RCRT: Rate-Controlled Reliable Transport for Wireless Sensor Networks

Jeongyeup Paek
Embedded Networks Laboratory
University of Southern California
jpaek@enl.usc.edu

Ramesh Govindan
Embedded Networks Laboratory
University of Southern California
ramesh@usc.edu

Abstract

Emerging high-rate applications (imaging, structural monitoring, acoustic localization) will need to transport large volumes of data concurrently from several sensors. These applications are also loss-intolerant. A key requirement for such applications, then, is a protocol that reliably transport sensor data from many sources to one or more sinks without incurring congestion collapse. In this paper, we discuss RCRT, a rate-controlled reliable transport protocol suitable for constrained sensor nodes. RCRT uses end-to-end explicit loss recovery, but places all the congestion detection and rate adaptation functionality in the sinks. This has two important advantages: efficiency and flexibility. Because sinks make rate allocation decisions, they are able to achieve greater efficiency since they have a more comprehensive view of network behavior. For the same reason, it is possible to alter the rate allocation decisions (for example, from one that ensures that all nodes get the same rate, to one that ensures that nodes get rates in proportion to their demands), without modifying sensor code at all. We evaluate RCRT extensively on a 40-node wireless sensor network testbed and show that RCRT achieves more than twice the rate achieved by a recently proposed interference-aware distributed rate-control protocol, IFRC [23].

Categories and Subject Descriptors:
C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Wireless communication*

General Terms: Design, Experimentation, Performance

Keywords: Sensor networks, Transport protocol, Reliable, Centralized, End-to-end, Congestion Control

1 Introduction

As sensor network software and hardware matures, it is becoming increasingly possible to conceive of a class of applications that has received relatively little attention so far — applications requiring the transport of high-rate data. Sources for high-rate data include imagers, microphones, and accelerometers. These sensors in turn motivate several interesting applications in surveillance, precision agriculture, structural damage assessment, and military target tracking.

To support these emerging applications, we need to solve two problems. First, wireless sensors have limited radio bandwidth. A collection of sensors generating high-rate data can easily overwhelm the network to the point of congestion collapse, where the network is unable to perform useful work because its capacity is exceeded. Second, applications that use high-rate sensors of the kind described above are often loss-intolerant. For example, source localization algorithms [2] use time difference of arrival between comparable samples at different nodes, and structural monitoring algorithms estimate structural mode shapes [5] by correlating comparable samples observed at different nodes. In either case, the loss of samples can adversely affect the accuracy of the algorithm.

While many sensor network transport protocols have been studied in the literature, most of them solve one of the two problems identified above (Section 2): they either provide reliable end-to-end delivery of data from every sensor to a sink, or discuss a congestion control mechanism without ensuring end-to-end reliable delivery.

In this paper, we discuss the design and implementation of a transport protocol that ensures reliable delivery of sensor data from a collection of sensors to a base station, while avoiding congestion collapse. However, we place two other requirements on the design of this transport protocol. First, unlike most existing proposals which, implicitly or explicitly, support only a single stream of sensor data from each network node, we require the network to be able to support multiple concurrent streams from each sensor node. We foresee that future sensor network deployments will be multi-user systems, with concurrently executing applications. Second, while much existing work has assumed a specific way to allocate network capacity to all sensors (*e.g.*, a fair allocation), we require our solution to separate the capacity allocation policy from the underlying transport mechanisms. It is unclear, yet, if there exists a single traffic allocation policy that would satisfy the needs of all sensor network applications.

Our solution, RCRT, has many different components, many of which are novel (Section 3). It uses relatively standard mechanisms for end-to-end reliable delivery; the base

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'07, November 6–9, 2007, Sydney, Australia.
Copyright 2007 ACM 1-59593-763-6/07/0011 ...$5.00.

	Distributed Congestion Control	*Centralized Congestion Control*	*No Congestion Control*
Reliable	Flush, STCP	**RCRT**	Wisden, Tenet, RMST
Unreliable	IFRC, Fusion, CODA	QCRA, ESRT	Surge, CentRoute, RBC

Table 1—Sensor Network Transport Protocols: A Taxonomy

station (or *sink*) discovers missing packets and explicitly requests them from the sensors. However, its congestion control functionality, in a significant departure from much of the prior work, is implemented in the sink. The sink has a comprehensive view of the performance of the network, and it uses this perspective to control traffic allocation in a more efficient way than would be possible with decentralized congestion control. RCRT employs a novel congestion detection technique, in which the sink decides that the network is congested if the time to repair a loss is significantly higher than a round-trip time. Moreover, it de-couples rate adaptation from rate allocation; that is, the RCRT sink first decides how much the total traffic needs to be reduced (or increased) in response to congestion (or lack thereof), then separately decides how to allocate the increase or decrease to different sources. This decoupling allows a network administrator to assign different capacity allocation policies for different applications.

We have implemented RCRT's sink-side functionality on a PC-class platform (our code ports to embedded systems such as Stargates as well), and the sensor-side functionality on the Tmote Sky mote. Our detailed evaluation (Section 4) of RCRT performance brings out many of its features: its ability to dynamically respond to congestion, its flexibility, robustness, and its support for multiple applications. Most importantly, we show that RCRT is able to achieve more than twice the network throughput of IFRC [23], a recently proposed approach that implements decentralized congestion control but does not guarantee end-to-end reliability. RCRT is able to achieve this because its traffic control algorithms are able to better estimate and control the network capacity given the sink's comprehensive perspective into network performance.

2 Related Work

To place our work in context, we taxonomize sensor network transport protocols as shown in Table 1. We distinguish transport protocols by whether they provide end-to-end reliability or not, whether they implement congestion control or not, and if they do, whether the congestion control implementation is distributed or centralized (at a sink, for example). As Table 1 shows, RCRT is, to the best of our knowledge, the only instance of a reliable transport protocol that implements congestion control in a centralized manner. Furthermore, to our knowledge, although there is a large literature on congestion control in wired and wireless networks, this specific problem has not been addressed in those contexts. However, many of our individual design decisions draw from that literature; we cite the relevant pieces of work when we describe the detailed design of RCRT in Section 3. We now discuss each element of Table 1 in turn.

The simplest transport protocols are those that do not guarantee end-to-end reliability, and implement no congestion control. Surge [1] can be thought of as implementing such a simple transport protocol. CentRoute/DataRel [26] centrally computes efficient source routes to individual motes on demand and provides a TCP-like abstraction for transporting data from a mote to a nearest gateway in a tiered network. It implements a fixed number of end-to-end retransmissions to improve reliability, but does not incorporate congestion control. RBC [31] is a hop-by-hop reliable transport scheme optimized for real-time many-to-one delivery of bursty event data. RBC uses retransmission scheduling to increase reliability and reduce latency. However, it is not designed for continous flows, and it does not guarantee end-to-end reliabilty.

Next come the class of transport protocols that provide end-to-end reliability, but implement no congestion control. RMST (Reliable Multi-Segment Transport) [25] and the transport protocol implemented in Wisden [30] and Tenet [13] are examples of such protocols. RMST is a hop-by-hop reliable transport protocol built on top of Directed Diffusion [16] in which loss is repaired hop-by-hop using caches in the intermediate nodes. RMST guarantees reliability, but it is designed for larger and more capable platforms. Wisden's transport protocol (Tenet's is very similar) provides end-to-end reliability of sensor data transmitted from a field of sensors to a single sink. However, in both these systems, the rate at which this data is transmitted by a node must be manually set by a system administrator.

Some research has examined centralized congestion control without end-to-end reliability guarantees. QCRA [4] (Quasi-static Centralized Rate Allocation) is a centralized rate allocation scheme that tries to achieve fair and near-optimal rate allocation for each node given the topology, routing tree, and link loss rate information. ESRT (Event-to-Sink Reliable Transport) [24], is also a centralized rate control scheme for event-driven applications where the base station requests all source nodes to increase or decrease rate in order to achieve the desired event reliability. But ESRT assumes that the sink can communicate with all sources in one hop, and has only been evaluated in simulation.

There is a body of work that has designed distributed congestion control schemes without regard to end-to-end reliability. IFRC (Interference-aware Fair Rate Control) [23] is a distributed rate allocation scheme that uses queue size to detect congestion, shares the congestion state through overhearing, and converges to fair and efficient rates for each node. Even though IFRC does not provide end-to-end reliability, it is closest in spirit to our work in that it attempts to find a fair and efficient transmission rate that avoids congestion collapse. We quantitively compare IFRC and RCRT in a later section, but note that a) RCRT has greater flexibility than IFRC since many different traffic allocation policies can be implemented in the RCRT sink without changing any code at the sensor nodes, and b) RCRT does not require sophisticated parameter tuning for stability as IFRC does. Also different from RCRT is work on congestion mitigation:

Fusion [15] uses hop-by-hop congestion control and CODA (Congestion Detection and Avoidance) [29] uses end-to-end flow control along with hop-by-hop back-pressure for this purpose.

