University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Open Access Dissertations

9-2010

Data Management and Wireless Transport for Large Scale Sensor Networks

Ming Li
University of Massachusetts Amherst, mingli@cs.umass.edu

Follow this and additional works at: https://scholarworks.umass.edu/open_access_dissertations

Part of the Computer Sciences Commons

Recommended Citation
Li, Ming, "Data Management and Wireless Transport for Large Scale Sensor Networks" (2010). *Open Access Dissertations*. 290.
https://scholarworks.umass.edu/open_access_dissertations/290

This Open Access Dissertation is brought to you for free and open access by ScholarWorks@UMass Amherst. It has been accepted for inclusion in Open Access Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

DATA MANAGEMENT AND WIRELESS TRANSPORT FOR LARGE SCALE SENSOR NETWORKS

A Dissertation Presented

by

MING LI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2010

Department of Computer Science

© 2010 Ming Li

DATA MANAGEMENT AND WIRELESS TRANSPORT FOR LARGE SCALE SENSOR NETWORKS

A Dissertation Presented

by

MING LI

Approved as to style and content by:

Deepak Ganesan, Co-Chair

Arun Venkataramani, Co-Chair

Prashant Shenoy, Member

Jim Kurose, Member

Lixin Gao, Member

Andrew Barto, Department Chair
Department of Computer Science

To Lili and Eleanor.

ACKNOWLEDGMENTS

This chapter is dedicated to all the people who have helped me in my graduate career.

First I would like to thank my advisors: Professor Deepak Ganesan and Professor Arun Venkataramani for their continuous and precious advising during my ph.d study. They have been giving me tremendous help on both research and personal life, and have been encouraging me all the time to exploit new research domains and to attack hard research problems. Without their tremendous help, this thesis work is impossible. The research methods and personality I learned from them will continue to benefit my future career. I would also like to thank the rest of my thesis committee: Professor Prashant Shenoy, Professor Jim Kurose, and Professor Lixin Gao, for their valuable comments on my thesis work.

Next I would like to thank my colleagues in SENSORS lab, LASS lab and Networking lab, as well as others in the department. I had a great time collaborating, discussing, and enjoying my life with them. In particular, I am thankful to Devesh Agrawal, Peter Desnoyers, Purushottam Kulkami, Xiaotao Liu. Huan Li, Tim Wood, Gaurav Marthur, Jeremy Gummeson, Aruna Balasubramanian, Nilanjan Banerjee, Tingxin Yan, Xiaozheng Tie, for their help with my work.

I would also like to extend my appreciating to all the people in the Department of Computer Science at UMass. In particular, I am grateful to Karren Sacco, Leeanne Leclerc, Michelle Eddy, Tyler Trafford, and the members of CSCF for their kindness and incredible efficiency.

Finally I would like to thank my wonderful wife, Lili Cheng, and my lovely daughter, Eleanor Li. Without their support there is no way I can go this far.

ABSTRACT

DATA MANAGEMENT AND WIRELESS TRANSPORT FOR LARGE SCALE SENSOR NETWORKS

SEPTEMBER 2010

MING LI

B.Sc., HARBIN INSTITUTE OF TECHNOLOGY

M.Sc., TSINGHUA UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Deepak Ganesan, Professor Arun Venkataramani

Today many large scale sensor networks have emerged, which span many different sensing applications. Each of these sensor networks often consists of millions of sensors collecting data and supports thousands of users with diverse data needs. Between users and wireless sensors there are often a group of powerful servers that collect and process data from sensors and answer users' requests. To build such a large scale sensor network, we have to answer two fundamental research problems: i) what data to transmit from sensors to servers? ii) how to transmit the data over wireless links?

Wireless sensors often can not transmit all collected data due to energy and bandwidth constraints. Therefore sensors need to decide what data to transmit to best satisfy users' data requests. Sensor network users can often tolerate some data errors, thus sensors may transmit data in lower fidelity but still satisfy users' requests. There are generally two types of requests–raw data requests and meta-data requests. To answer users' raw data requests, we propose a model-driven data collection approach, PRESTO. PRESTO splits intelligence between sensors and servers, i.e., resource-rich servers perform expensive model training and resource-poor sensors perform simple model evaluation. PRESTO can significantly reduce data to be transmitted without sacrificing service quality. To answer users' meta-data request, we propose a utility-driven multi-user data sharing approach, MUDS. MUDS uses utility function to unify diverse meta-data metrics. Sensors estimate utility value of each data packet and sends packets with highest utility first to improve overall system utility.

After deciding what data to transmit from sensors to servers, the next question is how to transmit these data over wireless links. Wireless transport often suffers low bandwidth and unstable connectivity. In order to improve wireless transport, I propose a clean-slate re-design of wireless transport, Hop. Hop uses reliable per-hop block transfer as a building block and builds all other components including hidden-terminal avoidance, congestion avoidance, and end-to-end reliability on top of it. Hop is built based on three key ideas: a) hop-by-hop transfer adapts to the lossy and highly variable nature of wireless channel significantly better than end-to-end transfer, b) the use of blocks as the unit of control is more efficient over wireless links than the use of packets, and c) the duplicated functionalities in different layers in the network stack should be removed to simplify the protocol and avoid complex interaction.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . **v**

ABSTRACT . . . . **vi**

LIST OF TABLES . . . . **xi**

LIST OF FIGURES . . . . **xii**

CHAPTER

1 INTRODUCTION **1**

1.1 How to Reduce Data Transmissions? 1

1.1.1 Handling of raw data requests 2
1.1.2 Handling of meta-data requests 3

1.2 How to Improve Wireless Transport? 3
1.3 Contributions 4
1.4 Thesis Outline 5

2 FEEDBACK-DRIVEN DATA MANAGEMENT **6**

2.1 Introduction 6

2.1.1 Research Contributions 8

2.2 System Architecture 9
2.3 PRESTO Proxy 11

2.3.1 Modeling and Prediction Engine 11
2.3.2 Query Processing at a Proxy 15
2.3.3 Proxy Cache 15
2.3.4 Failure Detection 16

2.4 PRESTO Sensor 16
2.5 Adaptation in PRESTO 17

2.5.1 Adaptation to Data Dynamics 17
2.5.2 Adaptation to Query Dynamics 18

2.6 PRESTO Implementation 19
2.7 Experimental Evaluation 20

2.7.1 Microbenchmarks 20
2.7.2 Performance of Model-Driven Push 23
2.7.3 PRESTO Scalability 25

2.7.3.1 Impact of Network Size 25
2.7.3.2 Impact of Query Rate 26

2.7.4 PRESTO Adaptation 27
2.7.5 Failure Detection 29

2.8 Related Work 29
2.9 Conclusions 30

3 UTILITY-DRIVEN MULTI-USER DATA SHARING 32

3.1 Introduction 32

3.1.1 Research Contributions 32

3.2 Radar Sensor Networks 34

3.2.1 End User Applications 34
3.2.2 System Model and Problem Formulation 35

3.3 MUDS System Architecture 38
3.4 MUDS System Design 40

3.4.1 Multi-Query Aggregator 40
3.4.2 Progressive Compression Engine 41
3.4.3 Local Transmission Scheduler 43
3.4.4 Global Transmission Control 45

3.5 Experimental Evaluation 46

3.5.1 Determining the Utility Function 47
3.5.2 Impact of Weighting Policy 49
3.5.3 Performance of Progressive Compression 49

3.5.3.1 Compression Efficiency 49
3.5.3.2 Bandwidth Adaptation 50

3.5.4 Performance of Data Sharing 51

3.5.4.1 Temporally Overlapping Queries 51
3.5.4.2 Spatially Overlapping Queries 52

3.5.5 Performance of Local Scheduler 53
3.5.6 Performance of Global Control 54
3.5.7 System Scalability 55

3.5.7.1 Impact of Network Size 55
3.5.7.2 Impact of Query Load 57

3.6 Related Work 58
3.7 Conclusions 60

4 HIGH THROUGHPUT RELIABLE WIRELESS TRANSPORT 61

4.1 Introduction 61
4.2 Why reliable per-hop block transfer? 63
4.3 Design 64

4.3.1 Reliable per-hop block transfer . . . 64
4.3.2 Ensuring end-to-end reliability . . . 65
4.3.3 Backpressure congestion control . . . 66
4.3.4 Robustness to partitions . . . 68
4.3.5 Handling hidden terminals . . . 68
4.3.6 Packet scheduling . . . 69

4.4 Implementation . . . 70

4.4.1 MAC parameters . . . 70
4.4.2 Hop implementation . . . 70

4.5 Evaluation . . . 71

4.5.1 Single-hop microbenchmarks . . . 72

4.5.1.1 Randomly picked links . . . 73
4.5.1.2 Graceful performance degradation . . . 74

4.5.2 Multi-hop microbenchmarks . . . 75
4.5.3 Hop under high load . . . 76

4.5.3.1 Goodput . . . 77
4.5.3.2 Fairness . . . 78

4.5.4 Hop performance breakdown . . . 78
4.5.5 Hop with WLAN access points . . . 79
4.5.6 Hop delay for small file transfers . . . 80

4.5.6.1 Single-hop transfer delay for small files . . . 80
4.5.6.2 Multi-hop transfer delay for Web file sizes . . . 80

4.5.7 Robustness to partitions . . . 81
4.5.8 Hop with VoIP . . . 82
4.5.9 Network and link layer dynamics . . . 83
4.5.10 Hop under 802.11g . . . 84
4.5.11 Discussion: Hop vs. TCP . . . 85

4.6 Related work . . . 86

4.6.1 Proposed alternatives to TCP . . . 86
4.6.2 Implemented alternatives to TCP . . . 86
4.6.3 Other related work . . . 87

4.7 Conclusions . . . 87

5 CONCLUSIONS . . . 89

BIBLIOGRAPHY . . . 90

LIST OF TABLES

2.1 Energy micro-benchmarks for sensor nodes. 21

2.2 Round trip latencies using B-MAC 21

2.3 Asymmetry: Model estimation vs Model checking 22

3.1 Comparision of different weighting policies. 49

4.1 Summary of evaluation results. All protocols above are given the benefit of burst-mode (txop) and the maximum number of link-layer retransmissions (max-ARQ) supported by the hardware. 71

4.2 Fairness indexes for the 30 flow experiment. Parentheses show 95% confidence intervals. 78

4.3 Mean/median goodput and Fairness for a many-to-one “AP” setting. 95% confidence intervals shown in parenthesis 79

4.4 Goodput achieved by Hop and DTN2.5 in a partitioned network without an end-to-end path. 82

4.5 Impact of Hop and TCP on VoIP flows. Result shows the median/mean goodput, conditional loss probability, and MOS for VoIP with 95% confidence intervals in parentheses. 83

LIST OF FIGURES

2.1 The PRESTO data management architecture. . . . 10

2.2 The PRESTO proxy comprises a prediction engine, query processor and a cache of predicted and real sensor values. The PRESTO sensor comprises a model checker and an archive of past samples with the model predictions. . . . 12

2.3 Comparison of PRESTO SARIMA models with model-driven pull and value-driven push. . . . 22

2.4 Scalability of PRESTO: Impact of network size. . . . 23

2.5 Scalability of PRESTO: Impact of query rates. . . . 24

2.6 Adaptation in PRESTO to data and query dynamics as well as adaptation in an outdoor deployment. . . 28

2.7 Evaluation of failure detection 29

3.1 Multi-user Radar Sensor Networks 33

3.2 Multi-hop Radar Sensor Networks 36

3.3 Multiple incoming queries in an epoch are first aggregated by the multi-query aggregator at the radar. The merged query and the radar scan for the epoch are input to the progressive encoder which generates different compressed streams for different regions in the query. The streams are input to the utility-driven scheduler which schedules packets across all streams whose deadlines have not yet expired. . . . 37

3.4 In this scenario, 66% of stream 1 and 33% of stream 2 have been transmitted. The scheduler determines the marginal utility of transmitting a packet from each of the streams for the applications interested in the streams and decides which packet to transmit next. . . . 40

3.5 The routing topology of a 13-node wireless testbed with one proxy and twelve emulated radars. . . . 47

3.6 Utility functions for the three applications are derived by compressing and evaluating application performance on traces from the Oklahoma dataset. . . . 48

3.7 Comparison of SPIHT progressive compression against averaging compression. Each algorithm compresses data to the size that can be transmitted in one epoch for a given bandwidth. . . . 50

3.8 Comparison of progressive compression against non-progressive compression for different levels of bandwidth fluctuation. Bandwidth fluctuation follows a normal distribution with mean 40kbps; standard deviation is varied from 0kbps to 25kbps. . . . 52

3.9 Time series of bandwidth and MSE of decoded data. Bandwidth fluctuation follows a normal distribution with mean value at 40kbps and standard deviation of 25kbps. . . . 53

3.10 Performance for temporally overlapping queries. Two queries with different deadlines but same region of interest arrive at the radar every two epochs. 54

3.11 Evaluation of the impact of data sharing on utility. Two applications, tornado detection and 3D assimilation, with overlapping sectors are considered. 55

3.12 Comparison of utility-driven scheduling against random scheduling. 56

3.13 Performance of global transmission control. Utility is shown for differing numbers of hops from the proxy to nodes having high-priority queries. 57

3.14 Scalability to network size. Breakdown of contribution of each component of our system to the overall utility. 58

3.15 System scalability to the query load. 59

4.1 Structure of a block. 65

4.2 Timeline of TCP/802.11 vs. Hop 66

4.3 Virtual retransmission due to node failure. 67

4.4 Example showing need for backpressure. Without backpressure, Node A would allocate $1/5$th of outgoing capacity to each flow, resulting in queues increasing unbounded at nodes B through E. With backpressure, most data is sent to node F, thereby increasing utilization. 67

4.5 Experimental testbed with dots representing nodes. 72

4.6 Experiment with one-hop flows. Hop improves lower quartile goodput by 28×, median goodput by 1.6×, and mean goodput by 1.6× over TCP with the best link layer settings. . . 73

4.7 Experiment with one-hop flows. Box shows lower/median/upper quartile, lines show max/min, and dot shows mean. Increasing 802.11 ARQ limit and using txops helps TCP but Hop is still considerably better. UDP results show that ARQs incur significant performance overhead (35%). Hop is within 24% of UDP without ARQ (achievable goodput). 74

4.8 Graceful degradation to adverse channel conditions. First plot shows per-link goodputs from one-hop experiment sorted in TCP order. Second plot shows controlled experiments demonstrating impact of loss. In both cases, Hop is more robust and degrades far more gracefully than TCP. 75

4.9 Experiment with multi-hop flows. Hop improves lower quartile goodput by 2.7×, median goodput by 2.3×, and mean goodput by 2×. 76

4.10 Boxplot of multi-hop single-flow benchmarks. Hop has 2-3× median, and 2-4× mean improvements over other reliable transport protocols. Hop is comparable to UDP/no-ARQ/txop in terms of median/mean — the latter is extremely fast since it has no overhead, but experiences more loss. 76

4.11 Hop for 30 concurrent flows. Dots on each line shows mean goodput. Median gains of Hop over Hop-by-Hop TCP and regular TCP are huge (20× and 90× respectively) while mean gains are modest (roughly 25% improvement). 77

4.12 Hop performance breakdown showing contribution of ack withholding and backpressure. Ack withholding and backpressure improve Hop's performance by more than 4.8x under high load. 78

4.13 Hop for WLAN: Hop improves delay for all file sizes with improvements between 3-15× 80

4.14 Performance for web traffic: Except the 32KB bin, Hop has comparable or better delay, with gains upto 6× 81

4.15 Hop for 30 concurrent flows under dynamic routing and auto bit-rate. Dots on each line shows mean goodput. Median gains by Hop with fixed bit-rate are around 4× over TCP with OLSR and more than 90× over TCP with static routing. 83

4.16 Hop for 30 concurrent flows under 802.11g. Dots on each line shows mean goodput. Hop's median gain is 22× over TCP with bit-rate fixed at 24Mbps, and is 6× over TCP with auto-rate control. Hop's mean gain is 3× over TCP with auto-rate control. 84

CHAPTER 1

INTRODUCTION

The recent years have seen remarkable development of large scale sensing applications, e.g. traffic monitoring networks and weather monitoring networks. Each of these sensor networks often consists of millions of sensors and supports thousands of users with diverse data needs. For instance, in traffic monitoring networks large number of sensors such as cameras, speed sensors, and driver-carried smart phones are used to collect traffic related information in fine granularity. The collected data are useful to many users including drivers, people planing trips, as well as organizations such as the department of transportation. Between users and wireless sensors there are often a group of servers with rich computation, storage, and energy resources. These servers collect data from sensors, store and process the data, and answer to users' data requests. To build such large scale sensor networks, we have to address several research problems. First of all, wireless sensors often do not have enough energy and bandwidth resources to transmit all sensed data to servers. Therefore, how to reduce data to be transmitted without sacrificing service quality posts a major challenge to wireless sensing applications. Secondly, wireless transport suffers low bandwidth and unstable connectivity. How to improve the performance of wireless transport is an important question to wireless sensor network design. The rest of this chapter discusses these two research problems in detail and presents our contributions to address these problems.

1.1 How to Reduce Data Transmissions?

We first examine how to reduce data transmissions over wireless links. Wireless sensors may not be able to transmit all the data they collected due to energy and bandwidth constraints. To reduce data size without sacrificing data's fidelity, wireless sensors may do lossless compression on data. However, the computation load of lossless compression may exceed sensors' capacity, and the compressed data may still exceed wireless sensor's transmission capacity. On the other hand, sensor network users can often tolerant data errors to certain extend. For instance, a user of a temperature monitoring network may specify that she wants temperature readings with $\pm 1^o F$ accuracy. Therefore, it is possible that sensors can transmit data in lower fidelity but still satisfy users' requests. We consider two cases of user requests, the case where users just want raw

data and the case where users want high level meta data after processing, and propose different techniques of data reduction for each case.

1.1.1 Handling of raw data requests

To answer users' raw data requests, servers need not do any data processing but to forward raw data directly from sensors to servers. Users specify their data fidelity needs in raw data metric. Sensors and servers can reduce data transmitted over wireless links according to users' data fidelity requests. A data reduction technique used in this case is model-driven data collection. In model-driven data collection, models are built to capture spatial or temporal correlations in raw data. Then servers answer users' requests with model predictions without actually getting raw data from sensors. Data transmissions only happen when prediction results are not accurate enough to satisfy users' requests. People have proposed several model-driven data collection systems. According to where the intelligence of the model-driven process is placed, we classify these approaches to sensor-centric approaches and server-centric approaches.

In sensor-centric approach, intelligence such as model building and model evaluation is placed on the sensors. Sensors train spatial or temporal model on collected data and send trained model to servers. Both servers and sensors use the model to predict current data. Sensors compare predicted data to sensed data. If the prediction error is higher than what users request, sensors will push sensed data to servers. Using sensor-centric approach, only data anomalies that are not predictable by models are transmitted. However, the heavy computation load in the model-training process may exceed wireless sensors' capability. The server-centric approaches, on the other hand, perform model training and evaluation on the resource-rich servers. Servers train model using archived data, then use model predictions to answer users' requests. To evaluate prediction accuracy, servers calculate confidence interval of prediction and pull real sensed data from sensors when the confidence interval exceeds users' requests. The server-centric approaches perform computationally-expensive model-training on resource-rich servers while keep resource-poor sensors simple. However, since servers can only use confidence interval to estimate prediction accuracy, the server-centric approaches can not efficiently capture data anomalies. Since both of these approaches have major drawbacks, we propose a feedback driven data management approach, PRESTO, that splits intelligence between sensors and servers according to their computation capabilities. In PRESTO, resource-rich servers perform expensive model training, and sensors perform simple model evaluation. In this thesis, we show that PRESTO can greatly reduce data to be transmitted while efficiently capture data anomalies.

1.1.2 Handling of meta-data requests

In advanced sensing applications, users often requires meta-data after data processing. For instance in weather monitoring networks, instead of sending user raw radar data servers often need to run data processing applications such as tornado detection and precipitation estimation on the raw data, and send the processed result to users. Users specify data fidelity needs in meta-data metrics such as location and strength of tornado instead of in raw data metric. Since data reduction algorithms process raw data directly, it is necessary to translate users' meta-data fidelity request to raw data fidelity request. Furthermore, in such advanced sensing applications users often share the same raw data even they are requesting different raw data. For instance, users interested in tornado detection and users interested in precipitation estimation may actually use raw data from the same radar. Therefore, even they have different meta-data requests, they can still share the same raw data stream. This requires sensors to aggregate users' meta-data requests during reduction of raw data. In this thesis, I propose a utility-driven multi-user data sharing approach, MUDS, that uses utility function to covert raw data fidelity to meta-data fidelity, maximizes data sharing among users, and sends data with highest utility first to improve overall system utility.

The design of MUDS addresses three key challenges: how to define utility functions for networks with data sharing among end-users, how to compress and prioritize data transmissions according to its importance to end users, and how to gracefully degrade end-user utility in the presence of bandwidth fluctuations. A key contribution of our work is multi-query aggregation, where raw data streams are shared between multiple users with diverse meta-data requests, thereby maximizing total end-user utility. At the core of our system is a utility-driven progressive data compression and packet scheduling engine at each sensor. The progressive compression engine enables raw data to be compressed and ordered such that information of most interest to queries is transmitted first. Such an encoding enables our system to adapt gracefully to bandwidth fluctuations.The utility-driven scheduler compares the utility of different progressively compressed streams that are intended for different sets of queries, and transmits packets such that utility across all concurrent queries at a sensor is maximized.

1.2 How to Improve Wireless Transport?

Now we know what data to transmit, the next question is how to transmit these data. Sensors transmit data to servers through wireless links. However, transmitting data reliably with high throughput on wireless networks is hard due to several constraints of wireless links. For instance, TCP, the universal transport protocol for wireless transport, is ill-suited to 802.11 Wi-Fi wireless link layer protocol, which incurs low throughput and intermittent availability. We identify following three fundamental problems of the existing

network stacks over wireless: i) the use of per-packet control which incurs high overhead, ii) the use of end-to-end techniques which work poorly due to high link dynamics, and iii) the bad interactions between duplicated functions of different layers in the stack. Instead of patching existing stack, we propose a clean-slate re-design, Hop.

Hop uses reliable per-hop block transfer as a building block and builds all other components in the stack such as hidden-terminal avoidance, congestion avoidance, and end-to-end reliability on top of it. We argue that a) hop-by-hop transfer adapts to the lossy and highly variable nature of wireless channel significantly better than end-to-end transfer, b) the use of blocks as the unit of control is more efficient over wireless links than the use of packets, and c) the duplicated functionalities in different layers in the network stack should be removed to simplify the protocol and avoid complex interaction.

1.3 Contributions

This thesis work makes following three research contributions that address the data management and the wireless transport problems in wireless sensor networks.

- **PRESTO–feedback-driven data management.** In this work we consider the sensing applications in which the sensors have neither enough computation capability to do extensive data processing, nor enough bandwidth and energy to transmit the raw data to the users. We propose a novel two-tier sensor data management architecture, PRESTO, that comprises proxies and sensors that cooperate with one another for acquiring data and processing queries. PRESTO proxies construct time-series models of observed trends in the sensor data and transmit the parameters of the model to sensors. Sensors check sensed data with model-predicted values and transmit only deviations from the predictions back to the proxy. We argue that such a model-driven push approach is energy efficient, while ensuring that anomalous data trends are never missed.

- **MUDS–utility-driven multi-user data sharing.** In this work we focus on how to support diverse user needs in rich sensor networks. We propose a utility-driven approach to maximize data sharing across users while judiciously using limited network and computational resources. Our utility-driven architecture addresses three key challenges: how to define utility functions for networks with data sharing among end-users, how to compress and prioritize data transmissions according to its importance to end users, and how to gracefully degrade end-user utility in the presence of bandwidth fluctuations.

- **Hop-block-based high throughput reliable wireless transport.** In this work we focus on the networking problems in wireless sensor networks. We propose a high throughput reliable wireless transport protocol, Hop, that works across diverse wireless networks. Hop uses reliable per-hop block

transfer as a building block and builds all other components in the stack on top of block transfer. We argue that a) hop-by-hop transfer adapts to the lossy and highly variable nature of wireless channel significantly better than end-to-end transfer, b) the use of blocks as the unit of control is more efficient over wireless links than the use of packets, and c) the duplicated functionalities in different layers in the network stack should be removed to simplify the protocol and avoid complex interaction.

1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we propose a feedback-driven data management system for data reduction in case of raw data requests. In Chapter 3, we propose a utility-driven multi-user data sharing system for data reduction in case of meta-data requests. In Chapter 4, we propose a block-based high throughput reliable wireless transport protocol that works across diverse wireless networks. Finally, in Chapter 5, we summarize the main contribution.

CHAPTER 2

FEEDBACK-DRIVEN DATA MANAGEMENT

2.1 Introduction

Many sensor networks today use a two-tier architecture — a proxy tier consists of a few resource rich base stations, each controlling tens of wireless sensors at the lower tier. The base stations are usually tethered PCs with much richer computation and energy resources than the sensors. Given such a hierarchical architecture, a natural design question is how to explore the rich resources on the base stations to assist the resource-poor sensors. In this work, we argue that splitting data processing load between sensors and proxies is more efficient than placing data processing solely on the sensors or on the proxies, and propose a feedback-driven data management approach that achieves both high energy efficiency and low query latency.

We consider resource-constrained sensors with very limited energy and computation resources. Energy is usually the most critical resource due to the fact that these sensors are often battery-powered, and are deployed in remote or hard-to-reach areas so have to run for long time without changing battery. Consequently, energy-efficient data management is a key problem in these sensor applications. Computation capability is also limited on these low-end sensors, for instance, the widely used TelosB Mote[22] that has 8MHz micro-controller with 10kB RAM. Thus, these sensors can not perform complex data processing efficiently.

Data management approaches in sensor networks have centered around two competing philosophies. Early efforts such as Directed Diffusion [15] and Cougar [27] espoused the notion of the sensor network as a database. The framework assumes that intelligence is placed at the sensors and that queries are pushed deep into the network, possibly all the way to the remote sensors. Direct querying of remote sensors reduces communication needs by processing query at (or close to) the data source, therefore, it is energy efficient since wireless communications are usually the most energy expensive operations on sensors. However, direct querying assumes that remote sensors have sufficient processing resources to handle query processing, an assumption that may not hold in untethered networks of inexpensive sensors (e.g., Berkeley Motes [23]). In contrast, efforts such as TinyDB [17] and acquisitional query processing [9] from the database community have adopted an alternate approach. These efforts assume that intelligence is placed at the edge of the network, while keeping the sensors within the core of the network simple. In this approach, data is pulled from remote sensors by edge elements such as base-stations, which are assumed to be less resource- and energy-

constrained than remote sensors. Sensors within the network are assumed to be capable of performing simple processing tasks such as in-network aggregation and filtering, while complex query processing is left to base stations (also referred to as micro-servers or sensor proxies). In acquisitional query processing [9], for instance, the base-station uses a spatio-temporal model of the data to determine when to pull new values from individual sensors; data is refreshed from remote sensors whenever the confidence intervals on the model predictions exceed query error tolerances.

While both of these philosophies inform our work, existing approaches have several drawbacks.

- *Need to capture unusual data trends:* Sensor applications need to be alerted when unusual trends are observed in the sensor field; for instance, a sudden increase in temperature may indicate a fire or a break-down in air-conditioning equipment. Although rare, it is imperative for sensor applications, particularly those used for monitoring, to detect these unusual patterns and to do so with low latency. Both TinyDB [17] and acquisitional query processing [9] rely on a pull-based approach to acquire data from the sensor field. A pure pull-based approach can never guarantee that all unusual patterns will always be detected, since the anomaly may be confined between two successive pulls. Further, increasing the pull frequency to increase anomaly detection probability has the harmful side-effect of increasing energy consumption at the tetherless sensors.

- *Support for archival queries:* Many existing efforts focus on querying and processing of current (live) sensor data, since this is the data of most interest to the application. However, support for querying historical data is also important in many applications such as surveillance, where the ability to retroactively "go back" is necessary to determine, for instance, how an intruder broke into a building. Similarly, archival sensor data is often useful to conduct postmortems of unexpected and unusual events to better understand them for the future. Architectures and algorithms for efficiently querying archival sensor data have not received much attention in the literature.

- *Adaptive system design:* Long-lived sensor applications that are deployed for months or years need to adapt to data and query dynamics while meeting user performance requirements. As data trends evolve and change over time, the system needs to adapt accordingly to optimize sensor communication overhead. Similarly, as the workload—query characteristics and error tolerance—changes over time, the system needs to adapt by updating the parameters of the models used for data acquisition. Such adaptation is key for enhancing the longevity of the sensor application.

2.1.1 Research Contributions

In this chapter we present PRESTO, a two-tier sensor architecture that comprises sensor proxies at the higher tier, each controlling tens of remote sensors at the lower tier. PRESTO[1] proxies and sensors interact and cooperate for acquiring data and processing queries. PRESTO strives to achieve energy efficiency and low query latency by exploiting resource-rich proxies, while respecting constraints at resource-poor sensors. Like TinyDB, PRESTO puts intelligence at the edge proxies while keeping the sensors inside the network simple. A key difference though is that PRESTO endows sensors with the ability to asynchronously push data to proxies rather than solely relying on pulls. Our design of PRESTO has led to the following contributions.

Model-driven Push: Central to PRESTO is the use of a feedback-based model-driven push approach to support queries in an energy-efficient, accurate and low-latency manner. PRESTO proxies construct a model that captures correlations in the data observed at each sensor. The remote sensors check the sensed data against this model and push data only when the observed data deviates from the values predicted by the model, thereby capturing anomalous trends. Such a model-driven push approach reduces communication overhead by only pushing deviations from the observed trends, while guaranteeing that unusual patterns in the data are never missed. An important requirement of our model is that it should be very inexpensive to check at resource-poor sensors, even though it can be expensive to construct at the resource-rich proxies. PRESTO employs seasonal ARIMA-based time series models to satisfy this *asymmetric* requirement.

Support for archival queries: Whereas PRESTO supports queries on current data using model-driven push, it also supports queries on historical data using a novel combination of prediction, interpolation, and local archival. By associating confidence intervals with the model predictions and caching values predicted by the model in the past, a PRESTO proxy can directly respond to such queries using cached data so long as it meets query error tolerances. Further, PRESTO employs interpolation methods to progressively refine past estimates whenever new data is fetched from the sensors. PRESTO sensors also log all observations on relatively inexpensive flash storage; the proxy can fetch data from sensor archives to handle queries whose precision requirements can not be met using the local cache. Thus, PRESTO exploits the proxy cache to handle archival queries locally whenever possible and resorts to communication with the remote sensors only when absolutely necessary.

