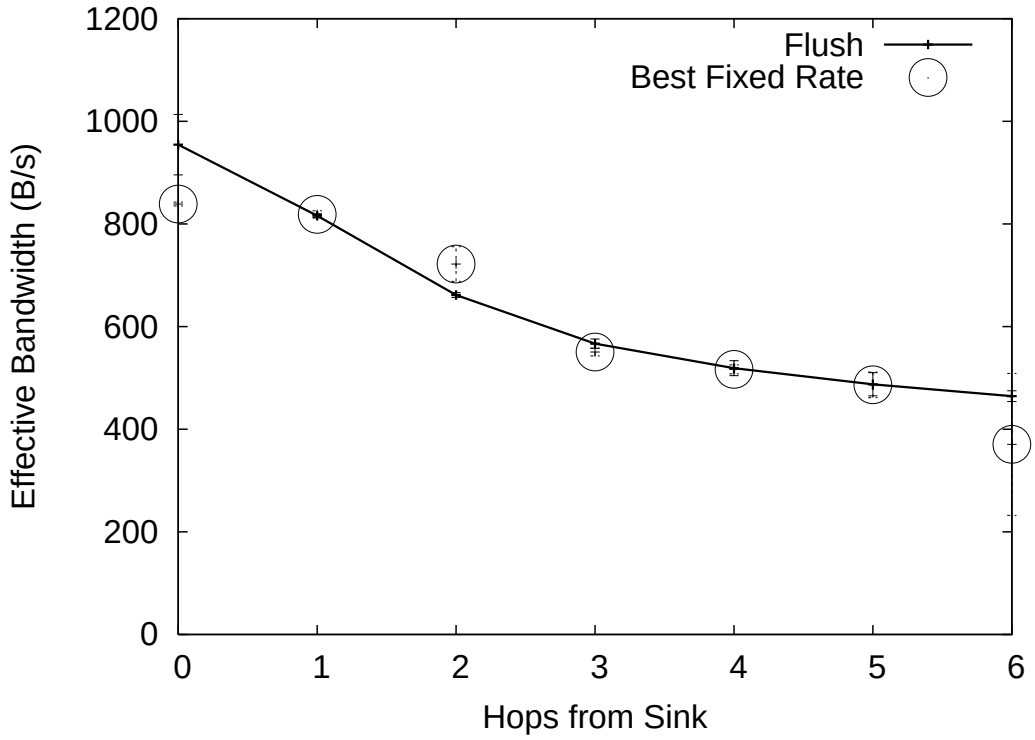

Figure 6.13. Effective bandwidth of Flush compared to the best fixed rate at each hop, taken from Figure 6.11. Flush's protocol overhead reduces the effective data rate.

link between a basestation mote and a basestation PC has 100 packets per second throughput. It is not clear how much two transfers can overlap. Assuming they perfectly overlap, then an effective packet throughput through two links will become 100 packets per second. With a 29-byte payload at the MAC layer, an effective bandwidth at this layer is 23,200bps. The routing layer (MintRoute) uses a 7-byte header reducing effective bandwidth to 17,600bps. For reliable data collection, 2 bytes are further used to include a sequence number. Flush also uses 3 bytes for rate control information. Then possible bandwidth becomes 13,600bps. Flush has overhead of a topology query, end-to-end retransmissions, and an integrity check. 69.8% of time is used for actual data transfer in this case. Then the final possible effective bandwidth becomes 9,490bps. Therefore, assuming that a basestation mote perfectly overlaps wireless and wired transfers, an effective bandwidth of 6,530bps is 68.8% of a possible capacity. If there can be no overlap, a possible capacity is 5,840bps, which is smaller than what is achieved by Flush. Investigation of how much overlap actually happens is future work. However, through this process, it is shown that the bandwidth of Flush is in a reasonable range and not a severe underutilization.

Figure 6.14 compares the efficiency, in terms of losses, of the different alternatives from the experiment above. The X-axis represents hop count from the sink and the Y-axis represents the average number of transmissions of nodes along the path. We use this average number of packets sent *per hop* in our transfer of 1000 packets as an indicator for the efficiency of each sending rate. Effective bandwidth is negatively correlated with the number of messages transmitted, as the transfers with a small fixed interval lose many packets due to queue overflows. As in the previous graphs, Flush performs close to the best fixed rate at each path length. Note that the extra packets transmitted by Flush and by the "Fixed 40ms" flow are mostly due to link-level retransmissions, which depend on the link qualities along the path. Flush and "Fixed 40ms" flow *experienced no losses due to queue overflows.* In contrast, the

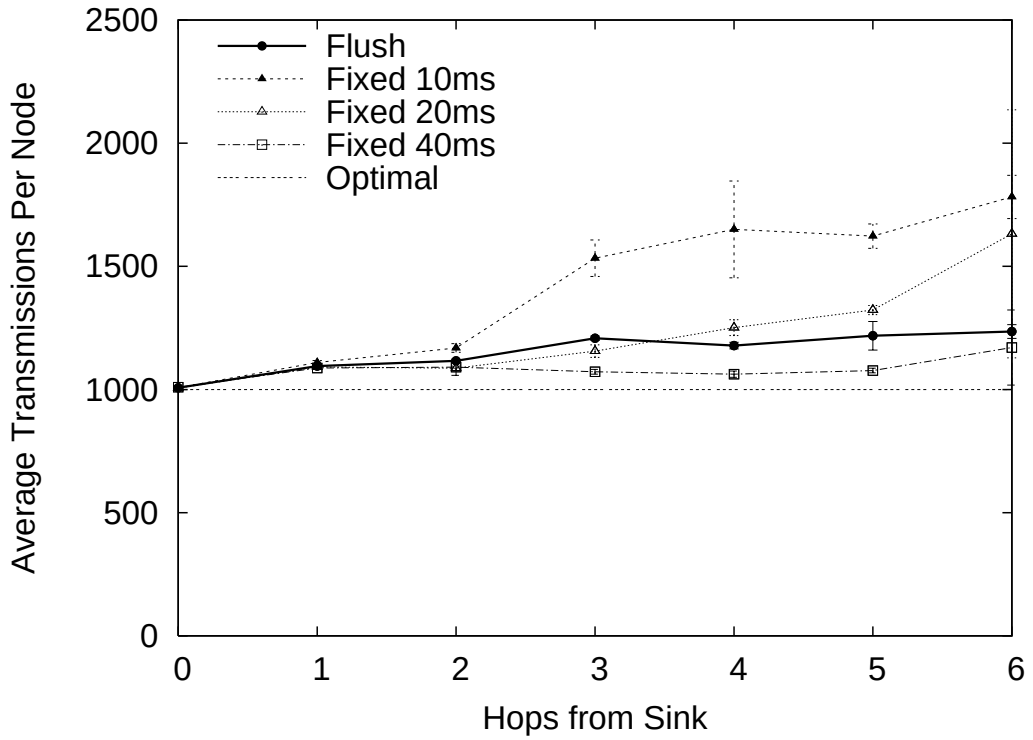

Figure 6.14. Average number of transmissions per node for sending an object of 1000 packets. The optimal algorithm assumes no retransmissions. Losses at Fixed 40ms is only due to losses not interference. Flush closely tracks the efficiency of this case.

retransmissions of the "Fixed 10ms" and "Fixed 20ms" curves include both the link-level retransmissions and end-to-end retransmissions for packet losses due to queue overflows at intermediate nodes.

6.8.2 Performance Breakdown

We now explore why Flush performs well compared with fixed rates and how Flush performs at a high level. Sending a packet during the acknowledgment phase is more expensive than sending a packet during the transfer phase both in terms of the number of packets needed and the total transfer time required. In response to a single NACK, only a few packets can be sent, but this requires one more packet transfer and one additional RTT delay. Therefore, an important design goal is to maximize the

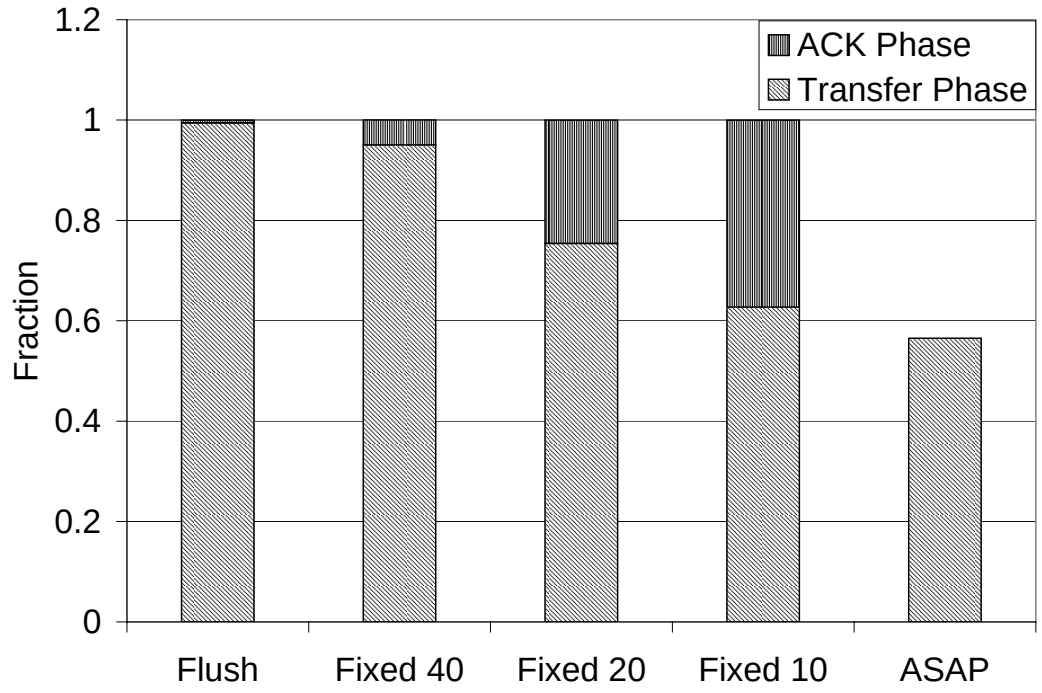

Figure 6.15. The fraction of data transferred from the 6th hop during the transfer phase and the acknowledgment phase. Greedy best-effort routing (ASAP) exhibits a loss rate of 43.5%. A higher than sustainable rate leads to a high loss rate.

data received during the transfer phase. The higher the goodput during the transfer phase, the greater the effective bandwidth becomes.

Figure 6.15 shows the fraction of packets received during the transfer phase from a node that is 6 hops away from the basestation. The data set is the same one as that in Subsection 6.8.1. Flush collects 99.5% of packets during the transfer phase. In contrast, the "Fixed 10ms" rate flow collects only 62.7% during the transfer phase. The spacing of packets at a 10ms interval is so small that it suffers severe loss from intra-path interference. This high loss rate explains why "Fixed 10ms" in Figure 6.14 sends a lot more packets, losing efficiency. "ASAP" is a greedy best-effort transfer that uses the underlying routing layer, MintRoute [86]. Packets are sent as quickly as possible, in quick succession. "ASAP" exhibits an even higher loss rate of 43.5%.