Finally, some work has examined distributed congestion control of end-to-end reliable transport. STCP [17] proposes to use RED [10]-style congestion detection in sensor nodes and a slightly modified form of TCP end-to-end. To our knowledge, STCP has not been evaluated on a real wireless testbed. By contrast, Flush [20] is a reliable transport protocol designed for large diameter sensor networks. Unlike in RCRT, at any instant at most one node can transport its data to the sink using Flush.

Not included in the taxonomy described in Figure 1 are protocols for reliable dissemination. PSFQ [28], Deluge [14], and TRD [13] are reliable dissemination protocols designed for reprogramming or retasking the sensor network from a base station, and not for transport of data in the other direction. Also omitted from Figure 1 is prior work on congestion sharing in wired networks (for example, Ensemble-TCP [8], and Integrated Congestion Management Architecture [3]). These proposals, though not directly applicable to sensor networks, bear some resemblance to RCRT's centralized congestion control.

3 Rate Controlled Reliable Transport

In this section, we start by describing the goals of RCRT and discuss how its overall design meets those goals. We then delve into the details of individual components of RCRT: end-to-end reliability, congestion detection, rate adaptation and rate allocation. We conclude with a discussion of RCRT's limitations.

3.1 Design Goals

RCRT is designed for sensor networks in which sensor readings are transmitted from one or more sensors (*sources*) to a base station (or *sink*). It is also applicable to tiered sensor networks [13], where sensors may be transporting sensor readings to the nearest gateway (master, in the terminology of [13]), which in turn routes the messages to a designated upper-tier node (which we call the *sink*). RCRT does *not* require all sensors to be transmitting data. More generally, sensors may start and end data transmissions at arbitrary times that are not known *a priori*.

Six goals guide the design of RCRT. They are:

Reliable end-to-end transmission. Our first goal is to achieve complete end-to-end reliability of all data transmitted by each sensor to a sink. Of course, this is only possible if the network is not partitioned: that is, for each source, there exists a network level path to the sink where each link has a non-zero packet reception rate. This goal is motivated by emerging high data rate applications which are loss-intolerant; examples of these applications include networked imaging [21], acoustic source localization [2], and structural health monitoring [6]. In each of these cases, processing algorithms are extremely sensitive to packet loss, and they are rarely interested in inter-node data aggregation. For example, source localization algorithms use time difference of arrival between comparable samples at different nodes, and structural monitoring algorithms estimate structural mode shapes by correlating comparable samples observed at different nodes. In either case, the loss of samples can adversely affect the accuracy of the algorithm.

Network Efficiency. Our second goal is to maintain the network at an efficient operating point. Specifically, we wish to avoid *congestion collapse* [9], a regime in which sources are sending data faster than the network can transport them to the base station. In this regime, no useful work gets done by the network, since packets are repeatedly lost and continually retransmitted. In addition, we wish to ensure that sources transmit their sensor readings to the base station at as high a rate as possible. Since sources may start transmitting sensor readings at arbitrary times, this rate cannot be determined *a priori*, but must be adaptively discovered.

Support for concurrent applications. While much prior work on sensor network transport has focused on supporting a single sensing application, we wish to explicitly support multiple concurrent applications in RCRT. Future sensor network deployments are likely to evolve to being multi-user or multi-application systems, so it is important to consider this design criterion at the outset.

Flexibility. Another goal is to allow different applications to choose different *capacity allocation policies*. A capacity allocation policy determines how the overall network capacity is divided up among the different sources. In some homogeneous deployments, where all the sensors are generating data at exactly the same rates, it may be necessary to divide up the capacity equally (in a *fair* manner). In other cases, some sources might need a proportionally larger allocation since, for example, they might be transmitting images. An important sub-goal is that this flexibility should not come at the expense of requiring code modifications on the sensors; this is clearly desirable. This goal distinguishes our work from distributed congestion control mechanisms that implicitly embed a traffic allocation policy within the network. For example, IFRC can only support fair and weighted fair allocations [23].

Minimal Sensor Functionality. We wish to keep as much of the protocol functionality out of the sensors as possible. This goal is motivated by the constraints of the current generation of sensors. More generally, however, it is motivated by the observation that, for our problem, it is possible to achieve overall system simplicity by moving as much of the intelligence as possible out of the sensor network and into the sink.

Robustness. Finally, we require that RCRT be robust to routing dynamics and to nodes entering and leaving the system. This implies that traffic allocations to sensors can dynamically change, as can the locations of congested nodes. The system must be able to dynamically adapt to these changes.

3.2 RCRT Overview

To describe RCRT, we introduce the following notation and terminology. We define a *sink* as an entity (software program, or part thereof) which runs on a base station (or an upper-tier node in a tiered network) and which collects data from one or more sensors (sources). RCRT is oblivious to the kind of data sourced by the sensors: they can be raw samples, processed time series, images, and so forth. More than one sink can be running concurrently in the sensor network. We

Function	Where	How
End-to-end retransmissions	Source and sink	End-to-end NACKs
Congestion Detection	Sink	Based on time to recover loss
Rate Adaptation	Sink	Based on total traffic, with additive increase and decrease based on loss rate
Rate Allocation	Sink	Based on application-specified capacity allocation policy

Table 2—RCRT Components

use the notation f_{ij} to denote the *flow* of data from source i for sink j. Of course, this flow is delivered from the source over (possibly) multiple hops to the base station. A sensor i may source several flows f_{ij} for different sinks j. Finally, each sink j is associated with a capacity allocation policy P_j which determines how network capacity is divided up across flows f_{ij} for $\forall i$. The simplest P_j is one in which each flow f_{ij} gets an equal share of the network capacity (a *fair* allocation). We give more examples of P_j later.

RCRT provides reliable, sequenced delivery of flows f_{ij} from source i to sink j. Furthermore, RCRT ensures that, for a given application j, the available network capacity is allocated to each flow according to policy P_j. Specifically, each flow f_{ij} is allocated a rate $r_{ij}(t)$ at each instant t that is in accordance with policy P_j. Thus, for a fair allocation policy, all sensors would receive equal $r_{ij}(t)$.

How does RCRT achieve all this? Table 2 describes the various components of RCRT. End-to-end reliability is achieved using end-to-end negative acknowledgments. A particularly novel aspect of RCRT is that its *traffic management functionality resides at the sink*. Specifically, each sink determines congestion levels and makes rate allocation decisions. Once sink j decides the rate r_{ij}, it either piggybacks this rate on a negative acknowledgment packet, or sends a separate feedback packet, to source i.

At the sink, RCRT has three distinct logical components. The *congestion detection* component observes the packet loss and recovery dynamics (which packets have been lost, how long it takes to recover a loss) across every flow f_{ij}, and decides if the network is congested. Once it determines that the network is congested, the *rate adaptation* component estimates the total sustainable traffic $R(t)$ in the network. Then, the *rate allocation* component decreases the flow rates $r_{ij}(t)$ to achieve $R(t)$, while conforming to policy P_j. Conversely, when the network is not congested, the rate adaptation component additively increases the overall rate $R(t)$, and the rate allocation component determines $r_{ij}(t)$. In our current implementation, multiple sinks do not cooperate with each other to determine congestion, adapt or allocate rates; doing so is likely to provide higher efficiency gains and a more equitable rate allocation, and we have left this to future work.

RCRT satisfies our design goals (Section 3.1). By design, it provides end-to-end reliability and attempts to keep the network operating efficiently while supporting multiple applications, each with its own capacity allocation policy. Much of the traffic management functionality in RCRT is centralized at the sink, keeping the sensors as simple as possible.

Finally, our assumptions about the link layer and the routing layer are largely consistent with current practice. Although our experiments are conducted using the default CSMA MAC layer in TinyOS (the widely-used sensor node operating system), the design of RCRT does not depend on any features specific to a particular MAC layer. Link-level retransmissions at the MAC layer can improve the performance of RCRT but are not essential for its correctness. We discuss this in more detail in Section 3.8.

We assume that the sensor nodes run a routing protocol that dynamically selects a path from each node to the sink and also a reverse path from the sink to each node. Also, RCRT does not assume symmetric routing between a source and the sink. Our experiments (Section 4) are conducted using TinyOS's tree-based routing protocol, MultiHopLQI, but it can work with other routing protocols such as CentRoute [26] which satisfy these requirements.

Finally, RCRT focuses on the transport protocol itself. Cross-layer techniques for congestion control that exploit radio, MAC or routing protocol features are beyond the scope of this work.