Adaptation to Data and Query Dynamics: Long-term changes in data trends are handled by periodically refining the parameters of the model at the proxy, which improves prediction accuracy and reduces the number of pushes. Changes in query precision requirements are handled by varying the threshold used at a sensor to trigger a push. If newer queries require higher precision (accuracy), then the threshold is reduced to

[1] PRESTO is an acronym for PREdictive STOrage.

ensure that small deviations from the model are reported to the proxy, enabling it to respond to queries with higher precision. Overall, PRESTO proxies attempt to balance the cost of pushes and the cost of pulls for each sensor.

We have implemented PRESTO using a Stargate proxy and Telos Mote sensors. We demonstrate the benefits of PRESTO using an extensive experimental evaluation. Our results show that PRESTO can scale up to one hundred Motes per proxy. When used in a temperature monitoring application, PRESTO imposes an energy requirements that is one to two orders of magnitude less than existing techniques that advocate on-demand, proactive, or model-driven pulls. At the same time, the average latency for queries is within six seconds for a 1% duty-cycled five hop sensor network, which is an order of magnitude less than a system that forwards all queries to remote sensor nodes, while not significantly more than a system where all queries are answered at the proxy.

The rest of this chapter is structured as follows. Section 2.2 provides an overview of PRESTO. Sections 2.3 and 2.4 describe the design of the PRESTO proxy and sensors, respectively, while Section 2.5 presents the adaptation mechanisms in PRESTO. Sections 2.6 and 2.7 present our implementation and our experimental evaluation. Finally, Sections 2.8 and 2.9 discuss related work and our conclusions.

2.2 System Architecture

System Model: PRESTO envisions a two-tier data management architecture comprising a number of sensor proxies, each controlling several tens of remote sensors (see Figure 2.1). Proxies at the upper tier are assumed to be rich in computational, communication, and storage resources and can use them continuously. The task of this tier is to gather data from the lower tier and answer queries posed by users or the application. A typical proxy configuration may be comprised of an Intel Stargate [25] node with multiple radios—an 802.11 radio that connects it to an IP network and a low-power 802.15.4 radio that connects it to sensors in the lower tier. Proxies are assumed to be tethered or powered by a solar cell. A typical deployment will consist of multiple geographically distributed proxies, each managing tens of sensors in its vicinity. In contrast, PRESTO sensors are assumed to be low-power nodes, such as Telos Motes [22], equipped with one or more sensors, a micro-controller, flash storage and a wireless radio. The task of this tier is to sense data, transmit it to proxies when appropriate, while archiving all data locally in flash storage. The primary constraint at this tier is energy—sensor nodes are assumed to be untethered, and hence battery-powered, with a limited lifetime. Sensors are assumed to be deployed in a multi-hop configuration and are aggressively duty-cycled; standard multi-hop routing and duty-cycled MAC protocols can be used for this purpose. Since

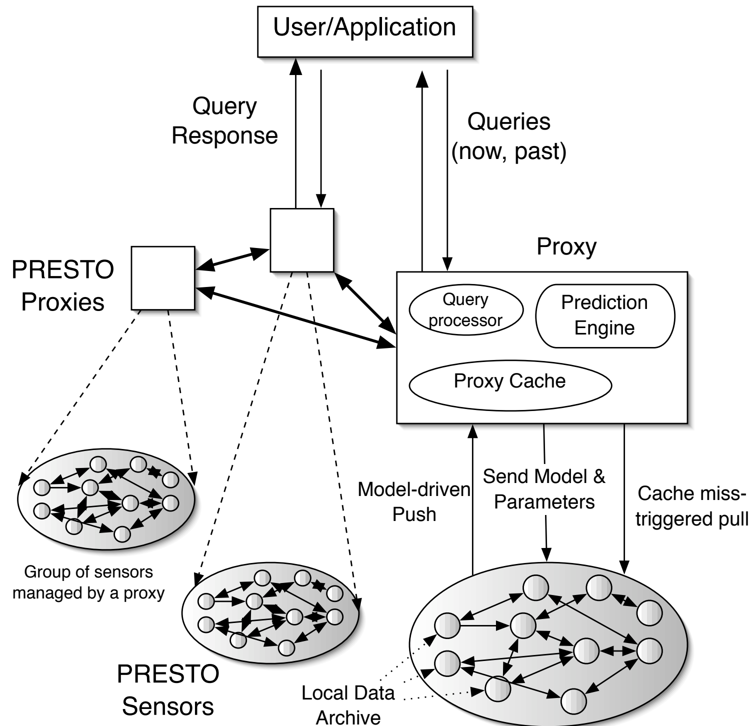


Figure 2.1. The PRESTO data management architecture.

communication is generally more expensive than processing or storage [10], PRESTO sensors attempt to trade communication for computation or storage, whenever possible.

System Operation: Assuming such an environment, each PRESTO proxy constructs a model of the data observed at each sensor. The model uses correlations in the past observations to predict the value likely to be seen at any future instant t. The model and its parameters are transmitted to each sensor. The sensor then executes the model as follows: at each sampling instant t, the actual sensed value is compared to the value predicted by the model. If the difference between the two exceed a threshold, the model is deemed to have "failed" to accurately predict that value and the sensed value is pushed to the proxy. In contrast, if the difference between the two is smaller than a threshold, then the model is assumed to be accurate for that time instant. In this case, the sensor archives the data locally in flash storage and does not transmit it to the proxy. Since the model is *also known to the proxy*, the proxy can compute the predicted value and use it as an approximation of the actual observation when answering queries. Thus, so long as the model accurately predicts observed values, no communication is necessary between the sensor and the proxy; the proxy continues to use the predictions to respond to queries. Further, any deviations from the model are always reported to the proxy and anomalous trends are quickly detected as a result.

Given such a model-driven push technique, a query arriving at the proxy is processed as follows. PRESTO assumes that each query specifies a tolerance on the error it is willing to accept. Our models are capable of generating a *confidence interval* for each predicted value. The PRESTO proxy compares the query error tolerance with the confidence intervals and uses the model predictions so long at the query error tolerance is not violated. If the query demands a higher precision, the proxy simply pulls the actual sensed values from the remote sensors and uses these values to process the query. Every prediction made by the model is cached at the proxy; the cache also contains all values that were either pushed or pulled from the remote sensors. This cached data is used to respond to historical queries so long as query precision is not violated, otherwise the corresponding data is pulled from the local archive at the sensors.

Since trends in sensed values may change over time, a model constructed using historical data may no longer reflect current trends. A novel aspect of PRESTO is that it updates the model parameters online so that the model can continue to reflect current observed trends. Upon receiving a certain number of updates from a sensor, the proxy uses these new values to refine the parameters of the model. These parameters are then conveyed back to the corresponding sensor, when then uses them to push subsequent values. Thus, our approach incorporates *active feedback* between the proxy and each sensor—the model parameters are used to determine which data values get pushed to the proxy, and the pushed values are used to compute the new parameters of the model. If the precision demanded by queries also changes over time, the thresholds used by sensors to determine which values should be pushed are also adapted accordingly—higher precision results in smaller thresholds. Next, we present the design of the PRESTO proxy and sensor in detail.

2.3 PRESTO Proxy

The PRESTO proxy consists of four key components (see Figure 2.2): (i) *modeling and prediction engine*, which is responsible for determining the initial model parameters, periodic refinement of model parameters, and prediction of data values likely to be seen at the various sensors, (ii) *query processor*, which handles queries on both current and historical data, (iii) *local cache*, which is a cache of all data pushed or pulled by sensors as well as all past values predicted by the model, and (iv) a *fault detector*, which detects sensor failures. We describe each component in detail in this section.

2.3.1 Modeling and Prediction Engine

The goal of the modeling and prediction engine is to determine a model, using a set of past sensor observations, to forecast future values. The key premise is that the physical phenomena observed by sensors exhibit long-term and short-term correlations and past values can be used to predict the future. This is true for weather phenomena such as temperature that exhibit long-term seasonal variations as well as short-term

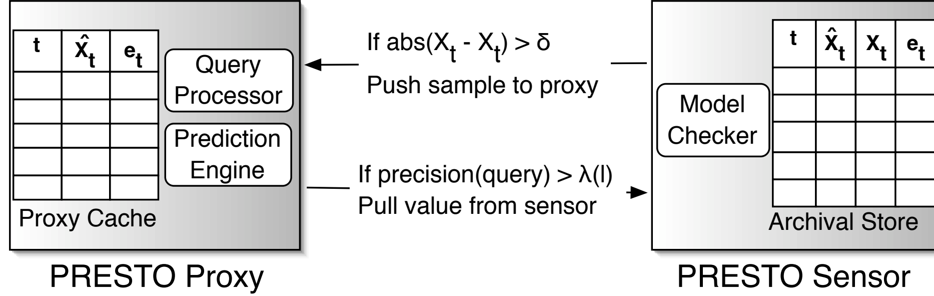


Figure 2.2. The PRESTO proxy comprises a prediction engine, query processor and a cache of predicted and real sensor values. The PRESTO sensor comprises a model checker and an archive of past samples with the model predictions.

time-of-day and hourly variations. Similarly phenomena such as traffic at an intersection exhibit correlations based on the hour of the day (e.g., traffic peaks during "rush" hours) and day of the week (e.g., there is less traffic on weekends). PRESTO proxies rely on *seasonal ARIMA* models; ARIMA is a popular family of time-series models that are commonly used for studying weather and stock market data. Seasonal ARIMA models (also known as SARIMA) are a class of ARIMA models that are suitable for data exhibiting seasonal trends and are well-suited for sensor data. Further they offer a way to deal with non-stationary data *i.e.* whose statistical properties change over time [1]. Last, as we demonstrate later, while seasonal ARIMA models are computationally expensive to construct, they are inexpensive to check at the remote sensors—an important property we seek from our system. The rest of this section presents the details of our SARIMA model and its use within PRESTO.

Prediction Model: A discrete time series can be represented by a set of time-ordered data $(x_{t_1}, x_{t_2}, ..., x_{t_n})$, resulting from observation of some temporal physical phenomenon such as temperature or humidity. Samples are assumed to be taken at discrete time instants $t_1, t_2, \ldots$. The goal of time-series analysis is to obtain the parameters of the underlying physical process that governs the observed time-series and use this model to forecast future values.

PRESTO models the time series of observations at a sensor as an *Autoregressive Integrated Moving Average (ARIMA)* process. In particular, the data is assumed to conform to the Box-Jenkins SARIMA model [1]. While a detailed discussion of SARIMA models is outside the scope of this thesis, we provide the intuition behind these models for the benefit of the reader. An SARIMA process has four components: *auto-regressive (AR)*, *moving-average (MA)*, *one-step differencing*, and *seasonal differencing*. The AR component estimates the current sample as a linear weighted sum of previous samples; the MA component captures relationship between prediction errors; the one-step differencing component captures relationship between adjacent samples; and the seasonal differencing component captures the diurnal, monthly, or yearly patterns in the data. In

SARIMA, the MA component is modeled as a zero-mean, uncorrelated Gaussian random variable (also referred to as white noise). The AR component captures the temporal correlation in the time series by modeling a future value as a function of a number of past values.

In its most general form, the Box-Jenkins seasonal model is said to have an order $(p, d, q) \times (P, D, Q)_S$; the order of the model captures the dependence of the predicted value on prior values. In SARIMA, p and q are the orders of the auto-regressive (AR) and moving average (MA) processes, P and Q are orders of the seasonal AR and MA components, d is the order of differencing, D is the order of seasonal differencing, and S is the seasonal period of the series. Thus, SARIMA is family of models depending on the integral values of p, q, P, Q, d, D, S. [2]

Model Identification and Parameter Estimation: Given the general SARIMA model, the proxy needs to determine the order of the model, including the order of differential and the order of auto-regression and moving average. That is, the values of p, d, q, P, D and Q need to be determined. This step is called model identification and is typically performed once during system initialization. Model identification is well documented in most time series textbooks [1] and we only provide a high level overview here. Intuitively, since the general model is actually a family of models, depending on the values of p, q, etc., this phase identifies a particular model from the family that best captures the variations exhibited by the underlying data. It is somewhat analogous to fitting a curve on a set of data values. Model identification involves collecting a sample time series from the field and computing its auto-correlation function (ACF) and partial auto-correlation function (PACF). A series of tests are then performed on the ACF and the PACF to determine the order of the model [1].

Our analysis of temperature traces has shown that the best model for temperature data is a Seasonal ARIMA of order $(0, 1, 1) \times (0, 1, 1)_S$. The general model in Equation 2.1 reduces to

$$(1 - B)(1 - B^S)X_t = (1 - \theta B)(1 - \Theta B^S)e_t \tag{2.2}$$

where θ and Θ are parameters of this $(0, 1, 1) \times (0, 1, 1)_S$ SARIMA model and capture the variations shown by different temperature traces. B is the backward operator and is short-hand for $B^i X_t = X_{t-i}$. S is the seasonal period of the time series and e_t is the prediction error.

---

[2] While not essential for our discussion, we present the general Box-Jenkins seasonal model for sake of completeness. The general model of order $(p, d, q) \times (P, D, Q)_S$ is given by the equation

$$\Phi_P(B^S) \cdot \phi_p(B) \cdot (1 - B)^d (1 - B^S)^D X_t = \theta_q(B)\Theta_Q(B^S)e_t \tag{2.1}$$

where B is the backward operator such that $B^i X_t = X_{t-i}$, S is the seasonal period, θ, Θ are parameters of the model, and e_t is the prediction error.

When employed for a temperature monitoring application, PRESTO proxies are seeded with a $(0, 1, 1) \times (0, 1, 1)_S$ SARIMA model. The seasonal period S is also seeded. The parameters θ and Θ are then computed by the proxy during the initial training phase before the system becomes operational. The training phase involves gathering a data set from each sensor and using the least squares method to estimate the values of parameters θ and Θ on a per-sensor basis (see [1] for the detailed procedure for estimating these parameters). The order of the model and the values of θ and Θ are then conveyed to each sensor. Section 2.5 explains how θ and Θ can be periodically refined to adapt to any long-term changes in the sensed data that occurs after the initial training phase.

Model-based Predictions: Once the model order and its parameters have been determined, using it for predicting future values is a simple task. The predicted value X_t for time t is simply given as:

$$\begin{aligned} X_t &= X_{t-1} + X_{t-S} - X_{t-S-1} \\ &+ \theta e_{t-1} - \Theta e_{t-S} + \theta\Theta e_{t-S-1} \end{aligned} \tag{2.3}$$

where θ and Θ are known parameters of the model, X_{t-1} denotes the previous observation, X_{t-S} and X_{t-S-1} denotes the values seen at this time instant and the previous time instant in the previous season. For temperature monitoring, we use a seasonal period S of one day, and hence, X_{t-S} and X_{t-S-1} represent the values seen *yesterday* at this time instant and the previous time instant, respectively. e_{t-k} denotes the prediction error at time $t-k$ (the prediction error is simply the difference between the predicted and observed value for that instant).

Since PRESTO sensors push a value to the proxy only when it deviates from the prediction by more than a threshold, the actual values of X_{t-1}, X_{t-S} and X_{t-S-1} seen at the sensor may not be known to the proxy. However, since the lack of a push indicates that the model predictions are accurate, the proxy can simply use the corresponding model predictions as an approximation for the actual values in Equation 2.3. In this case, the corresponding prediction error e_{t-k} is set to zero. In the event X_{t-1}, X_{t-S} or X_{t-S-1} were either pushed by the sensor or pulled by the proxy, the actual values and the actual prediction errors can used in Equation 2.3.

Both the proxy and the sensors use Equation 2.3 to predict each sampled value. At the proxy, the predictions serve as a substitute for the actual values seen by the sensor and are used to answer queries that might request the data. At the sensor, the prediction is used to determine whether to push—the sensed value is pushed only if the prediction error exceeds a threshold δ.

Finally, we note the *asymmetric* property of our model. The initial model identification and parameter estimation is a compute-intensive task performed by the proxy. Once determined, predicting a value using the model *involves no more than eight floating point operations* (three multiplications and five additions/subtractions, as shown in Equation 2.3). This is inexpensive even on resource-poor sensor nodes such as Motes and can be approximated using fixed point arithmetic.

2.3.2 Query Processing at a Proxy

In addition to forecasting future values, the prediction engine at the proxy also provides a confidence interval for each predicted value. The confidence interval represents a bound on the error in the predicted value and is crucial for query processing at the proxy. Since each query arrives with an error tolerance, the proxy compares the error tolerance of a query with the confidence interval of the predictions, and the current push threshold, δ. If the confidence interval is tighter than the error tolerance, then the predicted values are sufficiently accurate to respond to the query. Otherwise the actual value is fetched from the remote sensor to answer the query. Thus, many queries can be processed locally even if the requested data was never reported by the sensor. As a result, PRESTO can ensure low latencies for such queries without compromising their error tolerance. The processing of queries in this fashion is similar to that proposed in the BBQ data acquisition system [9], although there are significant differences in the techniques.

For a Seasonal ARIMA $(0,1,1) \times (0,1,1)_S$ model, the confidence interval of l step ahead forecast, $\lambda(l)$ is:

$$\lambda(l) = \pm u_{\varepsilon/2}(1 + \sum_{j=1}^{l-1}(1-\theta)^2)^{1/2}\sigma \qquad (2.4)$$

where $u_{\varepsilon/2}$ is value of the unit Normal distribution at $\varepsilon/2$, σ is the variance of 1 step ahead prediction error.

2.3.3 Proxy Cache

Each proxy maintains a cache of previously fetched or predicted data values for each sensor. Since storage is plentiful at the proxy—microdrives or hard-drives can be used to hold the cache—the cache is assumed to be infinite and all previously predicted or fetched values are assumed to be stored at the proxy. The cache is used to handle queries on historical data—if requested values have already been fetched or if the error bounds of cached predictions are smaller than the query error tolerance, then the query can be handled locally, otherwise the requested data is pulled from the archive at the sensor. After responding to the query, the newly fetched values are inserted into the cache for future use.

A newly fetched value, upon insertion, is also used to improve the accuracy of the neighboring predictions using interpolation. The intuition for using interpolation is as follows. Upon receiving a new value from the sensor, suppose that the proxy finds a certain prediction error. Then it is very likely that the predictions immediately preceding and following that value incurred a similar error, and interpolation can be used to scale those cached values by the prediction error, thereby improving their estimates. PRESTO proxies currently use two types of interpolation heuristics: forward and backward.

Forward interpolation is simple. The proxy uses Equation 2.3 to predict the values and Equation 2.4 to re-estimate the confidence intervals for all samples between the newly inserted value and the next pulled or pushed value. In backward interpolation, the proxy scans backwards from the newly inserted value and modifies all cached predictions between the newly inserted value and the previous pushed or pulled value. To do so, it makes a simplifying assumption that the prediction error grows linearly at each step, and the corresponding prediction error is subtracted from each prediction.

$$X'_t = X_t - \frac{t - T'}{T - T'} e_T \tag{2.5}$$

where X_t is the original prediction, X'_t is the updated prediction, T denotes the observation instant of the newly inserted value, T' is time of the nearest pushed or pulled value before T.

2.3.4 Failure Detection

Sensors are notoriously unreliable and can fail due hardware/software glitches, harsh deployment conditions or battery depletion. Our predictive techniques limit message exchange between a proxy and a sensor, thereby reducing communication overhead. However, reducing message frequency also affects the latency to detect sensor failures and to recover from them. In this work, we discuss mechanisms used by the PRESTO proxy to detect sensor failures. Failure recovery can use techniques such as spatial interpolation, which are outside the scope of this thesis.

The PRESTO proxy flags a failure if pulls or feedback messages are not acknowledged by a sensor. This use of implicit heartbeats has low communication energy overhead, but provides an interesting benefit. A pull is initiated by the proxy depending on the confidence bounds, which in turn depends on the variability observed in the sensor data. Consequently, failure detection latency will be lower for sensors that exhibit higher data variability (resulting in more pushes or pulls). For sensors that are queried infrequently or exhibit low data variability, the proxy relies on the less-frequent model feedback messages for implicit heartbeats; the lack of an acknowledgment signals a failure. Thus, proxy-initiated control or pull messages can be exploited for failure detection at no additional cost; the failure detection latency depends on the observed variability and confidence requirements of incoming queries. Explicit heartbeats can be employed for applications with more stringent needs.

2.4 PRESTO Sensor

PRESTO sensors perform three tasks: (i) use the model predictions to determine which observations to push, (ii) maintain a local archive of all observations, and (iii) respond to pull requests from the proxy.

The PRESTO sensor acts as a mirror for the prediction model at the proxy—both the proxy and the sensor execute the model in a completely identical fashion. Consequently, at each sampling instant, the sensor knows the exact estimate of the sampled value at the proxy and can determine whether the estimate is accurate. Only those samples that deviate significantly from the prediction are pushed. As explained earlier, the proxy transmits all the parameters of the model to each sensor during system initialization. In addition, the proxy also specifies a threshold δ that defines the worst-case deviation in the model prediction that the proxy can tolerate. Let X_t denote the actual observation at time t and let $\hat{X}_t$ denote the predicted value computed using Equation 2.3. Then,

$$\text{If } |\hat{X}_t - X_t| > \delta, \text{Push } X_t \text{ to Proxy.} \tag{2.6}$$

As indicated earlier, computation of $\hat{X}_t$ using Equation 2.3 involves reading of a few past values such as X_{t-S} from the archive in flash storage and a few floating point multiplications and additions, all of which are inexpensive.

PRESTO sensors archive all sensed values into an energy-efficient NAND flash store; the flash archive is a log of tuples of the form: $(t, X_t, \hat{X}_t, e_t)$. A simple index is maintained to permit random access to any entry in the log. A pull request from a proxy involves the use of this index to locate the requested data in the archive, followed by a read and a transmit.

2.5 Adaptation in PRESTO

PRESTO is designed to adapt to long-term changes in data and query dynamics that occur in any long-lived sensor application. To enable system operation at the most energy-efficient point, PRESTO employs active feedback from proxies to sensors; this feedback takes two forms—adaptation to data and query dynamics.

2.5.1 Adaptation to Data Dynamics

Since trends in sensor observation may change over time, a model constructed using historical data may no longer reflect current trends—the model parameters become stale and need to be updated to regain energy-efficiency. PRESTO proxies periodically retrain the model in order to refine its parameters. The retraining phase is similar to the initial training—all data since the previous retraining phase is gathered and the least squares method is used to recompute the model parameters θ and Θ [1]. The key difference between the initial training and the retraining lies in the data set used to compute model parameters.

For the initial training, an actual time series of sensor observations is used to compute model parameters. However, once the system is operational, sensors only report observations when they significantly deviate from the predicted values. Consequently, the proxy only has access to a small subset of the observations made at each sensor. Thus, the model must be retrained with *incomplete information*. The time series used during the retraining phase contains all values that were either pushed or pulled from a sensor; all missing values in the time series are substituted by the corresponding model predictions. Note that these prior predictions are readily available in the proxy cache; furthermore, they are guaranteed to be a good approximation of the actual observations (since these are precisely the values for which the sensor did not push the actual observations). This approximate time series is used to retrain the model and recompute the new parameters.

For the temperature monitoring application that we implemented, the models are retrained at the end of each day.[3] The new parameters θ and Θ are then pushed to each sensor for future predictions. In practice, the parameters need to be pushed only if they deviate from the previously computed parameters by a non-trivial amount (i.e., only if the model has actually changed).

2.5.2 Adaptation to Query Dynamics

Just as sensor data exhibits time-varying behavior, query patterns can also change over time. In particular, the query tolerance demanded by queries may change over time, resulting in more or fewer data pulls. The proxy can adapt the value of the threshold parameter δ in Equation 2.6 to directly influence the fraction of queries that trigger data pulls from remote sensors. If the threshold δ is large relative to the mean error tolerance of queries, then the number of pushes from the sensor is small and the number of pulls triggered by queries is larger. If δ is small relative to the query error tolerance, then there will be many wasteful pushes and fewer pulls (since the cached data is more precise than is necessary to answer the majority of queries). A careful selection of the threshold parameter δ allows a proxy to balance the number of pushes and the number of pulls for each sensor.

To handle such query dynamics, the PRESTO proxy uses a moving window average to track the mean error tolerance of queries posed on the sensor data. If the error tolerance changes by more than a pre-defined threshold, the proxy computes a new δ and transmits it to the sensor so that it can adapt to the new query pattern.

[3] Since the seasonal period is set to one day, this amounts to a retraining after each season.

2.6 PRESTO Implementation

We have implemented a prototype of PRESTO on a multi-tier sensor network testbed. The proxy tier employs Crossbow Stargate nodes with a 400MHz Intel XScale processor and 64MB RAM. The Stargate runs the Linux 2.4.19 kernel and EmStar release 2.1 and is equipped with two wireless radios, a Cisco Aironet 340-based 802.11b radio and a hostmote bridge to the Telos mote sensor nodes using the EmStar transceiver. The sensor tier uses Telos Mote sensor nodes, each consisting of a MSP430 processor, a 2.4GHz CC2420 radio, and 1MB external flash memory. The sensor nodes run TinyOS 1.1.14. Since sensor nodes may be several hops away from the nearest proxy, the sensor tier employs MultiHopLEPSM multi-hop routing protocol from the TinyOS distribution to communicate with the proxy tier.

Sensor Implementation: Our PRESTO implementation on the Telos Mote involves three major tasks: (i) model checking, (ii) flash archival, and (ii) data pull. A simple data gathering task periodically obtains sensor readings and sends the sample to the model checker. The model checking task uses the most recent model parameters (θ and Θ) and push delta (δ) obtained from the proxy to determine if a sample should be pushed to the proxy as per Equation 2.6. Each push message to the proxy contains the id of the mote, the sampled data, and a timestamp recording the time of the sampling. Upon a pull from the proxy, the model checking task performs the forward and backward updates to ensure consistency between the proxy and sensor view. For each sample, the archival task stores a record to the local flash that has three fields: (i) the timestamp when the data was sampled, (ii) the sample itself, and (iii) the predicted value from the model checker. The final component of our sensor implementation is a pull task that, upon receiving a pull request, reads the corresponding data from the flash using a temporal index-based search, and responds to the proxy.

Proxy Implementation: At the core of the proxy implementation is the prediction engine. The prediction engine includes a full implementation of ARIMA parameter estimation, prediction and update. The engine uses two components, a cache of real and predicted samples, and a protocol suite that enables interactions with each sensor. The proxy cache is a time-series stream of records, each of which includes a timestamp, the predicted sensor value, and the prediction error. The proxy uses one stream per node that it is responsible for, and models each node's data separately. The prediction engine communicates with each sensor using a protocol suite that enables it to provide feedback and change the operating parameters at each sensor.

Queries on our system are assumed to be posed at the appropriate proxy using either indexing [10] or routing [16] techniques. A query processing task at the proxy accepts queries from users, checks whether it can be answered by the prediction engine based on the local cache. If not, a pull message is sent to the corresponding sensor.

Our proxy implementation includes two enhancements to the hostmote transceiver that comes with the EmStar distribution [2]. First, we implemented a priority-based 64-length FIFO outgoing message queue

in the transceiver to buffer pull requests to the sensors. There are two priority levels — the higher priority corresponds to parameter feedback messages to the sensor nodes, and the lower priority corresponds to data pull messages. Prioritizing messages ensures that parameter messages are not dropped even if the queue is full as a result of excess pulls. Our second enhancement involves emulating the latency characteristics of a duty-cycling MAC layer. Many MAC-layer protocols have been proposed for sensor networks such as BMAC [21] and SMAC [28]. However, not all these MAC layers are supported on all platforms — for instance, neither BMAC nor SMAC is currently supported on the Telos Motes that we use. We address this issue by benchmarking the latency introduced by BMAC on Mica2 sensor nodes, and using these measurements to drive our experiments. Thus, the proxy implementation includes a MAC-layer emulator that adds duty-cycling latency corresponding to the chosen MAC duty-cycling parameters.

2.7 Experimental Evaluation

In this section, we evaluate the performance of PRESTO using our prototype and simulations. The testbed for our experiments comprises one Stargate proxy and twenty Telos Mote sensor nodes. One of the Telos motes is connected to a Stargate node running a sensor network emulator in Emstar[12]. This emulator enables us to introduce additional virtual sensor nodes in our large-scale experiments that share a single Telos mote radio as the transceiver to send and receive messages. In addition to the testbed, we use numerical simulations in Matlab to evaluate the performance of the data processing algorithms in PRESTO.

Our experiments involve both replays of previously gathered sensor data as well as a live deployment. The first set of experiments are trace-driven and use a seven day temperature dataset from James reserve [26]. The first two days of this trace are used to train the model. In our experiments, sensors use the values from the remainder of these traces—which are stored in flash memory—as a substitute for live data gathering. This setup ensures repeatable experiments and comparison of results across experiments (which were conducted over a period of several weeks). We also experiment with a live, four day outdoor deployment of PRESTO at UMass to demonstrate that our results are representative of the “real world”.

In order to evaluate the query processing performance of PRESTO, we generate queries as a Poisson arrival process. Each query requests the value of the temperature at a particular time that is picked in a uniform random manner from the start of the experiment to the current time. The confidence interval requested by the query is chosen from a normal distribution.

2.7.1 Microbenchmarks

Our first experiment involves a series of microbenchmarks of the energy consumption of communication, processing and storage to evaluate individual components of the PRESTO proxy and sensors. These

microbenchmarks are based on measurements of two sensor platforms — a Telos mote, and a Mica2 mote augmented with a NAND flash storage board fabricated at UMass. The board is attached to the Mica2 mote through the standard 51-pin connector, and provides a considerably more energy-efficient storage option than the AT45DB041B NOR flash that is loaded by default on the Mica2 mote [19]. The NAND flash board enables the PRESTO sensor to archive a large amount of historical data at extremely low energy cost.