Figure 6.16 shows a breakdown of how time is spent. As we argued previously, the "Acknowledgment Phase" is expensive. In a case of the "Fixed 10ms" rate, a

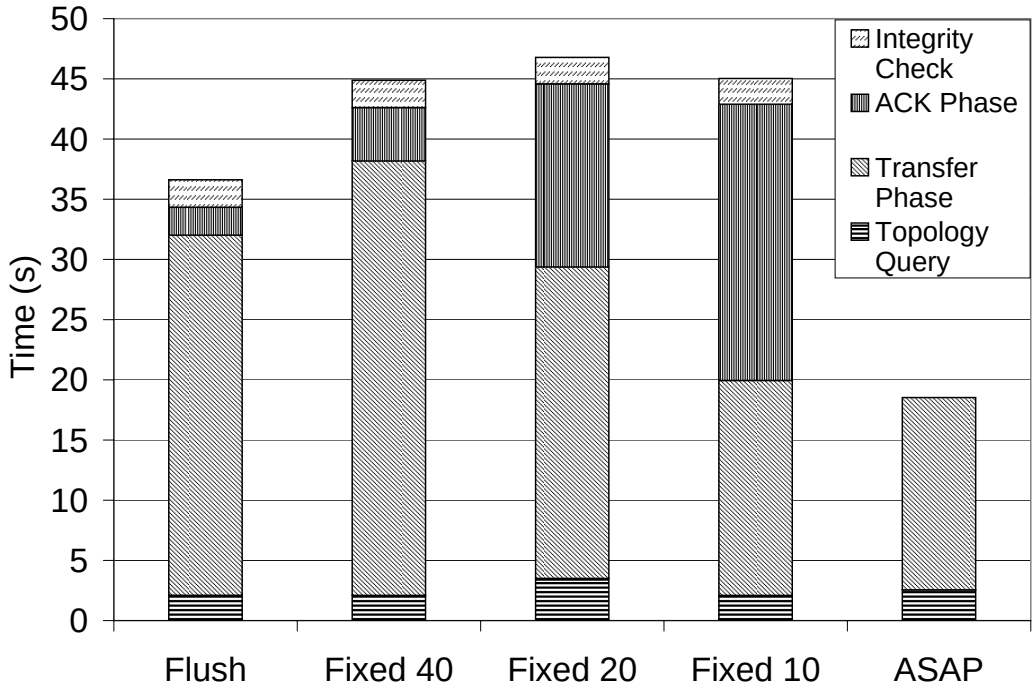

Figure 6.16. The fraction of time spent in different stages. A retransmission during the acknowledgment phase is expensive, and leads to poor throughput. Greedy best-effort routing (ASAP) does not have the acknowledgment phase nor the integrity check phase.

modest fraction of packets are transferred during the transfer phase, and more time is spent in the Acknowledgment phase. While both "Fixed 10ms" and "ASAP" spend less time than Flush in the transfer phase, they also deliver less data than Flush in these phases. To quantitatively compare the amount of time and data involved at the transfer phase, we compute an effective goodput. This indicates the effectiveness of the Flush rate control algorithm.

Figure 6.17 shows fraction of data transferred during the transfer phase and the acknowledgment phase in Flush. As hop count from the sink increases, more data is transferred in the acknowledgment phase, except where the source is 5 hops away. Except this case, most of the data is transferred in the transfer phase, implying Flush does not lose many packets. Figure 6.18 shows the fraction of time spent in different stages of Flush. As hop count from the sink increases, more time is taken overall, however time spent in the topology query phase and the integrity check phase remains similar. When the source is closer to the sink, time spent in the transfer phase is shorter, and relative overhead of time spent in other phases gets larger.

Figure 6.19 shows an effective goodput of bandwidth during the transfer phase. These bandwidth measurements include an adjustment factor for payload size. To see how Flush's rate control algorithm compares with fixed rate schemes, we present effective goodput in terms of packet throughput in Figure 6.20. We see that starting from the 2nd hop, and continuing for all greater hop counts, Flush provides a similar goodput to the best case among fixed rates and the greedy best-effort. For a basestation (0th hop) and the 1st hop, "Fixed 10ms" and "ASAP" provide a higher goodput. However, they suffer higher loss rates and pay a high price during the acknowledgment phase. Overall, Flush provides competitive goodput during the transfer phase.

In summary, Flush provides comparable effective goodput during the transfer phase as Figure 6.20 shows but with very low loss rates as Figure 6.15 shows. Flush

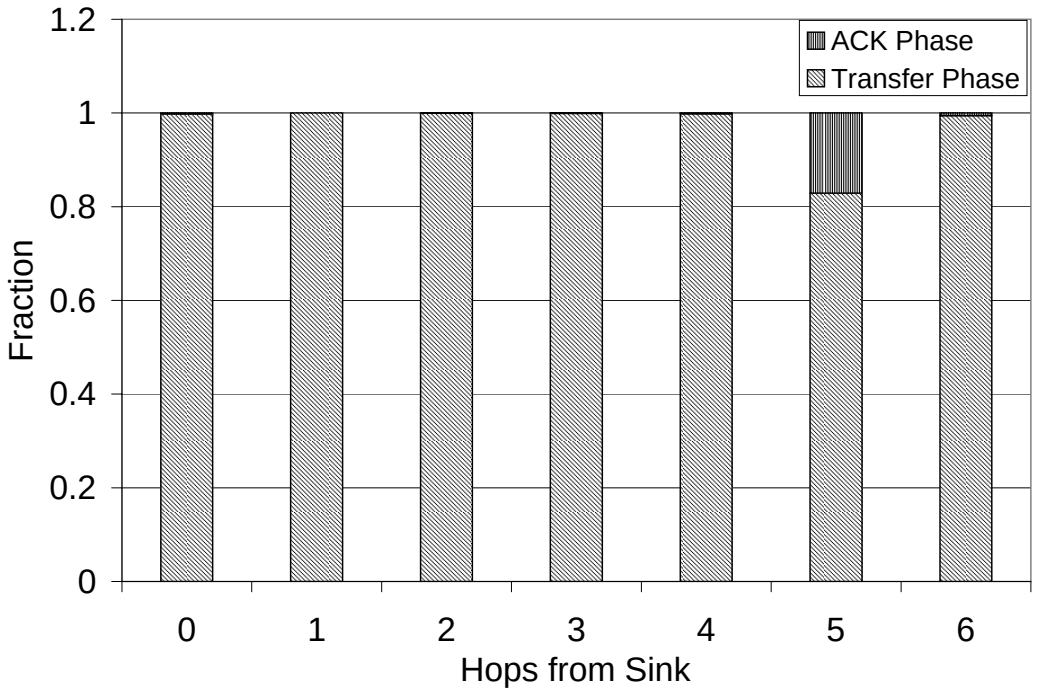

Figure 6.17. The fraction of data transferred during the transfer phase and the acknowledgment phase in Flush. In many cases, most of the data is transferred during the transfer phase.

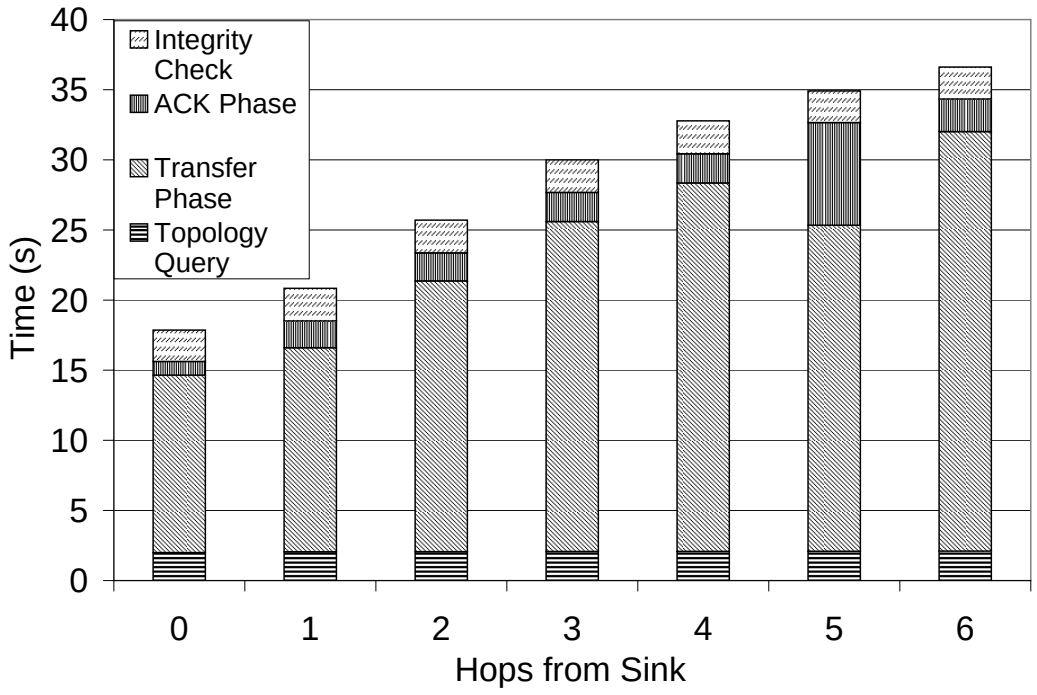

Figure 6.18. The fraction of time spent in different stages in Flush. When the source is close to the sink, time spent in the transfer phase is short, and the relative overhead of the time spent in other phases is large.

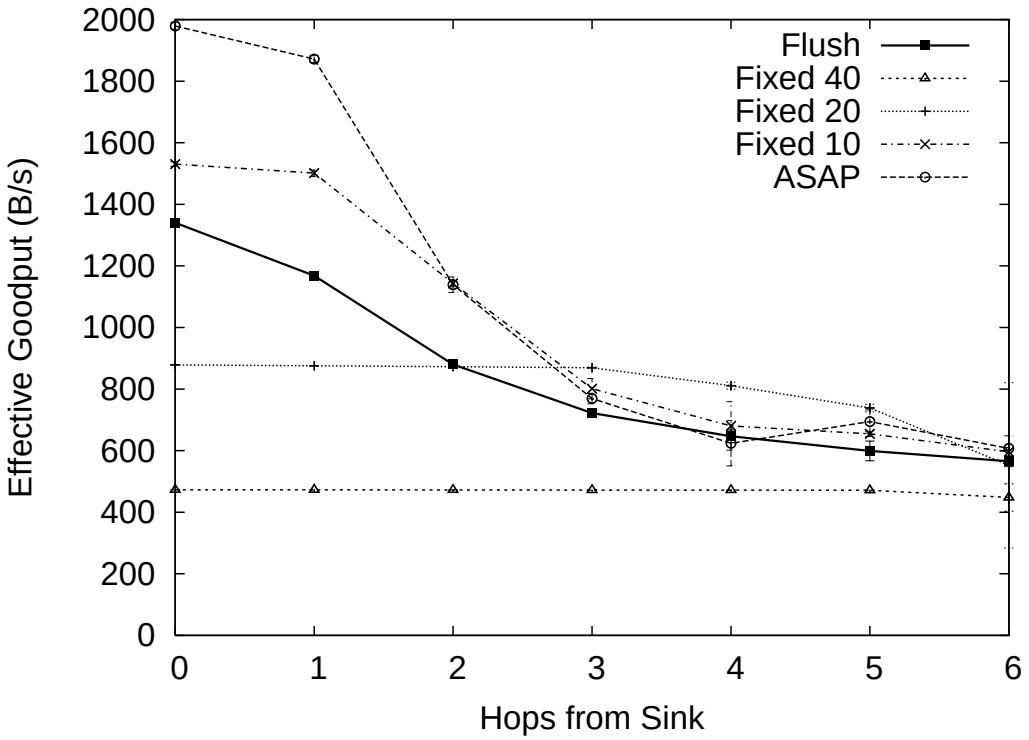

Figure 6.19. Effective goodput during the transfer phase. Effective goodput is computed as the number of unique packets received over the duration of the transfer phase.