In the following sections, we discuss the detailed design of RCRT. As described above, RCRT sinks act independently in rate allocation decisions. So, in what follows, we drop the subscript j, with the understanding that all the behavior described below refers to a sink and its associated flows.

3.3 End-to-end Reliability

Unlike TCP [18], RCRT implements a NACK-based end-to-end loss recovery scheme to guarantee 100% reliable data delivery. The sink detects packet losses and repairs them by requesting end-to-end retransmissions from source nodes. This design leverages the fact that the base station has significantly more memory and can keep track of all missing packets. Each sensor node stores a copy of every data packet that it originates in its local retransmit buffer when transmitting the packet to the sink. The sink keeps track of sequence numbers of packets that it receives on each flow. A gap in the sequence number of received packets indicates packet loss. The sink maintains a list of missing packets per flow. When losses are detected, the sequence numbers of the lost packets are inserted into this list. Entries in this list of missing packets are sent as negative acknowledgments (NACKs) by the sink to each source. The use of NACKs avoids an 'ACK implosion' [11], where the acknowledgments of successful receptions sent by the sink overwhelm the network. Upon receiving a NACK, the source retransmits the requested packets to repair the losses. Also, the source determines when it can safely overwrite packets in the retransmit buffer by looking at the cumulative ACK sequence number piggybacked in all feedback packets (Section 3.7 describes cumulative ACK and feedback packets).

The sink also maintains a list of out-of-order packets for each flow to provide in-order delivery of data packets to the application. This list contains packets that are received at the sink but have not been passed to the application layer because there are one or more gaps in the sequence numbers. For example, if sequence numbers $[1,2,3,5,6]$ have been received so far, packets 5 and 6 are inserted into the out-of-order packets list. When packet 4 is received, the sink passes packets $4,5,6$ to the application.

RCRT uses the per-flow lists of missing and out-of-order packets for detecting congestion and adapting rates, as we shall discuss below.

3.4 Congestion Detection

An important technical challenge for RCRT is the design of a mechanism for congestion detection. Various techniques have been proposed in the wireless and sensor network literature to measure the level of congestion *at a node*. Broadly speaking, these techniques either directly measure the channel utilization around a node [29], or measure the queue occupancy at the node [15, 23]. These techniques are similar in spirit to approaches that attempt to detect incipient congestion in Internet routers [10, 19]. By contrast, RCRT attempts to measure, at the sink, whether *the network is congested*. This approach is closer in spirit to approaches in wired networks that have attempted to detect congestion at the ends [18, 22], but with one important difference: in RCRT, a sink has a more extensive view of network performance than, for example, a TCP sender, since the sink receives traffic from many sources.

However, it would be incorrect to merely borrow congestion detection methods used in wired networks. For example, TCP uses a single packet drop to infer that the network is congested. In a wireless network, it is well known that such a congestion detection mechanism can result in extremely poor transport performance because wireless links tend to exhibit relatively high packet loss rates.

Accordingly, RCRT bases its congestion detection mechanism on a completely different approach. This approach is in line with RCRT's primary goal of providing end-to-end reliability. Its congestion detection mechanism is based on the following intuition: that the network is uncongested as long as end-to-end losses are repaired "quickly enough". This intuition permits a few end-to-end losses caused by transient congestion, or by poor wireless links. Furthermore, it permits the sources to transmit at a higher rate even if there are occasional end-to-end losses, since those losses can be recovered by the mechanism described in the previous section. For this reason, RCRT uses the *time to recover loss* as a congestion indicator.

Recall that RCRT maintains a per-flow list of packets that have been received out of order (Section 3.3). Now, the length of this list indicates how many packets have been received after the first unrecovered packet loss, which reflects how much time has passed since the first unrecovered loss. Ideally, we would want the average time taken to recover a loss to be around one round trip time (RTT). If that property holds, the network is not congested in the sense that loss recovery is functioning properly. If it takes more than two RTTs to recover the losses, then the network is more likely to be congested. Since the expected number of packets received during one RTT is $r_i RTT_i$, if the out-of-order packet list length is $r_i RTT_i$, then roughly one RTT has passed since the first unrecovered loss (recall that r_i is the rate assigned to source i; we have omitted the time dimension in the notation for simplicity). Denote by L_i the length of source i's out-of-order packet list at some instant. Then,

$$L_{norm,i} = \frac{L_i}{r_i RTT_i}$$

is a measure of the number of RTTs it would take to recover the loss. RCRT uses the exponentially weighted moving average of this value as the measure of average congestion level, denoted by C_i, for source i. Thus, $C_i = 2$ means that it takes around 2 RTT_i's on average to repair losses for node i. (Section 3.7 explains how RTT is estimated.)

RCRT detects congestion by using a simple thresholding technique. If the C_i exceeds an upper threshold U for *any* source i, we say that the network is congested. RCRT declares the network uncongested when C_i falls below a lower threshold L for *every* i. The gap between U and L is the desired congestion level in steady state that allows for some losses to achieve higher throughput while ensuring timely loss recovery. RCRT updates the C_is for every flow whenever a packet is received from that flow. It then decides whether the network is congested or uncongested, based on the algorithm described above. RCRT performs different actions in the congested and uncongested states, as described in Section 3.5.

We use $L = 1$ and $U = 4$ as our thresholds. The value of U is twice what one might expect from our discussion above. We choose this more conservative estimate since a flow's RTT_i increases dramatically when the network transitions from an uncongested to a congested state. Choosing these values ensures that RCRT does not react to congestion earlier than it should.

3.5 Rate Adaptation

The second major challenge in RCRT is the design of a mechanism to adapt transmission rates in response to congestion (or lack thereof). Rate adaptation techniques have been studied widely in the literature. Most transport protocols additively increase flow rates (or, as in TCP [27], windows) in the absence of congestion, and multiplicatively decrease flow rates when congestion is detected. Chiu *et al.* [7] show that this AIMD approach guarantees stability and convergence to a fair and efficient allocation. While RCRT uses AIMD, it adapts the total aggregate rate of all the flows as observed by the sink, rather than the rate of a single flow. In spirit, this is similar to XCP [19], but there are some qualitative differences: XCP's decisions are made at a bottlenecked router, and XCP adapts rate based on the excess capacity at a link. We describe RCRT's rate adaptation design now.

Let $R(t)$ denote the sum of the currently assigned rates $r_i(t)$ for all flows i. RCRT uses AIMD on $R(t)$. When the network is not congested, RCRT increases $R(t)$ additively,

- Increase: $R(t+1) = R(t) + A$

where A[1] is a constant. When the network is congested, RCRT decreases $R(t)$ multiplicatively:

- Decrease: $R(t+1) = M(t)R(t)$

where $M(t)$ is a time-dependent multiplicative decrease factor. An alternative design would have been to individually control the rates of each flow. However, since the sink has a greater perspective into network performance, designing the

[1] We have used A = 0.5 pkts/sec for our experiments, roughly 0.5% of the experimentally-determined maximum total rate at the base station.

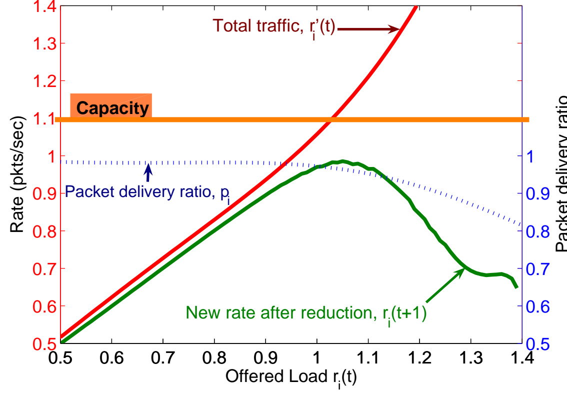

Figure 1—Plot of p_i, corresponding $r'_i = \frac{r_i(2-p_i)}{p_i}$, and effect of $M(t)$ where p_i has been polynomial curve-fitted from the experimentally collected data. X-axis is the offered load r_i. Y-axis represent both packet delivery ratio p_i, and the expected total traffic; $r'_i = \frac{r_i(2-p_i)}{p_i}$

rate adaptation to control the total aggregate rate of the network allows the control algorithm to be independent of the number of flows, and less oscillatory when there are a large number of flows.

Two questions remain: When are rate adaptation decisions made? How is $M(t)$ determined? We address these questions next.