Module	Component	Operation	Energy
NAND flash-enabled Mica2	NAND Flash	Read + Write + Erase 1 sample	21nJ
	ATmega128L Processor	1 Prediction	240nJ
	CC1000 Radio	Transmit 1 sample + Receive 1 ACK	20.3μJ
Telos Mote	ST M25P80 Flash	Read + Write + Erase 1 sample	2.14μJ
	MSP430 Processor	1 Prediction	27nJ
	CC2420 Radio	Transmit 1 sample + Receive 1 ACK	3.3μJ

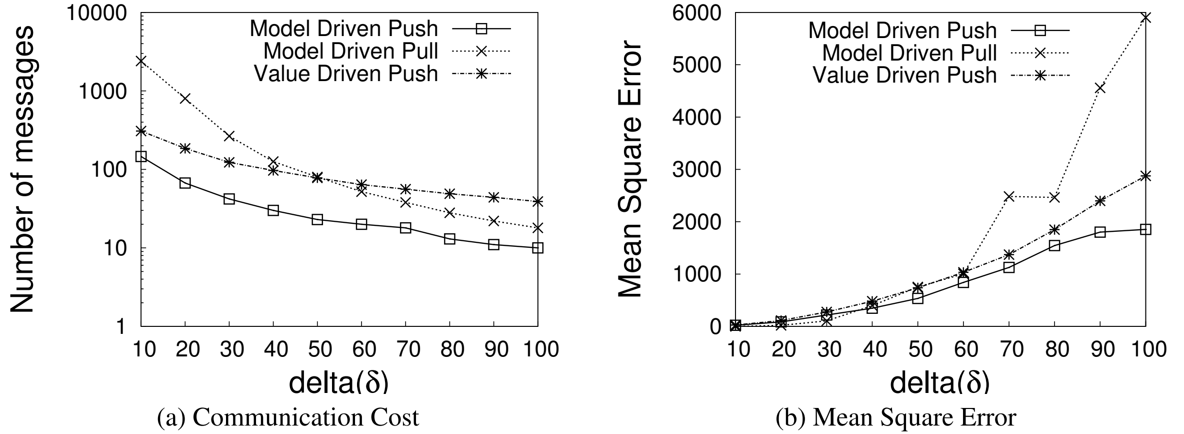
Table 2.1. Energy micro-benchmarks for sensor nodes.

	Round Trip Latency(ms)		
Routing Hops	1%	7.53%	35.5%
1-hop	2252	350	119
2-hop	4501	695	235
3-hop	6750	1040	347
4-hop	8999	1388	465
5-hop	11249	1733	580

Table 2.2. Round trip latencies using B-MAC

Energy Consumption: We measure the energy consumption of three components—computation per sample at the sensor, communication for a push or pull, and storage for reads, writes and erases. Table 2.1 shows that the results depend significantly on the choice of platform. On the Mica2 mote with external NAND flash, storage of a sample in flash is an order of magnitude more efficient than the ARIMA prediction computation, and three orders of magnitude more efficient than communicating a sample over the CC1000 radio. The Telos mote uses a more energy-efficient radio (CC2420) and processor (TI MSP 430), but a less efficient flash than the modified Mica2 mote. On the Telos mote, the prediction computation is the most energy-efficient operation, and is 80 times more efficient than storage, and 122 times more efficient than communication. The high cost of storage on the Telos mote makes it a bad fit for a storage-centric architecture such as PRESTO.

In order to fully exploit state-of-art in computation, communication and storage, a new platform is required that combines the best features of the two platforms that we have measured. This platform would use the TI MSP 430 microcontroller and CC2420 radio on the Telos mote together with NAND flash stor-



(a) Communication Cost (b) Mean Square Error

Figure 2.3. Comparison of PRESTO SARIMA models with model-driven pull and value-driven push.

age. Assuming that the component-level microbenchmarks in Table 2.1 hold for the new platform, storage and computation would be roughly equal cost, whereas communication would be two to three orders of magnitude more expensive than both storage and communication. We note that the energy requirements for communication in all the above benchmarks would be even greater if one were to include the overhead due to duty-cycling, packet headers and multi-hop routing. These comparisons validate our key premise that in future platforms, storage will offer a more energy-efficient option than communication and should be exploited to achieve energy-efficiency.

Component	Operation	Latency	Energy
Stargate (PXA255)	Model Estimation	21.75ms	11mJ
Telos Mote (MSP430)	Predict One Sample	18μs	27nJ

Table 2.3. Asymmetry: Model estimation vs Model checking

Communication Latency: Our second microbenchmark evaluates the latency of directly querying a sensor node. Sensor nodes are often highly duty-cycled to save energy, *i.e.* their radios are turned off to reduce energy use. However, as shown in Table 2.2, better duty-cycling corresponds to increased duration between successive wakeups and worse latency for the CC1000 radio on the Mica2 node. For typical sensor network duty-cycles of 1% or less, the latency is of the order of many seconds even under ideal 100% packet delivery conditions. Under greater packet-loss rates that are typical of wireless sensor networks [29], this latency would increase even further. We are unable to provide numbers for the CC2420 radio on the Telos mote since there is no available TinyOS implementation of an energy-efficient MAC layer with duty-cycling support for this radio.

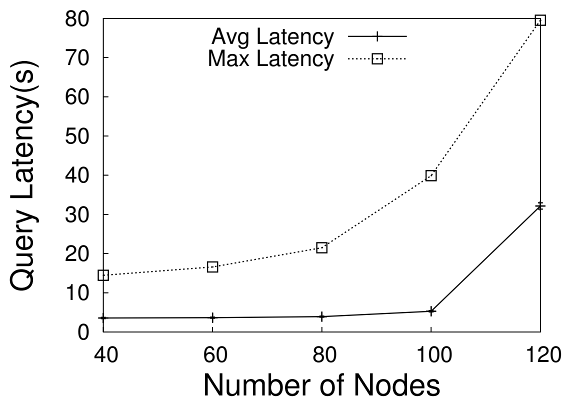


Figure 2.4. Scalability of PRESTO: Impact of network size.

Our measurements validate our claim that directly querying a sensor network incurs high latency, and this approach may be unsuitable for interactive querying. To reduce querying latency, the proxy should handle as many of the queries as possible.

Asymmetric Resource Usage: Table 2.3 demonstrates how PRESTO exploits computational resources at the proxy and the sensor. Determining the parameters of the ARIMA model at the proxy is feasible for a Stargate-class device, and requires only 21.75 ms per sensor. This operation would be very expensive, if not infeasible, on a Telos Mote due to resource limitations. In contrast, checking if the model is correct at the Mote consumes considerably less energy since it consists of only three floating point multiplications (approximated using fixed point arithmetic) and five additions/subtractions corresponding to Equation 2.3. This validates the design choice in PRESTO to separate model-building from model-checking and to exploit proxy resources for the former and resources at the sensor for the latter.

2.7.2 Performance of Model-Driven Push

In this section, we validate our claim that intelligently exploiting both proxy and sensor resources offers greater energy benefit than placing intelligence only at the proxy or only at the sensor. We compare the performance of model-driven push used in PRESTO against two other data-acquisition algorithms. The first algorithm, model-driven pull, is representative of the class of techniques where intelligence is placed solely at the proxy. This algorithm is motivated by the approach proposed in BBQ [9]. In this algorithm, the proxy uses a model of sensor data to predict future data and estimate the confidence interval in the prediction. If the confidence interval exceeds a pre-defined threshold (δ), the proxy will pull data from the sensor nodes, thus keeping the confidence interval bounded. The sensor node is simple in this case, and performs neither local storage nor model processing. While BBQ uses multi-variate Gaussians and dynamic Kalman Filters

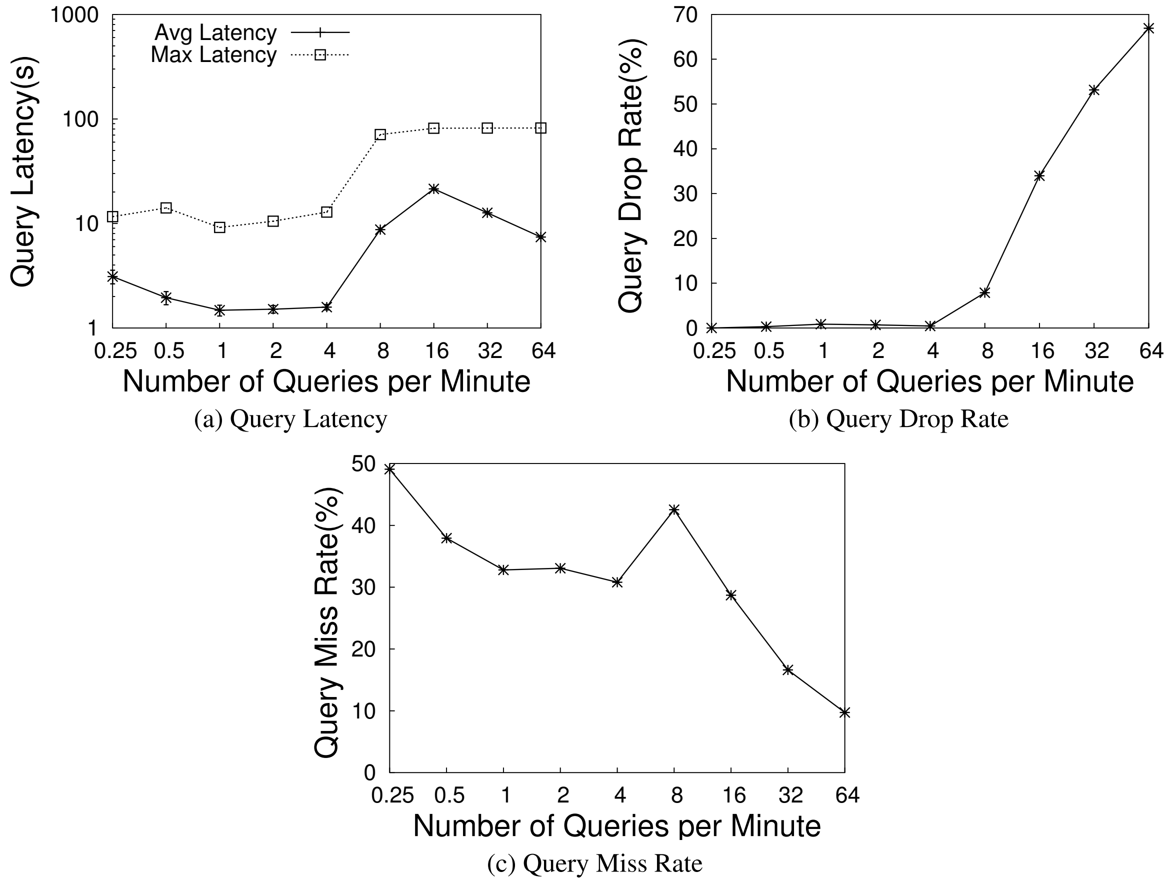


Figure 2.5. Scalability of PRESTO: Impact of query rates.

in their model-driven pull, our model-driven pull uses ARIMA predictions to ensure that the results capture the essential difference between the techniques and not the difference between the models used. The second algorithm that we compare against is a relatively naive value-driven push. Here, the sensor node pushes the data to the proxy when the difference between current data and last pushed data is larger than a threshold (δ). The proxy assumes that the sensor value does not change until the next push from the sensor. In general, a pull requires two messages, a request from the proxy to the sensor and a response, whereas push requires only a single message from the sensor to the proxy.

We compared the three techniques using Matlab simulations that use real data traces from James Reserve. Each experiment uses 5 days worth of data and each data point is the average of 10 runs. Figure 2.3 compares these three techniques in terms of the number of messages transmitted and mean-square error of predictions. In communication cost, PRESTO out-performs both the other schemes irrespective of the choice of δ. When δ is 100, the communication cost of PRESTO is half that of model-driven pull, and 25% that of value-driven push. At the same time, the mean square error in PRESTO is 30% that of model-driven pull, and 60% that

of value driven push. As δ decreases, the communication cost increases for all three algorithms. However, the increase in communication cost for model-driven pull is higher than that for the other two algorithms. When δ is 50, value driven push begins to out perform model-driven pull. When δ reaches 10, the number of messages in model-driven pull is 20 times more than that of PRESTO, and 8 times more than that of value driven push. This is because in the case of model-driven pull, the proxy pulls samples from the sensor whenever the prediction error exceeds δ. However, since the prediction error is often an overestimate and since each pull is twice as expensive as a push, this results in a larger number of pull messages compared to PRESTO and value-driven push. The accuracies of the three algorithms become close to each other when δ decreases. When δ is smaller than 40, model-driven pull has slightly lower mean square error than PRESTO but incurs 4 times the number of messages.

2.7.3 PRESTO Scalability

Scalability is an important criteria for sensor algorithm design. In this section, we evaluate scalability along two axes — network size and the number of queries posed on a sensor network. Network size can vary depending on the application (e.g: the Extreme Scaling deployment [11] used 10,000 nodes, whereas the Great Duck Island deployment [18] used 100 nodes). The querying rate depends on the popularity of sensor data, for instance, during an event such as an earthquake, seismic sensors might be heavily queried while under normal circumstances, the query load can be expected to be light.

The testbed used in the scalability experiments comprises one Stargate proxy, twenty Telos mote sensor nodes, and an EmStar emulator that enables us to introduce additional virtual sensor nodes and perform larger scale experiments. Messages are exchanged between each sensor and the proxy through a multihop routing tree rooted at the proxy. Each sensor node is assumed to be operating at 1% duty-cycling. Since MAC layers that have been developed for the Telos mote do not currently support duty-cycling, we emulate a duty-cycling enabled MAC-layer. This emulator adds appropriate duty-cycling latency to each packet based on the microbenchmarks that we presented in Table 2.2.

2.7.3.1 Impact of Network Size

A good data management architecture should achieve energy-efficiency and low-latency performance even in large scale networks. Our first set of scalability experiments test PRESTO at different system scales on five days of data collected from the James Reserve deployment. Queries arrive at the proxy as as a Poisson process at the rate of one query/minute per sensor. The confidence interval of queries is chosen from a normal distribution, whose expectation is equal to the push threshold, $\delta = 100$.

Figure 2.4 shows the query latency and query drop rate at system sizes ranging from 40 to 120. For system sizes of less than 100, the average latency is always below five seconds and has little variation. When the system size reaches 120, the average latency increases five-fold to 30 seconds. This is because the radio transceiver on the proxy gets congested and the queue overflows.

The effect of duty-cycling on latency is seen in Figure 2.4, which shows that the maximum latency increases with system scale. The maximum latency corresponds to the worst case of PRESTO when a sequence of query misses occur and result in pulls from sensors. This results in queuing of queries at the proxy, and hence greater latency. An in-network querying mechanism such as Directed Diffusion [15] that forwards every query into the network would incur even greater latency than the worst case in PRESTO since every query would result in a pull. These experiments demonstrate the benefits of model-driven pushes for user queries. By the use of caching and models, PRESTO results in low average-case latency by providing quick responses at the proxy for a majority of queries. We note that the use of a tiered architecture makes it easy to expand system scale to many hundreds of nodes by adding more PRESTO proxies.

2.7.3.2 Impact of Query Rate

Our second scalability experiment stresses the query handling ability of PRESTO. We test PRESTO in a network comprising one Stargate proxy and twenty Telos mote sensor nodes under different query rates ranging from one query every four minutes to 64 queries/minute for each sensor. Each experiment is averaged over one hour. We measure scalability using three metrics: the query latency, query miss rate, and query drop rate. A query miss corresponds to the case when it cannot be answered at the proxy and results in a pull, and a query drop results from an overflow at the proxy queue.

Figure 2.5 shows the result of the interplay between model accuracy, network congestion, and queuing at the proxy. To better understand this interplay, we analyze the graphs in three parts, *i.e.*, 0.25-4 queries/minute, 4-16 queries/minute and beyond 16 queries/minute.

Region 1: Between 0.25 and 4 queries/minute, the query rate is low, and neither queuing at the proxy nor network congestion is a bottleneck. As the query rate increases, greater number of queries are posed on the system and result in a few more pulls from the sensors. As a consequence, the accuracy of the model at the proxy improves to the point where it is able to answer most queries. This results in a reduction in the average latency. This behavior is also reflected in Figure 2.5(c), where the query miss rate reduces as the rate of queries grows.

Region 2: Between 4 and 16 queries/minute, the query rate is higher than the rate at which queries can be transmitted into the network. The queue at the proxy starts building, thereby increasing latency for query responses. This results in a sharp increase in average latency and maximum latency, as shown in Figure 2.5(a).

This increase is also accompanied by an increase in query drop rate beyond eight queries/minute, as more queries are dropped due to queue overflow. We estimate that eight queries/minute is the breakdown threshold for our system for the parameters chosen.

Region 3: Beyond sixteen queries/minute, the system drops a significant fraction of queries due to queue overflow as shown in Figure 2.5(b). Strangely, for the queries that do not get dropped, both the average latency (Figure 2.5(a)), and the query miss rate (Figure 2.5(c)) drop! This is because with each pull, the model precision improves and it is able to answer a greater fraction of the queries accurately.

The performance of PRESTO under high query rate demonstrates one of its key benefits — the ability to use the model to alleviate network congestion and queuing delays. This feature is particularly important since sensor networks can only sustain a much lower query rate than tethered systems due to limited wireless bandwidth.

2.7.4 PRESTO Adaptation

Having demonstrated the scalability and energy efficiency of PRESTO, we next evaluate its adaptation to query and data dynamics. In general, adaptation only changes what the sensor does for future data and not for past data. Our experiments evaluate adaptation for queries that request data from the recent past (one hour).

In our first experiment, we run PRESTO for 12 hours. Every two hours, we vary the mean of the distribution of query precision requirements thereby varying the query error tolerance. The proxy tracks the mean of the query distribution and notifies the sensor if the mean changes by more than a pre-defined threshold, in our case, 10. Figure 2.6(a) shows the adaptation to the query distribution changes. Explicit feedback from the proxy to each sensor enables the system to vary the δ corresponding to the changes in query precision requirements. From the figure, we can see that there is a spike in average query latency and the energy cost every time the query confidence requirements become tighter. This results in greater query miss rate and hence more pulls as shown in Figure 2.6(a). However, after a short period, the proxy provides feedback to the sensor to change the pushing threshold, which decreases the query miss rate and consequently, the average latency. The opposite effect is seen when the query precision requirements reduce, such as at the 360 minute mark in Figure 2.6(a). As can be seen, the query miss rate reduces dramatically since the model at the proxy is too precise. After a while, the proxy provides feedback to the sensors to increase the push threshold and to lower the push rate. A few queries result in pulls as a consequence, but the overall energy requirements of the system remains low. In comparison with a non-adaptive version of PRESTO that kept a fixed δ, our adaptive version reduces latency by more than 50%.

In our second experiment, we demonstrate the benefits of adaptation to data dynamics. PRESTO adapts to data dynamics by model retraining, as described in Section 2.5. We use a four day dataset, and at the end of

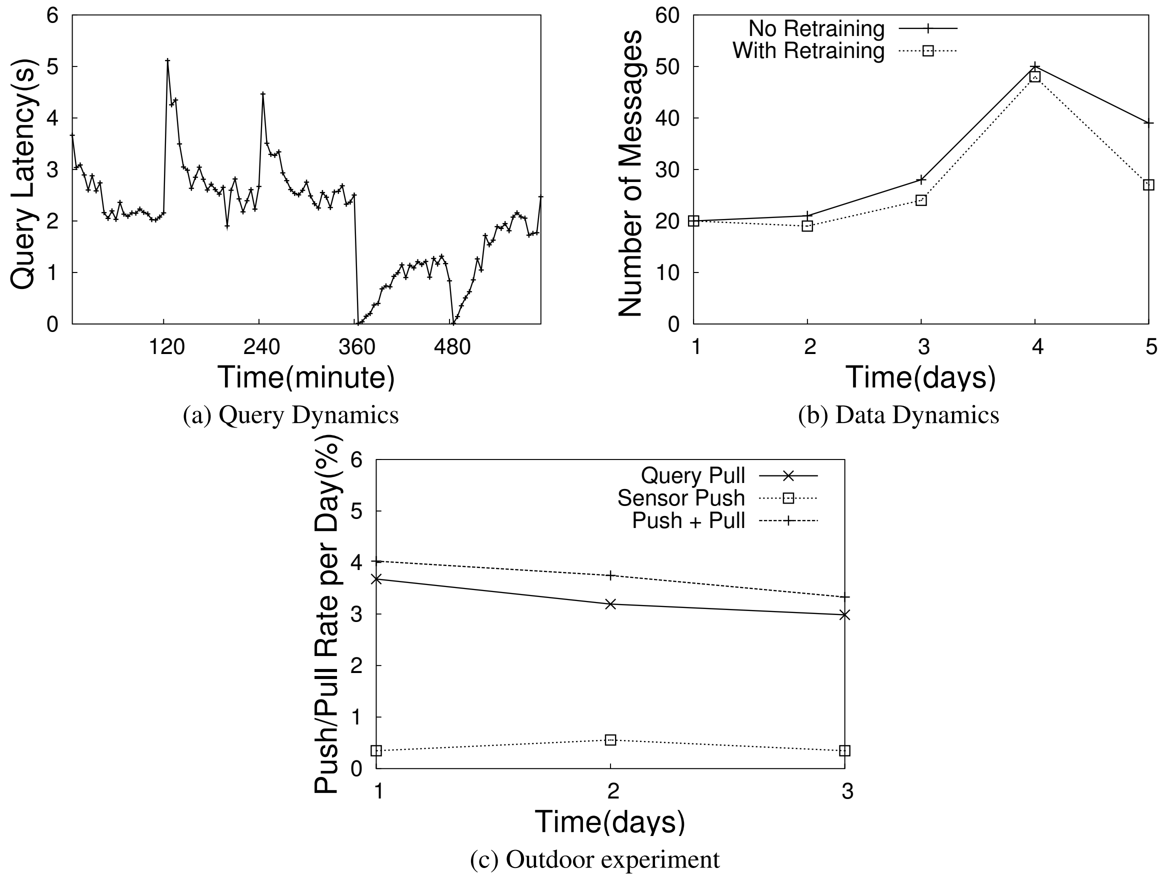


Figure 2.6. Adaptation in PRESTO to data and query dynamics as well as adaptation in an outdoor deployment.

each day, the proxy retrains the model based on the pushes from the sensor for the previous day, and provides feedback of the new model parameters to the sensor. Our result is shown in Figure 2.6(b). For instance, on day three, the data pattern changes considerably and the communication cost increases since the model does not follow the old patterns. However, at the end of the third day, the PRESTO proxy retrains the model and send the new parameters to the sensors. As a result, the model accuracy improves on the second day and reduces communication. The figure also shows that the model retraining reduces pushes by as much as 30% as compared to no retraining.

While most of our experiments involved the use of temperature traces as a substitute of live temperature sampling, we conducted a number of experiments with a live outdoor deployment of PRESTO using one proxy and four sensors. These experiments corroborate our findings from the trace-driven testbed experiments. The result of one such experiment is shown in Figure 2.6(c). The figure shows that, over a period of three days, as the model adapts via retraining, the frequency of pulls as well as the total frequency of pushes and pulls falls.

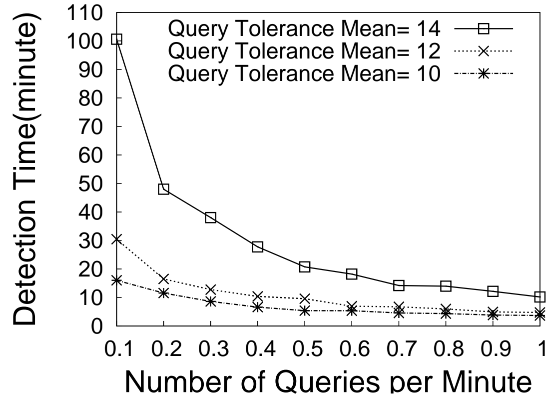


Figure 2.7. Evaluation of failure detection

2.7.5 Failure Detection

Detecting sensor failure is critical in PRESTO since the absence of pushes is assumed to indicate an accurate model. Thus, failures are detected only when the proxy sends a pull request or a feedback message to the sensor, and obtains no response or acknowledgment.

Figure 2.7 shows the detection latency using implicit heartbeats and random node failures. The detection latency depends on the query rate, the model precision and the precision requirements of queries. The dependence on query rate is straightforward—an increased query rate increases the number of queries triggering a pull and reduces failure detection latency. The relationship between failure detection and the model accuracy is more subtle. Model accuracy depends on two factors—the time since the last push from the sensor, and model uncertainty that captures inaccuracies in the model. As the time period between pushes grows longer, the model can only provide progressively looser confidence bounds to queries. In addition, for highly dynamic data, model precision degrades more rapidly over time triggering a pull sooner. Hence, even queries with low precision needs may trigger a pull from the sensor. The failure detection time also reduces with increase in precision requirements of queries. For instance, for a query rate of 0.1 queries/minute, the detection latency increases from 15 minutes when queries require high precision to 100 minutes when the queries only require loose confidence bounds.

The worst-case time taken for failure detection is one day since this is the frequency with which a feedback message is transmitted from the proxy to each sensor. However, this worst-case detection time occurs only if a sensor is very rarely queried.

2.8 Related Work

In this section, we review prior work on distributed sensor data management and time-series prediction.

Sensor data management has received considerable attention in recent years. As we described in Section 2.1, approaches include in-network querying techniques such as Directed Diffusion [15] and Cougar [27], stream-based querying in TinyDB [17], acquisitional query processing in BBQ [9], and distributed indexing techniques such as DCS [24]. Our work differs from all these in that we intelligently split the complexity of data management between the sensor and proxy, thereby achieving longer lifetime together with low-latency query responses.

The problem of sensor data archival has also been considered in prior work. ELF [3] is a log-structured file system for local storage on flash memory that provides load leveling and Matchbox is a simple file system that is packaged with the TinyOS distribution [14]. Our prior work, TSAR [10] addressed the problem of constructing a two-tier hierarchical storage architecture. Any of these techniques can be employed as the archival framework for the techniques that we propose in this thesis.

There has been much work on model-driven data management for sensor networks. We broadly classify them into model-driven push, model-driven pull, and fully distributed approaches. PAQ[4] and asynchronous in-network prediction[5] are model-driven push approaches where sensors train prediction models and push trained models to proxies for prediction. Instead PRESTO trains models on the resource-rich proxy side, therefore can use more sophisticate models. Ken [6] is another model-driven push approach that uses replicated dynamic probabilistic models. Our work differs in that we use the ARIMA models that impose much lower computation load on the sensor side compared to the dynamic probabilistic models. BBQ [9] is a model-driven pull approach that uses multi-variate Gaussian models to address spatial correlations, and dynamic Kalman filters to address temporal correlations. BBQ puts all intelligence on the proxy side, therefore can not efficiently capture sudden change in data trend. PRIDE[7] and [8] are fully-distributed approaches where model training, propagation, and prediction are done in completely distributed manor among sensors. There is no centralized proxy control. Any sensor can answer any queries using model prediction techniques.PRIDE replicates trained models to the whole network, while [8] replicates models along a tree.

2.9 Conclusions

In this chapter we described PRESTO, a model-driven predictive data management architecture for hierarchical sensor networks. In contrast to existing techniques, our work makes intelligent use of proxy and sensor resources to balance the needs for low-latency, interactive querying from users with the energy optimization needs of the resource-constrained sensors. A novel aspect of our work is the extensive use of an asymmetric prediction technique, Seasonal ARIMA [1], that uses proxy resources for complex model parameter estimation, but requires only limited resources at the sensor for model checking.

Our experiments showed that PRESTO yields an order of magnitude improvement in the energy required for data and query management, simultaneously building a more accurate model than other existing techniques. Also, PRESTO keeps the query latency within 3-5 seconds, even at high query rates, by intelligently exploiting the use of anticipatory pushes from sensors to build models, and explicit pulls from sensors. Finally, PRESTO adapts to changing query and data requirements by modeling query and data parameters, and providing periodic feedback to sensors.

CHAPTER 3

UTILITY-DRIVEN MULTI-USER DATA SHARING

3.1 Introduction

In the previous chapter, we address the data reduction problem when users request for raw data. Now, we consider the more advanced multi-user sensing applications where users request for meta-data after processing and a single sensor network has to support diverse user demands. For instance, in a radar sensor network shown in Figure 3.1, scientists may desire access to raw data to conduct research, while meteorological applications may require data that has undergone intermediate processing, and end-users may only need the "final processed result". Further, a tornado detection application will require timely notifications of important events, while other users are less sensitive to delay in sensor updates (e.g., end-users are tolerant to slight delays in weather updates).

How to support these diverse user needs in the presence of resource constraints poses a unique design challenge. A simple way is to handle the different user needs separately, but this model ignores one of the most important characteristics of multi-user sensor networks — all the users of a sensor network operate on the *same* data streams and the data relevant to one user can potentially be used to handle the needs of other users. Thus, rather than separately handling user needs, an approach that jointly considers user needs to maximize data sharing among users is better suited to make judicious use of the limited computational and network resources. Since the workload seen by such networks can dynamically vary over time as user needs and interests change—for instance, the workload imposed by users can increase significantly during an intense storm or a major traffic problem—such data sharing techniques must also adapt to dynamic load conditions.