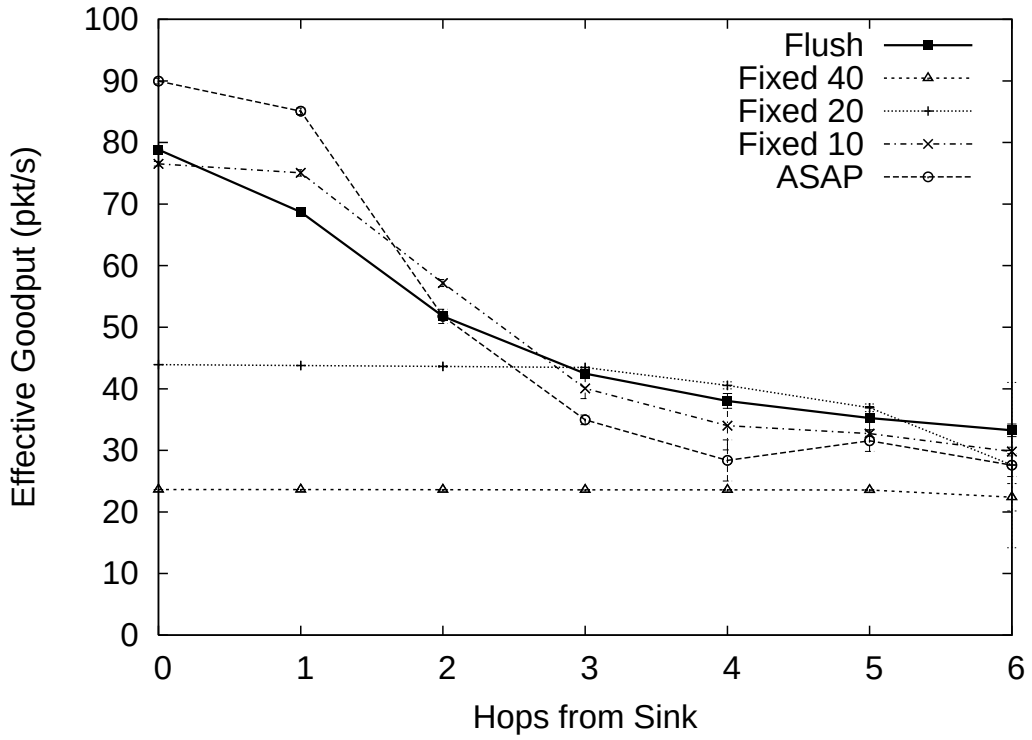

Figure 6.20. Effective goodput during the transfer phase. Effective goodput is computed as the number of unique packets received over the duration of the transfer phase. Flush provides comparable goodput as a lower loss rate which reduces the time spent in the expensive acknowledgment phase, which increases the effective bandwidth.

also spends much less time in the expensive acknowledgment phase as Figure 6.16 shows. This combination makes Flush's overall transfer time relatively short, and explains Flush's overall good bandwidth delivery.

6.8.3 A More Detailed Look

We now take a more detailed look at Flush's operation. In the following two subsections, Flush's rate control header fields account for 6 bytes (δ, f, and D each require 2 bytes). leaving 14 bytes for the payload. At this initial stage, 2 bytes are used just in case a value exceeds 255. We discovered, however, that the δ, f, and D values never exceeded 255, so these fields were reduced to a single byte each in other bandwidth and scalability experiments.

"Flush-e2e" is a variation of Flush which only *limits* the rate at the *source*, even though the intermediary nodes still estimate the delays and propagate them as described in Subsection 6.5.3. Using the detailed logs collected for a sample transfer of 900 packets (12600 bytes) over a 7-hop path, we are able to look at the real sending rate at each node, as well as the instantaneous queue length at each node as each packet is transmitted. These can be done in a time-correlated manner using globally synchronized time stamps. These traces are similar to TCP traces.

Figure 6.21 shows the sending rate of one node over a particular interval, where the rates are averaged over the last k packets received. We set k to 16, which is the maximum queue length. Other nodes had very similar curves. We compare Flush and Flush-e2e with the best performing fixed-rate sending interval at this path length, 30ms. Sending at this interval did not congest the network. As expected, under stable network conditions, the fixed-rate algorithm maintains a stable rate. Although Flush and Flush-e2e showed very similar high-level performance in terms of throughput and

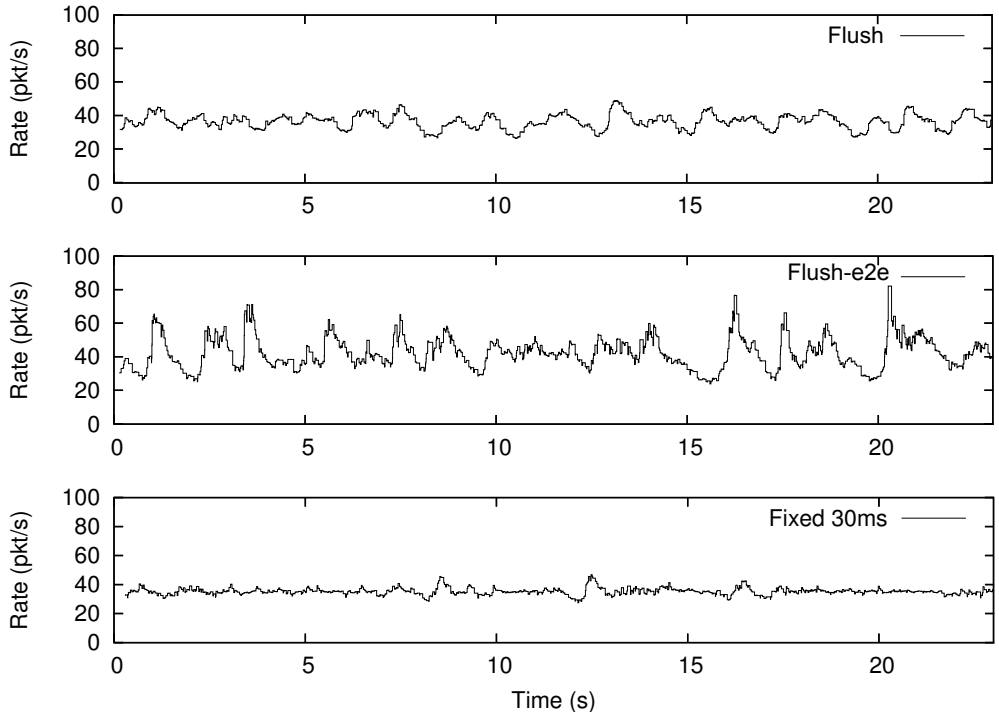

Figure 6.21. Packet rate over time for a source node, which is 7 hops away from the base station. Packet rate averaged over 16 values, which is the max size of the queue. Flush approximates the best fixed rate with the least variance.

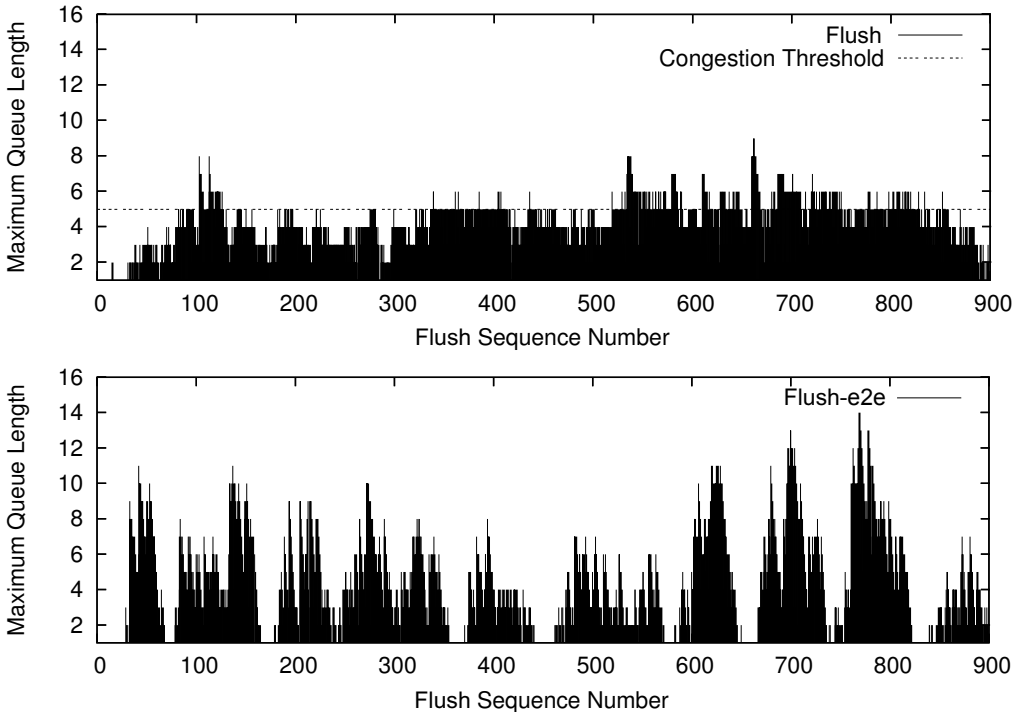

Figure 6.22. Maximum queue occupancy across all nodes for each packet. Flush exhibits more stable queue occupancies than Flush-e2e. Fluctuating queue occupancy together with a change in an environment can lead to a queue overflow and packet loss.

bandwidth, we see here that the Flush is much more stable, although not to the same extent as the fixed interval transfer.

Another benefit of the in-network rate limiting, as opposed to source-only limiting, can be seen in Figure 6.22. This plot shows the maximum queue occupancy for all nodes in the path, versus the packet sequence number. Note that we use sequence number here instead of time because two of the nodes were not properly time-synchronized due to errors in the timesync protocol. The results are very similar, though the rates do not vary much. The queue length in Flush is always close to 5, which is the congestion threshold we set for increasing the advertised delay (c.f. Subsection 6.5.3 and Subsection 6.6.4). Our simple scheme of advertising our delay as doubled when the queue is above the threshold seems to work well in practice. It

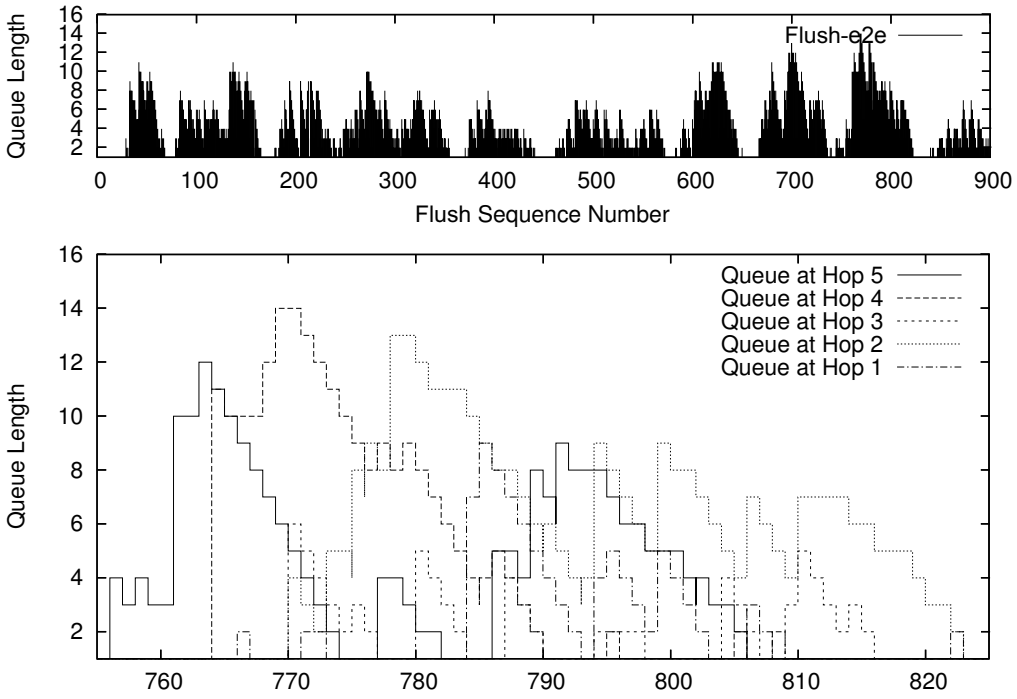

Figure 6.23. Detailed view of instantaneous queue length for Flush-e2e in Figure 6.22. Queue fluctuations ripple through nodes along a flow.

is actually good to have some packets in the queue, because it allows the node to quickly increase its rate if there is a sudden increase in available bandwidth.