Whenever RCRT determines the network is congested (Section 3.4), it applies the rate decrease step described above, computes a new rate allocation for all the flows (Section 3.6), and sends the new rate r_i for each flow to the corresponding source. It doesn't make another rate adaptation decision until it observes the effects of the previous decision. To do this, RCRT waits conservatively for at least twice the maximum value of RTT_i. This usually allows enough time for the rate feedback to reach the sources, and for the sources to send enough packets so that congestion measures at the sink are appropriately updated. Thereafter, if a packet is received from some flow i, and C_i is still above the upper threshold U, another rate decrease step is triggered, but only if f_i reports that it is using the rate assigned in the previous rate adaptation step. This last check ensures that RCRT does not react to stale information; especially in times of congestion, RCRT's RTT estimates can be slightly off and this step ensures that RCRT reacts only after the source has had a chance to respond to the previous rate adaptation. Finally, if the network is uncongested, RCRT applies the increase rule above, but only if twice the maximum RTT_i has expired since the last rate adaptation decision. Furthermore, RCRT checks that if a packet from f_i triggered the rate adaptation decision, then that flow is using its most recent rate allocation.

In most transport protocols, the multiplicative decrease factor is constant (0.5 in TCP). However, because RCRT has a comprehensive view of network behavior across different sources, it can do better than simply halve $R(t)$. In RCRT, when a packet is received from flow f_i and that packet causes C_i to exceed the upper threshold U, $M(t)$ is computed based on the loss rate experienced by f_i. What is the intuition for this? Suppose that f_i's packet delivery ratio is p_i. This means that, every second, $r_i p_i$ packets out of r_i are being delivered to the sink without end-to-end loss. Furthermore, feedback traffic roughly proportional to $r_i(1-p_i)$ must be delivered by the sink to the source in response to these losses.[2] Then the expected amount of traffic to and from the sink is $\frac{r_i}{p_i}$ and $\frac{r_i(1-p_i)}{p_i}$ respectively. This adds up to total traffic of at least $\frac{r_i(2-p_i)}{p_i}$ for node i in the network.

We know that this level of traffic was not sustainable, since the flow was congested. But, the last time that a rate adaptation decision was made, a source rate of r_i was deemed sustainable, assuming no end-to-end losses. Given a packet delivery rate p_i, it would now be safe to set f_i's rate such that the *total amount of traffic* is r_i. To do this, we should reduce f_i's rate to: $r'_i = \frac{p_i}{2-p_i} r_i$. However, RCRT is a little bit more conservative than that. It reduces the *overall* total rate $R(t)$ by that factor, by setting the multiplicative decrease factor $M(t)$ to be:

$$M(t) = \frac{p_i(t)}{2-p_i(t)}$$

$M(t)$ is larger than 0.5 for all p_i greater than 0.67. So, in regimes where the end-to-end packet reception rate is high, RCRT can more efficiently adapt to congestion than protocols that employ a fixed multiplicative decrease factor of 0.5.

We now give some intuition for how $M(t)$ behaves for different values of $r_i(t)$. To do this, we conducted an experiment consisting of 40 nodes, and measured the *average* or smoothed value of p_i as a function of r_i. This is shown in Figure 1. Also shown on that figure is the *total* network traffic corresponding to a given r_i (labeled $r'_i(t)$), and the value $r_i(t+1)$, which is the rate that would be assigned to node i by RCRT after its multiplicative decrease. For this network, the network was empirically-determined to be saturated at 1.1 pkts/sec per node(Section 4). We'd like to point out two important properties of $M(t)$. First, regardless of the value of r_i, $M(t)$ is always successful in assigning a rate $r_i(t+1)$ that is lower than capacity. Second, $M(t)$ is more aggressive when $r'_i(t)$ is well above the network's capacity. These two properties together ensure that RCRT avoids congestion collapse.

Loss Rate Estimation. Thus far, we have not described how $p_i(t)$ is calculated. Finally, RCRT keeps track of estimated path loss rate $(1-p_i(t))$ for each flow using the *Average Loss Interval* method discussed in [12]. It uses the average interval (in number of packets) between loss events to estimate the loss rate of a flow. Denote the interval between m-th and $m+1$-th loss on flow i as s_{im}. Then, for the recent $1 \le m \le n$ losses, the average loss event interval for flow i $\hat{s}_i$ is calculated as,

$$\hat{s}_{i(1,n)} = \frac{\sum_{m=1}^{n} w_m s_{im}}{\sum_{m=1}^{n} w_m}$$

$$\hat{s}_{i(0,n-1)} = \frac{\sum_{m=0}^{n-1} w_m s_{im}}{\sum_{m=1}^{n} w_m}$$

$$\hat{s}_i = max(\hat{s}_{i(1,n)}, \hat{s}_{i(0,n-1)})$$

[2] In RCRT, a list of missing information can be packed into a single NACK. In this analysis, we assume that only one missing packet is included in each NACK packet. This can happen when the losses are infrequent and spread out uniformly over time.

where s_0 is the interval since the most recent loss and w_m is the weight given to each loss interval in the history. We have used $n = 8$ and $w = [1,1,1,1,0.8,0.6,0.4,0.2]$ as our parameters. Intuitively, these choices give greater weight to recent loss periods, and lesser weight to more distant loss events. Then we compute $p_i(t)$ from

$$1 - p_i(t) = \frac{1}{\hat{s}_i(t)}$$

We chose the Average Loss Interval (ALI) method over others after experimentally verifying that it was more robust to parameter choices than the alternatives. A window-based method that estimates loss rates based on some window of the number of packets or time is highly sensitive to the choice of window. An EWMA approach is sensitive to the choice of gain, and includes a significant history of losses. By contrast, the ALI method always looks at a fixed number of loss events, and preferentially weighs the recent ones. This helps RCRT be more responsive to the onset of congestion.

3.6 Rate Allocation

Once the total rate $R(t)$ is calculated by the rate adaptation mechanism, the role of RCRT's rate allocation component is to implement the capacity allocation policy P_j associated with its sink. This is a novel aspect of RCRT; to our knowledge, most prior work has (implicitly or explicitly) embedded a capacity allocation policy within the rate allocation mechanism.

RCRT's rate allocation component essentially assigns rates $r_i(t)$ to each flow in keeping with the rate allocation policy P_j such that the individual flow rates sum up to $R(t)$. Because RCRT decouples rate adaptation from rate allocation, it is possible to obtain this flexibility. We see this flexibility as being crucial, since it is unclear that there is, *a priori* a preferred policy for sensor networks (unlike the Internet, which is a shared infrastructure, and for which some form of fairness makes sense). RCRT's rate allocation component is similar to XCP's [19] fairness controller, but offers greater flexibility.

Our current prototype contains three different policies.

Demand-proportional. In this policy, each flow expresses a desired rate, that we call its *demand* d_i. This policy allocates rate r_i to each node i proportional to its demand d_i such that the fraction $(r_i(t)/d_i)$ is the same for all i;

$$r_i(t)/d_i = \rho(t) \quad \forall i$$

As long as we use same $\rho(t) = \rho_i(t)$ for all node i, demand-proportional allocation follows. We compute $\rho(t)$ as follows:

$$R(t) = \sum_i^N r_i(t) = \sum_i^N \rho(t) d_i = \rho(t) D$$

$$\rho(t) = \frac{R(t)}{D}$$

where D is the sum of all d_is. Then, the new rate for each node i simply becomes $r_i(t) = \rho(t) d_i$.

Demand-limited. In this policy, $R(t)$ is divided among all the flows equally, except that no flow gets a higher rate than d_i. Given $R(t)$, there exists a simple greedy algorithm for computing a demand-limited rate allocation.

Fair. This allocation policy assigns an equal share of $R(t)$ to all flows, regardless of d_i. Flow demands are ignored in this policy.

While we have not experimented with other policies, our prototype software is written modularly to easily accommodate other policies. For example, it is easy to implement a weighted allocation policy, where each source gets a rate allocation proportional to a configured weight w_i (weighted allocation is similar to demand-proportional allocation, with the only difference that in the latter, a source is not assigned a rate higher than its demand). This weighted allocation policy is a generalized form of priority allocation, in which some nodes are given higher priority than others. Finally, new rate allocation policies should not require any changes to the protocol (or to code on the sensors) — they can simply be configured on the sink.

3.7 Other Details

In our description, we have left out several details of RCRT. We describe them here briefly for completeness.

Source Node Behavior: In RCRT, each source node initiates a flow by first establishing an end-to-end connection with the sink. RCRT uses a three-way handshake connection establishment mechanism similar to that of TCP where the third ACK is substituted with the first data packet. While doing this, the source tells the sink its desired data rate d_i (although this information can, in principle, also be just configured at the sink), and the sink tells the node the rate r_i. This three-way handshake also allows RCRT to guess an initial RTT estimate. Once a connection has been established, the node transmits packets on this connection. Each node originates data at rate at most r_i assigned by the sink. The rate r_i is enforced by a token bucket with rate r_i. Each packet contains sequence number, flow ID, and the rate of the corresponding flow. The rate r_i regulates only the new packets that are sourced by this node. End-to-end retransmissions are not constrained by this rate, nor are forwarding and link-level retransmissions performed the lower layers of the protocol stack.