3.1.1 Research Contributions

In this chapter, we describe a novel utility-driven architecture that maximizes data sharing among diverse users in a sensor network. We believe that maximizing utility across diverse end-user queries using *multi-user data sharing* techniques (henceforth referred to as MUDS) is a key challenge for designing more scalable sensor networks. Our architecture is designed for hierarchical sensor networks where sensors are streaming data over a multi-hop wireless network to a sensor proxy. These incoming data streams at the proxy are

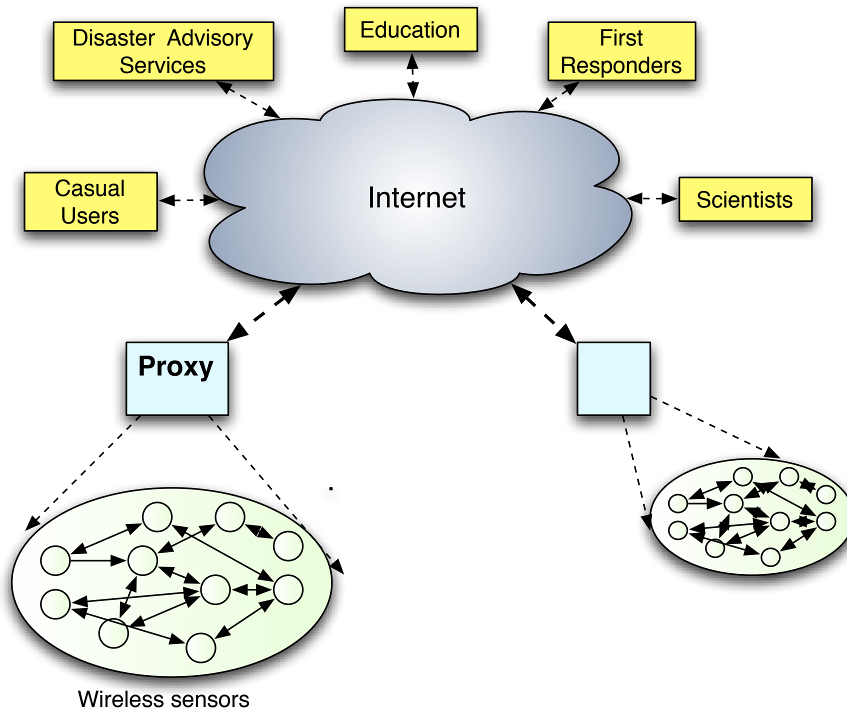


Figure 3.1. Multi-user Radar Sensor Networks

used to answer queries from different users. The proxy and the sensors interact continually to maximize data sharing across queries while simultaneously adapting to bandwidth variations, and changing query needs of users. We instantiate this architecture in the context of ad-hoc networks of wireless radar sensors for severe weather prediction and monitoring. Our work has three main contributions:

- **Multi-query Aggregation:** A key contribution of our work is multi-query aggregation, where radar data streams are shared between multiple and diverse end-user queries, thereby maximizing total end-user utility. We demonstrate that different end-user application needs, spatial areas of interest, deadlines, and priorities, can be combined into a single aggregated query, thereby enabling more optimized use of bandwidth resources.

- **Utility-driven Compression and Scheduling:** At the core of our system is a utility-driven progressive data compression and packet scheduling engine at each radar. The progressive compression engine enables radar data to be compressed and ordered such that information of most interest to queries is transmitted first. Such an encoding enables our system to adapt gracefully to bandwidth fluctuations.The utility-driven scheduler compares the utility of different progressively compressed streams that are in-

tended for different sets of queries, and transmits packets such that utility across all concurrent queries at a radar is maximized.

- **Global Transmission Control:** In addition to local utility-driven techniques, our system supports global utility optimization mechanisms driven by the proxy. The proxy continually monitors the utility of incoming data from different radars and decides how to control streams to maximize total utility across the entire network. Such a global control mechanism enables the system to adapt to uneven query distribution across the network, and to deal with disparities in available bandwidth among different radars due to wireless contention. This is especially important when some nodes in the network are observing important events such as tornadoes, and need to obtain more bandwidth than other nodes that are transmitting data for less critical queries.

In our experiments, we measure, evaluate and demonstrate the performance of our architecture and algorithms for radar sensor networks for severe weather monitoring. We have implemented the system on a testbed of Linux machines that form an 802.11-based wireless mesh network. Using a combination of simulations and experiments with real and emulated radar traces, we show that our system provides more than an order of magnitude (11x) improvement in query accuracy and utility for a 12 node network, when compared to an existing utility-agnostic non-progressive approach. Our system also degrades gracefully with network size — when the network size increases from one nodes to twelve nodes, the average utility achieved by each radar in our system only decreases by 25%, whereas the average utility of the existing *NetRad*[57] approach decreases by 80%. Further, our system adapts better to bandwidth variations with only 15% reduction in utility when the bandwidth drops from 150kbps to 10kbps.

The rest of this chapter is structured as follows. Section 3.2 provides an overview of radar sensor networks and the challenges in these networks. Section 3.3 provides an overview of our architecture, while Section 3.4 describes the design of the key components of our architecture. Sections 3.5 describes our implementation and evaluation. Finally, Sections 3.6 and 3.7 discuss related work and our conclusions.

3.2 Radar Sensor Networks

In this section, we provide an overview of the diverse end-user applications that use a radar sensor network, followed by the formulation of the problem addressed in this thesis.

3.2.1 End User Applications

A network of weather sensing radar sensors can be used by diverse users such as automated weather monitoring applications, meteorologists, scientists, teachers and emergency personnel. Several different weather

monitoring applications may be in use, each of which continuously requests and processes data sensed by various radars:

- *Hazardous weather detection:* Applications in this class are responsible for detecting hazardous weather such as storm cells, tornadoes, hail, and severe winds in real-time (e.g. [39]). This class of applications focuses on sharp changes in weather patterns; a tornado detection application, for instance, looks for sharp changes in wind speed and direction that are indicative of a tornado.

- *3D wind direction estimation:* This application constructs a 3D map by computing the direction of the wind at each point in 3D space. Since a single radar can only determine wind direction in a single dimension (radial axis), the application needs to merge data from two or more overlapping radars in order to estimate the 3D wind direction. Due to the need to merge data, only regions of overlap between adjacent radars are useful, and data from other areas need not be transmitted.

- *3D assimilation:* This application integrates data from multiple radars into a single 3D view to depict areas of high reflectivity (intense rain) that occur in the region.

We note that the first application is of interest to meteorologists for real-time weather forecasting, the second is useful to researchers, while the third is useful to emergency managers to visualize weather in their jurisdiction. In addition to these applications, end-users may pose other ad-hoc queries for data or instantiate continual queries that continuously request and process data to detect certain events or conditions.

3.2.2 System Model and Problem Formulation

Our MUDS radar sensing network comprises three tiers as shown in Figure 3.2 (i) applications and end-users who pose queries and request field data, (ii) sensor proxies that act as the gateway between the Internet and the radar sensor field, execute user queries, and manage the radar sensor network, and (iii) a wireless network of remote radar sensors that implement utility-driven services and stream their data to the proxy.

Each radar node comprises a mechanically steerable radar attached to an embedded PC controller; the embedded PC with dual-core Intel processor that runs Linux is equipped with 1GB RAM and a 802.11 wireless interface. A typical deployment will comprise many tens of radars distributed over a wide geographic area. The radars are "small" and are designed to be deployed in areas with no infrastructure using solar-powered rechargeable batteries; they can also be deployed on cellphone towers or on building rooftops where infrastructure such as A/C power is readily available. In either case, we assume that the radars connect to the proxy node using a multi-hop 802.11 wireless mesh network.

Each mechanically steerable radar has two degrees of freedom (θ, ϕ) which enable control over the *orientation* and the *altitude* where the radar points and senses data. The radar scans the atmosphere by first

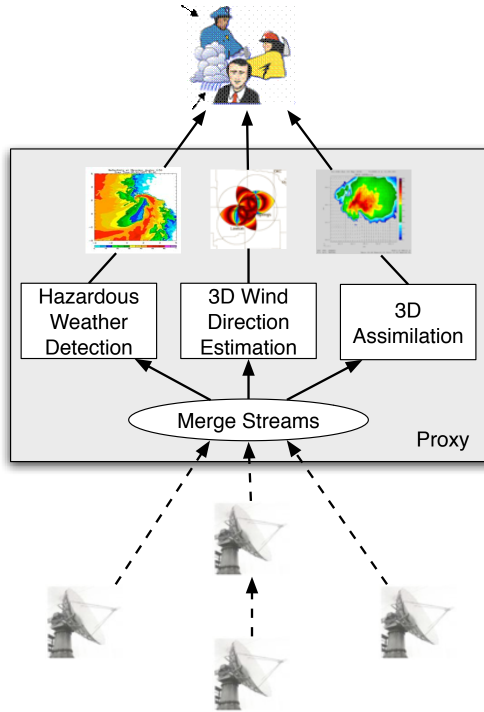


Figure 3.2. Multi-hop Radar Sensor Networks

positioning itself to point at an altitude ϕ and then conducts a *scan* by rotating θ degrees and scanning while rotating. The MUDS system operates in rounds, where each round is referred to as an *epoch*. For this thesis, we assume an epoch of 30 seconds. Before each epoch begins, the proxy collects all queries for a particular radar. Each query represents a request for data from a weather monitoring application (e.g., the tornado detection) or from end-users (who may issue ad-hoc queries). A query can request *any subset* of the region covered by a radar scan—for instance, a tornado detection algorithm may only request data from regions where intense weather has been detected. Each query has a priority and a deadline associated with it, which is then used to to assign a *weight* to each region that the radar can scan. The weight represents the relative importance of scanning and transmitting data from the region in the next epoch.[1] Thus, region-specific weights represent collective needs of all queries that have requested data in that epoch. Although the radar follows the 30-seconds sensing epoch, the deadline of queries is determined by end-user and can be of arbitrary lengths. A recent work[58] in radar sensor networks studies requirements from end-users like meteorologists, first responders etc., and shows the deadlines for different queries. We use this study as a baseline for setting deadlines for different queries in this thesis.

[1]For instance, a region that is not requested by any query will receive a weight of zero and need not be scanned by the radar.

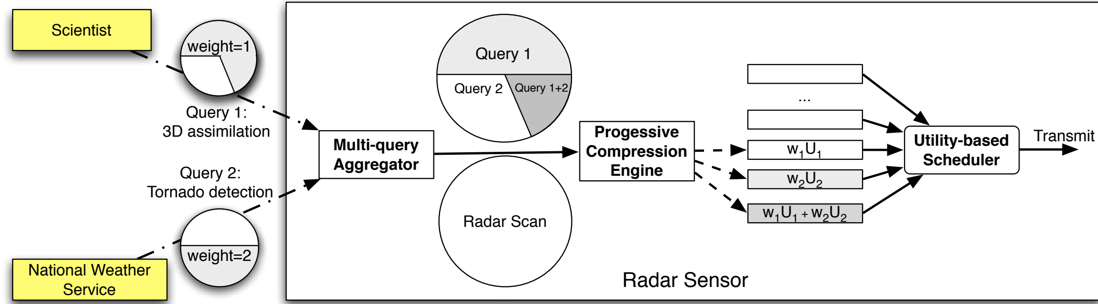


Figure 3.3. Multiple incoming queries in an epoch are first aggregated by the multi-query aggregator at the radar. The merged query and the radar scan for the epoch are input to the progressive encoder which generates different compressed streams for different regions in the query. The streams are input to the utility-driven scheduler which schedules packets across all streams whose deadlines have not yet expired.

Assuming the weights are computed before each epoch begins, the radar then scans all regions with non-zero weights during the epoch. Each scan is assumed to produce tens of Megabytes of raw data, which is typically much higher than bandwidth available to each radar in a multi-hop 802.11 mesh network. Thus, the primary constraint at a radar node is bandwidth, and the radar node must determine how to intelligently transmit the results of the scan back to the proxy.

Each proxy is assumed to be a server (or a server cluster) with significant processing and memory resources. The weather monitoring applications described in Section 3.2.1 are assumed to execute at the proxy, processing data streams from various radars in real-time. Each application is assumed to process data from an epoch and issues per-radar queries for data that it needs in the next epoch.

Assuming such a system, this thesis addresses the following questions:

- How can the radar sensor system merge and jointly handle queries with diverse high-level needs such as tornado detection, 3D wind direction estimation and 3D assimilation?

- Since the raw data from a scan exceeds the available network bandwidth and this bandwidth can vary significantly over time, how should a radar node intelligently compress the raw data prior to transmission?

- How should the radar prioritize the transmission of this compressed scan result back to the proxy node so that application overall utility is maximized?

- Since the query load on different radars can be uneven and data from some radars may be more critical than others during intense storms, how should the proxy globally control transmissions across radars to ensure that important data gets priority?

The following section discusses techniques employed by the MUDS system to address these questions. For simplicity of exposition and because optimizing the radar scan strategy is not the goal of our work, we assume each radar points at a fixed altitude ϕ and performs a 360° scan of the atmosphere resulting in a full 2D scan. It is straightforward to extend the discussion to three dimensional partial scans where both the altitude ϕ and the rotation θ are varied in a scan. Also, since our focus is on multi-user data sharing in a wireless environment, we do not focus on the design issues of long range wireless mesh networks, and assume that existing techniques such as [34, 88] can be used.

3.3 MUDS System Architecture

The proxy and sensor in the MUDS system interact continually to maximize utility under query and bandwidth dynamics. This interaction has four major parts: (a) a multi-query aggregation phase at the proxy and radar to compute a single unified query per epoch, (b) progressive compression of the radar scan at each radar by using the unified query as input, (c) a utility-driven scheduling phase at each radar where packets are prioritized by overall utility gain, and (d) a global transmission control phase driven by the proxy to optimize transmissions from different radars.

Multi-query aggregation: The first phase of our system operation is the multi-query aggregation phase where multiple user queries in an epoch are combined to generate a single unified query. This is done both by the proxy as well as the radars — the proxy uses the unified query for global transmission control, and the radar uses it for progressive compression and scheduling. Each user query is associated with a weight, a spatial region of interest, and a deadline. The weight of a query is dependent on the priority of the user (e.g. the National Weather Service is a high priority user), and the priority of the query to the user (e.g. a tornado detection query has higher priority during times of severe weather). Each query is also associated with a spatial area of interest, for instance, the wind direction estimation query is only meaningful for overlapping regions between radars. Queries are executed in batches — queries that arrive within a single epoch are merged to generate a joint spatial query map that captures the needs of all concurrent queries. An example of the spatial map that merges a tornado detection and a 3D assimilation query is shown in Figure 3.3. The merging of queries results in their weights being accumulated for shared regions of interest. The set of queries in an epoch is communicated by the proxy to the individual radar sensors whenever there is a change due to the arrival of new queries.

Progressive compression: Each radar scan produces tens of Megabytes of raw data that must then be transmitted back to the proxy node. Since the raw data rate is significantly higher than the bandwidth available per radar on the mesh, the data rate must somehow be reduced prior to transmission. The existing *NetRad*[57]

system employs a simple averaging technique to down-sample data—neighboring readings are averaged and replaced by this mean; the larger the number of neighboring readings over which the mean is computed, the greater the reduction in data rate. Rather than using a naive averaging technique, our system relies on the query map to intelligently reduce the data rate using a *progressive compression* technique. The progressive compression engine uses the unified query map and compresses data in two steps. First, the weights of different regions in the map are used to split the radar scan into multiple smaller regions, such that each region has a fixed weight and a fixed set of associated queries. Thus, the radar scan in Figure 3.3 is split into three regions with weights 1, 2, and 3 respectively. Each of these regions is then progressively encoded using a wavelet-based progressive encoder. The encoder compresses and orders data in each region such that most important features in the data is transmitted first, and less important features are transmitted later. Finally, the progressively encoded streams corresponding to different regions are input to a utility-based scheduler at the radar.

Utility-driven packet scheduling: The utility-based scheduler schedules packets between different streams from different epochs, and makes a decision regarding which packet to send from among the streams. This decision is based on the weight associated with the stream and the utility of the packet to the queries that are interested in the stream. For example, stream 3 in Figure 3.3 is of interest to both queries; therefore transmitting a packet improves the utility for both the queries. In order to compute the utility of a packet, the radar uses *a priori* knowledge of how application utility relates to the mean square error (MSE) of the data. This provides a mechanism for the scheduler to observe *error in the compressed raw data* and determine how this error would translate to *application error*. As we describe later, the mean square error of the data influences utility in different ways for different applications. The scheduler computes the total benefit (computed as the product of marginal utility of the packet and weight assigned) that would result from transmitting the first packet from each stream, and picks the stream with greatest increase in benefit. Figure 3.4 provides an illustration of the scheduling decision. In the example, 66% of the first stream has been transmitted but only 33% of the second stream has been transmitted. Therefore, the difference in mean square error is likely to be higher by transmitting a packet from the second stream. However, there are two additional factors to consider. The first stream corresponds to a tornado detection query, which requires high resolution data in order to precisely pinpoint the location of the tornado, whereas the second stream corresponds to a 3D assimilation query and 3D wind direction estimation queries, each of which needs only less precise data. On the other hand, a packet from the second stream is useful to two concurrent queries, whereas a packet from the first stream is only useful for tornado detection. Thus, the decision of what packet to choose depends on the mean square error of the data, number of queries interested in the data, weights of the queries, and importantly, the utility function of the queries.

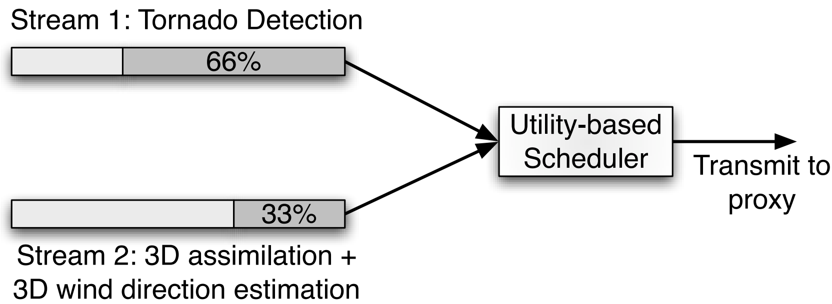


Figure 3.4. In this scenario, 66% of stream 1 and 33% of stream 2 have been transmitted. The scheduler determines the marginal utility of transmitting a packet from each of the streams for the applications interested in the streams and decides which packet to transmit next.

Global Transmission Control: While the progressive encoding and utility-driven scheduling at each sensor optimize for multiple queries at a single radar, there is a need for global control of transmissions to maximize overall utility across the network because radars within the same wireless contention domain share the wireless media and contend with each other while transmitting. In particular, this is useful when queries are not evenly distributed across the network, and some nodes that are handling higher priority queries need more bandwidth than others. In this case more bandwidth should be allocated to the radars that are achieving higher marginal utility. The proxy uses a simple global transmission control policy where it monitors the marginal utility of incoming packets from different radars. If there is a great imbalance in the marginal utility of streams from different radars, it notifies the radar with lower marginal utility to stop its stream temporarily. This has the effect of reducing contention in the network, especially at nodes close to the proxy, thereby potentially enabling a radar with more important data to obtain more bandwidth to the proxy.

3.4 MUDS System Design

We describe each component of the MUDS architecture in greater detail in this section.

3.4.1 Multi-Query Aggregator

The multi-query aggregator is central to the data sharing goals of our system. Aggregating multiple user queries into a single aggregated query has two benefits. First, it minimizes the number of scans performed by the radar (which is time and energy-intensive) since each radar scan is used to answer a batch of queries. Second, it allows the data in a single scan to be transmitted once but shared to answer multiple queries, thereby maximizing query utility in limited bandwidth settings. In contrast, a system that scans and transmits data separately for each query would be extremely inefficient both due to increased scanning overhead, as well as the duplication of data transmitted.

The proxy batches all queries that are posed in each epoch, and at the beginning of the next epoch, it sends to each radar a list of queries that require data from that radar. An alternative model could have been for the proxy to merge the queries and transmit only the merged query to the radar sensor, but we eschewed this option since it would consume more bandwidth than just sending the queries to the radar. Each query is specified by a 4-tuple <*QueryType, ROI, Priority, Deadline*> that shows the type, region of interest, priority, and the deadline of the query. In our system, the region of interest is represented by a sector or a rectangle for simplicity, although our approach can be easily extended to handle more arbitrary regions of interest. The priority can be either specified by the query or implicitly specified by the proxy — for instance, if the user is a high priority user like the National Weather Service — or can be determined as a combination of the two.

The multi-query aggregator then combines multiple user queries into a single aggregated query plan. The query plan that is generated is a spatial map in which the spatial area corresponding to the region covered by the radar is pixelated. For each pixel in the scan data, the corresponding pixel in the query plan is a list of 3-tuples <*QueryType, Weight, Deadline*>, that show the type, weight, and the deadline of queries interested in data sensed at that pixel.

The weight value of a pixel for each query represents the "importance" of transmitting data sensed from that pixel to that query. We use a heuristic for determining pixel weights in order to maximize application utility. Let p_i, I_i , and d_i represent the priority, the region of interest, and the deadline of query i. Priority p_i is represented as a scalar value; region of interest, I_i, is represented as a 2D map where $I_i(u, v)$ is 1 if the pixel (u, v) is within the region of interest of i, and 0 otherwise; and deadline, d_i is in seconds.

Let $w_i(u, v)$ represent the weight of pixel (u, v) for query i. We would like the following three criteria to be satisfied: i) the weight for the pixel should be greater if the query has higher priority than other queries, ii) the weight for the pixel should be greater if the query's deadline is shorter than other queries since higher weight will result in the data being transmitted first, and iii) the weight for the pixel should be zero if the pixel is not in the region of interest of query i. Thus, the weight $w_i(u, v)$ is defined as:

$$w_i(u, v) = p_i I_i(u, v) \frac{1}{d_i} \tag{3.1}$$

3.4.2 Progressive Compression Engine

Data compression is an integral component of rich sensor networks where the data rates can be considerably higher than available bandwidth. In our system, we use progressive encoding to compress raw data. Progressive compression of data yields two benefits: (a) it enables the system to use all available wireless bandwidth to transmit data, thereby adapting to bandwidth fluctuations, and (b) it enables us to order data packets based on utility of data to queries, thereby maximizing overall utility.

Progressive encoding (also known as embedded encoding) compresses data into a bit stream with increasing accuracy. This means that as more bits are added to the stream, the decoded data will contain more detail. In our system, we use a wavelet-based progressive encoding algorithm called *set partitioning in hierarchical trees (SPIHT)* [53]. The choice of a wavelet encoder is well-suited for radar data processing applications since meteorological tornado detection algorithms use wavelet-based processing in order to detect discontinuities in reflectivity and velocity signals [35, 43]. Moreover, SPIHT orders the bits in the steam such that the most important data is transmitted first. Thus, the decoded data can achieve high fidelity even with few packets transmitted.

We provide a brief overview of the SPIHT algorithm next (refer [53] for a detailed discussion). The input data for the algorithm is assumed to be a two-dimension matrix. Before SPIHT encoding, the matrix is first transformed into subbands of different frequencies using the wavelet transform. Then the subbands are formed into a pyramid in ascending order of frequency from top to bottom. The subband with the lowest frequency is on the top of the pyramid. A hierarchical tree is built on the pyramid, naturally defines the spatial relationship on the pyramid. Each node of the tree corresponds to a pixel in current subband. Its direct descendants correspond to the pixels of the same spatial orientation in the subbands in next higher frequency of the pyramid. The SPIHT encoding iterates through the hierarchical tree starting from the root node. In each iteration, the most significant bit of each node is output into a stream and is removed from that node. In the generated stream, the most important data is at the head of the stream because most natural images like photos or radar scans have energy concentrated in the low frequency components so the significance of a point decreases as we move from the highest to the lowest levels of the tree.

Besides generating the progressive stream, the SPIHT encoder also generates an incremental trace of the encoded stream that shows what the mean square error of the decoded data would be after sending each byte of the stream. As described in the next section, this feature is essential to perform utility-driven scheduling of packets.

We made a few modifications to the standard SPIHT encoder to adapt it to our needs. The progressive encoding engine in our system first splits each scan into multiple regions such that all pixels in a region share the same list of three tuples, *<QueryType, Weight, Deadline>* in the aggregated query map. Although this may result in an exponential number of regions with respect to the number of queries in the worst case, in practice we find the number of regions to be small for radar queries. Each of these regions is encoded to generate a progressively compressed stream per region. One practical problem is that the standard wavelet transform that expects a square matrix, but each region can be of arbitrary shape. To deal with this, we use a shape adaptive wavelet coding scheme[54] to encode each region. The shape adaptive wavelet coding encodes arbitrarily shaped object without additional overhead, i.e., the number of coefficients after the transform is

identical to the number of pixels in the original arbitrarily shaped object. After the encoding, the generated streams are buffered and fed into the local transmission scheduler.

3.4.3 Local Transmission Scheduler

At any given time, a radar may have multiple streams that are buffered and being transmitted by the local transmission scheduler. The goal of this scheduler is to optimize the transmission order of the data in the streams in order to maximize overall application utility despite fluctuating bandwidth conditions. We describe this in detail next.

Each stream buffered by the scheduler comprises packets of the same length (1KB in our implementation). The local transmission scheduler optimizes the transmission order of the packets based on their marginal utility to the set of queries corresponding to the stream. The marginal utility of a packet is the increase in utility resulting from the transmission of that packet. Informally, the utility of a prefix of a stream is determined by the application error that results from decoding and processing that prefix.

Formally, let p denote some prefix of a stream and let i denote a query corresponding to that stream. The utility $U_i(p)$ of p to query i is given by

$$U_i(p) = \begin{cases} w_i & \text{if } \mathit{err}_i(p) < \mathit{req_err}_i(p) \\ w_i \frac{\mathit{max_err}_i(p) - \mathit{err}_i(p)}{\mathit{max_err}_i(p) - \mathit{req_err}_i(p)} & \text{if } \mathit{err}_i(p) \geq \mathit{req_err}_i(p) \end{cases} \quad (3.2)$$

where w_i is the weight of the query i; $\mathit{err}_i(p)$ is the application error that results from decoding and processing p; $\mathit{max_err}_i(p)$ is the maximum value of the application error (computed as the error corresponding to a 1KB prefix of the stream); and $\mathit{req_err}_i(p)$ is the error value below which the user is satisfied with the result. Thus, the utility decreases linearly with the application error and stops decreasing when the user-specified limit is reached. The marginal utility of a packet to a query is the difference in utility to the query just before and after sending the packet.

How does the scheduler compute the application error $\mathit{err}_i(p)$? It is impractical for the scheduler to measure $\mathit{err}_i(p)$ by running the application on each prefix of the stream because of the huge computation overhead of decompressing data and executing the application. Thus, we need a simple and accurate method to determine $\mathit{err}_i(p)$ given just the compressed stream. One possibility is to use a data-agnostic metric such as the compression ratio as an indicator of application error. However, since the progressive encoder could be encoding different scans with very different features, this metric is only weakly correlated with application error.

Fortunately, our empirical evaluation confirms that a data-centric metric, the mean square error of the data stream, is highly correlated to the application error. We leverage this observation to estimate application error

as follows. We seed the scheduler with a function $\textit{seed_err}_i(\textit{mse})$ that maps mean square error of the decoded data to application error. Such a function is generated a priori for each application using training data from past radar scans. In the training procedure, scans are compressed into a progressive stream using the SPIHT compression algorithm. The stream is cut off at different prefix lengths, giving us decoded data of varying fidelity. For each such prefix, the application is run on the decoded data, and the error of the decoded data as well as the application error are measured. Based on this measured data, we build a function $\textit{seed_err}_i(\textit{mse})$ for each application and seed each radar with this function.

Finally, during regular operation, the scheduler needs to compute *mse* corresponding to the decoded prefix just after sending the packet. The *mse* can be obtained from the error trace generated by the progressive compressor as described in Section 3.4.2. The scheduler estimates $\textit{err}_i(p)$ as $\textit{seed_err}_i(\textit{mse})$ by simply performing a lookup table. The weight of the query w_i is incorporated in Equation 3.2 so that more urgent queries have higher utility. Note that by construction, all pixels in a region have the same weight.

The total marginal utility of a packet x is its marginal utility across all queries corresponding to the stream. To understand this, suppose there are m queries corresponding to a stream. Let $U_i(p)$ be the utility of prefix p to query i just before sending packet x, and $U_i(p+x)$ just after. Then, the overall marginal utility of packet is given by

$$\Delta U(p) = \sum_{i=1\cdots m} (U_i(p+x) - U_i(p)), \tag{3.3}$$

where the operator '$+$' denotes extending the prefix to include the next packet. Based on Equation 3.3 the scheduler can calculate the marginal utility of the packet at the head of each stream. Given the utility, the scheduler picks in each round the packet with maximum marginal utility across all packets at the heads of existing streams, and transmits that packet. Such a scheduling algorithm can be implemented efficiently in practice. First, we note that packets within a stream are already present in order of decreasing marginal utility, so only the packet at the head of each stream needs to be examined for a scheduling decision. The marginal utility of the packet at the head of each stream can be computed efficiently with a small number of table lookups — one lookup to identify the MSE difference resulting from transmitting the packet, and one lookup per query to identify the marginal utility for the query from decoding the packet. Finally, the packet with the highest marginal utility across all streams needs to be chosen. Since the number of streams is small, our implementation simply uses a linear insert and search procedure; it is straightforward to use a heap instead.

The above packet scheduling algorithm achieves the maximum total utility across all the concurrent streams at each point in time if $U(p)$ is concave, i.e., the marginal utility is strictly decreasing. This can

be proved by reducing it to the knapsack problem[59]. Our empirical evaluation confirms that the marginal utility decreases with the length of the progressively encoded stream.

Example: We exemplify our methodology for computing the $\textit{seed_err}()$ function for the three applications. We first consider tornado detection. This application uses a clustering-based technique to detect tornadoes, and generates the centroids and intensities of each tornado. In order to determine the error in tornado detection, we run the application on scans that were decoded after compressing them to different compression ratios. Let the data MSE for a decoded scan be $\textit{mse}_1$. There are three cases to consider to determine $\textit{seed_err}(mse_1)$. First, if the result on the decompressed scan detects a tornado, t, within 300m of the result on the raw scan, then this is a positive result. The choice of 300m as the threshold for positive detection was made based on discussions with meteorologists. In this case, tornado detection error $tornado_err(t)$ is computed as follows: $\textit{tornado_err}(t) = |(\textit{RI}(t) - \textit{DI}(t))| \cdot \frac{d(t)}{300}$ where $\textit{RI}(t)$ is the intensity of the tornado as determined from processing the raw data, $\textit{DI}(t)$ is the intensity from processing the decoded data, and $d(t)$ is the distance between the actual centroid from the raw data, and the computed centroid from the decoded data. Second, if a tornado, t, is detected in the decoded scan but no tornado is detected in the raw scan within 300m, then it is considered a false positive. In this case, $\textit{tornado_err}(t) = \textit{DI}(t)$. Finally, if a tornado, t, is detected in the raw scan but no tornado is detected within 300m of its centroid in the decoded scan, then this is considered a false negative, and $\textit{tornado_err}(t) = \textit{RI}(t)$.

The total error, $\textit{seed_err}(mse_1)$ is the sum over of the above errors over all tornadoes detected in the raw scan and the compressed scan. Determining the error function for the 3D wind direction estimation and 3D assimilation applications is more straightforward. Here, the applications are run on the raw radar scan and the decompressed scan, and the mean square error of the difference between these results is used as the error for the application.