In contrast, Flush-e2e produces highly variable queue lengths. The peak occupancy of Flush-e2e is already 14. A little more fluctuation or packets from other services like a routing layer can easily overflow a queue leading to packet loss. The lack of rate limiting at intermediary nodes induces a cascading effect in queue lengths, as shown in Figure 6.23. The top graph is the same graph in Figure 6.22 (bottom). The bottom graph provides a closer look at the queue lengths for 5 out of the 7 nodes in the transfer during a small subset of the entire period. The queue is drained as fast as possible when bandwidth increases, thus increasing the queue length at the next hop. This fast draining of queues also explains the less stable rate shown in Figure 6.21.

6.8.4 Adapting to Network Changes

We also conduct experiments to assess how well Flush adapts to changing network conditions. Our first experiment consists of introducing artificial losses for a link in the middle of a 6-hop path in the testbed for a limited period of time. We did this by programmatically having the link layer drop each packet sent with a 50% probability. This effectively doubled the expected number of transmissions along the link, and thus the delay.

Figure 6.24 provides the instantaneous sending rate over the link with the forced losses for Flush, Flush-e2e, and Fixed 30ms. Again, 30ms was the best fixed rate for this path before the link quality change was initiated. In the test, the link between two nodes, 3 and 2 hops from the sink, respectively, has its quality halved between the 7 and 17 second marks, relative to the start of the experiment. We see that the static algorithm rate becomes unstable during this period; due to the required retransmissions, the link can no longer sustain the fixed rate. Flush adapts gracefully to the change: the sending rate decreases smoothly. The variability remains constant during the entire experiment. Flush-e2e is not very stable when we introduce the extra losses, and is also less stable after the link quality is restored.

Figure 6.25 compares the queue lengths for the same experiment for all three algorithms, and the reasons for the rate instability becomes apparent, especially for the fixed rate case. The queue at the lossy node becomes full as its effective rate increases, and is rapidly drained once the link quality is reestablished.

The last experiment looks at the effect of a route change *during a transfer* on the performance of Flush. We started a transfer over a 5 hop path, and approximately 21 seconds into the experiment forced the node 4 hops from the sink to switch its next hop. Consequently, the entire subpath from the node to the sink changed (from 1a, 2a, 3a to 1b, 2b, 3b). Note that this scenario does not simulate node failure,

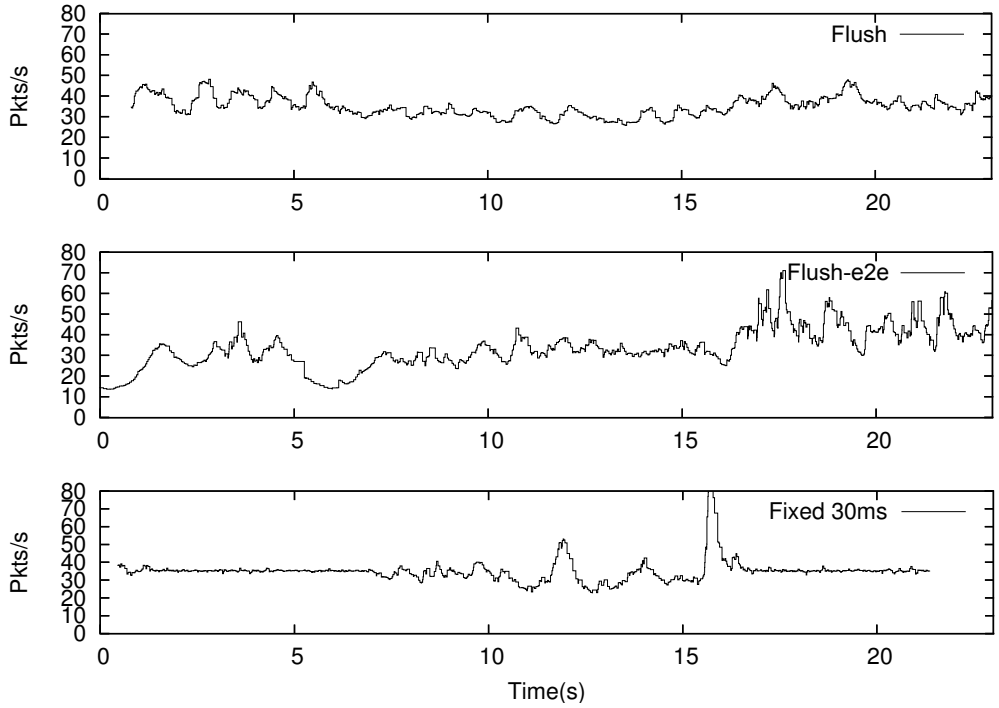

Figure 6.24. Sending rates at the lossy node for the forced loss experiment. Packets were dropped with 50% probability between 7 and 17 seconds. Both Flush and Flush-e2e adapt while the fixed rate overflows its queue.

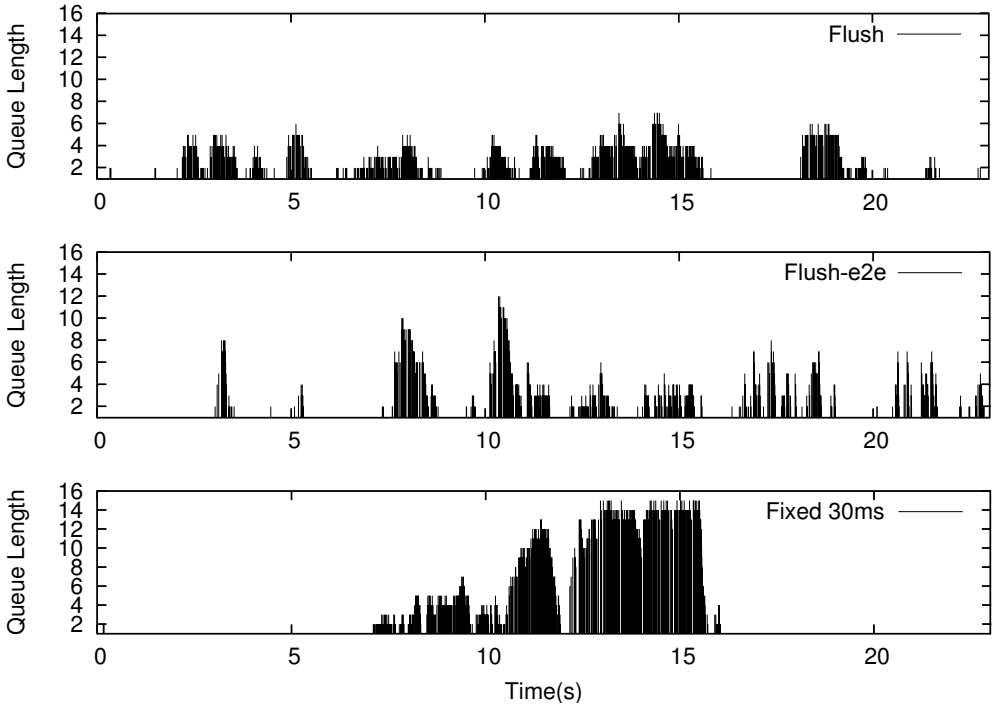

Figure 6.25. Queue length at the lossy node for the forced loss experiment. Packets were dropped with 50% probability between 7 and 17 seconds. Flush and Flush-e2e adapt while the fixed rate overflows its queue.

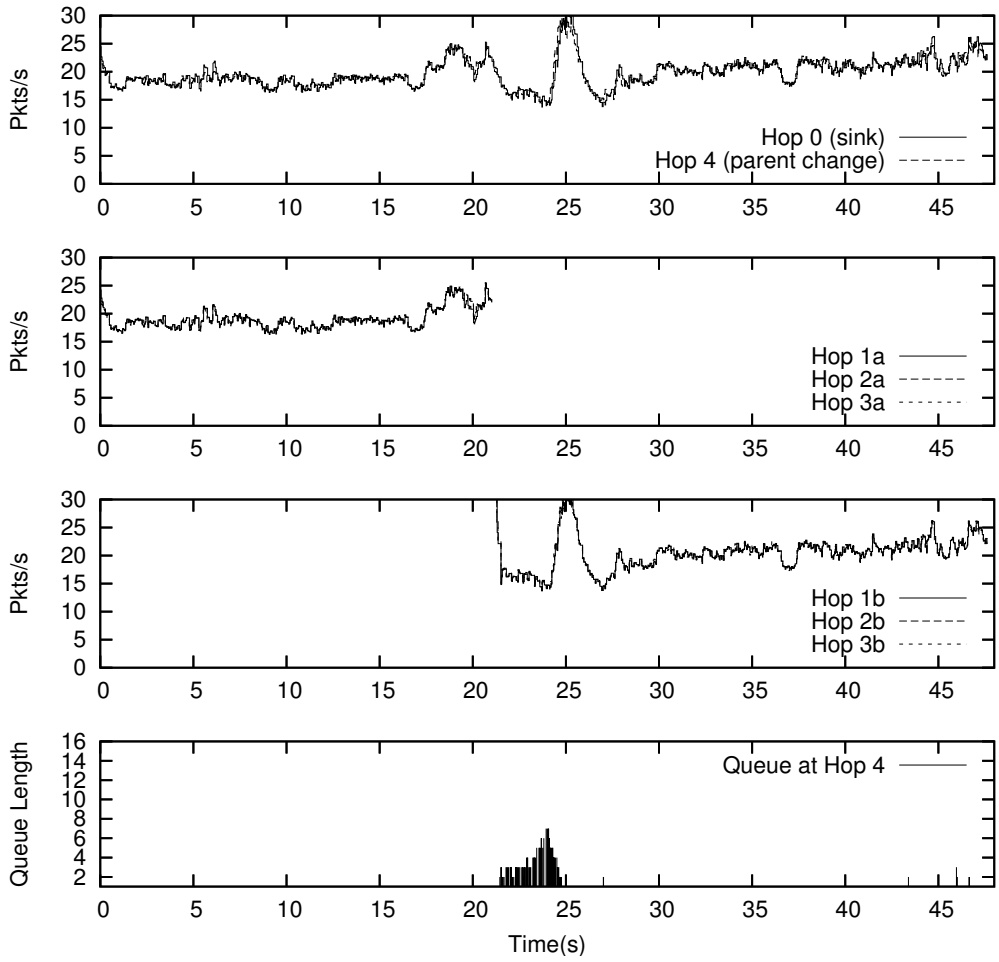

Figure 6.26. Detailed look at the route change experiment. Node 4's next hop is changed, changing all nodes in the subpath to the root (from 1a, 2a, 3a to 1b, 2b, 3b). Top 3 graphs show rates at each node. The bottom graph shows a queue length at hop 4. No packets were lost, and Flush adapted quickly to the change. The only noticeable queue increase was at node 4. This figure shows Flush adapts when the next hop changes suddenly.

but rather a change of the next hop in a routing topology, so packets should not be lost. The high level result is that the change had a negligible effect on performance. Figure 6.26 presents a detailed look at the rates for all nodes (top 3 graphs), and the queue length at the node that had its next hop changed (bottom graph). There was no packet loss, and the rate control algorithm was able to quickly reestablish a stable rate. Right after the change there was a small increase in the affected node's queue, but that was rapidly drained once the delays were adjusted.