Since rate allocation and loss recovery are controlled end-to-end by the sink, each node simply reacts to the sink. When a node receives a packet from the sink with *new rate* r_i' (this is usually sent in a feedback packet which may or may not contain NACK information, see below), the node adjust the token bucket rate accordingly. When a node receives a feedback packet with *NACK* information, the node retransmits those missing packets.

Finally, each source has a fixed-size retransmit buffer which contains sent packets that have not been acknowledged. Clearly, a source cannot store an infinite number of packets for potential retransmission. To efficiently manage the retransmit buffer, each feedback packet (described below) includes a cumulative ACK sequence number, which indicates the last sequence number that the sink has received contiguously without any missing packets. The source can safely remove packets in the retransmit buffer that are covered by the cumulative ACK sequence number.

When the retransmit buffer is full, the source stops sending new packets and retransmits the packet whose sequence number is one higher than the last acknowledged packet at a rate of $min(\frac{r_i}{2}, \frac{1}{RTT})$. The source does this until it is told that the sink has received those packets and hence it is safe to overwrite the buffers. When this packet has been received at the sink, loss recovery is triggered, the rate is adjusted for all nodes, and a feedback packet is sent. The retransmit buffer rarely fills up since a cumulative ACK is piggybacked in every feedback packet. Additionally, to prevent a stall, each source requests explicit feedback from the sink when its retransmit buffer is more than half full. It does this by setting a bit in every outgoing packet. To minimize the number of feedback transmissions while allowing for enough packets to be in transit, the buffer must be large enough ($>> 2r_i/RTT_i$ packets).

Feedback packets: Every feedback packet sent to a source i contains the assigned rate r_i, a list of *NACK*ed sequence numbers, a cumulative ACK, and the RTT_{avg} (see below) value for that node. Thus, every feedback packet is completely self-contained so that, even if one of these is lost, a subsequent feedback packet suffices to adjust the node's rate, and recover from loss.

RCRT has to be careful in sending feedback packets, since they represent significant overhead. When a packet is received at the sink from node i, the sink sends a feedback to the node only when at least one of these four conditions have been met: one or more missing packets have been detected; the node is sending at a rate different from the assigned rate; a duplicate packet with an already acknowledged sequence number has been received; or, a feedback packet has been explicitly requested by the node. However, RCRT is careful not to send feedback more often than once every RTO_i, defined as $RTT_{avg} + 2 * RTT_{var}$, where RTT_{avg} is the average RTT (see below), and RTT_{var} is the mean deviation of RTT. This rate limit trades-off increased convergence time for lower overhead, especially in times of congestion. Finally, recall that rate adaptation decisions are made on RTT time-scales, also reducing the rate of sending feedback packets.

RTT estimation: Finally, the sink estimates the RTT of each node using feedback packets. RCRT does not require any time synchronization mechanism to do this. The sink records the time T_i when it sends a feedback packet to node i. Node i remembers the time T_i' at which it had received the feedback. When node i next sends a packet to the sink at time T_i'', it calculates the interval $T_i'' - T_i'$ and piggybacks this value in the packet. Upon reception of this packet at time T_i''', the sink can calculate the instantaneous RTT sample value $RTT_{inst,i}$ by $(T_i''' - T_i) - (T_i'' - T_i')$. RCRT uses an exponentially-weighted moving average of this value to get the estimated $RTT_{avg,i}$ for node i. Earlier in this section, when we have referred to RTT_i, we have meant $RTT_{avg,i}$.

3.8 Discussion

Three minor details of RCRT are worth mentioning. One is that any NACK-based scheme suffers from not being able to detect the loss of the last packet (or the contiguous loss of the last few packets), since it relies on the successful reception of a later packet. In RCRT, we detect and recover from such losses during connection tear-down: after they are done sending data, sources explicitly tear-down the transport connection, at which point they synchronize sequence numbers and repair the last loss (if any). The second is that in large networks, the maximum RTT can be high and since RCRT makes decisions on RTT timescales, the network may converge slowly. The third is the case when several consecutive packets are lost due to high congestion. Until a subsequent packet is received, the sink will not notice the congestion, hence RCRT's reaction may be delayed.

A fourth detail requires a little more explanation. Our current RCRT implementation uses link-level retransmissions, and a natural question to ask is: How would RCRT perform in the absence of link-layer retransmissions? To us, this question is ill-posed: in wireless networks with link loss rates approaching 30%, trying to design an end-to-end ARQ-based reliability mechanism without link-layer retransmissions is a fundamentally bad idea, since it would rely on expensive end-to-end retransmissions to repair loss. That said, with a limit on the number of retransmissions as in RCRT, there may be scenarios in which the end-to-end packet delivery rate is still low. In this case, p_i (Section 3.5) will be low, and the multiplicative decrease factor could be lower than 0.5 (the constant multiplicative decrease used by TCP). Thus, in this regime, RCRT may be more conservative than TCP's AIMD, but that mechanism it not known to work well in lossy conditions anyway. In those conditions, the best approach would be to re-provision the network by re-deploying some nodes, or adding others.

Finally, we address the question: does RCRT really avoid congestion collapse? There are two cases to consider. When source nodes hear feedback from the sink, RCRT's multiplicative decrease function $M(t)$ (Figure 1) aggressively reduces source sending rates, allowing the network to recover from congestion. When a source does not hear feedback, the source sends at $r_i(t)$ for a while, but it eventually fills up its retransmit buffer and stalls, effectively sending one packet per RTT (Section 3.7). This is a conservative solution, since the source needs to "probe" by sending at least one packet to recover from transient path failures, and the RTT is the right timescale to do this. (Of course, more conservative approaches, like exponentially backing-off the probe interval are possible, and we have left these to future work). When it is stalled, the source does not congest the network, allowing the network to recover (if, indeed, the loss of feedback packets was caused by congestion).

There are several open questions in the design of RCRT. First, we have not examined inter-sink cooperation. Having such a mechanism in place would help administrators control capacity allocation across different applications and provide higher efficiency gains. Second, our current design does not support a policy which gives excess bandwidth to unconstrained sources. In most of the topologies we have experimented with, almost all sources are constrained by one bottleneck wireless region. However, there can exist topologies where some sources may not be so constrained, and it might be beneficial to allocate higher rates to these sources while rate limiting the congested sources only. Since RCRT does not have insight into the network, it cannot easily distinguish between these sources. Even if the sink observed a set of

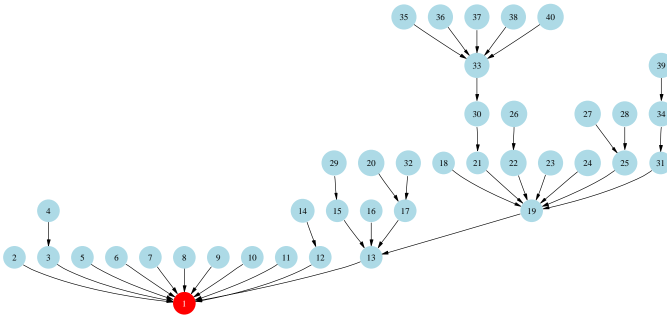

Figure 2—A snapshot of routing topology constructed by MultihopLQI

Design Goal	Section
Reliable end-to-end transmission	All
Network Efficiency	Section 4.2.1 Section 4.2.2 Section 4.2.5
Support for concurrent applications	Section 4.2.4
Flexibility	Section 4.2.4 Section 4.2.3
Robustness	Section 4.2.3

Table 3—Experiments to demonstrate that RCRT meets its goals.

uncongested flows, it cannot determine how those flows interfere with other flows. So, trying to allocate higher rate to those flows might exacerbate congestion on other flows. We are currently experimenting with adding minimal protocol functionality to help with this task. For example, each packet could contain the largest queue size that it traversed, and this information could be used to segregate the constrained from the unconstrained flows. Also, since the sink maintains the estimated loss rate of each flow, this information could be used to isolate exceptionally unreliable flows. Finally, we have not discussed how the “applications” running on the sensors can adapt their sampling rates (for example), based on the r_is they receive. We intend to address these in the near future.

4 Evaluation

In this section, we present results from an extensive performance evaluation of our implementation of RCRT on a 40-node wireless sensor testbed.