3.4.4 Global Transmission Control

While the local transmission scheduler uses the weight map to optimize what order to transmit packets from each radar, the global transmission controller performs a decision across all the concurrent streams on all the radars. Radars compete with each other for wireless bandwidth in a number of ways: (i) radars within the same wireless contention domain contend with each other when transmitting, (ii) in multi-hop communication, all the nodes in the same routing branch share the bandwidth of a forwarding node, and (iii) the proxy's incoming bandwidth is shared among all the radars in the network. As a result, maximizing local utility at each radar may not optimize global utility across all radars in the network. A radar with higher utility data might have much lower available bandwidth than a radar with lower utility due to a number of factors.

This necessitates global control of transmissions from radars, in addition to local utility optimization. Global transmission control in wireless networks has been the subject of significant work (e.g. [41]). Most of these approaches use the idea of a conflict graph that captures the interference patterns between nodes in the network. Such a conflict graph can be used as the foundation for scheduling transmissions from nodes such that spatial reuse is maximized, in addition to throughput.

While the use of conflict graphs is the subject of our future research in the area, we use a simple but effective heuristic in this work. In our approach, the proxy monitors the incoming streams from the radars, and stops the transmission of streams that will not improve overall utility much. Specifically, the proxy stops a stream when its utility reaches 95% of its maximal utility. The proxy knows the maximum utility since it has a locally generated version of the aggregated query plan. Since utility is a concave function of the length of the transmitted data stream, the utility of a stream grows very slowly after having achieved 95% of its maximal value. Therefore stopping the stream does not affect overall utility significantly. However, stopping a stream can benefit other streams since there will be less channel contention, and less forwarded data to the proxy. We experimentally demonstrate the effectiveness of such a threshold-based global transmission control in Section 3.5.

3.5 Experimental Evaluation

In this section, we evaluate the performance of our system using a radar trace-driven prototype implementation as well as trace-driven simulations. We use two data traces in our experiments. The first is the *Oklahoma dataset* collected from a 4-radar testbed deployed in Oklahoma (obtained from meteorologists [33]). Each radar in the testbed generates 107MB Doppler readings per 360-degree scan every 30 seconds. We collected 30 minutes of trace data from each radar. To obtain a larger scale dataset for scalability experiments, we also obtained an emulated radar data set generated by the Advanced Regional Prediction System (ARPS) emulator. The ARPS emulator is a comprehensive regional-to-stormscale atmospheric modeling system designed by the Center for Analysis and Prediction of Storms, which can simulate weather phenomena like storms and tornadoes, and generate data at the same rate as the real radars in the Oklahoma testbed. We emulated 12 radars in the emulator and collected 30 minutes of trace data from each of them. We refer to this trace as the *ARPS dataset*. The ARPS emulator takes days to generate a 30 minute trace, hence larger traces were prohibitively time consuming. Note that the actual raw data from radars can be up to an order of magnitude larger than the two datasets that we used. We were limited to collecting smaller datasets by the bandwidth and storage capacity in the Oklahoma network, and the speed of the ARPS emulator.

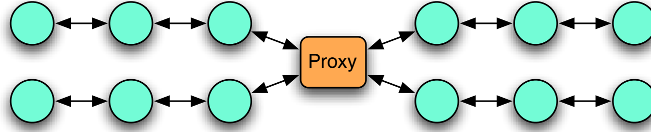


Figure 3.5. The routing topology of a 13-node wireless testbed with one proxy and twelve emulated radars.

Our radar network prototype comprises 13 radar nodes, each emulated by a Apple Mac Mini computer with an 802.11 b/g wireless card. We manually configure the nodes into a 3-hop wireless topology (shown in Figure 3.5) by setting their routing tables appropriately. The proxy is a server running a proxy process that collects data from radars and processes user queries. The other twelve nodes run radar processes that encode and transmit radar data. To simplify our protocol design, we use TCP as our transmission protocol, since the progressively stream needs to be received reliably and in-order for decoding. Two TCP/IP connections are built between each radar and the proxy—one for transmitting data from the radar to the proxy, the other for sending control information from the proxy to the radar. The progressive compression engine was adapted from the open-source QccPack library [38] that provides an implementation of SPIHT for images.

To evaluate performance of individual components of our system under controlled conditions, we augment prototype experiments with simulations using real traces. In order to evaluate the query processing performance of our system, we implement a query generator. Each generated query is a 4-tuple $< Type, ROI, Deadline, Priority >$. The *Type* field is the application type which can be tornado detection, wind direction estimation or 3D assimilation. The *ROI* field shows the query's region of interest which is represented by a sector of the radar's circular sensing range. The *Deadline* field represents the query's reply deadline in seconds. The *Priority* field represents the query's priority, which is determined by the user's preference to this query. We implemented two query arrival models: (i) a *Poisson arrival model* in which queries arrive at each radar as a Poisson process with configurable average arrival rate, (ii) a *deterministic model* in which queries arrive at each radar in fixed order at fixed rate. For the tornado detection query, we designed a additional model, in collaboration with meteorologists, that models query patterns during a tornado. In this model, the priority of the tornado query, and the nodes on which it is posed depends on where the tornado is predicted to be localized.

3.5.1 Determining the Utility Function

At the core of our system is a utility function that captures application-perceived utility as a function of the mean square error of data being transmitted by the radars. To evaluate the utility functions for the three applications, we ran the applications on lossily compressed versions of the Oklahoma dataset. We lossily

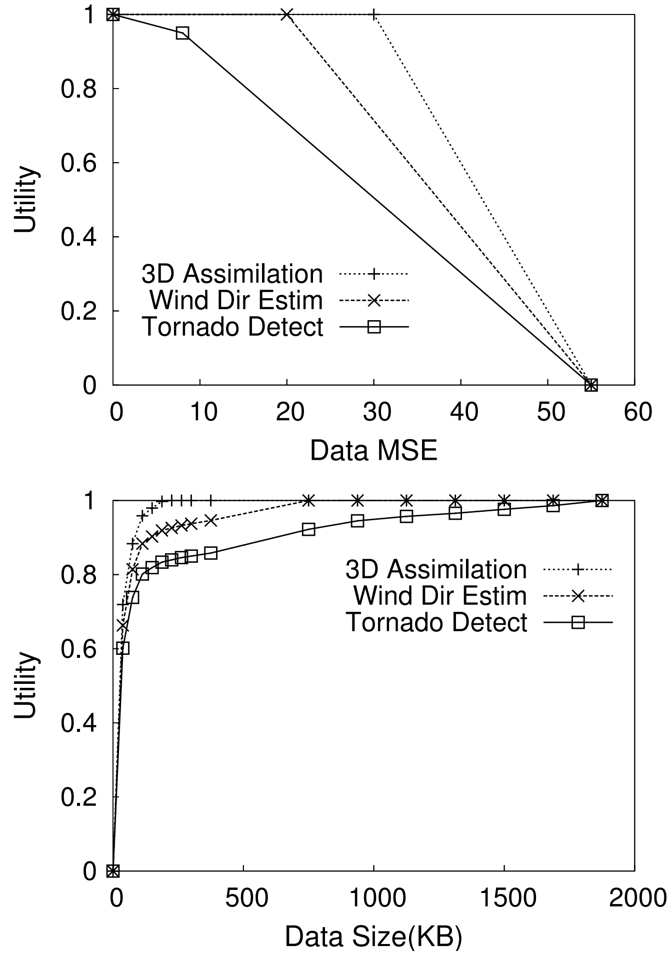


Figure 3.6. Utility functions for the three applications are derived by compressing and evaluating application performance on traces from the Oklahoma dataset.

compress the data traces to $\frac{1}{2^i}$ of original size with i ranging from 1 to 13. For each of these compression ratios, we measure the mean square error of the resulting data after decompression, as well as the application error after executing it on the decompressed data. Given the application error, the utilities for the applications are generated using Equation 3.2. Here we use fixed user requirement E_{user} in the experiments so that the utility functions only need to computed once. We fit piece-wise linear functions to utility functions, and use these functions as the utility functions in the rest of our experiments. The graph on the top of Figure 3.6 shows the piece-wise utility functions of the three applications obtained from our empirical evaluation. The bottom graph shows an example of how this utility function would translate to actual number of packets when a scan is compressed.

3.5.2 Impact of Weighting Policy

The weight value of a pixel quantifies the importance of data sensed from that pixel to the queries. In MUDS system, we use Equation 3.1 to determine the weight of a pixel, in which the area of interest of the query, its priority as well as its deadline are taken into account. We compare this policy against three other weighting policies. We use a policy that only takes the area of interest into account, i.e. $w_{AOI} = I_i(u, v)$, as a baseline of our comparison. Then we consider two variants of our policy: i) $w_{deadline} = I_i(u, v)/d_i$ in which the area of interest and the deadline are taken into account, and iii) $w_{priority} = I_i(u, v)p_i$ in which the area of interest and the priority are taken into account.

We evaluate the four policies using trace-driven simulations with the Oklahoma dataset. Table 3.1 shows the average utility per epoch achieved using different weighting policies. The weighting policy used in MUDS performs the best and achieves 1.6 folds more utility than the baseline, while the priority-based policy achieves 30 percent more utility than the deadline-based policy, which shows that the priority has higher impact than the deadline. This comparision demonstrates that our weighting policy decently quantifies the importantce of data.

Policy	w_{AOI}	$w_{deadline}$	$w_{priority}$	w_{MUDS}
Utility	0.612	0.783	0.976	1.633

Table 3.1. Comparision of different weighting policies.

3.5.3 Performance of Progressive Compression

In this section, we evaluate two main benefits of the SPIHT progressive compression algorithm: (i) higher compression rate, and (ii) adaptation to bandwidth fluctuation.

3.5.3.1 Compression Efficiency

The extreme data generation rates of radar sensors makes compression an essential component of radar sensor system design. In this section, we compare the compression efficiency of the SPIHT algorithm that we employ against an *averaging* compression algorithm that is currently used in the *NetRad* radar system. Each radar scan is represented as a matrix of gates x azimuths, where the radial axis is divided into gates, and the angular dimension is divided into azimuths. The averaging compression algorithm compresses data simply by averaging along the azimuth dimension. In order to compress data n times, the averaging compression algorithm averages values from n adjacent azimuths in the same gate position . The compressed data has n times fewer azimuths than the original data.

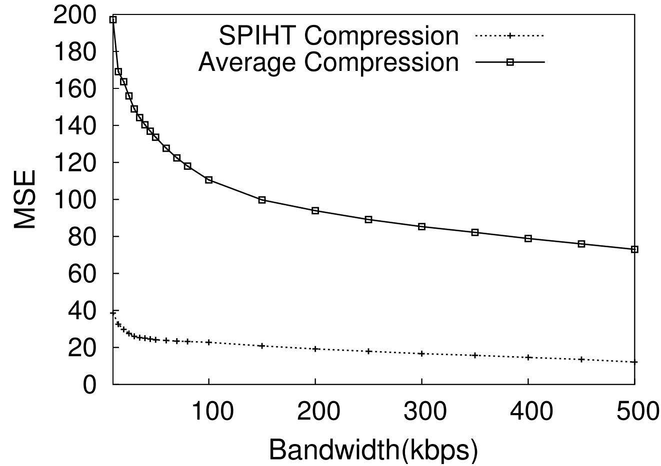


Figure 3.7. Comparison of SPIHT progressive compression against averaging compression. Each algorithm compresses data to the size that can be transmitted in one epoch for a given bandwidth.

We compare the two compression algorithms using trace-driven simulations with the Oklahoma dataset. Each scan in the trace is compressed to the size s that can be sent in one epoch (30 seconds) under a fixed bandwidth B, i.e., $s = 30 \cdot B$. The MSE of the compressed data is measured for different bandwidth settings ranging from 10kbps to 500kbps. Figure 3.7 shows MSE as a function of bandwidth. With increasing bandwidth, the MSE of the SPIHT algorithm decreases much more quickly than the averaging algorithm since SPIHT captures the key features of the radar scan using very few packets. Even at extremely low bandwidths such as 20kbps, the MSE of the SPIHT compressed stream is 20, whereas the MSE of the same stream with averaging compression is an order of magnitude higher at 200. This shows that SPIHT is an extremely efficient compression scheme for radar data.

3.5.3.2 Bandwidth Adaptation

Next, we evaluate the ability of SPIHT to adapt to bandwidth fluctuations. SPIHT adapts to fluctuations naturally because of its progressive feature, i.e., data can be decoded progressively without receiving the entire compressed data stream. We compare it against a non-progressive compression algorithm under different levels of bandwidth fluctuation. The non-progressive algorithm is implemented by simply removing the progressive feature from SPIHT. In other words, the non-progressive SPIHT encoder first estimates how much bandwidth is highly likely to be available until the deadline of the stream, and would compresses data to that size before transmission. The data can be decoded by the proxy only after the entire compressed stream is received since no partial decoding is possible.

Unlike progressive compression where the receiver can decode even a partially transmitted stream, a non-progressive compression-based scheme has to rely on a conservative estimate of the available bandwidth to ensure the compressed data can be fully transmitted and received before the query deadline. We use a moving window estimation algorithm in our implementation. The non-progressive encoder considers a window of bandwidth values in last w epochs. The values are sorted in descending order and the 95th percentile value is taken as the estimated bandwidth. We use a window size of 20 in the experiments.

We perform a trace-driven simulation using the Oklahoma dataset where the available bandwidth in each epoch is chosen from a normal distribution with mean 40kbps. The standard deviation of the distribution is varied from 0kbps to 25kbps in steps of 5, and the resulting MSE from the two schemes is measured. Figure 3.8 shows MSE of the decoded data as a function of the standard deviation of the distribution. At a standard deviation of zero, the two compression algorithms achieve the same accuracy since they utilize the same amount of bandwidth. As the standard deviation increases, the bandwidth utilized by the non-progressive algorithm drops quickly, because it estimates available bandwidth conservatively. Therefore, the accuracy of the non-progressive algorithm degrades much more quickly than the progressive algorithm. For the highest standard deviation, the MSE of the non-progressive algorithm is four times more than that of the progressive algorithm.

Figure 3.9 gives us a time-series view of how bandwidth fluctuation impacts the two schemes. While the non-progressive scheme has high MSE due to its conservative estimate, the MSE for the progressive compression scheme follows the fluctuations in bandwidth since it is able to exploit the entire available bandwidth. The R-value for the bandwidth and MSE time-series for the progressive algorithm is -0.79, indicating robust anti-correlation: *i.e.* bandwidth is inversely correlated to the MSE.

3.5.4 Performance of Data Sharing

In multi-user sensor networks with diverse end-user needs, sharing data among queries greatly improves utility of the system. We evaluate the ability of our system to handle two types of data sharing: i) queries with identical regions of interest but with different deadlines, and ii) queries with identical deadlines but with overlapping regions of interest.

3.5.4.1 Temporally Overlapping Queries

We first consider the case where queries have the same region of interest but have different deadlines. In this case, the progressive compression engine generates a single progressively compressed stream for both queries. The query processor decodes the compressed stream as it is received, and processes the two queries when their deadlines arise. In contrast, a system using non-progressive compression cannot easily share data

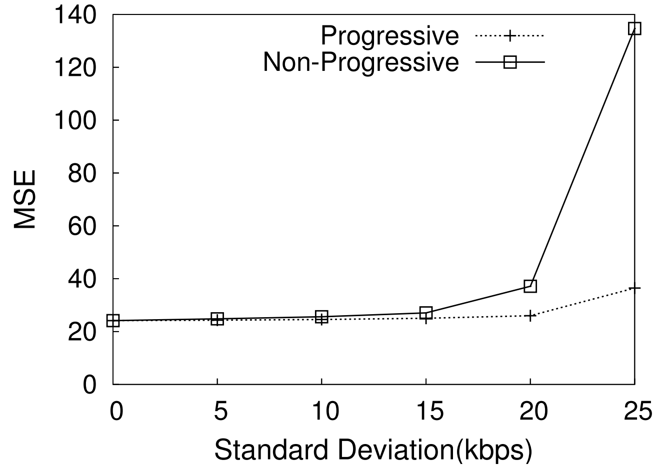


Figure 3.8. Comparison of progressive compression against non-progressive compression for different levels of bandwidth fluctuation. Bandwidth fluctuation follows a normal distribution with mean 40kbps; standard deviation is varied from 0kbps to 25kbps.

between queries. We compare our approach against a non-progressive compression scheme in which data is compressed and transmitted separately for each query individually.

We evaluate the two schemes using trace-driven simulations with the Oklahoma dataset. Two queries—tornado detection and 3D assimilation—arrive at a radar every two epochs. They have different deadlines but both ask for all the data from a 360-degree scan. The tornado detection query has a deadline of one epoch, and the 3D assimilation query has a deadline of two epochs. Figure 3.10 shows the utility of the two schemes as bandwidth is varied from 10kbps to 100kbps. At bandwidth of 10kbps, our system achieves five times the utility of the non-progressive scheme. As the bandwidth increases, both schemes can get significant data through to the proxy, therefore the relative utility gains from our system reduces.

3.5.4.2 Spatially Overlapping Queries

We next evaluate our system's ability to handle queries with the same deadline and overlapping regions of interest. Regions that are of interest to multiple queries are weighted higher than regions of interest to just one query, and are therefore transmitted earlier and with higher fidelity than regions that are only of interest to one of the queries. We compare our scheme to a scheme without data-sharing. For the non-data-sharing scheme, data for different queries are sent separately even when there is overlap between the queries. We consider two queries, tornado detection and 3D assimilation, each of which requires data from a 180-degree sector. The degree of overlap between the regions of interest for the two queries is varied from 0 degrees to 180 degrees in steps of 30 degrees.

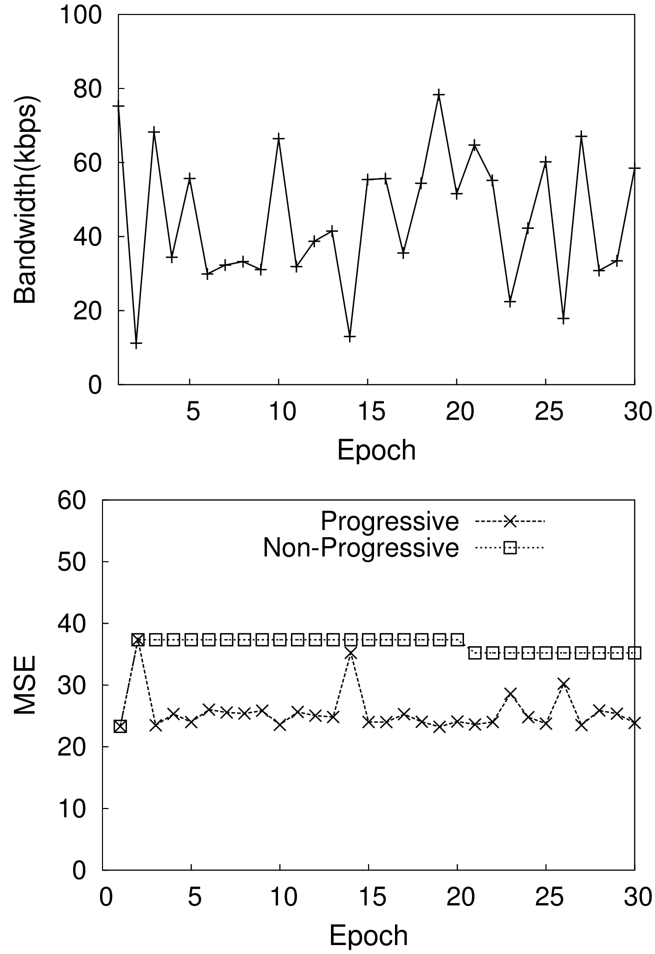


Figure 3.9. Time series of bandwidth and MSE of decoded data. Bandwidth fluctuation follows a normal distribution with mean value at 40kbps and standard deviation of 25kbps.

Figure 3.11 shows the end-user utility achieved by our scheme and the non-data-sharing scheme as the angle of overlap of the two queries is varied. As the angle of overlap increases, the utility gain from our scheme increases. For an overlap of 180 degrees (maximum overlap), our scheme achieves 21% higher utility than the non-data-sharing scheme.

3.5.5 Performance of Local Scheduler

We now evaluate the benefit of the local transmission scheduler, which always transmits the packet with the highest utility gain first. We compare this approach against an approach that uses a random transmission scheduler, which picks packets randomly from heads of the data streams. In the experiments, we simulate one radar and one server and control the available bandwidth. The tornado detection, wind direction estimation, and 3D assimilation queries arrive in round robin order at the radar at the beginning of each epoch. All

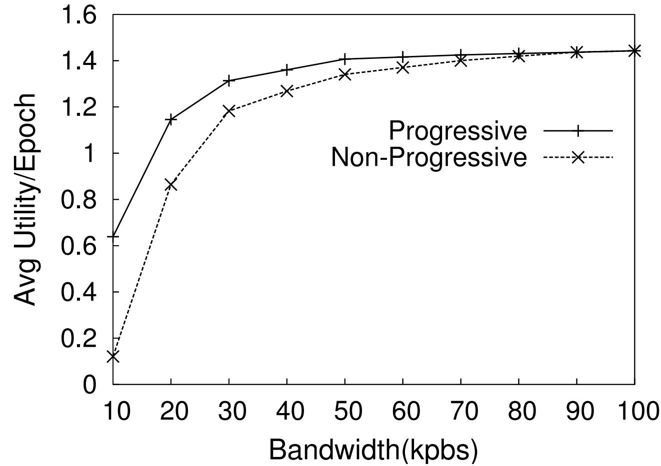


Figure 3.10. Performance for temporally overlapping queries. Two queries with different deadlines but same region of interest arrive at the radar every two epochs.

queries have the same priority and the same deadline of three epochs. We run the two systems at bandwidth ranging from 10kbps to 150kbps, and seeded with the Oklahoma dataset.

Figure 3.12 shows the average utility per epoch as a function of bandwidth. For bandwidth lower than 150kbps, the utility-driven scheduler always achieves higher utility than the random scheduler, with as much as 100% increase in utility at low bandwidth. As bandwidth increases, the utilities of the two systems become closer. Our system performs better under low bandwidth conditions because the most important data are always sent in the first packets. When bandwidth is high enough to send all data in high fidelity, e.g., at 150kbps, there is negligible benefit from utility-driven scheduling.

3.5.6 Performance of Global Control

Having evaluated the performance of local transmission control, we next consider global transmission control by the proxy. Such an optimization is beneficial when there is an imbalance in query load across different regions in the network. We designed an uneven query pattern as follows - a tornado detection query with priority=3 arrives at each radar which is i-hops (i varies from 1 to 3) away from the server in each epoch, while each of the other radars has a wind direction estimation or 3D assimilation query with priority=1 in each epoch. We use the testbed consisting of one server and twelve radars as shown in Figure 3.5. Each radar in the testbed is seeded with a radar trace from the ARPS dataset.

Figure 3.13 shows the average utility per epoch with increasing number of hops from the proxy. The utility decreases for both of the approaches as queries with high priority arrive at nodes farther from the proxy. This is because nodes on the edge of the routing topology usually have less available bandwidth

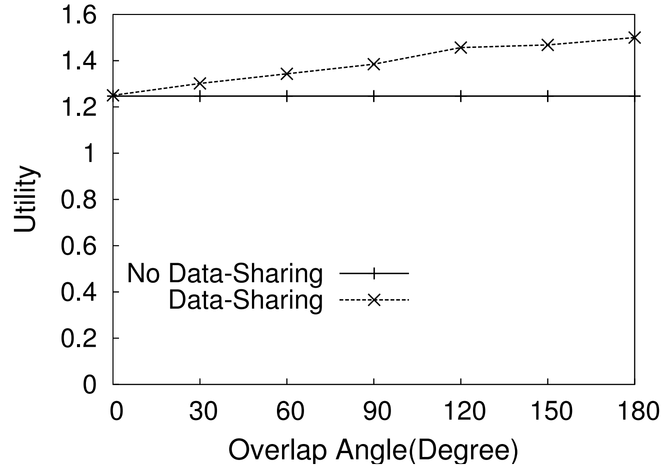


Figure 3.11. Evaluation of the impact of data sharing on utility. Two applications, tornado detection and 3D assimilation, with overlapping sectors are considered.

than nodes closer to the proxy, as packet loss probability increases as packets travel more hops. Thus, a query arriving at an edge node cannot achieve high utility because of the limited bandwidth, therefore, the contribution of the tornado detection query to the overall utility is reduced. However, the global control-based approach degrades much slower than the approach without global control. For instance, when the tornado query is posed three hops from the proxy, the global control-based approach achieves twice the utility of the approach without such control. This shows that global transmission control provides a simple but effective approach to deal with imbalanced query loads.

3.5.7 System Scalability

Until now, we have characterized the performance of individual components of our system. We now turn to full system measurement and evaluation on our testbed. Our goals are two-fold: i) to demonstrate that our system as a whole scales well with network size and number of queries per epoch, and, ii) to provide a breakdown of the utility gains provided by the different components of our system.

3.5.7.1 Impact of Network Size

Our first set of scalability experiments test our system at different network scales. In the experiments we use different number of nodes in the testbed shown in Figure 3.5 — the one and four node experiments are for a one hop topology, the eight node experiments are for a two hop topology, and the twelve node experiments are for a three hop topology. Each radar is seeded with data traces from the ARPS dataset.

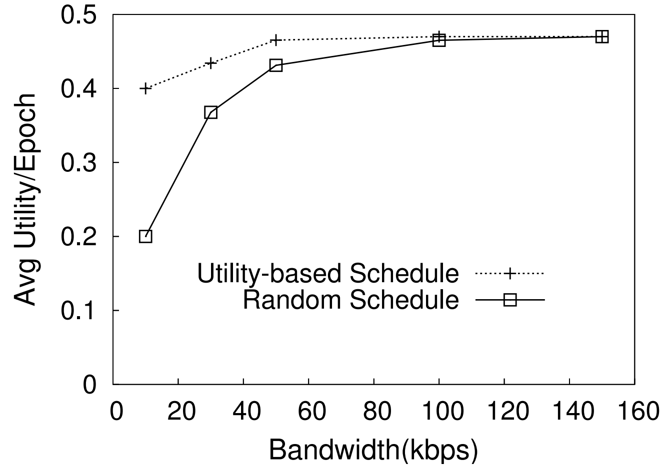


Figure 3.12. Comparison of utility-driven scheduling against random scheduling.

The query distribution for our experiments was designed, in collaboration with meteorologists, to realistically model query patterns during a tornado. The three queries — tornado detection, wind direction estimation, and 3D assimilation — arrives at each radar as a Poisson process with average arrival rate of one query per three epochs and standard deviation of one query per epoch. The wind direction estimation queries and 3D assimilation queries are assigned weights of one or two randomly.

The priority of the tornado query, and the nodes on which it is posed depends on where the tornado is predicted to be. Meteorologists use tracking algorithms such as Extended Kalman Filters to track tornado trajectories, thereby predicting its likely location. Therefore, in our query model, we assume that the priority of tornado detection queries is three on radars where the tornado is predicted to be observed by the tracker, and is one otherwise. To generate this query pattern, we use a visual estimate from the ARPS emulator data to determine the likely centroid of the tornado.

We compare four schemes in this experiment. The existing *NetRad* system with averaging compression and conservative bandwidth estimation (described in Section 3.5.3.2) provides us a baseline for comparison. Then, we consider three variants of our system: first, we turn on progressive compression only, then we turn on progressive compression as well as local transmission scheduling, and finally, we include global control as well. Figure 3.14 shows the average utilities per epoch of *NetRad* and the three variations of our system.

For small networks (1 or 4 nodes), our gains over the *NetRad* system are primarily due to progressive compression. For instance, when there is only one radar in the network, just the addition of progressive compression gives us 3x as much utility as the NetRad scheme. Both local scheduling and global control have limited impact for the one and four node network settings, because there is limited contention and

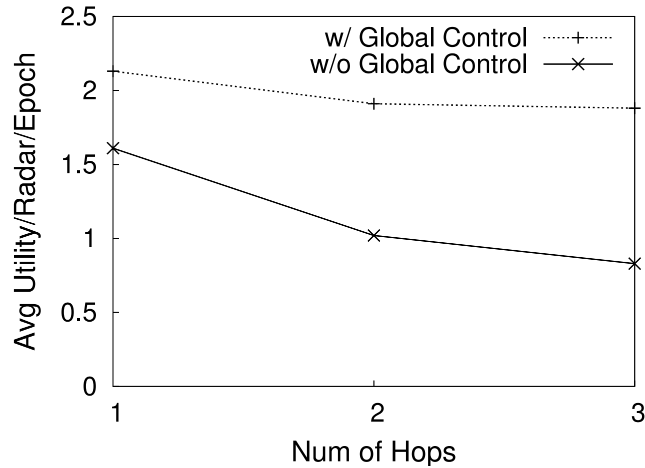


Figure 3.13. Performance of global transmission control. Utility is shown for differing numbers of hops from the proxy to nodes having high-priority queries.

considerable available bandwidth from each node to the proxy. Thus, at a network size of one, the addition of local scheduling achieves only 4% more utility than just having progressive compression. Global control has no impact at network size 1, and limited impact at network size 4.

As system size increases, contention between nodes also increases. There is less available bandwidth per radar and more bandwidth fluctuation due to increased contention and collisions, and consequent variations in TCP window size. As a result, both local scheduling as well as global control give more gains. The benefit from these schemes increases with growing network size. For instance, the addition of local scheduling to progressive compression increases utility from 15% at network size four to 38% at network size 12. The inclusion of global control improves utility by only 4% at network size 4, but provides a 30% improvement at network of size 12.

Another point to note is the increasing difference in performance between the *NetRad* scheme and our full system. With all three techniques enabled, our system achieves more than an order of magnitude improvement in utility over the NetRad system for network size at 12. As network size increases from one to twelve, the utility of our system only decreases by 25%, whereas the utility of NetRad decreases by 80%; this comparison demonstrates the scalability of our system.