While we do not show any results for node failure, we expect the algorithm will considerably slow down the source rate, because the node before the failure will have to perform a large number of retransmissions. If the routing layer selects a new route in time, the results we have lead us to believe Flush would quickly readjust itself to use the full bandwidth of the new path.

6.8.5 Scalability

Finally, to evaluate the scalability of Flush, we deployed an outdoor network consisting of 79 MicaZ nodes in an outdoor setting at the Richmond Field Station (RFS) [12]. For the following experiments, we increased the data payload size to 38 bytes (from 20 bytes used previously) for the fixed rate and 35 bytes (from 17 bytes used previously) for Flush. The size of the Flush rate control header was 3 bytes, leaving us with a protocol overhead of about 8%. We transfer a 26,600 byte data object from the node with a depth of 48 (node 79), and then perform similar transfers from nodes at depths 40, 30, 20, 15, 10, 7, 5, 4, 3, 2, and 1. The experiment is repeated for Flush, and fixed rates of 20ms, 40ms, and 60ms. Each experiment is performed twice and the results are averaged. We omit error bars for clarity. Figure 6.27 shows the results of this experiment. The results indicate that Flush efficiently transfers data over very long networks – 48 hops in this case.

6.9 Discussion

In this section, we discuss two important issues that we have largely ignored until now. The first deals with whether a multihop collection protocol that scales to tens of hops is needed and the second deals with the interactions between Flush and routing that led us to freeze the collection tree over the duration of a transfer.

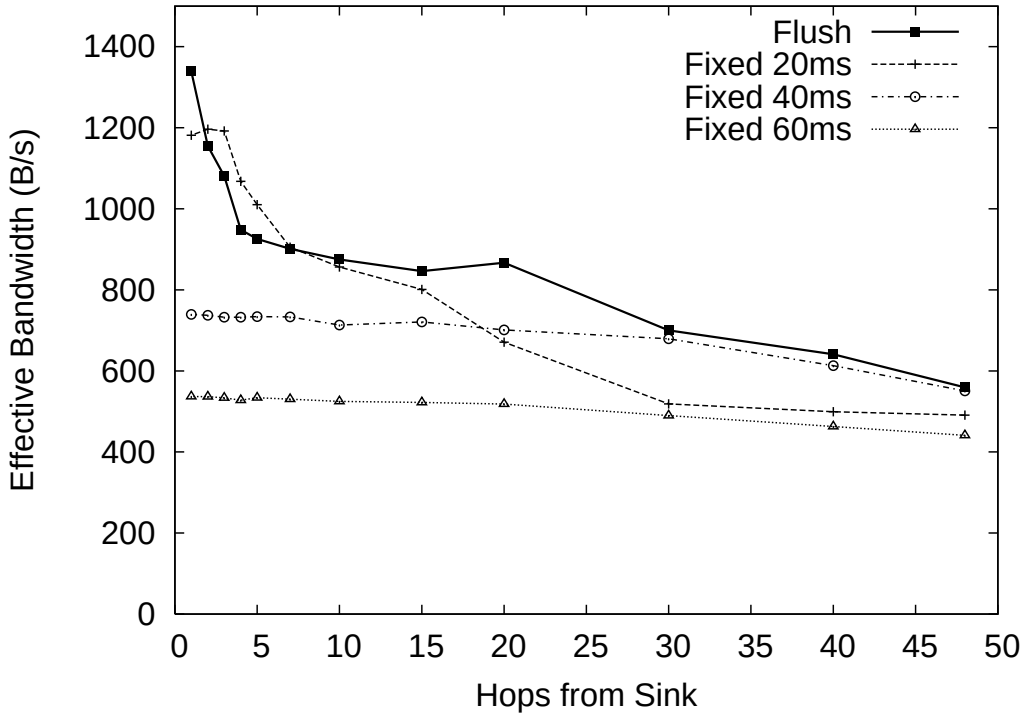

Figure 6.27. Effective bandwidth from the real-world scalability test where 79 nodes formed a 48-hop network at RFS. The Flush header is 3 bytes and the Flush payload is 35 bytes (versus a 38-byte payload for the fixed rates). Flush closely tracks or exceeds the best possible fixed rate across all hop distances that we tested.

6.9.1 Density of Nodes

We claimed that delivery of bulk data to the network edge is complicated by the vagaries of wireless communication: efficiency and reliability are hampered by lossy links, intra-path interference is hard to avoid, inter-path interference is hard to cope with, and transient rate mismatches can overflow queues. One compelling solution to these challenges might be to simply sidestep them. By judiciously placing high-power radios within a low-power sensornet and increasing the density of nodes, a network administrator might be able to reduce it to a single-hop problem, remove the complexities that multihop introduces, and allow simple, robust solutions.

Unfortunately, it is not always possible to convert a deployment of low-power, short-range nodes into an equivalent network of high-power, long-range nodes. In other words, it is not possible to arbitrarily increase the density of nodes to a degree where all nodes are within a reach of all others. Some applications are deployed in challenging radio environments that do not provide a clear line-of-sight over multiple hops. For example, signals do not propagate well in steel-framed buildings, steel truss bridges, or dense foliage. When physically large networks are deployed in such environments, multihop delivery is often the enabler, so eliminating it eliminates the application as well.

6.9.2 Interactions with Routing

We freeze the MintRoute collection tree immediately prior to a Flush transfer and then let the tree thaw after the transfer. This freeze-thaw cycle prevents collisions between routing beacons and Flush traffic. MintRoute [86] generates periodic routing beacons but these beacons use a separate, unregulated data path to the radio. With no rate control at the link layer, the beacons are transmitted at inopportune times,

collide with Flush traffic, and are lost. Since MintRoute depends on these beacons for route updates and it evicts stale routes aggressively, Flush freezes the MintRoute state during a transfer to avoid stale route evictions. A more general statement of a problem is how to handle an external contention, and this problem is also observed in the deployment at the Golden Gate Bridge, see Section 5.1.

Our freeze-thaw approach sidesteps the issue and works well in practice. Over small time scales on the order of a Flush session, routing paths are generally stable, even if instantaneous link qualities vary somewhat. Our results show that Flush can easily adapt to these changes. In Figure 6.26, we also showed that Flush adapts robustly to a sudden change in the next hop. If the underlying routing protocol can find an alternate route, then Flush will adapt to it. But if the physical topology changes, and routing cannot adapt, then new routes will need to be rebuilt and the Flush session will have to be restarted.

It may seem that forcing all traffic to pass through a single (rate-limited) queue would address the issue, but it does not. Nodes located on the flow path *would* be able to fairly queue routing and Flush traffic. However, nodes located off the flow path but within interference range of the flow would not be able to contend for bandwidth and successfully transmit beacons. Hence, if the physical topology changes along the flow path *during* a transfer, the nodes along the path may not be able to find an alternate route since beacons from these alternate routes may have been lost. A solution may be an interference-aware fair MAC. Many pieces are already in place [30, 44, 73] but the complete solution would require rate-controlling all protocols at the MAC layer across all nodes within interference range of the path.

6.10 Summary

In this chapter, we present *Flush*, a reliable, single-flow transport protocol for transferring bulk data from a source to a sink over a multihop wireless sensor network. Flush achieves end-to-end reliability through selective negative acknowledgments and with its adaptive rate control algorithm, Flush can automatically match or exceed the best fixed rate for a multihop flow. We show that Flush is scalable; it provides an effective bandwidth of approximately 550 bytes/second over a 48-hop wireless network, approximately one-third of the rate achievable over one hop. Flush achieves these results by allowing just one flow at a time (a reasonable restriction for many sensornet applications) and by following two simple rules. First, a node should only transmit when its successor is free from interference. At each node, Flush attempts to send as fast as possible without causing interference at the next hop along the flow. Second, a node's sending rate cannot exceed the sending rate of its successor. Again, Flush attempts to send as fast as possible without increasing the average queue occupancy at the next hop along the flow. These two rules, applied recursively along the path, explain Flush's performance.

Chapter 7

Data Analysis of the Golden Gate Bridge

This chapter analyzes data from the deployment at the Golden Gate Bridge. Before a detailed analysis is presented, let us first take a look at a high-level deployment history and characteristics.

The operation of a system can be divided mainly into two phases. Each phase uses one battery set. Each battery set can last about 5 weeks. There was a non-operational period between the two phases, which was about one month. There also was an addition of nodes before the second phase. In the first phase, 8 nodes were deployed on the south tower and 51 nodes were deployed on the west side of the main span only. In total, 59 nodes were deployed. In the second phase, 2 nodes were added to the west side of the main span to provide better connectivity. 3 nodes were deployed on the east side of the main span to distinguish the torsional mode of a vibration from the vertical mode of a vibration. The total count of nodes increased to 64 nodes.

The topology that was formed at the bridge was close to a linear topology, there

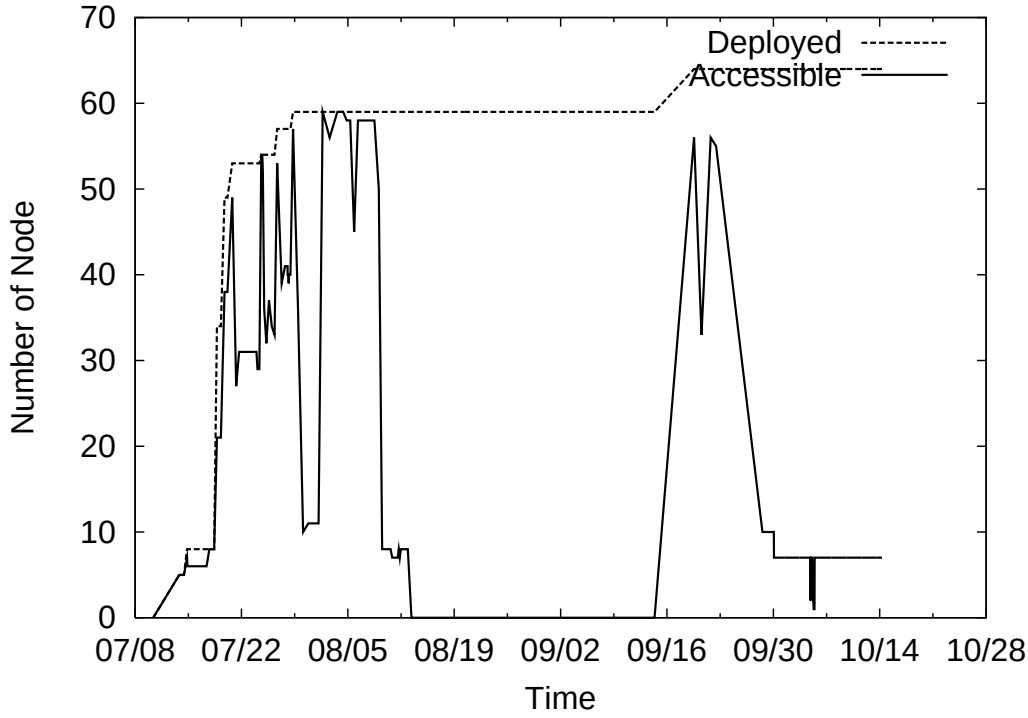

Figure 7.1. Number of nodes deployed and number of nodes accessible. About one month after the first set of batteries depleted, the second set of batteries was deployed with a few more nodes.

were only a couple of candidates for the next hop. The failure of one node sometimes led to a failure of all the nodes beyond that point. The topology also showed an enormous depth. Large initial RTT and timeout values slowed down the reaction to a loss.