4.1 Implementation and Methodology

We have implemented RCRT in TinyOS 1.x for the motes and in C for a PC-class sink device running Linux. The RCRT module on the motes provides a transport layer interface that a sensor application can use to initiate a flow to the sink and send data packets. Also, the module implements a token bucket, whose rate is set to the rate allocated by the sink, to regulate data packets generated at that node. The memory footprint of RCRT for the mote is approximately 5252 code bytes and 374 bytes of RAM for a packet payload size of 64 bytes. The code size excludes RCRT-independent basic components such as timer, flash, MAC, and routing. It uses 64KB of external flash for a retransmit buffer. All other mechanisms described in Section 3 including loss detection, rate adaptation, rate allocation, congestion detection, and RTT estimation are implemented at the sink.

We have evaluated our RCRT implementation on a 40-node indoor wireless sensor network testbed. Each sensor node in our testbed is a Moteiv Tmote with an 8MHz TI-MSP430 micro controller, IEEE 802.15.4-compatible CC2420 radio chip, 10KB RAM, and a 1MB external serial flash memory. These motes are deployed over 1125 square meters of a large office floor.

We have used MultihopLQI in TinyOS as our routing tree protocol for our experiments. In MultihopLQI, each node dynamically selects its parent to construct a routing tree to the base station using the link quality indicator provided by the CC2420 radio chip. Since a sink-to-mote reverse path is required in RCRT (for the feedback packets), we have added a data-driven reverse path construction mechanism. This works as follows. Each node maintains a routing table. When it receives a packet with source address S from a neighbor N, it adds a route entry to S with next hop N. Feedback packets are forwarded using this routing table. Finally, our implementation uses link-layer retransmissions based on chip-level acknowledgments with up to 4 retransmissions.

Figure 2 is a snapshot of the routing tree constructed by the MultihopLQI during one of our experiments. Due to changes in wireless link quality over time, the routing tree changes. Figure 12 shows how frequently each node changed its parent in one of our experiments. Thus, our experiments were conducted in a dynamic environment, with significant routing variability. However, in all of our experiments, the routing protocol consistently produced 6 to 8-hop deep routing trees.

In each of our RCRT experiments, each source originated at least 1000 data packets. This traffic is synthetic, and does not represent the workload generated by any sensor; however, since RCRT is oblivious to the actual data, this is an appropriate methodology. Each experiment ran from 30 minutes to an hour depending on the achieved rate. We logged every packet received at the sink along with the current allocated rate at the time of packet reception.

4.2 Results

In this section, we describe experimental results that validate our RCRT design, demonstrate that RCRT achieves the goals discussed in Section 3.1, and show that RCRT outperforms the state-of-the-art in distributed congestion control. Table 3 summarizes our methodology: for each high-level goal described in Section 3.1, we have designed at least one experiment to validate or quantify that goal. (One exception is the goal of minimal sensor functionality, which follows from RCRT’s design). Some experiments are used to validate or quantify more than one goal.

In most cases, we evaluate RCRT’s performance using its *rate allocation profile* which is the plot of the assigned sensor rates r_is as a function of time. In some cases, particularly to show the efficacy of RCRT’s capacity allocation policies, we plot the average goodput achieved by the node during the experiment.

We must emphasize that we have run RCRT experiments under very general settings. All experiments reported here are from an actual implementation running on a real testbed. The underlying routing and MAC layers are not optimized in

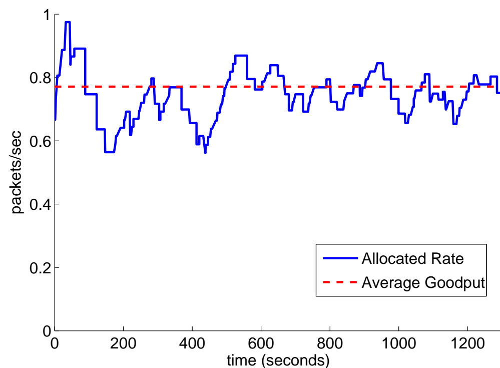

Figure 3—Rate r_i allocated to every node in the 40-node experiment with fair rate policy

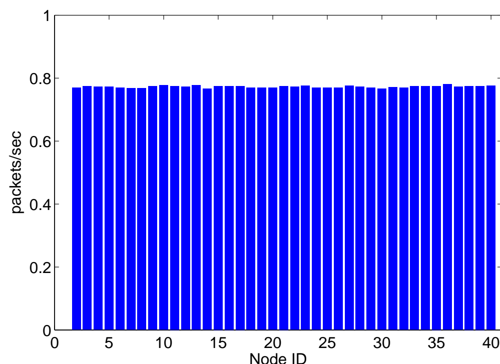

Figure 4—Per-flow goodput in the 40-node experiment with fair rate policy

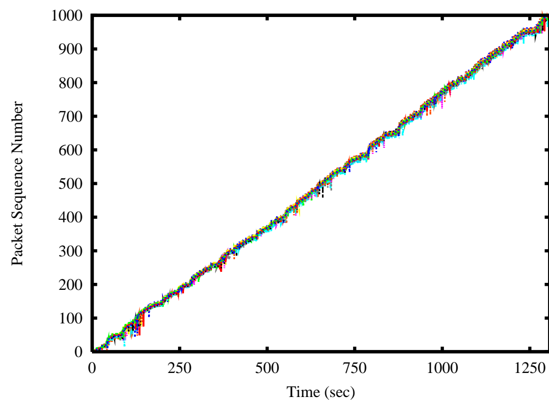

Figure 5—Packet reception plot for all nodes in the 40-node experiment with fair rate policy

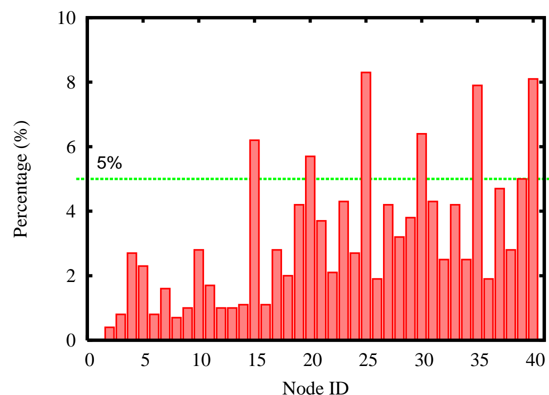

Figure 6—Percentage of packet repaired by end-to-end loss recovery mechanism in the 40-node experiment

any way, nor have the experiments been run at special times to avoid interference. Our testbed is susceptible to significant interference both from other 802.15.4 radios and from 802.11 radios, and this interference is highly time-varying.

Finally, we note that RCRT achieves 100% reliable packet delivery *in all experiments* we have conducted for this paper. For this reason, we do not focus on RCRT's end-to-end reliable transmission mechanism, but instead focus on how well RCRT's congestion control works: what rates are assigned, how RCRT reacts to dynamics, and so on.

4.2.1 Baseline

We start with a simple baseline experiment that illustrates some of the important features of RCRT. In this experiment, RCRT runs on a 40-node network. One node is a basestation, and the others are programmed to send data back to the sink. The fair rate allocation policy is used at the sink.

Figure 3 shows the rate allocation profile $r_i(t)$ allocated to every node by the sink as a function of time. The solid line represents the instantaneous allocated rate logged at the time of every packet reception. The dashed line shows the average achieved goodput from all nodes. Since a fair-rate policy was used, all nodes received the same goodput. This graph shows the efficiency of RCRT's rate adaptation mechanism. Unlike other AIMD schemes that drastically reduce the rate in response to congestion by halving the rate (or, as in TCP, the window) RCRT tries to stay near the steady state average rate by making small adaptive reductions. At the beginning of the experiments, the allocated rate over-shoots to almost 1 pkts/sec, and drops down to around 0.55 pkts/sec (almost half). This is because when the nodes first start sending packets, the network was not congested, and the RTT estimate takes some time to stabilize. But after this transient, the allocated rate converges and stays within 25% of the average goodput. This less oscillatory behavior results from RCRT's rate adaptation design which makes rate adaptation decisions based on the overall traffic, rather than on a single flow.

Figure 4 shows the per-flow goodput achieved at the sink. Each bar represent the average goodput achieved by each source during the entire experiment. This graph shows that nodes achieved approximately fair goodput: the difference between the largest and the smallest goodput is only 0.015pkts/sec! This is a surprising result since one might expect that allocating *same* sending rate to all nodes in a multi-hop environment would penalize nodes farther away from the sink, since they traverse more hops. RCRT maintains fairness because its rate adaptation mechanism conservatively adapts to the source that experiences congestion most along the path to the sink.