3.5.7.2 Impact of Query Load

Our second scalability experiment stresses the query handling ability of our system. We compare our system against the *NetRad* system under different query loads. Since the query processor aggregates the same type of queries into a single query in each epoch, there are at most three queries posted on each radar

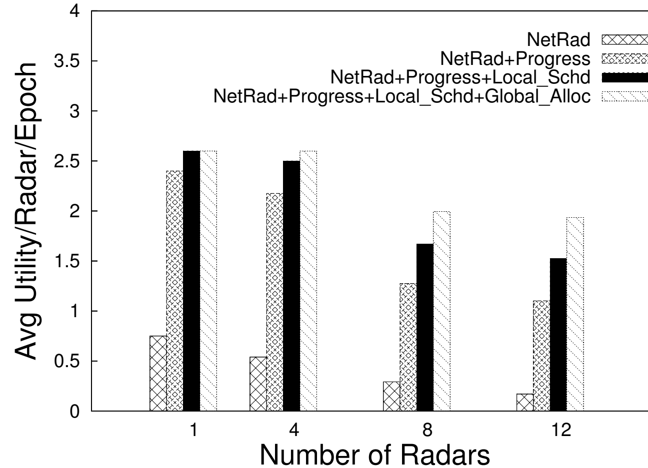


Figure 3.14. Scalability to network size. Breakdown of contribution of each component of our system to the overall utility.

per epoch. We run the experiments on the wireless testbed at network size 12. Each radar node is seeded with a data trace from the ARPS emulator. We use constant query arrival rate in the query distribution for our experiments. In each epoch at most three queries of different types arrive at each radar. The priorities of the wind direction estimation queries and 3D assimilation queries are assigned one or two randomly. The priority of tornado detection queries is three on radars which the tornado is predicted to be observed by the tracker, and is one otherwise. We evaluate the two systems under different query rate ranging from one to three queries per epoch.

Figure 3.15 shows the average utility per query as a function of query rate. In our system, as the query rate increases, each query still gets data with sufficient accuracy to achieve high utility. Thus, the utility of NetRad system decreases by 25% when the query rate increases from one to three, whereas the utility of our system only decreases by 15%. This demonstrates the scalability of our system to high query load.

3.6 Related Work

We discuss related work not covered in previous sections.

Radar Sensor Network: The work most related to MUDS is the Meteorological Command and Control (MC&C)[56, 48, 49] system deployed in NetRad radar network that schedules sensing tasks of radars. MC&C allocates resources such as beam position to satisfy end-user's needs. Based on the sensing schedules from MC&C, our MUDS system optimizes data transmission to maximize the total utility gain.

Multi-query Optimization: A few approaches have addressed multi-query optimization in sensor networks [45, 55, 50]. For instance, [45] considers a limited form of multi-user sharing where different users request

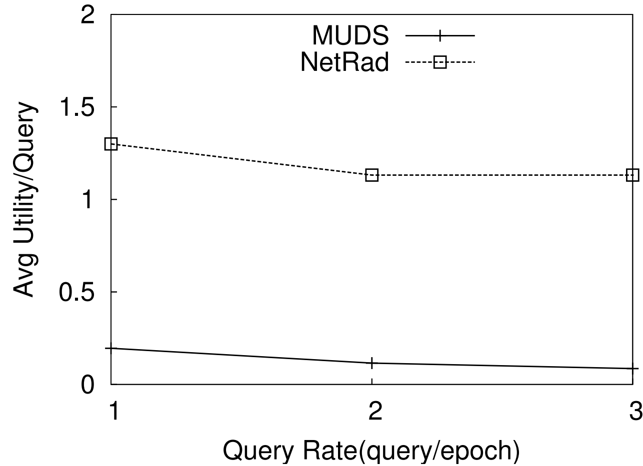


Figure 3.15. System scalability to the query load.

data at different rates from different sensors, [55] considers a multi-query optimization for arbitrary SQL queries and do simple data aggregrations such as min, max, sum, count and average, and [50] considers spatial overlaps between queries and builds an aggregation tree in each area of interest. In contrast, we consider data sharing for considerably more complex applications involving spatial and temporal data sharing, but focuses on the specific set of queries used in radar sensor networks.

Utility-based Design: There is a growing body of research on utility-based approaches to address different problems in sensor networks including resource allocation in SORA [44], and sensor placement [31]. Much of this work is only peripherally related to our work. For instance, SORA employs a reinforcement learning and an economic approach for energy optimization in sensor networks [44]. The work is not designed for multi-user scenarios.

Data compression: Many techniques have used data compression to reduce communication energy overhead in sensor networks. For instance, Sadler et al. [52] consider data compression algorithms such as LZW for networks of energy-constrained devices. [51] introduces a compression layer between the link and the routing layers and allows sensors to tune between computation intensive and transmission intensive. However, the use of progressive compression together with multi-query optimization on resource-rich platforms is a novel approach that has not been studied in the past.

Utility in Internet-based Systems: For Internet-like networks, Kelly [42] pioneered a utility-theoretic framework for rate control and, in particular, for deconstructing TCP like protocols. Such approaches have also been used for jointly optimizing routing and rate control [42, 40]. These schemes attempt to allocate resources such as bandwidth across users without consideration to data sharing between the users. Multicast

rate control schemes exploit data sharing across users; but they apply to a one-to-many environment unlike MUDS that is designed for many-to-one or many-to-many environments.

3.7 Conclusions

In this chapter, we focused on a network of rich sensors that are geographically distributed and argued that the design of such networks poses very different challenges from traditional "mote-class" sensor network design. We identified the need to handle the diverse requirements of multiple users to be a major design challenge, and proposed a utility-driven approach to maximize data sharing across users while judiciously using limited network and computational resources. Our utility-driven architecture addresses three key challenges: how to define utility functions for networks with data sharing among end-users, how to compress and prioritize data transmissions according to its importance to end-users, and how to gracefully degrade end-user utility in the presence of bandwidth fluctuations. We instantiated this architecture in the context of geographically distributed wireless radar sensor networks for weather, and presented results from an implementation of our system on a multi-hop wireless mesh network that uses real radar data with real end-user applications. Our results demonstrated that our progressive compression and transmission approach achieves an order or magnitude improvement in application utility over existing utility-agnostic non-progressive approaches, while also scaling better with the number of nodes in the network.

CHAPTER 4

HIGH THROUGHPUT RELIABLE WIRELESS TRANSPORT

4.1 Introduction

Having discussed how to reduce data to be transmitted, we turn to challenges of how to transmit the data over wireless networks. In particular, we ask how to make wireless transport more reliable and can support higher through to better support above data management systems. High throughput reliable transport is desirable for many high data rate sensing applications. For instance, MUDS requires reliable transport to ensure decodability of the process streams, and can take advantage of high throughput to transmit more data to the proxy to improve the system utility.

Reliable high-throughput transport, although desirable for many wireless applications, is hard to achieve due to several fundamental constraints of wireless networks. Many studies have shown that TCP, the universal transport protocol for reliable transport, is ill-suited for multi-hop wireless networks. There are three key reasons for this mismatch. First, multi-hop wireless networks exhibit a range of loss characteristics depending on node separation, channel characteristics, external interference, and traffic load, whereas TCP performs well only under low loss conditions. Second, many emerging multi-hop wireless networks such as long-distance wireless mesh networks, delay-tolerant networks, and highly duty-cycled sensor networks exhibit intermittent disconnections or persistent partitions. TCP assumes a contemporaneous end-to-end route to be available and breaks down in partitioned environments [73]. Third, TCP has well-known fairness issues due to interactions between its rate control mechanism and CSMA in the link layer, e.g., it is common for some flows to get completely shut out when many TCP/802.11 flows contend simultaneously [98]. Although many solutions (e.g. [79, 93, 99]) have been proposed to address parts of these problems, these have not gained much traction and TCP remains the dominant available alternative today.

Our position is that a clean slate re-design of wireless transport necessitates re-thinking three fundamental design assumptions in legacy transport protocols, namely that 1) a packet is the unit of reliable wireless transport, 2) end-to-end rate control is the mechanism for dealing with congestion, and 3) a contemporaneous end-to-end route is available. The use of a small packet as the granularity of data transfer results in increased overhead for acknowledgements, timeouts and retransmissions, especially in high contention and loss conditions. End-to-end rate control severely hurts utilization as end-to-end loss and delay feedback is

highly unpredictable in multi-hop wireless networks. The assumption of end-to-end route availability stalls TCP during periods of high contention and loss, as well as during intermittent disconnections.

Our transport protocol, Hop uses *reliable per-hop block transfer* as a building block, in direct contrast to the above assumptions. Hop makes three fundamental changes to wireless transport. First, Hop replaces packets with *blocks*, i.e., large segments of contiguous data. Blocks amortize many sources of overhead including retransmissions, timeouts, and control packets over a larger unit of transfer, thereby increasing overall utilization. Second, Hop does not slow down in response to erroneous end-to-end feedback. Instead, it uses hop-by-hop backpressure, which provides more explicit and simple feedback that is robust to fluctuating loss and delay. Third, Hop uses hop-by-hop reliability in addition to end-to-end reliability. Thus, Hop is tolerant to intermittent disconnections and can make progress even when a contemporaneous end-to-end route is never available, i.e., the network is always partitioned [62].

Large blocks introduce two challenges that Hop converts into opportunities. First, end-to-end block retransmissions are considerably more expensive than packet retransmissions. Hop ensures end-to-end reliability through a novel retransmission scheme called *virtual retransmissions*. Hop routers cache large in-transit blocks. Upon an end-to-end timeout triggered by an outstanding block, a Hop sender sends a token corresponding to the block along portions of the route where the block is already cached, and only physically retransmits blocks along non-overlapping portions of the route where it is not cached. Second, large blocks as the unit of transmission exacerbates hidden terminal situations. Hop uses a novel *ack withholding* mechanism that sequences block transfer across multiple senders transmitting to a single receiver. This lightweight scheme reduces collisions in hidden terminal scenarios while incurring no additional control overhead.

In summary, our main contribution is to show that reliable per-hop block transfer is fundamentally better than the traditional end-to-end packet stream abstraction through the design, implementation, and evaluation of Hop. The individual components of Hop's design are simple and perhaps right out of an undergraduate networking textbook, but they provide dramatic improvements in combination. In comparison to the best variant of 1) TCP, 2) Hop-by-hop TCP, and 3) DTN 2.5, a delay tolerant transport protocol [69],

- Hop achieves a median goodput benefit of 1.6× and 2.3× over single- and multi-hop paths respectively. The corresponding lower quartile gains are 28× and 2.7× showing that Hop degrades gracefully.

- Under high load, Hop achieves over an order of magnitude benefit in median goodput (e.g., 90× over TCP with 30 concurrent large flows), while achieving comparable or better aggregate goodput and transfer delay for large as well as small files.

- Hop is robust to partitions, and maintains its performance gains in well-connected WLANs and mesh networks as well as disruption-prone networks. Hop also co-exists well with delay-sensitive VoIP traffic.

4.2 Why reliable per-hop block transfer?

In this section, we give some elementary arguments for why reliable per-hop block transfer with hop-by-hop flow control is better than TCP's end-to-end packet stream with end-to-end rate control in wireless networks.

Block vs. packet: A major source of inefficiency is transport layer per-packet overhead for timeouts, acknowledgements and retransmissions. These overheads are low in networks with low contention and loss but increase significantly as wireless contention and loss rates increase. Transferring data in blocks as opposed to packets provides two key benefits. First, it amortizes the overhead of each control packet over larger number of data packets. This allows us to use additional control packets, for example, to exploit in-network caching, which would be prohibitively expensive at the granularity of a packet. Second, it enables transport to leverage link-layer techniques such as 802.11 burst transfer capability [60], whose benefits increase with large blocks.

Transport vs. link-layer reliability: Wireless channels can be lossy with extremely high raw channel loss rates in high interference conditions. In such networks, the end-to-end delivery rate decreases exponentially with the number of hops along the path, severely degrading TCP throughput. The state-of-the-art response today is to use a sufficiently large number of 802.11 link-layer acknowledgements (ARQ) to provide a reliable channel abstraction to TCP. However, 802.11 ARQ 1) interacts poorly with TCP end-to-end rate control as it increases RTT variance, 2) increases per-packet overhead due to more carrier sensing, backoffs, and acknowledgments, especially under high contention and loss (in Section 4.5.1.1, we show that 802.11b ARQ has 35% overhead). Note that TCP's woes cannot be addressed by just setting the 802.11 ARQ limit to a large value as it would reduce the overall throughput by disproportionately using the channel for transmitting packets over bad links. Unlike TCP, Hop relies solely on transport-layer reliability and avoids link-layer retransmissions for data, thereby avoiding negative interactions between the link and transport layers.

Hop-by-hop vs. end-to-end congestion control: Rate control in TCP occurs in response to end-to-end loss and delay feedback reported by each packet. However, end-to-end feedback is error-prone and has high variance in multi-hop wireless networks as each packet observes significantly different wireless interference across different contention domains along the route. This variance hurts TCP's utilization as: 1) its window

size shrinks conservatively in response to loss, and 2) it experiences frequent retransmission timeouts when no data is sent.

Our position is that fixing TCP's rate control algorithm in environments with high variability is fundamentally difficult. Instead, we circumvent end-to-end rate control, and replace it with hop-by-hop backpressure. Our approach has two key benefits: 1) hop-by-hop feedback is more robust than end-to-end feedback as it involves only a single contention domain, and 2) block-level feedback provides an aggregated link quality estimate that has less variability than packet-level feedback.

In-network caching: The use of reliable per-hop block transfer enables us to exploit caching at intermediate hops for two benefits. First, caching obviates redundant retransmissions along previously traversed segments of a route. Second, caching is more robust to intermittent disconnections as it enables progress even when a contemporaneous end-to-end route is unavailable. Hop can also use secondary storage if needed in partitionable networks with long disconnections and reconnections.

4.3 Design

This section describes the Hop protocol in detail. Hop's design consists of six main components: 1) reliable per-hop block transfer, 2) virtual retransmissions for end-to-end reliability, 3) backpressure congestion control, 4) handling routing partitions, 5) ack withholding to handle hidden terminals, and 6) a per-node packet scheduler.

4.3.1 Reliable per-hop block transfer

The unit of reliable transmission in Hop is a *block*, i.e., a large segment of contiguous data. A block comprises a number of txops (the unit of a link layer burst), which in turn consists of a number of frames (Figure 4.1). The protocol proceeds in rounds until a block B is successfully transmitted. In round i, the transport layer sends a BSYN packet to the next-hop requesting an acknowledgment for B. Upon receipt of the BSYN, the receiver transmits a bitmap acknowledgement, BACK, with bits set for packets in B that have been correctly received. In response to the BACK, the sender transmits packets from B that are missing at the receiver. This procedure repeats until the block is correctly received at the receiver.

Control Overhead: Hop requires minimal control overhead to transmit a block. At the link layer, Hop disables acknowledgements for all data frames, and only enables them to send control packets: BSYN and BACK. At the transport layer, a BACK acknowledges data in large chunks rather than in single packets. The reduced number of acknowledgement packets is shown in Figure 4.2, which contrasts the timeline for a TCP packet transmission alongside a block transfer in Hop. For large blocks (e.g. 1 MB), Hop requires orders

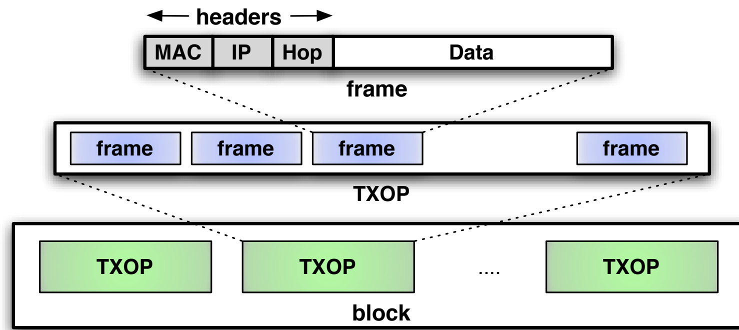


Figure 4.1. Structure of a block.

of magnitude fewer acknowledgements than for an equivalent number of packets using TCP with link-layer acknowledgements. In addition, Hop reduces idle time by ensuring that packets do not wait for link-layer ACKs, and at the transport layer by disabling rate control. Thus, Hop nearly always sends data at a rate close to the link capacity.

Spatial Pipelining: The use of large blocks and hop-by-hop reliability can hurt spatial pipelining since each node waits for the successful reception of a block before forwarding it. To improve pipelining, an intermediate hop forwards packets as soon as it receives at least a txop worth of new packets instead of waiting for an entire block. Thus, Hop leverages spatial pipelining as well as the benefits of burst transfer at the link layer.

4.3.2 Ensuring end-to-end reliability

Hop-by-hop reliability is insufficient to ensure reliable end-to-end transmission. A block may be dropped if 1) an intermediate node fails in the middle of transmitting the block to the next-hop, or 2) the block exceeds its TTL limit, or 3) a cached block eventually expires because no next-hop node is available for a long duration.

Hop uses virtual retransmissions together with in-network caching to limit the overhead of retransmitting large blocks. Hop routers store all packets that they overhear. Thus, a re-transmitted block is likely cached at nodes along the original route until the point of failure or drop, and might be partially cached at a node that is along a new path to the destination but overheard packets transmitted on the old path. Hence, instead of retransmitting the entire block, the sender sends a *virtual retransmission*, i.e., a special BSYN packet, using the same hop-by-hop reliable transfer mechanism as for a block. Virtual retransmissions exploit caching at

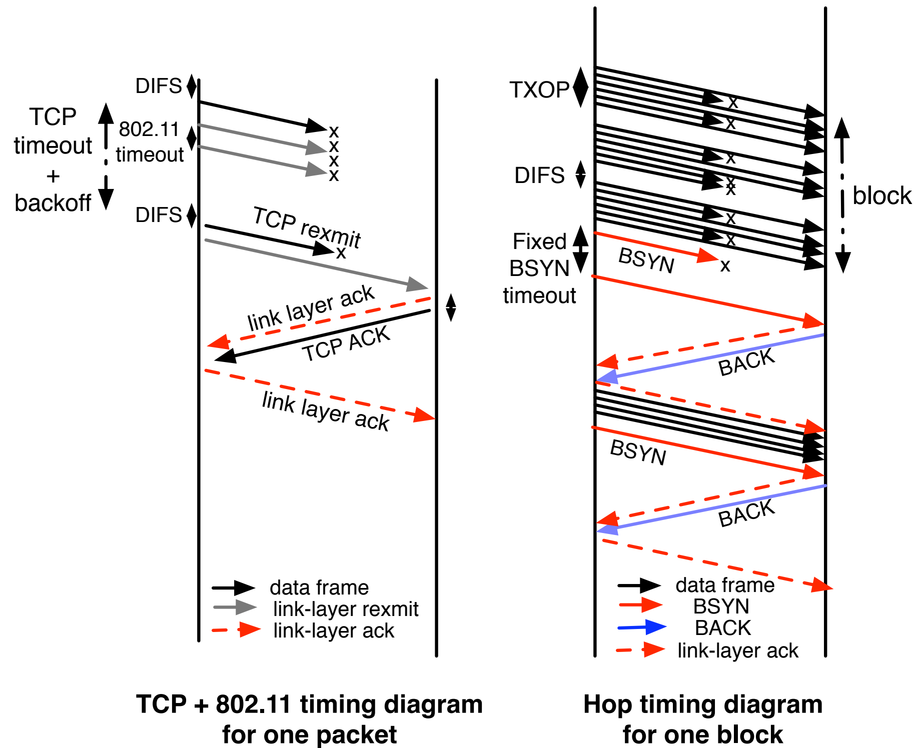


Figure 4.2. Timeline of TCP/802.11 vs. Hop

intermediate nodes by only transmitting the block (or parts of the block) when the next hop along the route does not already have the block cached as shown in Figure 4.3.

A premature timeout in TCP incurs a high cost both due to redundant transmission as well as its detrimental rate control consequence, so a careful estimation of timeout is necessary. In contrast, virtual retransmissions due to premature timeouts do little harm, so Hop simply uses the most recent round-trip time as its timeout estimate.

4.3.3 Backpressure congestion control

Rate control in response to congestion is critical in TCP to prevent congestion collapse and improve utilization. In wireless networks, congestion collapse can occur both due to increased packet loss due to contention [71], and increased loss due to buffer drops [70]. Both cases result in wasted work, where a packet traverses several hops only to be dropped before reaching the destination. Prior work has observed that end-to-end loss and delay feedback has high variance and is difficult to interpret unambiguously in wireless networks, which complicates the design of congestion control [61, 93].

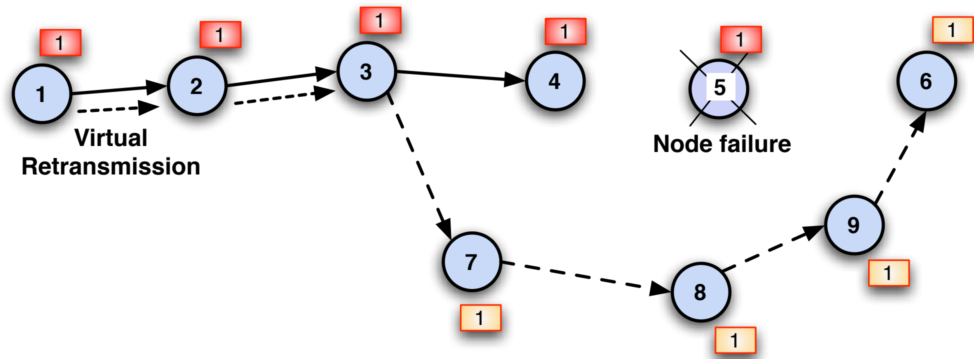


Figure 4.3. Virtual retransmission due to node failure.

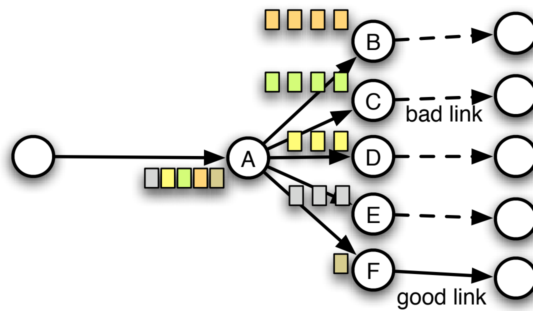


Figure 4.4. Example showing need for backpressure. Without backpressure, Node A would allocate $1/5$th of out-going capacity to each flow, resulting in queues increasing unbounded at nodes B through E. With backpressure, most data is sent to node F, thereby increasing utilization.

Hop relies only on hop-by-hop backpressure to avoid congestion. For each flow, a Hop node monitors the difference between the number of blocks received and the number reliably transmitted to its next-hop as shown in Figure 4.4. Hop limits this difference to a small fixed value, H, and implements it with no additional overhead to the BSYN/BACK exchange. After receiving H complete blocks, a Hop node does not respond to further BSYN requests from an upstream node until it has moved at least one more block to its downstream node. The default value of H is set to 1 block.

Backpressure in Hop significantly improves utilization. To appreciate why, consider the following scenario where flows $1, \ldots, k$ all share the first link with a low loss rate. Assume that the rest of flow 1's route has a similar low loss rate, while flows $2, \ldots, (k-1)$ traverse a poor route or are partitioned from their destinations. Let C be the link capacity, p_1 be the end-to-end loss observed by the first flow, and p_2 be the end-to-end loss rate observed by other flows ($p_1 \ll p_2$). Without backpressure, Hop would allocate a $1/k$

fraction of link capacity to each flow, yielding a total goodput of $C\frac{((1-p_1)+(1-p_2)\cdot(k-1))}{k}$. And the number of buffered blocks at the next-hops of the latter $k-1$ flows grows unbounded. On the other hand, limiting the number of buffered blocks for each flow yields a goodput close to $C\cdot(1-p_1)$ in this example.

Why does Hop limit the number of buffered blocks, H, to a small default value? Note that the example above can be addressed simply by choosing the block corresponding to the flow with the largest differential backlog (along A-F). Indeed, classical backpressure algorithms known to achieve optimal throughput [94] work similarly. Hop limits the number of buffered blocks to a small value in order to ensure small transfer delay for finite-sized files, as well as to limit intra-path contention.

4.3.4 Robustness to partitions

A fundamental benefit of Hop is that it continues to make progress even when the network is intermittently partitioned. Hop transfers a blocks in a hop-by-hop manner without waiting for end-to-end feedback. Thus, even if an end-to-end route is currently unavailable, Hop continues to make progress along other hops.

The ability to make progress during partitions relies on knowing which next-hop to use. Unlike typical mesh routing protocols [86, 63], routing protocols designed for disruption-tolerance expose next-hop information even if an end-to-end route is unavailable (e.g. RAPID [62], DTLSR [68]). In conjunction with such a disruption-tolerant routing protocol, Hop can accomplish data transfer even if a contemporaneous end-to-end route is never available, i.e., the network is always partitioned.

In disruption-prone networks, a Hop node may need to cache blocks for a longer duration in order to make progress upon reconnection. In this case, the backpressure limit needs to be set taking into account the fraction of time a node is partitioned and the expected length of a connection opportunity with a next-hop node along a route to the destination (see §4.5.7 for an example).

4.3.5 Handling hidden terminals

The elimination of control overhead for block transfer improves efficiency but has an undesirable side-effect — it exacerbates loss in hidden terminal situations. Hop transmits blocks without rate control or link-layer retransmissions, which can result in a continuous stream of collisions at a receiver if the senders are hidden from each other. While hidden terminals are a problem even for TCP, rate control mitigates its impact on overall throughput. Flows that collide at a receiver observe increased loss and throttle their rate. Since different flows get different perceptions of loss, some reduce their rate more aggressively than others, resulting in most flows being completely shut out and bandwidth being devoted to one or few flows [98]. Thus, TCP is highly unfair but has good aggregate throughput.

Hop uses a novel *ack withholding* technique to mitigate the impact of hidden terminals. Here, a receiver acknowledges only one BSYN packet at any time, and withholds acknowledgement to other concurrent BSYN packets until the outstanding block has completed. In this manner, the receiver ensures that it is only receiving one block from any sender at a given time, and other senders wait their turn. Once the block has completed, the receiver transmits the BACK to one of the other transmitters, which starts transmitting its block.

Although ack withholding does not address hidden terminals caused by flows to different receivers, it offers a lightweight alternative to expensive and conservative techniques like RTS/CTS for the common single-terminal hidden terminal case. The high overhead of RTS/CTS arises from the additional control packets, especially since these are broadcast packets that are transmitted at the lowest bit-rate. The use of broadcast also makes RTS/CTS more conservative since a larger contention region is cleared than typically required [106]. In contrast, ack withholding requires no additional control packets (as BSYNs and BACKs are already in place for block transfer).

4.3.6 Packet scheduling

Hop's unit of link layer transmission is a txop, which is the maximum duration for which the network interface card (NIC) is permitted to send packets in a burst without contending for access [60]. Hop's scheduler leverages the burst mode and sends a txop's worth of data from each concurrent flow at a time in a round-robin manner.

Hop traffic is isolated from delay-sensitive traffic such as VoIP or video by using link-layer prioritization. 802.11 chipsets support four priority queues—voice, video, best-effort, and background in decreasing order of priority—with the higher priority queues also having smaller contention windows [60]. Hop traffic is sent using the lowest priority background queue to minimize impact on delay-sensitive datagrams.

The design choices that we have presented so far can be detrimental to delay for small files (referred to as micro-blocks) in three ways: 1) the initial BSYN/BACK exchange increases delay for micro-blocks, 2) a sender may be servicing multiple flows, in which case a micro-block may need to wait for multiple txops, and 3) ack-withholding can result in micro-blocks being delayed by one or more large blocks that are acknowledged before its turn. Hop employs three techniques to optimize delay for micro-blocks. First, micro-blocks of size less than a fixed BSYN batch threshold (few tens of KB) are sent piggybacked with the BSYN with link-layer ARQ via the voice queue. This optimization eliminates the initial BSYN/BACK delay, and avoids having to wait for a BACK before proceeding, thereby circumventing ack-withholding delay. Second, the packet scheduler at the sender prioritizes micro-blocks over larger blocks. Finally, Hop use a block-size based ack-withholding policy that prioritizes micro-blocks over larger blocks.

4.4 Implementation

We have implemented a prototype of Hop with all the features described in §4.3. Hop is implemented in Linux 2.6 as an event-based user-space daemon in roughly 5100 lines of C code. Hop is currently implemented on top of UDP (i.e., there is a UDP header in between the IP and Hop headers in each frame in Figure 4.1). Below, we describe important aspects of Hop's implementation.

4.4.1 MAC parameters

Our implementation uses the Atheros-based wireless chipset and the Madwifi open source 802.11 device driver [81], a popular commodity implementation. By default, the MadWifi driver (as well as other commodity implementations) supports the 802.11e QoS extension. However, MadWiFi supports the extension only in the access point mode, so we modify the driver to enable it in the ad-hoc mode as well. Hop uses default 802.11 settings, except for the following. The transmission opportunity (txop) for the background queue is set to the maximum value permitted by the MadWifi driver (8160 μs or roughly 8KB of data). Link-layer ARQ is disabled for all data frames sent via Hop but enabled for control packets (BSYN, BACK, etc).

4.4.2 Hop implementation

4.4.2.0.1 Parameters A large block size increases batching benefits, so we set the default maximum block size to 1MB. Note that this means that a Hop block is allowed to be up to 1MB in size, but may be any smaller size. Hop never waits idly in anticipation of more application data in order to obtain batching benefits. The BSYN batch threshold for micro-blocks is set to a default value of 16KB, and the backpressure limit, H, is set to 1. The virtual retransmission timeout is set to an initial value of 60 seconds and simply reset to the round-trip block delay reported by the most recent block. The TTL limit for a virtual retransmissions is set to 50 hops. In the current implementation, an intermediate Hop node keeps all the blocks that it has received in memory.

4.4.2.0.2 Header format: The Hop header consists of the following fields. All frames contain the `msg_type` that identifies if the frame is a data, BSYN, BACK, virtual retransmission BSYN, or an end-to-end BACK frame; the `flow_id` that uniquely identifies an end-to-end Hop connection; and the `block_num` identifies the current block. Data frames also contain the `packet_num` that is the offset of the packet in the current block. The `packet_num` is also used to index into the bitmap returned in a BACK frame.