7.1 Lifetime Analysis

Figure 7.1 shows how many nodes were deployed and how many of them were actually accessible over time.

'*Deployed*' means the number of nodes deployed over time. At the beginning only a few nodes were deployed, and their operation was observed. They worked fine, so

we cranked up the number of nodes from July 18th. We can see the beginning of the second phase, when there is a sudden increase on September 15th.

'*Accessible*' indicates how many nodes were replying to a ping command. From an initial small-scale deployment, the size of the network ramped up quickly. Then there was a sudden drop on July 21st, and the drop was sustained for a few days. It took a few days to find a problem, but finally it was figured out on July 24th. This drop was due to a loose connection between the accelerometer board and the MicaZ mote. All enclosures were opened. Screws were added to secure the connection between the board and the mote. The number of accessible nodes increased unstably, but dropped again on July 29th. The weather became very foggy and windy. It also took a few days to figure out the problem. The cause was a combination of strong wind and bi-directional patch antenna. Under heavy wind, the patch antenna flapped like a fish tail. Zip ties were applied to every antenna. The number of accessible nodes increased again on August 1st. The battery depleted on August 13th. The operation of the second phase starts with a replacement of batteries on September 19th. However, due to rusted battery connectors, nodes did not operate well from September 23rd. On October 14th, there was a shortage of power in a control room in the south tower where the base station server was located. The base station server also suffered power failure, and it was decided to finish the second phase.

7.2 Failure Analysis

Each packet has a 16-bit cyclic redundancy check (CRC). An entire object, which is the entire data set from one node, has a 16-bit checksum. A few data objects have a checksum mismatch. It implies that there are packets which have multiple data corruptions, yet still pass a CRC check at the packet level. It is calculated how many packets belong to this situation out of how many one-hop packet transfers.

There were 2630 Straw transfers. Out of 389, 472, 400 one-hop packet transfers, 17 or more packets were corrupted (when an object is corrupted, at least one packet in the object has multiple corruptions which CRC cannot detect). This ratio is equal to approximately 1 out of 23 million packets.

To analyze factors contributing to a failure of an operation, a subset of data is taken from August 1st to August 8th where hardware malfunctions are minimal. The data can be divided into two periods: before and after software upgrade. The software is upgraded on August 3rd reflecting lessons learned. The upgrade is only applied to software running on the base station PC and had the following improvements: when a routing tree becomes stale, the base station PC instructs the routing layer to rebuild the tree. When a received object has an incorrect checksum, the PC abandons the corrupted data and asks for a retransmission of that object. Table 7.1 and 7.2 show the causes and fractions of failures before and after the software upgrade respectively. '*Deployed, Not Accessible*' is a case where a node does not respond to a ping command. It is unknown whether the cause of a failure is a hardware failure or a transient disconnection. '*Accessible, Sentri Disconnected*' is when a node is connected to a network, however the application fails to communicate with that node. In '*Sentri Connected, Timesync Failed*', a time synchronization component (FTSP [61]) fails to synchronize, so sampling cannot be triggered. In both Table 7.1 and 7.2, this is the most common cause of a failure. '*Timesync Successful, Straw Disconnected*' indicates Straw failed in the middle of a communication due to a disconnection at a routing layer. After the software upgrade, when Straw suffers a disconnection in the middle of a transfer, the routing layer is instructed to rebuild the tree. Then, Straw starts the transfer again. Sometimes, a rebuilt tree can still suffer disconnections repeatedly. After the software upgrade, this type of failure decreased by a factor of 4. '*Straw Connected, Data Corrupted*' is when two checksums of an object at the source and at the sink do not match, indicating that the object is corrupted. After the software

Table 7.1. Causes and fractions of failures before software upgrade. A time synchronization failure is the most common cause of a failure.

Category	Number of Cases	Fraction
Deployed, Not Accessible	3	2.54%
Accessible, Sentri Disconnected	0	0.00%
Sentri Connected, Timesync Failed	13	11.02%
Timesync Successful, Straw Disconnected	2	1.69%
Straw Connected, Data Corrupted	1	0.85%
Successful	99	83.90%
Total	118	100.00%

Table 7.2. Causes and fractions of failures after software upgrade. Again a time synchronization failure is the most common cause of a failure. There is no case of data corruption because upgraded software retransmits a corrupted object.

Category	Number of Cases	Fraction
Deployed, Not Accessible	8	1.23%
Accessible, Sentri Disconnected	8	1.23%
Sentri Connected, Timesync Failed	61	9.40%
Timesync Successful, Straw Disconnected	3	0.46%
Straw Connected, Data Corrupted	0	0.00%
Successful	569	87.68%
Total	649	100.00%

upgrade, a corrupted object is retransmitted, so this failure disappeared thereafter. [29] compares gain and power consumption of error-correcting codes for four different wireless scenarios – very low power, low-power, average-power, and high-power. It will be an interesting future work to apply their findings to our data. '*Successful*' is when everything works correctly and data is sampled and collected successfully. After the software upgrade, 87.68% of cases showed a successful operation.

7.3 Vibration Data

A sample of vibration data collected on the bridge at 6 PM on the 21st of September, 2006, is presented in Figures 7.2 and 7.3. These data show acceleration time histories and frequency domain plots of the accelerations in the transverse direction at

two nodes, one located near the south quarter span of the bridge (about 365m north of the south tower) and one at the mid-span of the bridge. The accelerations were sampled at 1kHz, with every twenty samples averaged and logged to flash. Each figure includes a zoom to a 20s interval of the acceleration time history. The Power Spectral Density (PSD) of the signals were computed using the Welch method [56]. The time histories show that the signal in both orientations have an average amplitude of about 5mG with peaks of approximately 10mG, which most likely corresponds to the passing of large cars or trucks. The frequency analysis shows clearly defined peaks in the low frequencies, where the natural modes of vibrations of the bridge are expected to reside. For the vertical orientation, a peak at 0.11Hz matches the fundamental frequency of the bridge found by past studies [22]. Modal properties also match with the simulation model and the previous study; see Figures 7.4 and 7.5. Other resonant peaks of 0.17Hz, 0.22Hz, and 0.27Hz are consistently repeated in all the signals from the vertically oriented sensors, and are likely to be other fundamental modes of the bridge structure. More extensive analysis of this data will be presented in future publications.

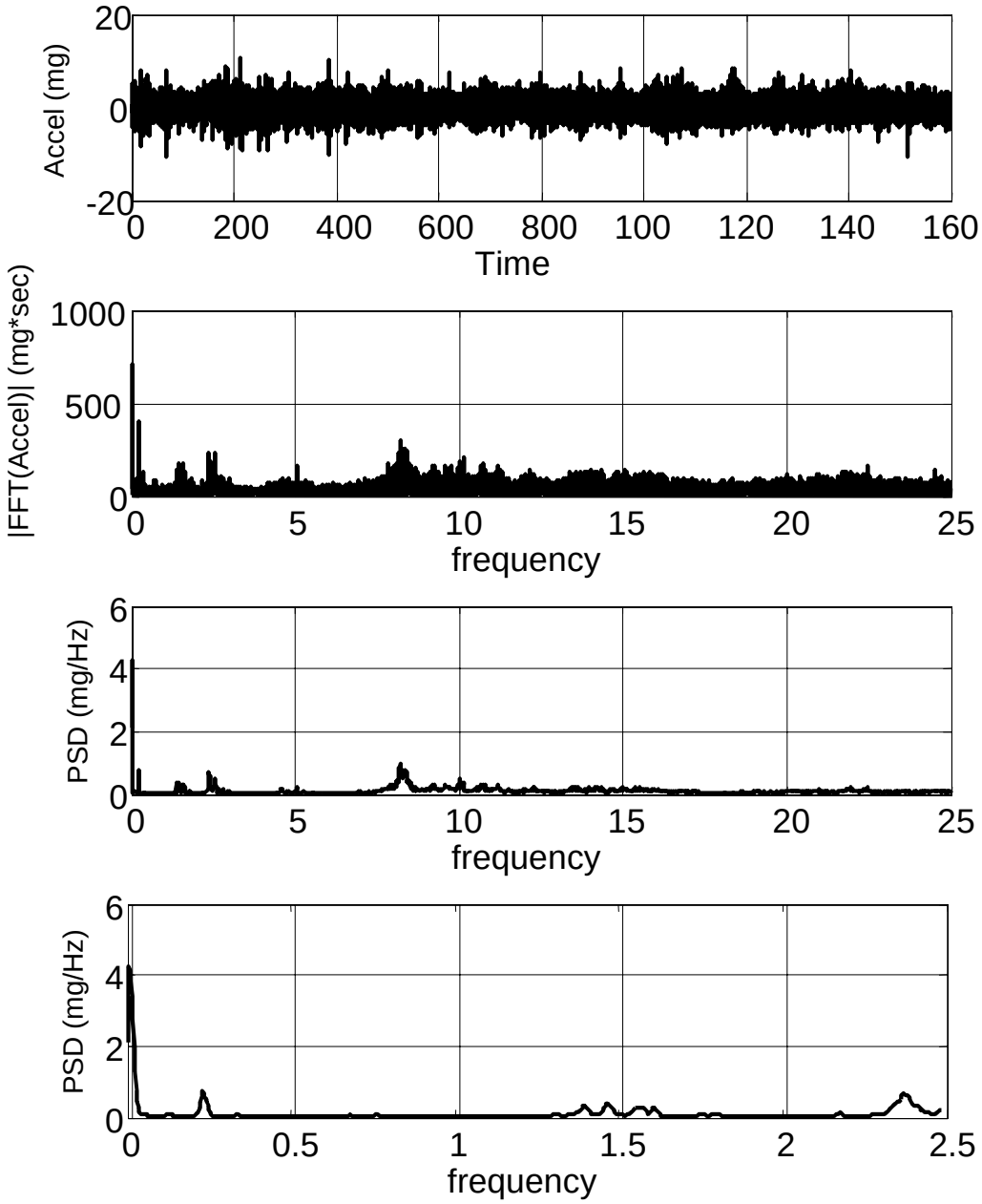

Figure 7.2. Time and Frequency Plots of Transverse (Horizontal) Sensor Located at Quarter span, 365m North of the South Tower. The data matches the fundamental frequency of the bridge in past studies [22].