Figure 5 shows packet reception plot for all the nodes in the network. Each point on the curve is the time at which a packet with a particular sequence number was received. Since all of the packets were eventually delivered, the progress in sequence numbers approximately corresponds to the progress in number of packets received. The slope of each

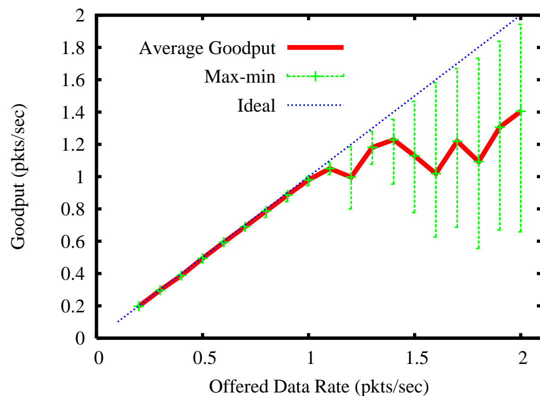

Figure 7—Average goodput achieved by best-effort transport. X-axis is the rate at which each node sourced packets. Y-Error bar represent the maximum and minimum goodput among all nodes

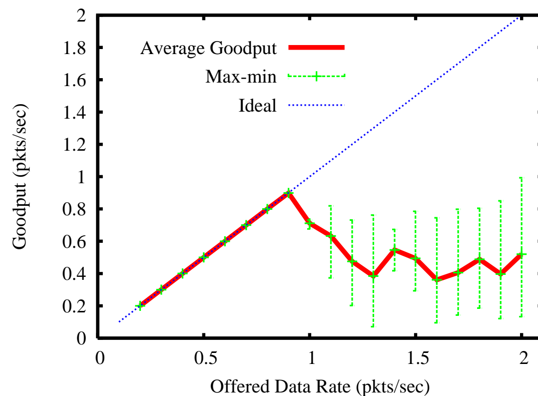

Figure 8—Average goodput achieved by reliable transport. X-axis is the rate at which each node sourced packets. Y-Error bar reprint the maximum and minimum goodput among all nodes

plot is the estimate of the instantaneous goodput that each node achieves at that point in time, and this figure shows that all nodes have approximately fair goodput throughout the experiment. The small spikes below the curve represent lost packets being repaired by RCRT's end-to-end loss recovery mechanism.

How many packets are recovered via end-to-end retransmission? Because of link-layer retransmissions, for all but 6 nodes, only 5% of the packets incurred end-to-end retransmissions, and for none of the nodes were more than 8% of the packets recovered end-to-end (Figure 6). This is encouraging, since one of the original design goals of RCRT's end-to-end reliability mechanism was to avoid a feedback implosion. We can quantify the overhead of feedback in RCRT. In this experiment, 4549 feedback packets were sent *in total* for 39000 data packets, representing an overhead of 11.6%. This feedback was used *both* to recover from losses and to adapt to congestion. Contrast this with TCP, in which every connection can see *half* as many ACK packets as data packets (most TCP implementations acknowledge every other packet).

4.2.2 Optimality

The baseline experiment demonstrates some of the salient features of RCRT's algorithms. The next question we address is: How close does RCRT's rate allocation get to the optimal? One way to evaluate the performance achieved by RCRT is to determine the maximum fair and reliable rate sustainable on the same network with same routing and MAC layers. We address this question by experimentally evaluating the goodput received by two different kinds of transport mechanisms at different offered loads. *Best-effort* transport sends data at a configured rate, but includes no end-to-end reliability and does not adapt to congestion. *Reliable* transport sends data at a configured rate, includes end-to-end reliability, but does not adapt to congestion.

Figures 7 and 8 plot the average goodput over all nodes for two transport mechanisms at various offered loads. The thick solid curve shows the average goodput achieved at the sink, the error-bars parallel to the y-axis indicate the maximum variation in node goodput at each offered load, and the straight dotted diagonal line represents the rate achievable

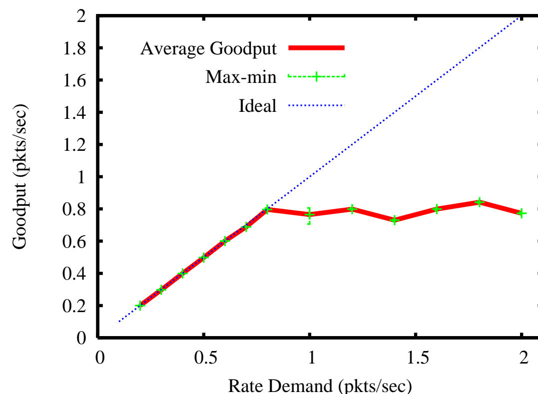

Figure 9—Average goodput achieved by RCRT. X-axis is the demand assigned to each node. Y-Error bar represent the maximum and minimum goodput among all nodes

with infinite capacity.

Figure 7 shows the maximum fair rate achievable without end-to-end reliability on our 40-node testbed. Until the offered load reaches 1.1 pkts/sec per node, all nodes achieve approximately fair goodput (small error bars) and 95.4% reliability (not shown). However, as the offered load increases above 1.2 pkts/sec, variability in goodput increases, and the reliability drops below 90%.

Figure 8 represents the achievable rate with end-to-end reliability, but no congestion control. Our network is able to sustain up to 0.9 pkts/sec per node. Thereafter, it experiences congestion collapse: acknowledgments and retransmissions use up much of the network capacity, resulting in less goodput and lower fairness than best-effort transport.

Since RCRT provides end-to-end reliability, we should compare its achieved rate with that of Figure 8. If we define 0.9 pkts/sec as the maximum sustainable rate for reliable transport, then, as Figure 9 shows, RCRT achieves nearly 0.8 pkts/sec per node, or 88% of the sustainable rate for reliable transport. In this experiment, we assigned all nodes equal demand. The figure plots this increasing demand on the x-axis. Of course, since RCRT is congestion-adaptive, sources only send at the assigned rates, not at their demands.

Another way to evaluate RCRT's optimality is to consider a single-source network. We conducted a single-source

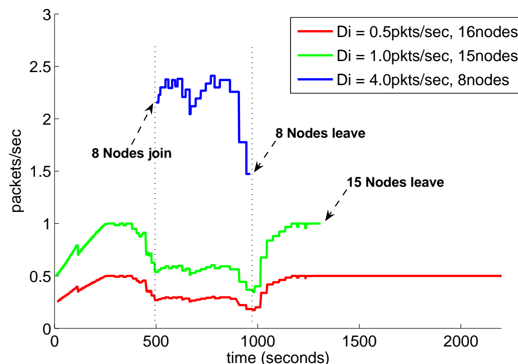

Figure 10—Rate r_i allocated to each node in the 40-node experiment with demand-proportional rate allocation policy when 8 nodes join in after 500 seconds

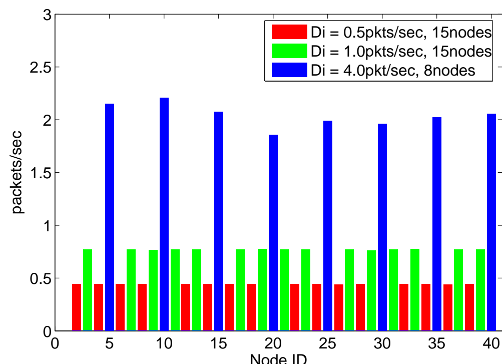

Figure 11—Per-flow goodput in the 40-node experiment with demand-proportional rate allocation policy when 8 nodes join in after 500 seconds

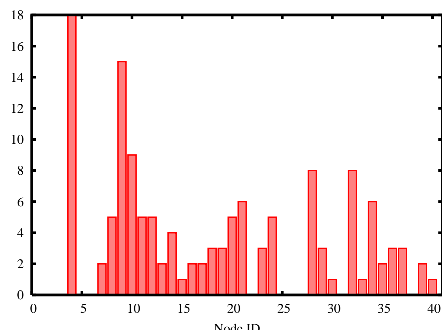

Figure 12—Number of routing parent changes during the 40-node experiment with demand-proportional rate allocation policy when 8 nodes join in after 500 seconds

single-sink experiment and found that best-effort transport can achieve up to 95 pkts/sec, and RCRT can achieve up to 60 pkts/sec on average. (As an aside, note that RCRT capable of achieving high goodput (60pkts/sec). In our previous experiments, the low per-node goodput (0.8pkts/sec) is mainly a function of the topology, not our protocol.) We can relate these experimental results to those observed on our larger topology as follows. In Figure 2, an instantaneous snapshot of the routing tree during one of our experiments, we estimate that node 13 is the most congested node: it has 26 children and 12 siblings. Let's say every node sends 1 unit of data per 1 unit of time. Then 13 receives 26 units of data and sends 27 units of data (including it's own data) per unit time. Also, 13 contends with its 12 siblings to reach the base-station. Hence, the channel capacity around 13 must be shared at least by total traffic of 65 units of data per unit time (= 26 + 27 + 12), even if we assume an ideal MAC. This means that the optimal sustainable rate in this network is $60/65 = 0.923$ for RCRT and $95/65 = 1.461$ for best-effort transport. These numbers match what we observe above, and confirms that RCRT achieves 87% of the sustainable rate.