4.4.2.0.3 End-to-end connection management: Because Hop is designed to work in partitionable networks, it does not use a three-way handshake like TCP to initiate a connection. A destination node sets up connection state upon receiving the first block. The loss of the first block due to a node failure or expiry or

Sec.	Experiment setup	Experiment	Result: Median (Mean)
§4.5.1	One single-hop flow	Hop vs. TCP	1.6× (1.6×)
§4.5.2	One multi-hop flow	Hop vs. TCP Hop vs. Hop-by-Hop TCP Hop vs. DTN2.5	2.3× (2×) 2.5× (2×) 2.9× (3.9×)
§4.5.3	Many multi-hop flows	Hop vs. TCP Hop vs. Hop-by-Hop TCP	90× (1.25×) 20 × (1.4×)
§4.5.4	Performance breakdown	Base Hop + ack withholding + backpressure + ack withholding + backpressure	(1×) (2.5×) (3.7×) (4.8×)
§4.5.5	WLAN AP mode	Hop vs. TCP Hop vs. TCP + RTS/CTS	2.7× (1.12×) 2× (1.4×)
§4.5.6	Single small file	Hop vs. TCP	3× to 15× lower delay
	Concurrent small files	Hop vs. TCP	Comparable or lower delay
§4.5.7	Disruption-tolerance	Hop vs. DTN2.5	2.8× (2.9×)
§4.5.8	Impact on VoIP traffic	Hop vs. TCP	Slightly lower MOS score but significantly higher throughput
§4.5.9	Network and link-layer dynamics	Hop vs. TCP + OLSR Hop vs. TCP + auto-rate Hop vs. TCP + OLSR + auto-rate	4× (1×) 95× (2.4×) 5× (1.8×)
§4.5.10	Under 802.11g	Hop vs. TCP Hop vs. TCP + auto-rate	22× (1×) 6× (3×)

Table 4.1. Summary of evaluation results. All protocols above are given the benefit of burst-mode (txop) and the maximum number of link-layer retransmissions (max-ARQ) supported by the hardware.

the loss of the first end-to-end BACK is handled naturally by virtual retransmissions. In our current implementation, a Hop node tears down a connection simply by sending a FIN message and recovering state; we have not yet implemented optimizations to handle complex failure scenarios.

4.5 Evaluation

We evaluate the performance of Hop in a 20-node wireless mesh testbed. Each node is an Apple Mac Mini computer running Linux 2.6 with a 1.6 Ghz CPU, 2 GB RAM and a built-in 802.11a/b/g Atheros/MadWiFi wireless card. Each node is also connected via an Ethernet port to a wired backplane for debugging, testing, and data collection. The nodes are spread across a single floor of the UMass CS building as shown in Figure 4.5.

All experiments, except those in Section 4.5.9 and Section 4.5.10, were run in 802.11b mode with bit-rate locked at 11 Mbps. There is significant inherent variability in wireless conditions, so in order to perform a meaningful comparison, a single graph is generated by running the corresponding experiments back-to-back interspersed with a short random delay. The compared protocols are run in sequence, and each sequence is repeated many times to obtain confidence bounds.

Figure 4.5. Experimental testbed with dots representing nodes.

We compare Hop against two classes of protocols: *end-to-end* and *hop-by-hop*. The former consists of 1) UDP, and 2) the default TCP implementation in Linux 2.6 with CUBIC congestion control [101]; we did not use the Westwood+ congestion control algorithm since it performed roughly 10% worse. The latter consists of 3) Hop-by-Hop TCP, and 4) DTN2.5 [69]. Hop-by-Hop TCP is our implementation of TCP with backpressure. It splits a multi-hop TCP connection into multiple one-hop TCP connections, and leverages TCP flow control to achieve hop-by-hop backpressure. Each node maintains one outgoing TCP socket and one incoming TCP socket for each flow. When the outgoing socket is full, Hop-by-Hop TCP stops reading from the incoming socket, thereby forcing TCP's flow control to pause the previous hop's outgoing socket. This "backpressure" propagates up to the source and forces the source to slow down. DTN2.5 is a reference implementation of the IEEE RFC 4838 and 5050 from the Delay Tolerant Networking Research Group [69] that reliably transfers a bundle using TCP at each hop. Hop and UDP were set to use the same default packet size as TCP (1.5KB). In all our experiments, the delay and goodput of TCP are measured after subtracting connection setup time.

4.5.1 Single-hop microbenchmarks

In this section, we answer two questions: 1) What are the best 802.11 settings for link layer acknowledgments (ARQ) and burst mode (txop) for TCP and UDP?, 2) How does Hop's performance compare to that of TCP and UDP given the benefit of these best-case settings?

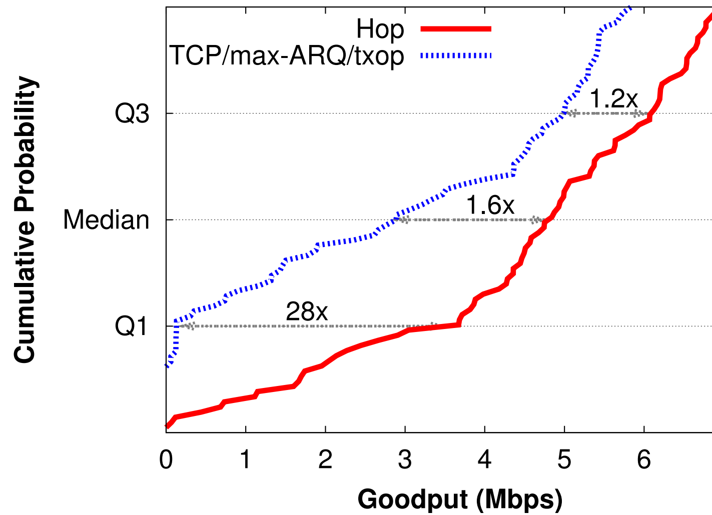


Figure 4.6. Experiment with one-hop flows. Hop improves lower quartile goodput by 28×, median goodput by 1.6×, and mean goodput by 1.6× over TCP with the best link layer settings.

4.5.1.1 Randomly picked links

In this experiment, we evaluate the single-hop performance of TCP, UDP, and Hop over 802.11 across links in our mesh testbed. The testbed has total of 56 unique links from which a random sequence of 100 links was sampled with repetition for this experiment. The average and median loss rates were 25% and 1% respectively. For each sampled link, a 10MB file is transferred using each protocol; for bad links, flows were cut off at 10 minutes, and goodput measured until the last received packet. The metric for comparison is the goodput that is measured as the total number of unique packets received at the receiver divided by the time until the last byte is received.

We compare Hop against TCP for three 802.11 settings: 1) 11 link layer retries (ARQ) with no txop, the default settings of the MadWifi driver, 2) 11 ARQ + txop, and 3) maximum permitted ARQ setting (18 for the Atheros card) + txop. We do not consider TCP with no ARQ since it (expectedly) performs poorly without 802.11 retransmissions on lossy links. We also compare against UDP under different 802.11 settings. Since UDP has no transport-layer control overhead, and transmits as fast as the card can transmit packets, it provides an upper bound on the achievable capacity on the link. For clarity of presentation, we show cumulative distributions (CDFs) for Hop and the best TCP combination and summary statistics for the other combinations.

Figure 4.6 shows that Hop significantly outperforms TCP/max-ARQ/txop, the best TCP combination. The Q1, Q2, and Q3 gains over TCP/max-ARQ/txop TCP combination are 28×, 1.6×, and 1.2× respectively. The Q1 gain is notable and shows Hop's robust performance on poor links compared to TCP.

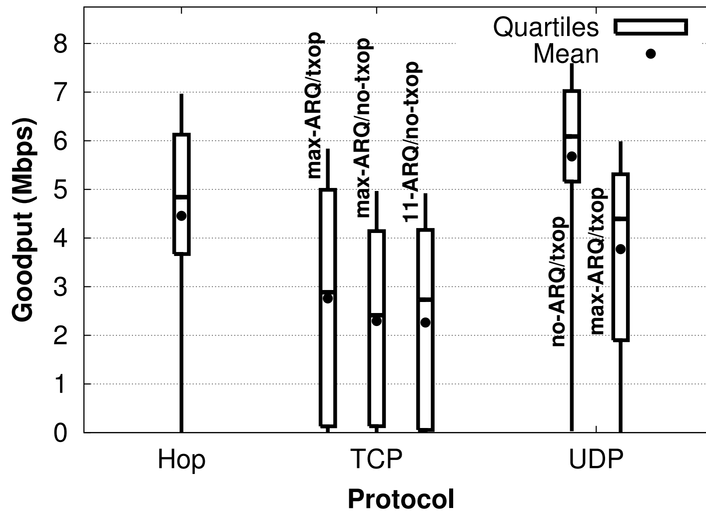


Figure 4.7. Experiment with one-hop flows. Box shows lower/median/upper quartile, lines show max/min, and dot shows mean. Increasing 802.11 ARQ limit and using txops helps TCP but Hop is still considerably better. UDP results show that ARQs incur significant performance overhead (35%). Hop is within 24% of UDP without ARQ (achievable goodput).

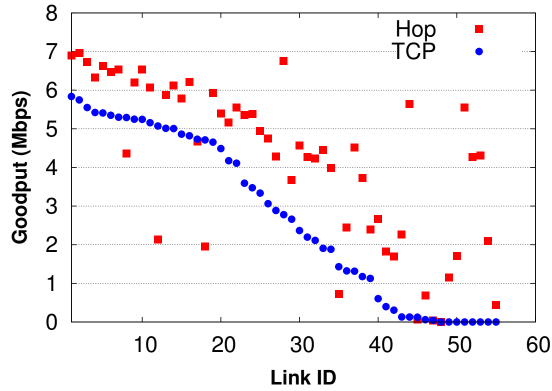
Figure 4.7 shows the summary statistics for Hop and two best TCP and UDP schemes using a box plot representation. The "box" shows the upper quartile (Q3), median (Q2) and lower quartile (Q1), and the "whiskers" show the maximum and minimum goodput. UDP/no-ARQ/txop is the best UDP combination and provides an upper bound on the achievable rate. The median Hop is about 24% lower than the achievable rate. Interestingly, turning on ARQ degrades UDP by 35% showing that ARQ in 802.11 comes at a high overhead and ARQ alone is not sufficient to fix TCP's problems.

As we find that TCP performance consistently improves by using txops and ARQ with the maximum possible limit, we give TCP and its variants the benefit of txop/max-ARQ in the rest of our evaluation.

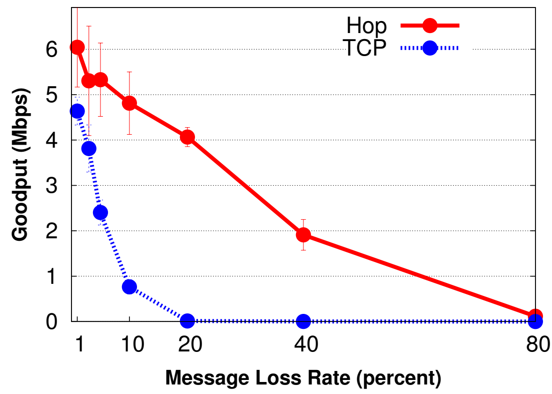
4.5.1.2 Graceful performance degradation

A key benefit of Hop is robustness, i.e., its performance gracefully degrades with increasing link losses and interference. To confirm this, we further analyze the data from the experiment in Section 4.5.1.1. Figure 4.8(a) shows the per-link throughput across the 56 links in the testbed (with multiple runs over the same link averaged) sorted by TCP goodput. Hop degrades gracefully to some of the poorest links in the testbed where TCP's throughput is near-zero. The average goodput for the worst 20 TCP flows is 334 Kbps, whereas Hop's goodput for the same flows is 2.37 Mbps, a difference of 7×.

To understand the cause of TCP's fragile behavior, we evaluate the impact of loss perceived at the transport layer on the performance of Hop and TCP. We start with a perfect link that has a near-zero loss rate and introduce loss by modifying the MadWifi device driver to randomly drop a specified fraction of incoming



(a) Sorted Single-hop Flows



(b) Impact of Loss.

Figure 4.8. Graceful degradation to adverse channel conditions. First plot shows per-link goodputs from one-hop experiment sorted in TCP order. Second plot shows controlled experiments demonstrating impact of loss. In both cases, Hop is more robust and degrades far more gracefully than TCP.

packets. Figure 4.8(b) shows that, unsurprisingly, TCP goodput drops to near-zero when loss rate is roughly 20%. Hop shows graceful near-linear degradation and is operational until the loss rate is about 80%.

4.5.2 Multi-hop microbenchmarks

How does Hop perform on multi-hop paths compared to existing alternatives? To study this question, we pick a sequence of 100 node pairs randomly with repetition from the testbed. Static routes are set up a priori between all node pairs to isolate the impact of route flux (considered in §4.5.3). The static routes were obtained by running OLSR with the default ETX metric until the routing topology stabilized at the beginning of the experiment. Among the 100 randomly chosen flows, 30% are two-hop, 30% are three-hop, 10% are four-hop, 20% are five-hop, and the remaining 10% are seven-hop flows. We compare the multi-hop goodput of Hop to TCP, Hop-by-Hop TCP, DTN2.5, and UDP.

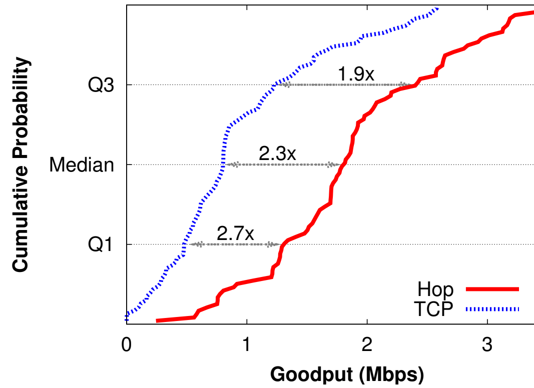


Figure 4.9. Experiment with multi-hop flows. Hop improves lower quartile goodput by 2.7×, median goodput by 2.3×, and mean goodput by 2×.

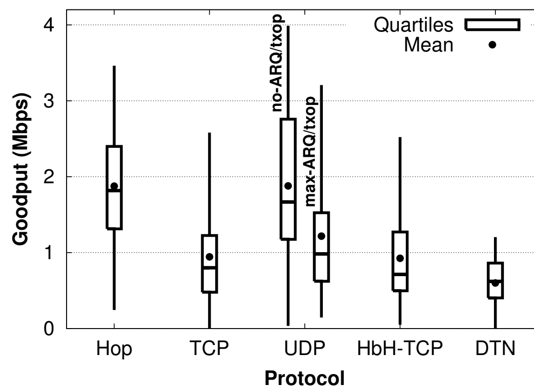


Figure 4.10. Boxplot of multi-hop single-flow benchmarks. Hop has 2-3× median, and 2-4× mean improvements over other reliable transport protocols. Hop is comparable to UDP/no-ARQ/txop in terms of median/mean — the latter is extremely fast since it has no overhead, but experiences more loss.

Figure 4.9 shows the CDF of goodput for just Hop and TCP, while Figure 4.10 shows the summary statistics for all the protocols. Hop consistently outperforms all other protocols. The Q1, Q2, and Q3 gains over TCP are 2.7×, 2.3× and 1.9× respectively. The Q1 gain over TCP is lower than for the single-hop experiment because only good links selected by OLSR are used in this experiment (as evidenced by the better performance of UDP/no-ARQ/txop compared to UDP/max-ARQ/txop). Over lossier paths, Hop's gains are much higher. We also find that the gains also grow with increasing number of hops. For example, the lower quartile gains grow from about 2.7× for two hops to more than 4× for five and six hops.

4.5.3 Hop under high load

The experiments so far considered one flow in isolation. Next, we evaluate Hop in a heavily loaded network to understand the effect of increased contention and collisions on Hop's performance and fairness.

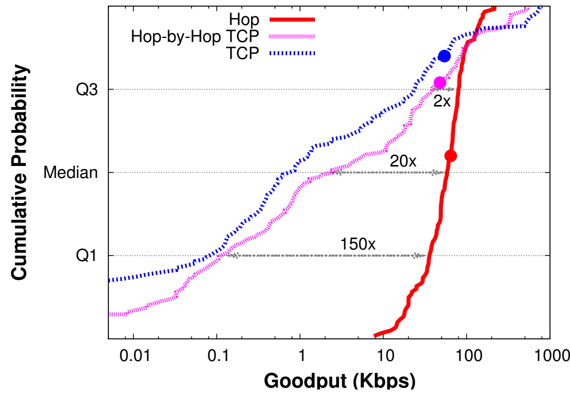


Figure 4.11. Hop for 30 concurrent flows. Dots on each line shows mean goodput. Median gains of Hop over Hop-by-Hop TCP and regular TCP are huge (20× and 90× respectively) while mean gains are modest (roughly 25% improvement).

We compare Hop, TCP, and Hop-by-Hop TCP. The experiment consists of thirty concurrent flows that transfer data continually between randomly chosen node pairs in the testbed. All protocols are run over a static mesh topology identical to Section 4.5.2. To focus on multihop benefits, we pick src-dst pairs that are not immediate neighbors of each other. We run the experiment five times, and for each run, we measure the goodputs of flows half an hour into the experiment, since the network reaches a steady state at this time.

4.5.3.1 Goodput

Figure 4.11 shows that Hop achieves a huge improvement in median goodput over TCP and Hop-by-Hop TCP. Hop achieves a median goodput of 54.9 Kbps whereas all the other protocols achieve less than 2.8 Kbps—an improvement of over an order of magnitude! Hop also improves the Q1 goodput by more than two orders of magnitude and upper quartile goodput by 2× over the other protocols. The exact numbers of Hop's median and Q1 gains over other protocols are sensitive to environmental conditions, but we consistently observe them to be large under different conditions. The figure also shows that Hop-by-Hop TCP achieves more than 4× improvement over TCP's median goodput. This shows that end-to-end rate control hurts TCP utilization and using hop-by-hop backpressure with TCP improves its performance. We also run UDP (not shown for clarity), but due to lack of congestion control, around 67% flows get zero goodput (i.e., the median is zero) and the mean goodput is 0.32Kbps.

Hop's mean gain over TCP is just 25%, which is not as impressive as the quartile gains. This is to be expected as TCP is highly unfair and starves a large number of flows to acquire the channel for only a few flows. In many cases, the top three TCP flows get around 90% of the total goodput. In contrast, Hop is significantly fairer and has higher throughput than most of the TCP flows.

	Fairness index
Hop	0.78 (0.09)
TCP	0.12 (0.04)
Hop-by-Hop TCP	0.21 (0.05)

Table 4.2. Fairness indexes for the 30 flow experiment. Parentheses show 95% confidence intervals.

4.5.3.2 Fairness

Table 4.2 shows the fairness index for different protocols. The fairness metric that we use is hop-weighted Jain's fairness index (JFI [89]). When there are n flows, with throughput x_1 through x_n and hop lengths h_1 through h_n, it is computed as follows: JFI $= \frac{(\sum_{i=1}^{n} x_i \cdot h_i)^2}{n \sum_{i=1}^{n} (x_i \cdot h_i)^2}$.

Hop is significantly fairer than both TCP-based protocols. It is noteworthy that while TCP sacrifices fairness for goodput, Hop is superior on both metrics.

4.5.4 Hop performance breakdown

How much do components of Hop individually contribute to its overall performance? To answer this question, we compare four versions of Hop: 1) the basic Hop protocol that only uses hop-by-hop block transfer, 2) Hop with ack withholding turned on, 3) Hop with backpressure turned on, and 4) Hop with both ack withholding and backpressure turned on. Since the impact of these mechanisms depends on the load in the network, we consider 10, 20 and 30 concurrent flows between randomly picked sender-receiver node pairs. A static mesh topology identical to Section 4.5.2 was used. The length of the randomly picked paths are between three and seven hops. The average path length is 3.9 hops in the 10 flow case, 4 hops in the 20 flow case, and 3.9 hops in the 30 flow case. Each flow transmits a large amount of data, and we take a snapshot of the measurements after half an hour.

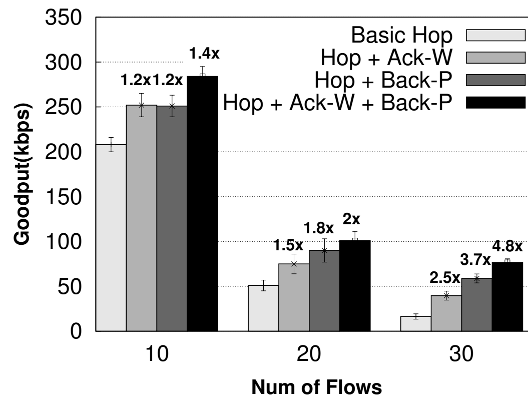


Figure 4.12. Hop performance breakdown showing contribution of ack withholding and backpressure. Ack withholding and backpressure improve Hop's performance by more than 4.8x under high load.

Figure 4.12 shows the performance of the different schemes. The benefit of ack withholding and backpressure increases with network load. In the 10 flow case, both ack withholding and backpressure increase goodput by around 20%. With greater network load, congestion increases dramatically, hence the gains due to backpressure is more than due to ack withholding. For example, in the 30 flow case, Hop with backpressure yields 3.7× improvement over basic Hop, whereas Hop with ack withholding yields 2.5× improvement. Furthermore, the benefits of using both backpressure and ack withholding are considerably more than using either one of them. For instance, the full-fledged Hop yields 4.8× improvement over basic Hop for the 30 flow case.

4.5.5 Hop with WLAN access points

Next, we evaluate how ack withholding in Hop compares to the 802.11 RTS/CTS mechanism for dealing with hidden terminals. We emulate a typical one-hop WiFi network where a number of terminals connect to a single access point. We setup a 7-to-1 topology for this experiment, by selecting a node in the center of our testbed to act as the "AP node", and transmitting data to this node from all its seven neighbors. Among the seven transmitters, six pairs were hidden terminals (i.e. they could not reach each other but could reach the AP). We verified this by checking to see if they could transmit simultaneously without degradation of throughput. In each run, the nodes transmit data continually, and we measure goodput after 30 minutes when the flow rates have stabilized.

	Mean	Median	Fairness
Hop	663 (24)	652 (33)	0.93 (0.01)
TCP	587 (88)	244 (142)	0.35 (0.06)
TCP + RTS/CTS	463 (20)	333 (87)	0.4 (0.05)

Table 4.3. Mean/median goodput and Fairness for a many-to-one "AP" setting. 95% confidence intervals shown in parenthesis

We compare Hop against TCP both with and without 802.11 RTS/CTS enabled. The results are presented in Table 4.3, and show that Hop beats TCP with or without RTS/CTS both in throughput and fairness. While the mean gains over TCP without RTS/CTS are only 12%, the median improvement is about 2.7×. TCP has a crafty way of maintaining high aggregate goodput amidst hidden terminals by squelching all but one of the flows and in effect serializing them. In contrast, Hop achieves almost perfectly fair allocation across the different flows. The addition of RTS/CTS to TCP hurts aggregate throughput but improves median throughput and fairness. However, Hop achieves 1.4× the aggregate throughput, 1.96× the median throughput, in addition to hugely improving fairness over TCP with RTS/CTS.

4.5.6 Hop delay for small file transfers

How does Hop impact the delay incurred by micro-blocks (small files)? Recall that Hop uses two mechanisms to speed micro-block transfers: 1) It piggybacks micro-blocks less than 16KB in size with the initial BSYN to reduce connection setup overhead, 2) It's ack withholding mechanism prioritizes micro-blocks.

4.5.6.1 Single-hop transfer delay for small files

First, we evaluate the benefits of Hop's size-aware ack withholding policy. To evaluate this, we pick a one-hop Wifi network where five nodes are connected to an AP (similar setup as our WLAN experiments). In each experiment, one of the five nodes (randomly chosen), transmits a micro-block to the AP at a random time, whereas the other four nodes continually transfer large amounts of data. Each experiment runs until the micro-block completes, at which point we compute the delay for the transfer. We compare against TCP with and without RTS/CTS, and report aggregate numbers over five runs. Figure 4.13 shows that the transfer delay of the micro-block with Hop is always lower than for TCP (with or without RTS/CTS). In many cases, the delay gains are significant, e.g., for file sizes less than 16KB, the gains range from 3× to 15×. This experiment shows that Hop can be used for delay-sensitive transfers like web transfers, ssh, and SMS in many-to-one AP settings.

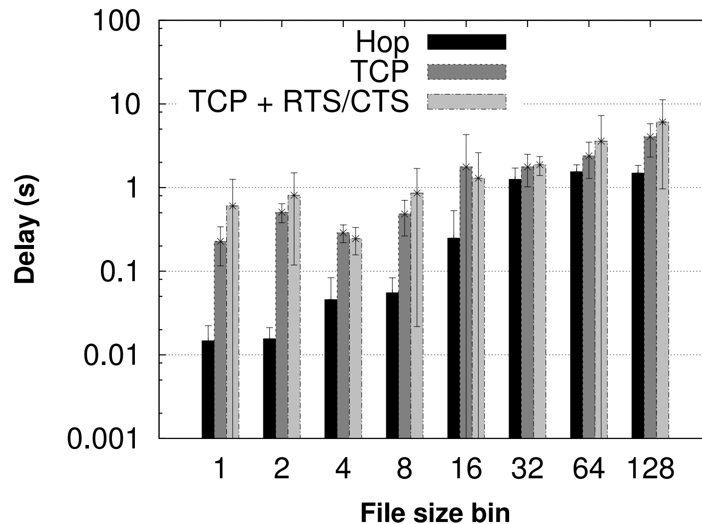


Figure 4.13. Hop for WLAN: Hop improves delay for all file sizes with improvements between 3-15×

4.5.6.2 Multi-hop transfer delay for Web file sizes

Next, we evaluate Hop and TCP over a larger workload that comprises predominantly of micro-blocks. (We do not consider TCP with RTS/CTS enabled, since it consistently introduces more delay.) In particular,

we consider a Web traffic pattern where most files are small web pages [65]. The flow sizes used in this experiment were obtained from a HTTP proxy server trace obtained from the IRCache project [72]. The CDF obtained was sampled to obtain the representative flow sizes used in this experiment. The distribution of file sizes is as follows: roughly 63% of the files are less than 10KB, 25% are between 10KB-100KB, and remaining are greater than 100KB. To stress multi-hop performance, the sender and receiver for each flow are chosen randomly among the node-pairs that were multiple hops away in our mesh network. Flows followed a Poisson arrival pattern with $\lambda = 2$ flows per second. We present results from 100 flows aggregated in bins of size $[2^{n-1}, 2^n]$ except the bins at the edge, *i.e.* $\leq$2KB, and $\geq$512.

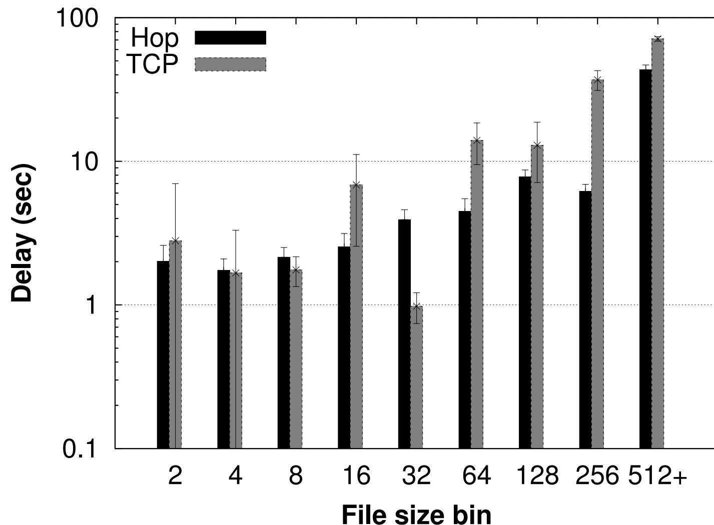


Figure 4.14. Performance for web traffic: Except the 32KB bin, Hop has comparable or better delay, with gains upto 6×

Figure 4.14 shows that Hop has less or comparable delay to TCP for almost all file sizes except those between 16K-32K. This dip occurs because 16KB is our threshold for piggybacking data with BSYNs. This suggests that a slightly larger threshold might be more effective, but we leave the optimization for future work. For other bins, delay with Hop is mostly lower than TCP (between 19% higher to 6× lower than TCP), demonstrating its benefits for micro-block transfer. Detailed file size microbenchmarks in isolation (i.e., without concurrent transfers) show a similar behavior.

4.5.7 Robustness to partitions

A key strength of Hop is its ability to operate even under disruptions unlike end-to-end protocols such as TCP. We now evaluate how, in a partitioned scenario, Hop compares to hop-by-hop schemes such as DTN2.5 that are designed primarily for disruption-tolerance. In this experiment, we pick a seven hop path and simulate a partition scenario by bringing down the third node and fifth node in succession along the path

for one minute each in an alternating manner. Table 4.4 shows the goodput obtained by Hop averaged over five runs under two different backpressure settings: 1) backpressure limit (H) is set to 1 and 2) backpressure limit is set to 100. Hop outperforms DTN2.5, a protocol specifically designed for partitioned settings, by $2\times$ when $H = 1$, and $3\times$ when $H = 100$. The results show that Hop achieves excellent throughput under partitioned settings, and a large backpressure limit improves throughput by about 15%. This result is intuitive as having a larger threshold enables maximal use of periods of connectivity between nodes. In contrast to Hop, TCP achieves zero throughput since a contemporaneous end-to-end path is never available.

	Goodput (Kbps)
Hop w/ H=1	320 (29)
Hop w/ H=100	457 (18)
DTN2.	159 (15)

Table 4.4. Goodput achieved by Hop and DTN2.5 in a partitioned network without an end-to-end path.

4.5.8 Hop with VoIP

In this experiment, we quantify the impact of Hop and TCP on Voice-over-IP (VoIP) traffic. We use two metrics: 1) the mean opinion score (MoS) to evaluate the quality of a voice call, and 2) the conditional loss probability (CLP) to measure loss burstiness. The MoS value can range from 1-5, where above 4 is considered good, and below 3 is considered bad. The MOS score for a VoIP call is estimated as in [67]. The CLP is calculated as the conditional probability that a packet is lost given that the previous packet was also lost.

The experiment consists of a single VoIP flow and multiple Hop/TCP flows that transmit data continually over randomly picked 3-hop paths in the testbed. We emulate the VoIP flow as a stream of 20 byte packets with data rate at 8 Kbps. We evaluate two cases: one VoIP flow with five Hop/TCP flows, and one VoIP flow with ten Hop/TCP flows.

Table 4.5 shows that Hop achieves significantly better throughput than TCP (in terms of median/mean) but has more impact on the quality of VoIP calls. This is to be expected as TCP starves most of its flows as evidenced by the abysmal median throughput (1-2 Kbps), and therefore has lower impact on the VoIP flow. In contrast, Hop obtains median throughput of a few hundreds of Kbps, while sacrificing a little VoIP quality. We believe that even this discrepancy can be reduced by exploiting 802.11e to set larger contention window parameters to the background queue (e.g. higher backoff), but have not experimented with this so far.