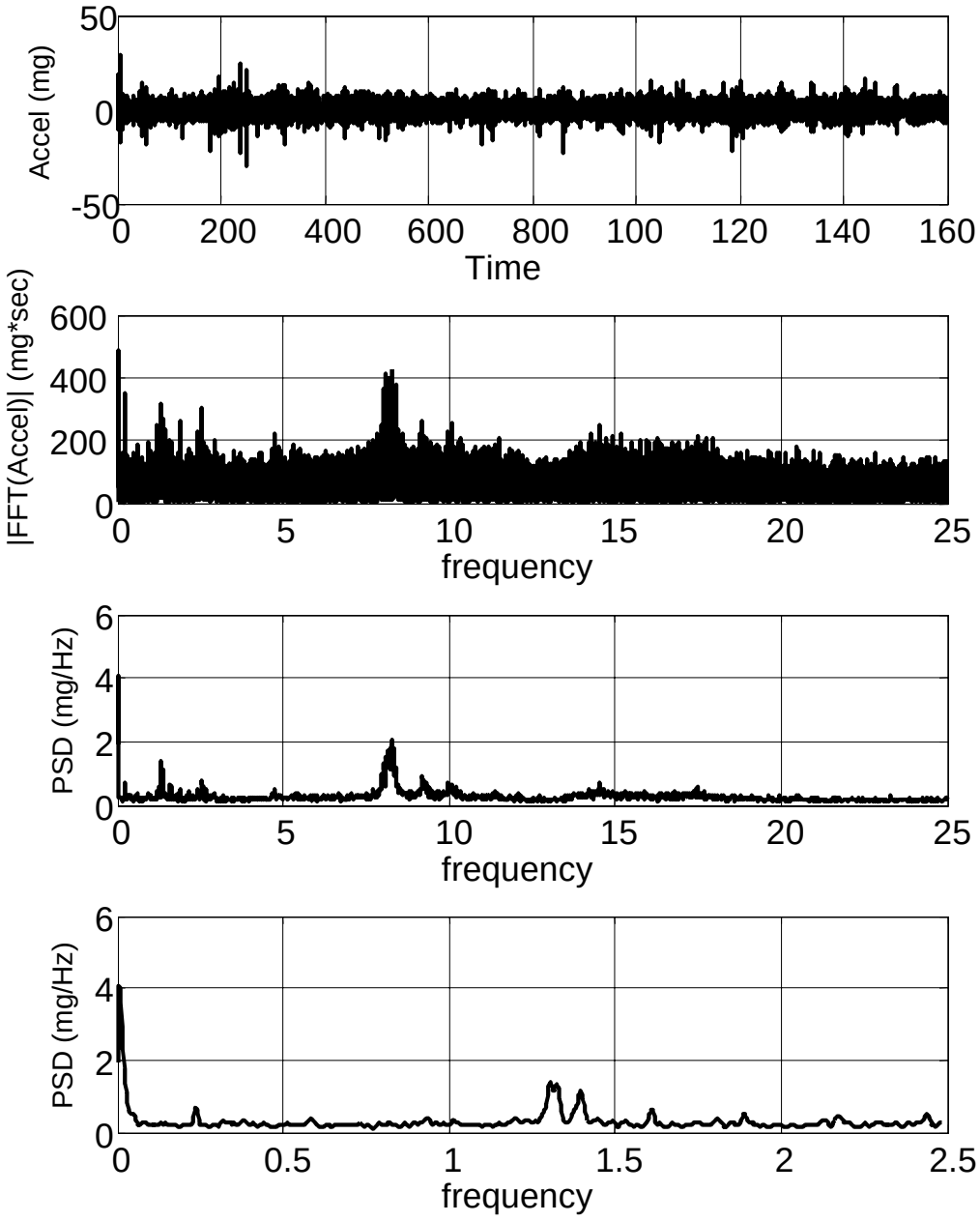

Figure 7.3. Transverse (Horizontal) Sensor, Mid-Span

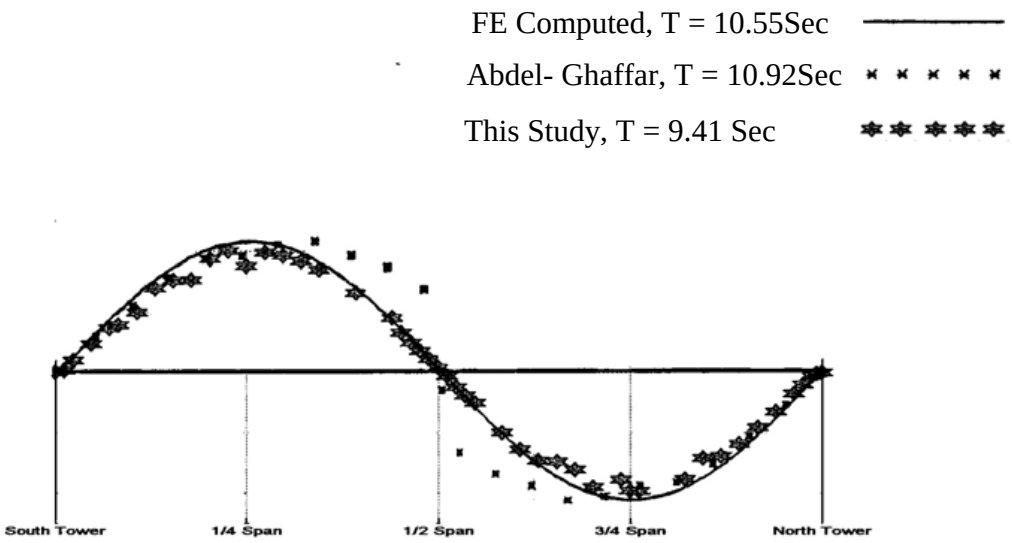

Figure 7.4. The vertical modal properties match among simulation model, previous study, and this study [22].

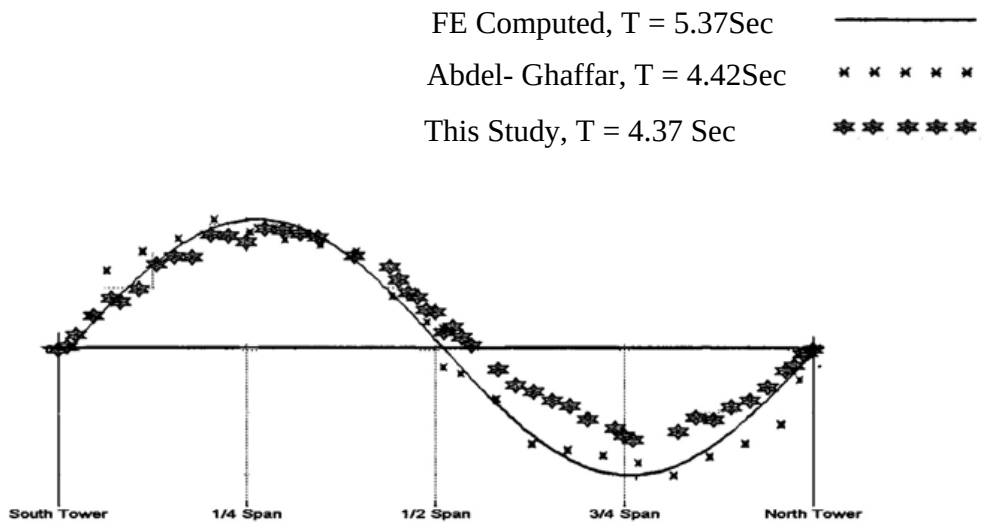

Figure 7.5. Torsional modes also match [22].

Chapter 8

Discussion

This work gives a few implications to WSN that can be interesting research topics.

8.1 Link Estimation Under Heavy Traffic

The routing layer and time synchronization protocol worked fine in laboratory tests, but revealed some problems in the deployment on the Golden Gate Bridge. One more difficulty confronted was that heavy traffic of reliable data collection prevented the routing layer (MintRoute) from estimating link quality correctly. Therefore, after some time of transmission, the routing layer broke down. The routing tree had to be frozen before each data collection. Our best guess is that heavy traffic increases the noise level so that the link estimator gets confused.

8.2 De-synchronization of Packet Transmission Schedule

In Flush, it is still not very clear what happens when a packet transmission schedule gets out of synchronization. CSMA helps in handling a transient mismatch. However, a link-level retransmission makes the problem more subtle. A packet loss will lead to a retransmission. This retransmission is out of schedule and can cause interference. However, the schedule of the following packet will not be influenced by this retransmission since the beginning of transmissions of individual packets are separated by a specified interval regardless of retransmissions.

A more complicated case happens when adjacent nodes have different packet intervals for a while. In this case, the beginning of transmissions can stay out of synchronization for a while, and the resolution of interference and collision may heavily rely on CSMA. CSMA suffers a hidden terminal problem and can lead to a decreased performance in this case. However, even in this case, Flush will decrease the rate to be suitable for reliance on CSMA. Another remaining question is whether Flush recovers back to a synchronized state.

8.3 CSMA versus TDMA

Flush tries to control its rate and schedule packet transmissions in order to avoid intra-path interference. Flush performs these tasks at the transport layer. TDMA also controls a share of the channel for each node and schedules transmissions at the MAC layer, however for a different reason. TDMA tries to coordinate different flows fairly and avoid inter-path collisions.

Controlling packet scheduling and transmission times at the MAC layer can be

more efficient than controlling these functions at the transport layer without any knowledge about lower layers. Therefore, coordination with a TDMA MAC might provide an opportunity for better interference control. However, in TDMA, the channel is wasted unless every node always has data to send in each slot. In the case of Flush, there is only one active flow at a time. Therefore, other slots cannot be utilized, leading to underutilization of potential capacity. TDMA can confront a difficulty when there is a node (A) which cannot be heard clearly. TDMA may exclude this node in scheduling at a node (B). However a node (A) may constantly interfere with transmissions of a node (B).

8.4 Practical Issues

In a Flush test at RFS, MicaZ motes show a shorter communication range under the sun than under a cloud. We do not completely understand which component contributes to a decrease in communication range when heated.

Packaging was a problem in GGB. In spite of enormous care to tightly seal the enclosures, a few nodes had water in their packages. It is not clear whether the water actually entered into the package or air with moisture was circulated into the package and was condensed.

To provide high reliability at the link layer, the hardware and deployment layout were planned and adjusted. A bi-directional patch antenna was used as a hardware solution. When nodes were deployed, the distance between adjacent nodes was adjusted to provide good link quality, Since the data is very important for the research of structural analysis, the system is somewhat overengineered to reduce the risk of losing data. There was not a chance to confront low link quality in normal operation,

except those due to hardware failures. A network with moderate loss could yield interesting observations in a future deployment.

Chapter 9

Conclusion

This work has four major contributions to wireless sensor networks. First, requirements are identified to obtain data of sufficient quality to have real scientific value to civil engineering researchers for structural health monitoring. An accurate data acquisition system, high-frequency sampling with low jitter and time synchronized sampling were not provided by previous work like Wisden [87] and Tenet [38], but are crucial for data to be useful for structural health monitoring, and are provided in this work. Second, the system is designed to scale to a large number of nodes to allow dense sensor coverage of real world structures. This is verified in a 64-node, 46-hop deployment over the main span and a tower of the Golden Gate Bridge. Third, this network is deployed in a real world structure solving a myriad of problems encountered in a real deployment in difficult conditions. As a result, the network provided reliable and calibrated data for analysis, which was not possible in previous studies. Finally, a reliable data collection protocol, Flush is provided. Reflecting on experiences from the deployment at the Golden Gate Bridge, Flush sustains high bandwidth even in a 48-hop network and adapts to environment dynamics.

Bibliography

[1] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/apps/HighFrequencySampling/MicroTimerM.nc. MicroTimer Code in TinyOS Code Repository.

[2] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/GGB/apps/Sentri. Sentri Code in TinyOS Code Repository.

[3] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/GGB/tos/lib/Straw. Straw Code in TinyOS Code Repository.

[4] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/tos/lib/Broadcast. Broadcast Code in TinyOS Code Repository.

[5] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/tos/system/BufferedLog.nc. BufferedLog Code in TinyOS Code Repository.

[6] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/tos/system/TimerC.nc. Timer Code in TinyOS Code Repository.

[7] http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/hsn/README_TinyAODV. TinyAODV Code in TinyOS Code Repository.

[8] http://en.wikipedia.org/wiki/A-law_algorithm. A-law Companding Algorithm).

[9] http://en.wikipedia.org/wiki/Unit_disk_graph. OKW Robust-Box C2012201.

[10] http://mathworld.wolfram.com/.

[11] http://mirage.berkeley.intel-research.net/. Mirage Testbed in Intel Research at Berkeley.

[12] http://rfs.berkeley.edu. Richmond Field Station (RFS).

[13] http://tinyos.net/scoop/special/hardware#mica2. Crossbow Mica2.

[14] http://tinyos.net/scoop/special/hardware#mica2dot. Crossbow Mica2Dot.

[15] http://www.eecs.berkeley.edu/~binetude/ggb/Sentri.htm. Manual for Sentri User Command.

[16] http://www.millennium.berkeley.edu/sensornets/. Soda Hall Testbed.

[17] http://www.okwenclosures.com/products/okw/robust.htm. OKW Robust-Box C2012201.

[18] http://www.rayovacindustrial.com/assets/pdf/marketing_data_sheets/928_003217.pdf. Rayovac 928 Heavy Duty Carbon Zinc Battery.

[19] http://www.superpass.com/SPPG24BD.html. Superpass 2.4GHz Bi-directional Antenna SPAPG24-BD.

[20] http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf. Crossbow MicaZ.

[21] http://www.yuasa-intl.com/1axis.html.

[22] Ahmed M. Abdel-Ghaffar. Ambient vibration studies of golden gate bridge. *Journal of Engineering Mechanics*, 111(4):483–499, April 1985.

[23] T. Blackwell, K. Chang, H.T. Kung, and D. Lin. Credit-based flow control for ATM networks. In *Proc. of the First Annual Conference on Telecommunications R&D in Massachusetts*, 1994.

[24] Johannes Blomer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An xor-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, 1995.

[25] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM '98*. ACM Press, 1998.

[26] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Antony Rowstron. Virtual ring routing: network routing inspired by dhts. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 351–362, New York, NY, USA, 2006. ACM Press.

[27] Juan M. Caicedo, Johannio Marulanda, Peter Thomson, and Shirley J. Dyke. Monitoring of bridges to detect changes in structural health. *the Proceedings of the 2001 American Control Conference, Arlington, Virginia, June 2527, 2001.*

[28] Penggen Cheng, Wenzhong John Shi, and Wanxing Zheng. Large structure health dynamic monitoring using gps technology.

[29] Claude Desset and Andrew Fort. Selection of channel coding for low-power wireless systems. In *Vehicular Technology Conference*, volume 3, pages 1920–1924, April 2003.

[30] Cheng Tien Ee and Ruzena Bajcsy. Congestion control and fairness for many-to-one routing in sensor networks. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 148–161. ACM Press, 2004.

[31] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *the Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA. December 2002.*

[32] Jonathan M. Engel, Lianhan Zhao, Zhifang Fan, Jack Chen, and Chang Liu. Smart brick - a low cost, modular wireless sensor for civil structure monitoring. *International Conference on Computing, Communications and Control Technologies (CCCT 2004), Austin, TX USA, August 14-17, 2004.*

[33] Rodrigo Fonseca, Sylvia Ratnasamy, Jerry Zhao, Cheng Tien Ee, David Culler, Scott Shenker, and Ion Stoica. Beacon vector routing: scalable point-to-point routing in wireless sensornets. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[34] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync protocol for sensor networks. *SenSys 03, November 5-7, 2003, Los Angeles, California, USA.*

[35] Deepak Ganesan. TinyDiffusion Application Programmer's Interface API 0.1. `http://www.isi.edu/scadds/papers/tinydiffusion-v0.1.pdf`.

[36] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[37] Steven D. Glaser, Min Chen, and Thomas E. Oberheim. Terra-scope - a mems-based vertical seismic array. *Smart Structure & Systems*, 2(2):115–126, 2006.

[38] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (Sensys '06)*. ACM Press, November 2006.

[39] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. *ACM SIGCOMM*, August 2003.

[40] Piyush Gupta and P. R. Kumar. The capacity of wireless networks. In *IEEE Transactions on Information Theory*, volume 46, pages 388–404. IEEE Transactions on Information Theory Society, March 2000.

[41] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), Cambridge, MA, USA*, pages 93–104. ACM Press, November 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[42] Jason Hill, Robert Szewczyk, Alec Woo, Philip Levis, Kamin Whitehouse, Joe Polastre, David Gay, Sam Madden, Matt Welsh, David Culler, and Eric Brewer. Tinyos: An operating system for sensor networks, 2003.

[43] 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.

[44] Bret Hull, Kyle Jamieson, and Hari Balakrishnan. Mitigating congestion in wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, November 2004.

[45] Van Jacobson and Michael J. Karels. Congestion avoidance and control. In *In Proceedings of the Sigcomm '88 Symposium*, Stanford, CA, August 1988.

[46] B.F. Spencer Jr., M. Ruiz-Sandoval, and N. Kurata. Smart sensing technology: Opportunities and challenges. *Journal of Structural Control and Health Monitoring, in press, 2004.*

[47] Sukun Kim. Wireless sensor networks for structural health monitoring. Master's thesis, University of California at Berkeley, May 2005.

[48] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *the Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07), Cambridge, MA*. ACM Press, April 2007.

[49] Young Kim, Ramesh Govindan, Brad Karp, and Scott Shenker. Geographic routing made practical. In *Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2005)*, Boston, MA, May 2005.

[50] Kishore Kothapalli, Christian Scheideler, Melih Onus, and Andrea Richa. Constant density spanners for wireless ad-hoc networks. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 116–125, New York, NY, USA, 2005. ACM Press.

[51] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (Sensys), San Diego*, pages 64–75. ACM Press, November 2005.

[52] Markus Krüger and Christian U. Grosse. Structural health monitoring with wireless sensor networks. *Otto-Graf-Journal*, 15:77–90, 2004.

[53] Philip Levis, Sam Madden, David Gay, Joe 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.

[54] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, March 2004.

[55] Jinyang Li, Charles Blake, Douglas S.J. De Couto, Hu Imm Lee, and Robert Morris. Capacity of ad hoc wireless networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 61–69, New York, NY, USA, 2001. ACM Press.

[56] Lennart Ljung. Prentice Hall PTR, Upper Saddle River, N.J., 2nd edition, 1999.

[57] Jerome P. Lynch. Overview of wireless sensors for real-time health monitoring of civil structures. *Proceedings of the 4th International Workshop on Structural Control (4th IWSC), New York City, NY, June 10-11, 2004.*

[58] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[59] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[60] Miklós Maróti. Directed flood-routing framework. Technical Report ISIS-04-502, ISIS, Vanderbuilt University, 2004.

[61] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, New York, NY, USA, 2004. ACM Press.

[62] Petar Maymounkov. Online codes. *NYU, Technical Report TR2002-833*, November 2002.

[63] Partho Pratim Mishra, Hemant Kanakia, and Satish K. Tripathi. On hop-by-hop rate-based congestion control. *IEEE/ACM Trans. Netw.*, 4(2):224–239, 1996.

[64] Thomas Moscibroda. The worst-case capacity of wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 1–10, New York, NY, USA, 2007. ACM Press.

[65] Vinayak Naik, Anish Arora, Prasun Sinha, and Hongwei Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005)*.

[66] Clement Ogaja, Chris Rizos, Jinling Wang, and James Brownjohn. Toward the implementation of on-line structural monitoring using rtk-gps and analysis of results using the wavelet transform.

[67] Jeongyeup Paek, Krishna Chintalapudi, John Cafferey, Ramesh Govindan, and Sami Masri. A wireless sensor network for structural health monitoring: Performance and experience. In *Proceedings of the Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*.

[68] Shamim N. Pakzad, Sukun Kim, Gregory L Fenves, Steven D. Glaser, David E. Culler, and James W. Demmel. Multi-purpose wireless accelerometers for civil infrastructure monitoring. In *Proceedings of the 5th International Workshop on Structural Health Monitoring (IWSHM 2005)*, September 2005.

[69] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, third edition, May.

[70] Marina Petrova, Janne Riihijarvi, Petri Mahonen, and Saverio Labella. Performance study of ieee 802.15.4 using measurements and simulations. In *Wireless Communications and Networking Conference 2006 (WCNC 2006)*, volume 1, pages 487–492, April 2006.

[71] 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.

[72] Pei Qiang, Guo Xun, and Zhao Chang-you. A wireless structural health monitoring system in civil engineering. *The Third International Conference on Earthquake Engineering (3ICEE), Nanjing, China, October 18-20, 2004*.

[73] Sumit Rangwala, Ramakrishna Gummadi, Ramesh Govindan, and Konstantinos Psounis. Interference-aware fair rate control in wireless sensor networks. In *SIGCOMM 2006*, Pisa, Italy, August 2006.

[74] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, April 1997.

[75] Y. Sankarasubramaniam, O. Akan, and I. Akyildiz. ESRT: Event-to-sink reliable transport in wireless sensor networks. In *In Proceedings of MobiHoc*, June 2003.

[76] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 1–12, New York, NY, USA, 2004. ACM Press.

[77] Prasun Sinha, Thyagarajan Nandagopal, Narayanan Venkitaraman, Raghupathy Sivakumar, and Vaduvur Bharghavan. WTCP: A reliable transport protocol for wireless wide-area networks. *Wireless Networks*, 8(2-3):301–316, 2002.

[78] Fred Stann and John Heidemann. Rmst: Reliable data transport in sensor networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pages 102–112. IEEE, April 2003.

[79] Karthikeyan Sundaresan, Vaidyanathan Anantharaman, Hung-Yun Hsieh, and Raghupathy Sivakumar. ATP: a reliable transport protocol for ad-hoc networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc '03)*, pages 64–75, 2003.

[80] Gilman Tolle and David Culler. Design of an application-cooperative management system for wireless sensor networks. In *the Proceedings of the 2nd European*

Workshop on Wireless Sensor Networks (EWSN 2005), Istanbul, Turkey, January 2005.

[81] 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. In *the Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (Sensys 05), San Diego*. ACM Press, November 2005.

[82] Amit K. Vyas and Fouad A. Tobagi. Impact of interference on the throughput of a multihop path in a wireless network. In *The Third International Conference on Broadband Communications, Networks, and Systems (Broadnets 2006)*.

[83] Chieh-Yih Wan, Andrew T. Campbell, and Lakshman Krishnamurthy. Psfq: a reliable transport protocol for wireless sensor networks. In *Proc. of the 1st ACM international workshop on Wireless sensor networks and applications*. ACM Press, 2002.

[84] Geoff Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *the Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005), Istanbul, Turkey*, January 2005.

[85] Alec Woo and David E. Culler. A transmission control scheme for media access in sensor networks. In *Proceedings of the seventh annual international conference on Mobile computing and networking*, Rome, Italy, July 2001.

[86] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

[87] Ning Xu, Sumit Rangwala, Krishna Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. *the Proceedings of the ACM Conference on Embedded Networked Sensor Systems, November 2004.*

[88] M. D. Yarvis, W. S. Conner, L. Krishnamurthy, A. Mainwaring, J. Chhabra, , and B. Elliott. Real-world experiences with an interactive ad hoc sensor network. In *Proceedings of the International Conference on Parallel Processing Workshop*, 2002.

[89] Jerry Zhao and Ramesh Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the First International Conference on Embedded Network Sensor Systems*, 2003.