Load		Goodput (Kbps)	CLP	MOS
5 flows	Hop	Median: 468.5 Mean: 1474 (51)	0.37	4.12
	TCP	Median: 2 Mean: 1372 (14)	0.48	4.19
10 flows	Hop	Median: 184 Mean: 336 (24.8)	0.57	3.92
	TCP	Median: 1.7 Mean: 260 (8.5)	0.31	4.16

Table 4.5. Impact of Hop and TCP on VoIP flows. Result shows the median/mean goodput, conditional loss probability, and MOS for VoIP with 95% confidence intervals in parentheses.

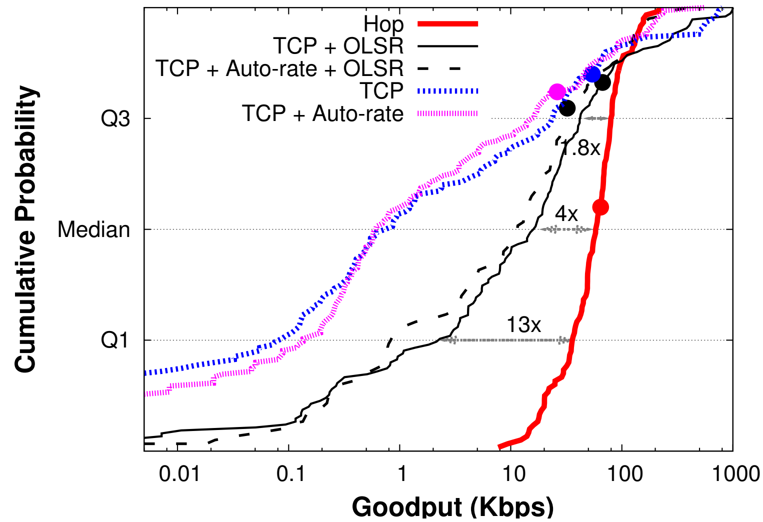


Figure 4.15. Hop for 30 concurrent flows under dynamic routing and auto bit-rate. Dots on each line shows mean goodput. Median gains by Hop with fixed bit-rate are around 4× over TCP with OLSR and more than 90× over TCP with static routing.

4.5.9 Network and link layer dynamics

Our experiments so far were run with static routes and with a fixed wireless bit-rate. Now, we evaluate the impact of dynamic routing using OLSR and auto bit-rate control using the default Madwifi *Sample* algorithm. We run TCP under all four combinations of static/dynamic routes and fixed/auto bit-rate selection. We compare these to Hop with a fixed bit-rate and static/dynamic routes. We are unable to evaluate Hop with auto-rate control as the current implementation of Hop disables link-layer ARQs that auto-rate control requires to estimates link quality. As in Section 4.5.3, we consider thirty concurrent long-lived flows between randomly chosen node pairs, and run the experiment five times.

Figure 4.15 shows that Hop is better than TCP across all combinations, with median gains of 4× over the best of them. (Hop behaves almost identically with dynamic or static routes, therefore we only show

the static case in the figure.) Surprisingly, we see that the best combination for TCP is with OLSR and fixed bit-rate. OLSR significantly improves TCP's median goodput or fairness, thereby reducing Hop's gain over TCP in comparison to the static case (Section 4.5.3). OLSR benefits TCP as it constantly changes the routing topology with concurrent TCP flows, which makes high goodput flows backoff and yield transmission opportunities to the previously low goodput flows. While the constant shuffling of flows increases TCP's median goodput, OLSR's impact on TCP's mean goodput is small (25%) because the links in the network are already heavily loaded. Auto-rate control makes almost no improvement to TCP since the testbed remains well-connected at 11 Mbps, and hence OLSR choses good links at this bit-rate.

4.5.10 Hop under 802.11g

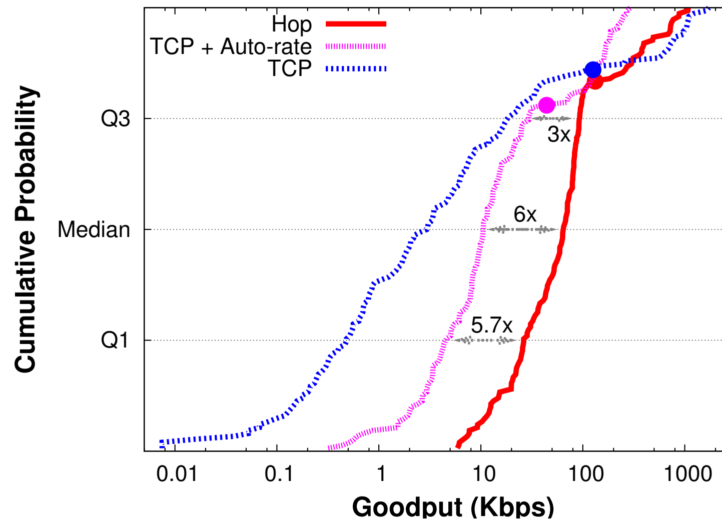


Figure 4.16. Hop for 30 concurrent flows under 802.11g. Dots on each line shows mean goodput. Hop's median gain is 22× over TCP with bit-rate fixed at 24Mbps, and is 6× over TCP with auto-rate control. Hop's mean gain is 3× over TCP with auto-rate control.

All of our experiments so far were done with 802.11b. How does Hop perform under higher bit-rates obtained using 802.11g? To answer this question, we consider an experiment similar to that in Section 4.5.3 with thirty long-lived concurrent flows between randomly chosen node pairs. We use a subset of our testbed (15 nodes) for this experiment as many nodes get disconnected under 802.11g. We ran this experiment with a static routing topology obtained by running OLSR under 802.11g. We consider Hop and TCP with a fixed 802.11g bit-rate of 24 Mbps that yields a reasonably connected topology, as well as TCP with auto-rate control.

Figure 4.16 shows that Hop improves median goodput by 6× over TCP with auto-rate control and by 22× over TCP with fixed bit-rate. The gains over TCP with auto-rate are lower than in the case of our

802.11b experiments in Section 4.5.3 because the maximum bit-rate in 802.11g is higher than the selected fixed bit-rate of 24 Mbps. Thus, TCP with auto-rate control can take advantage of the fact that the maximum bit-rate on 802.11g links is 54 Mbps, whereas Hop's bit-rate is fixed at 24 Mbps. As a result, the highest goodput achieved by a flow that uses TCP with auto-rate control is 23 Mbps, which is higher than Hop's maximum goodput of 16 Mbps. The fact that Hop shows considerable benefits despite using a static best bit-rate suggests that Hop with a good bit-rate selection scheme can benefit even more.

Figure 4.16 also shows that auto-rate control improves TCP's fairness (median goodput increases by 3.2×) but hurts network utilization (mean goodput decreases by 65%). This is because auto-rate improves the low goodput flows over lossy links by reducing the bit-rate (and thereby the loss rate), but impacts high goodput flows as flows over low bit-rate links are slow and consume a large portion of transmission opportunities.

4.5.11 Discussion: Hop vs. TCP

Although the above results show Hop's benefits across a wide range of scenarios, our evaluation has some limitations. First, our results are based on a 20-node indoor testbed, so we can not claim that they will hold in other wireless mesh networks. For example, it is conceivable that the benefits due to ack withholding are because of hidden terminals specific to our testbed's topology. Nevertheless, our experience with Hop has been encouraging. Over the last few months, we have experimented with different node placements, static topology configurations, and diurnal as well as seasonal variations in cross traffic and channel conditions, and have seen results consistent with those described in this thesis. Second, we have not compared Hop to a large number of proposed TCP modifications for multi-hop wireless networks for which implementations are not available (refer §4.6.1). We present Hop as a simple and robust alternative to end-to-end rate control schemes, but do not claim that end-to-end rate control can not be fixed to obtain comparable benefits at least in well-connected environments.

TCP's strengths are undeniable. Under high load, it is difficult to outperform TCP significantly in terms of aggregate throughput (refer Figures 4.11 and 4.16). TCP backs off aggressively on bad paths reducing contention for flows on good paths resulting in an efficient but unfair allocation. TCP has a similar effect on hidden terminals—by squelching most of the colliding flows, TCP in effect unfairly serializes them but ensures high throughput. Finally, despite its many woes in wireless environments, TCP enjoys the luxury of experience through widespread deployment, setting a high bar for alternate proposals.

Hop is not designed to be TCP-friendly. For example, in the 30 flow scenario, if we convert just 7 of the 30 TCP flows to use Hop instead of TCP, the median goodput of the remaining 23 drops by an order of

magnitude. This is unsurprising as Hop's bursty traffic increases the loss and contention perceived by TCP flows causing them to aggressively back off.

4.6 Related work

Wireless transport, especially the performance and fairness of TCP over 802.11, has seen large body of prior work. Our primary contribution is to draw upon this work and show that reliable per-hop block transfer is a better building block for wireless transport through the design, implementation, and evaluation of Hop.

4.6.1 Proposed alternatives to TCP

TCP performance: TCP's drawbacks in wireless networks include its inability to disambiguate between congestion and loss [61], and its negative interactions with the CSMA link layer. Proposed solutions include: 1) end-to-end approaches that try to distinguish between the different loss events [87], attempt to estimate the rate to recover quickly after a loss event [82], or reduce TCP congestion window increments to be fractional [84], 2) network-assisted approaches that utilize feedback from intermediate nodes, either for ECN notification [99], failure notification [80] or for rate estimation [93], and 3) link-layer solutions that use a fixed window TCP in conjunction with link-layer techniques such as neighborhood-based Random Early Detection ([70]) or backpressure flow control (RAIN [79]) to prevent losses due to link queues filling up.

TCP fairness: TCP unfairness over 802.11 stems primarily from: 1) excess time spent in TCP slow-start, which is addressed in [93] by use of better rate estimation, and 2) interactions between spatially proximate interfering flows [98, 90] by using neighborhood-based random early detection and rate control techniques.

In comparison to the above schemes, Hop does not rely on end-to-end rate control, and thereby eliminates the complex interaction between TCP and 802.11 that is the root of its performance and fairness problems. Instead, Hop uses simple mechanisms—batching, hop-by-hop backpressure and ack withholding—to improve performance as well as fairness. Hop requires no modifications to the 802.11 MAC protocol.

4.6.2 Implemented alternatives to TCP

Few implemented alternatives to TCP are available for reliable transport in 802.11 networks today. At the time of writing, we found only two such implementations—TCP Westwood+ and DTN2.5—both of which we compare against Hop. Hop's use of hop-by-hop reliability and backpressure is similar to a recent proposal, CXCC [92], but differs in its use of burst-mode, ack withholding, virtual retransmissions, etc. We could not compare Hop against CXCC as it is not implemented for 802.11.

Two recent systems, WCP [91] and Horizon [107], also address TCP's performance and fairness problems over 802.11. WCP, similar in spirit to NRED [98], augments TCP's end-to-end rate control with network-

assisted feedback about contention along the path. WCP shows significant gains in median throughput (or fairness) under load, but often reduces the mean throughput considerably. Horizon uses backpressure scheduling with multi-path routing as a shim between unmodified TCP and 802.11 layers, and shows improved fairness under load in a majority of experimental runs at the cost of mean throughput. In comparison, Hop consistently shows significant improvement in fairness and mild improvement in mean throughput under load. Although we have not performed a head-to-head comparison to Hop, we note that both WCP and Horizon rely on link-layer ARQ per frame that our experiments (Figures 4.7 and 4.10) suggest are inefficient for lossy wireless links.

4.6.3 Other related work

Backpressure: Backpressure was first investigated in ATM [108] and high-speed networks [83] to handle data bursts. A seminal paper by Tassiulas and Ephremides [94] showed that backpressure scheduling can achieve the stable capacity region of a wireless network. This paper sparked off a large body of theoretical work [112] on optimal scheduling, routing, and flow control in wireless networks. However, backpressure scheduling is NP-hard, incurs a high signaling overhead per transmission, and is difficult to implement with the 802.11 MAC layer, so few practical implementations exist.

In recent times, backpressure-like ideas have been adapted for congestion control as an alternative to TCP [92] or underneath TCP [79, 107, 109]; for unreliable hierarchical data aggregation in sensor networks [71]; for reliable bulk transport in linear sensor networks and a single flow [76], etc. Similar to Hop DiffQ[109] uses backpressure congestion control for multi-hop wireless networks. However, Hop performs backpressure over blocks to amortize the signaling overhead, uses ack withholding to to alleviate hidden terminal losses, and uses per-hop reliability with virtual retransmissions to efficiently deal with in-network losses.

Batching: The idea of batching frames to reduce wireless overhead has been employed in much existing work. FRJ[110] optimizes link-layer performance by combining frame batching, partial packet recovery, and bit-rate adaption techniques. Ng et al. [85] show that adapting the burst size of txop's in 802.11e to the load can improve TCP fairness in WLAN settings. WildNet [88] leverages batching with FEC and bulk acknowledgments at the link layer over long-distance unidirectional 802.11 links. Kim et al. [111] aggregate TCP frames using the 802.11n burst mode to amortize the MAC protocol overhead. In comparison, Hop jointly leverages batching both at the link and transport layers.

4.7 Conclusions

The last decade has seen a huge body of research on TCP's problems over wireless networks, but TCP for good reasons continues to to be the dominant real-world alternative today. One reason may be that TCP

is good enough in the common case of wireless LANs, and solutions proposed for more challenged environments do not perform well in the common case. A natural question is if we can have one simple transport protocol that yields robust performance across diverse networks such as WLANs, meshes, MANETs, sensornets, and DTNs. Our work on Hop suggests that this goal is achievable. Hop achieves significant throughput, fairness, and delay gains both in well-connected WLANs and mesh networks as well as disruption-prone networks.

CHAPTER 5

CONCLUSIONS

In this thesis, we examine the problems in data management and wireless transport on diverse sensor networks. Specifically, we focus on two topics, i.e., how to explore the resources on the edge of the network to achieve energy efficiency and data accuracy simultaneously, and how to optimize diverse user needs on the same sensor network in the presence of limited resources.

To explore the resources on the edge of the network, we proposed PRESTO, a model-driven predictive data management architecture for hierarchical sensor networks. PRESTO makes intelligent use of proxy and sensor resources to balance the needs for low-latency, interactive querying from users with the energy optimization needs of the resource-constrained sensors. Our experiments showed that PRESTO yields an order of magnitude improvement in the energy required for data and query management, simultaneously building a more accurate model than other existing techniques.

To optimize diverse user needs, we proposed a utility-driven approach, MUDS, to maximize data sharing across users while judiciously using limited network and computational resources. MUDS addresses three key challenges: how to define utility functions for networks with data sharing among end-users, how to compress and prioritize data transmissions according to its importance to end-users, and how to gracefully degrade end-user utility in the presence of bandwidth fluctuations. We instantiated this architecture in the context of geographically distributed wireless radar sensor networks for weather. The trace-driven simulations show that MUDS significantly improves system utility than the non-data-sharing approaches.

To optimize wireless transport, we presented Hop a fast, robust and simple wireless transport protocol. Hop is fast because it reduces sources of overhead for reliable per-hop block transfer, eliminates noisy end-to-end rate control, and nearly always send at link capacity. Hop is robust because it operates under high route flux and makes progress even in partitioned topologies. Hop is simple because it eliminates many complex interactions between the transport layer and 802.11. Our experiments on a real wireless mesh testbed show that Hop achieves orders of magnitude higher goodput than the existing reliable transport protocols.

BIBLIOGRAPHY

[1] G. E. P. Box and G. M. Jenkins. *Time Series Analysis*. Prentice Hall, 1991.

[2] Emstar: Software for wireless sensor networks. `http://cvs.cens.ucla.edu/emstar/`.

[3] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proc. ACM SenSys*, 2004.

[4] D. Tulone and S. Madden. PAQ: Time Series Forecasting For Approximate Query Answering in Sensor Networks In *Proc. EWSN*, 2006.

[5] Pavan Edara, Ashwin Limaye, and Krithi Ramamritham Asynchronous in-network prediction: Efficient aggregation in sensor networks In *ACM Transactions on Sensor Networks*, 2008

[6] D. Chu, A. Deshpande, J. Hellerstein and W. Hong. Approximate Data Collection in Sensor Networks Using Probabilistic Models. In *Proc. ICDE*, 2006.

[7] Woochul Kang, S.H. Son, and J.A. Stankovic PRIDE: A Data Abstraction Layer for Large-Scale 2-tier Sensor Networks, In *SECON*, 2009.

[8] A. Meliou, C. Guestrin and J. Hellerstein. Approximating Sensor Network Queries Using In-network Summaries In *Proc. IPSN*, 2009

[9] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. VLDB*, 2004.

[10] P. Desnoyers, D. Ganesan, and P. Shenoy. Tsar: A two tier storage architecture using interval skip graphs. In *Proc. ACM SenSys.*, 2005.

[11] A. Arora, et al. ExScal: Elements of an Extreme Scale Wireless Sensor Network. In *The 11th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005

[12] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proc. ACM SenSys*, 2004.

[13] J. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. In *Proc. the IEEE Intl. Conf. on Systems and Network Management*, 1999.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. ASPLOS-IX*, 2000.

[15] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. Mobicom*, 2000.

[16] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proc. Mobicom*, 2000.

[17] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acqusitional query processing system for sensor networks. In *ACM Transactions on Database Systems*, 2005.

[18] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[19] G. Mathur, P. Desnoyers, D. Ganesan and P. Shenoy. Ultra-low Power Data Storage for Sensor Networks. In *Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS)*, 2006.

[20] M. Philipose, K. P. Fishkin, M. Perkowitz, D. J. Patterson1, D. Hahnel, D. Fox, and H. Kautz. Inferring adls from interactions with objects. *IEEE Pervasive Computing*, 3(4):50–56, 2003.

[21] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. ACM SenSys*, 2004.

[22] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS)*, 2005.

[23] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. In *Proc. Hot Chips 16: A Symposium on High Performance Chips*, 2004.

[24] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT - a geographic hash-table for data-centric storage. In *First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[25] Stargate platform. `http://platformx.sourceforge.net/`.

[26] Center for Embedded Networked Sensing (CENS) - James Reserve Data Management System. `http://dms.jamesreserve.edu/`.

[27] Y. Yao and J. E. Gehrke. The cougar approach to in-network query processing in sensor networks. In *Sigmod Record*, 31(3), 2002.

[28] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proc. IEEE Infocom*, 2002.

[29] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. ACM SenSys*, 2003.

[30] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11b mesh network. In *Proc. SIGCOMM*, 2004.

[31] F. Bian, D. Kempe, et al. Utility based sensor selection. In *Proc. IPSN*, 2006.

[32] V. Bychkovsky, K. Chen, M. Goraczko, A. Miu, E. Shih, Y. Zhang, et al. Cartel: A distributed mobile sensor computing system. In *Proc. SenSys*, 2006.

[33] `http://www.caps.ou.edu/`. CAPS: Center for Analysis and Prediction of Storms.

[34] K. Chebrolu, B. Raman, and S. Sen. Long-distance 802.11b links: performance measurements and experience. In *Proc. MOBICOM*, 2006.

[35] P. R. Desrochers and S. Y. Yee. Wavelet-based algorithm for mesoCyclone detection. In *Proc. SPIE*, 1997.

[36] B. Donovan, D. J. McLaughlin, J. Kurose, et al. Principles and design considerations for short-range energy balanced radar networks. In *Proc. IGARSS*, 2005.

[37] `http://www.earthscope.org`.

[38] J. E. Fowler. QccPack: an open-source software library for quantization, compression and coding. In *Proc. SPIE*, 2000.

[39] R. Fritchie, K. K. Droegemeier, et al. Detection of hazardous weather phenomena using data assimilation techniques. In *32nd Conference on Radar Meteorology*, 2005.

[40] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley. Overlay TCP for multi-path routing and congestion control. In *Proc. of IMA Workshop on Measurements and Modeling of the Internet*, 2004.

[41] K. Jain, J. Padhye, V. N. Padmanabhan, and L. Qiu. Impact of interference on multi-hop wireless network performance. *Wireless Networks*, 2005.

[42] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. In *Journal of the Operational Research Society*, volume 49, 1998.

[43] S. Liu, M. Xue, and Q. Xu. Using wavelet analysis to detect tornadoes from doppler radar radial-velocity observations. In *Journal of Atmospheric Ocean Technology*, 2006.

[44] G. Mainland, D. Parkes, and M. Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proc. NSDI*, May 2005.

[45] R. Muller, G. Alonso, and D. Kossman. Efficient sharing of sensor networks. In *Proc. MASS*, 2006.

[46] R. Patra, S. Nedevschi, et al. WiLDNet: design and implementation of high performance WiFi-based long distance networks. In *Proc. NSDI*, 2007.

[47] B. Philips, D. Pepyne, et al. Integrating end user needs into system design and operation: the center for collaborative adaptive sensing of the atmosphere (CASA). In *Proceedings of the 87th AMS Annual Meeting, San Antonio, TX, USA*, Jan. 2007.

[48] D. Pepyne, D. Westbrook, B. Philips, E. Lyons, M. Zink, and J. Kurose Distributed Collaborative Adaptive Sensor Networks for Remote Sensing Applications In *Proc. American Control Conference*, 2008.

[49] M. Zink, E. Lyons, D. Westbrook, J. Kurose Closed-loop Architecture for Distributed Collaborative Adaptive Sensing of the Atmosphere: Meteorological Command & Control In *International Journal of Sensor Networks*, Vol 6, 2009.

[50] Z. Zhang, A. Kshemkalyani, S. Shatz Multi-root, Multi-Query Processing in Sensor Networks In *DCOSS*, 2008.

[51] X. Lu, M. Spear, K. Levitt, Wu. Felix Non-uniform Entropy Compression for Uniform Energy Distribution in Wireless Sensor Networks In *SENSORCOMM*, 2008.

[52] C. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proc. SenSys*, 2006.

[53] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6:243–250, 1996.

[54] S. Li, W. Li Shape-adaptive discrete wavelet transforms for arbitrarily shaped visual object coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 10: 725–743, 2000

[55] N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *Proc. DCOSS*, 2005.

[56] M. Zink, D. Westbrook, S. Abdallah, B. Horling, V. Lakamraju, E. Lyons, V. Manfredi, J. Kurose, and K. Hondl Meteorological Command and Control: An End-to-end Architecture for a Hazardous Weather Detection Sensor Network. In *Proc. EESR*, 2005.

[57] M. Zink, D. Westbrook, et al NetRad: Distributed, Collaborative and Adaptive Sensing of the Atmosphere. Calibration and Initial Benchmarks In *Proc. DCOSS*, 2005.

[58] J. Kurose, E. Lyons, D. McLaughlin, D. Pepyne, B. Philips, D. Westbrook, M. Zink An End-User-Responsive Sensor Network Architecture for Hazardous Weather Detection, Prediction and Response In *Proc. AINTEC*, 2006.

[59] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to algorithms *The MIT Press*, 2001.

[60] http://standards.ieee.org/getieee802/download/802.11e-2005.pdf. 802.11e: Quality of Service enhancements to 802.11.

[61] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. In *SIGCOMM*, 1996.

[62] A. Balasubramanian, B. Levine, and A. Venkataramani. Dtn routing as a resource allocation problem. *SIGCOMM*, 2007.

[63] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Morris. Architecture and Evaluation of an Unplanned 802.11b Mesh Network. In *MobiCom*, 2005.

[64] Sanjit Biswas and Robert Morris. Exor: opportunistic multi-hop routing for wireless networks. *SIGCOMM Comput. Commun. Rev.*, 35(4):133–144, 2005.

[65] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. *INFOCOM*, 1999.

[66] David L. Clark, Mark M. Lambert, and Lixia Zhang. RFC 998: Netblt: A bulk data transfer protocol, March 1987.

[67] R. G. Cole and J. H. Rosenbluth. Voice over ip performance monitoring. *SIGCOMM Comput. Commun. Rev.*, 2001.

[68] M. Demmer and K. Fall. Dtlsr: Delay tolerant routing for developing regions. *NSDR*, 2007.

[69] http://www.dtnrg.org/. Delay Tolerant Networking (DTN) Reference Group.

[70] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on tcp throughput and loss. In *INFOCOM'03*, 2003.

[71] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *SenSys*, pages 134–147, New York, NY, USA, 2004. ACM Press.

[72] http://www.ircache.net/. IRCache: The NLANR Web Caching Project.

[73] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *SIGCOMM*, 2004.

[74] S. Katti, D. Katabi, H. Balakrishnan, and M. Medard. Symbol-level network coding for wireless mesh networks. *SIGCOMM*, 2008.

[75] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. Xors in the air: practical wireless network coding. *SIGCOMM*, 2006.

[76] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. Flush: a reliable bulk transport protocol for multihop wireless networks. In *SenSys*, 2007.

[77] S. Kopparty, S. Krishnamurthy, M. Faloutsos, and S. Tripathi. Split-tcp for mobile ad hoc networks. In *IEEE GLOBECOM*, 2002.

[78] M. Li, D. Agrawal, D. Ganesan, A. Venkataramani, and H. Agrawal. Block-switched networks: A new paradigm for wireless transport. Technical Report TR 08-27, UMass Amherst, May 2006.

[79] Chaegwon Lim, Haiyun Luo, and Chong-Ho Choi. RAIN: A reliable wireless network architecture. In *Proceedings of IEEE ICNP*, 2006.

[80] S. Liu, J.; Singh. ATCP: Tcp for mobile ad hoc networks. *IEEE JSAC*, 2001.

[81] http://www.madwifi.org/. Madwifi Device Driver.

[82] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *Mobicom*, 2001.

[83] Partho P. Mishra and Hemant Kanakia. A hop by hop rate-based congestion control scheme. *SIGCOMM*, 1992.

[84] Kitae Nahm, Ahmed Helmy, and C.-C. Jay Kuo. Tcp over multihop 802.11 networks: issues and performance enhancement. In *MobiHoc*, 2005.

[85] Anthony C. H. Ng, David Malone, and Douglas J. Leith. Experimental evaluation of tcp performance and fairness in an 802.11e test-bed. In *E-WIND*, 2005.

[86] http://www.olsr.org/. Optimized Link State Routing Protocol.

[87] Sinha P., T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. Wtcp: a reliable transport protocol for wireless wide-area networks. *Wireless Networks*, 2002.

[88] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. WiLDNet: Design and Implementation of High Performance WiFi-based Long Distance Networks. In *NSDI*, 2007.

[89] Gojko Babic Raj Jain, Arjan Durresi. Throughput fairness index: An explanation, atm forum/99-0045, february 1999.

[90] S. Rangwala, R. Gummadi, R. Govindan, and K. Psounis. Interference-aware fair rate control in wireless sensor networks. *SIGCOMM*, 2006.

[91] S. Rangwala, A. Jindal, K. Jang, K. Psounis, and R. Govindan. Understanding congestion control in multi-hop wireless mesh networks. *Mobicom*, 2008.

[92] B. Scheuermann, C. Lochert, and M. Mauve. Implicit hop-by-hop congestion control in wireless multi-hop networks. *Ad Hoc Netw.*, 2008.

[93] K. Sundaresan, V. Anantharaman, H. Hsieh, and R. Sivakumar. Atp: A reliable transport protocol for ad-hoc networks. In *In Proceedings of MOBIHOC 2003*, 2003.

[94] L. Tassiulas. Adaptive back-pressure congestion control based on local information. In *IEEE Transactions on Automatic Control*, Feb 1995.

[95] I.; Sunghyun Choi Tinnirello. Efficiency analysis of burst transmissions with block ack in contention-based 802.11e wlans. *ICC*, 2005.

[96] UMassDieselNet: A Bus-based Disruption Tolerant Network. http://prisms.cs.umass.edu/diesel/.

[97] M. Vutukuru, K. Jamieson, and H. Balakrishnan. Harnessing exposed terminals in wireless networks. In *NSDI*, San Francisco, USA, April 2008.

[98] Kaixin Xu, Mario Gerla, Lantao Qi, and Yantai Shu. Enhancing tcp fairness in ad hoc wireless networks using neighborhood red. In *MobiCom*, 2003.

[99] Xin Yu. Improving tcp performance over mobile ad hoc networks by exploiting cross-layer information awareness. In *MobiCom*, 2004.

[100] B.; Xiaoqiao Meng; Songwu Lu Zhenghua Fu; Greenstein. Design and implementation of a tcp-friendly transport protocol for ad hoc wireless networks. *Proceedings of ICNP*, pages 216–225, 12-15 Nov. 2002.

[101] S. Ha and I. Rhee and L. Xu CUBIC: a new TCP-friendly high-speed TCP variant *Proceedings of SIGOPS*, pages 64–74 Nov. 2008.

[102] P. Sarolahti and M. Kojo and K. Raatikainen F-RTO: an enhanced recovery algorithm for TCP retransmission timeouts *SIGCOMM Comput. Commun. Rev.*, pages 51-63 2003.

[103] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, M. Turon Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks *IPSN*, 2007.

[104] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo The Collection Tree Protocol (CTP) *http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html*

[105] F. Stann, J. Heidemann RMST: reliable data transport in sensor networks. *SNPA*, 2003

[106] J. Kurose, K. Ross Computer networking: a top down approach. Addison Wesley, 2007

[107] B. Radunovic, C. Gkantsidis, D. Gunawardena, and P. Key. Horizon: Balancing tcp over multiple paths in wireless mesh network. In *MobiCom*, 2008.

[108] Cüneyt Özveren, Robert Simcoe, and George Varghese. Reliable and efficient hop-by-hop flow control. In *SIGCOMM*, 1994.

[109] Ajit Warrier, Sankararaman Janakiraman, Sangtae Ha and Injong Rhee DiffQ: Practical Differential Backlog Congestion Control for Wireless Networks In *InfoCom*, 2009

[110] Anand Padmanabha Iyer, Gaurav Deshpande, Eric Rozner, Apurv Bhartia, Lili Qiu Fast Resilient Jumbo Frames in Wireless LANs In *IWQoS*, 2009

[111] W. Kim, H. Wright and S. Nettles Improving the Performance of Multi-hop Wireless Networks using Frame Aggregation and Broadcast for TCP ACKs In *CoNext*, 2008

[112] L. Georgiadis, M. Neely, and L. Tassiulas. Resource allocation and cross-layer control in wireless networks. In *Foundations and Trends in Networking,* 1(1):1-144, 2006.