In summary, RCRT manages to assign near-optimal rates by having congestion control functionality at the sink. We have compared the various transport protocols with the same radio, MAC, and routing layers, to ensure that our results are not affected by differences in the underlying protocol layers. Our results show that it is possible to estimate and manage overall network capacity in a centralized manner, and achieve high efficiency.

4.2.3 Robustness

In this section, we conduct an experiment that demonstrates RCRT's robustness, and also validates its flexibility in capacity allocation. In this experiment, nodes join and leave dynamically, and different nodes are assigned different demands. The network is configured to use a demand-proportional allocation policy. We set up three sets of flows that request different demands. Specifically, in this experiment, 31 flows start at time 0. Sixteen of these (which we will call set *A*) demand 1.0 pkts/sec and the other 15 flows (set *B*) demand for 0.5 pkts/sec. The remaining 8 flows (set C) join in after 500 seconds with a demand of 4 pkts/sec. All flows send the same total number of packets, but set *C* finishes earlier because of its higher demand.

Figure 10 show the rate allocation profile for this experiment. Recall that RCRT allocates exactly the same rate to all flows having the same demand. During the first 500 seconds of the experiment, sets *A* and *B* were allocated roughly the rate that they demanded: 0.5 and 1 pkts/sec. When the third set of flows *C* joined in with significantly higher demand, the network immediately experienced congestion, and the flows in both *A* and *B* were forced to reduce their rates. Between 500 and 1000 seconds into the experiment, all three sets of flows were active, and were all allocated rates proportional to their demands. When the flows in set *C* had completed at around 1000 seconds, the network had enough capacity to satisfy the demands of flows *A* and *B*. This result shows that RCRT is robust to node joins and leaves, its congestion detection mechanism and the rate adaptation mechanism successfully adapted the network-wide aggregate rate to the network state, and the rate allocation mechanism indeed allocated rates proportional to the demands of each node.

Figure 11 plots the goodput achieved by each node for this experiment. While nodes with identical demands achieved comparable goodputs, the average goodput between different sets is, interestingly, *not* exactly proportional to their demands. Specifically, the average goodput achieved by sets *A*, *B*, and *C* is 0.44, 0.77, and 2.04 pkts/sec respectively, while their demands are 0.5, 1.0, and 4.0 pkts/sec respectively. This is because each set experienced network congestion for different fractions of their lifetimes. Since the flows in set *C* experienced congestion during their entire lifetime and only

Set	App. ID	Demand	Num.pkts	Num.nodes
A_1	1	1.0 pkts/sec	1000	19
B_1	1	0.5 pkts/sec	500	18
A_2	2	1.0 pkts/sec	500	19
B_2	2	0.5 pkts/sec	250	18

Table 4—Setup for 2-application, 2-sink experiment

achieved half the goodput of their demand, the average $\frac{r_i}{d_i}$ during this congested period is about 0.5 ($2.04/4.0 \approx 0.5$). Since set B experienced congestion and was assigned half the demanded rate for half of its lifetime, its expected goodput is around 75% of what it would have achieved in an uncongested network. This roughly matches the goodput that set B actually achieved ($0.5\frac{1}{2} + 1.0\frac{1}{2} = 0.75 \approx 0.77$). Finally, set A was assigned half the rate for $1/4$ of its lifetime, which amounts to $0.25\frac{1}{4} + 0.5\frac{3}{4} = 0.4325$ pkts/sec, which is close to the observed 0.44 pkts/sec.

Finally, Figure 12 shows how frequently each node changed its routing parent during this 38-minute experiment. Even though, on average, each node changed its parent 3.4 times, RCRT assigned rates correctly to all the flows. This also highlights the robustness of RCRT's design and implementation.

4.2.4 Flexibility

In this section, we demonstrate that RCRT achieves two more of its original goals: that it can support multiple concurrent applications, and that each application can use different capacity allocation policies.

We ran two separate "applications" with two sinks. Each application ran on a different sink and used different rate allocation policies: one used demand-proportional allocation, and the other used demand-limited allocation. The two sinks at the upper-tier was connected via 802.11b wireless. Nodes 15 and 30 were the gateway motes on our testbed connected to the two sinks. Each sink was a Stargate running Linux. The motes used a multi-sink version of MultiHopLQI, so that two trees were formed, one rooted at each sink. We used additional routing software that allowed both sinks to receive data packets from *all* motes. Thus, this set up represents two applications running on a tiered network.

In this experiment, we used 37 motes. The experiment was set up as shown in Table 4. The demand-proportional application comprised two sets of nodes A_1 and B_1, the first set being assigned twice the demand as the second. The demand-limited application comprised the same two sets (denoted A_2 and B_2) respectively, with identical respective demand assignments. The total aggregate demand is 56 pkts/sec, enough to saturate the network.

Figure 13 shows the rate allocation profile for this experiment. Flows in application 1 (sets A_1 and B_1) were assigned rates proportional to their demands, and flows in application 2 (sets A_2 and B_2) were assigned rates limited by their demands. Thus, notice that even though flows in A_1 and A_2 *had the same demand*, they each get different rate allocations because the applications use different capacity allocation policies. Also note that all flows in the demand-limited application get the same rate; the sustainable network rate was below the 0.5 pkts/sec demanded by B_2. All flows experienced congestion from time 0 till about 600 seconds when

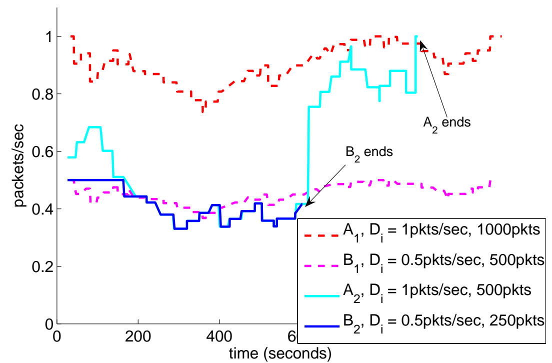

Figure 13—Rate allocated to each node in 39-node experiment with 2-applications running on2-sinks

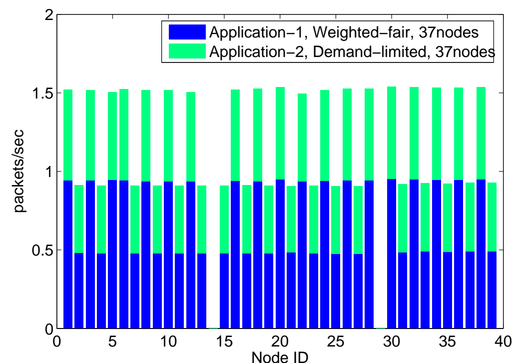

Figure 14—Per-node goodput (two flows for same node stacked together) in 39-node experiment with 2-applications running on 2-sinks

all four sets of flows were active (a total of 74 flows). After set B_2 finished at 600 seconds, the other flows were allocated higher rates to take advantage of the increased available capacity.

Finally, Figure 14 shows the goodput achieved at the sink by each node. Two flows (for two different applications) from the same node are stacked together to show the total goodput achieved by each node. The average goodputs achieved by the two applications are 0.718 pkts/sec and 0.508 pkts/sec respectively, which totals 1.226 pkts/sec. This brings up an important point. The total achieved goodput is approximately 60% higher than the single-sink 40-node experiment (Section 4.2.1). This comes from using a tiered network. The two sinks are near the center of the network and roughly have comparably sized sub-trees. Moreover, the two sinks use 802.11b radios to communicate with each other, which has at least an order of magnitude higher bandwidth. This experiment not only shows that RCRT can support multiple concurrent applications on a tiered network with multiple sinks, but also quantifies the capacity increase achievable by using RCRT on a tiered network. Furthermore, although we have left inter-sink cooperative rate control as future work, this experiment shows that independent congestion control decisions made by different sinks resulted in reasonable (although not perfect) behavior.

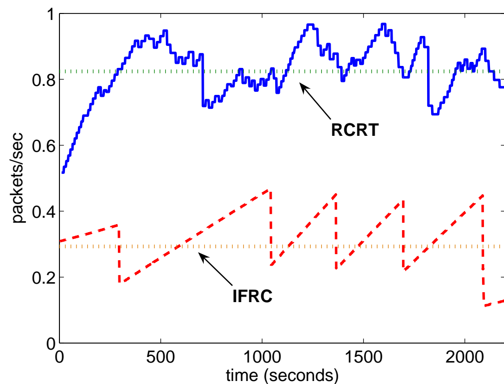

Figure 15—Rate achieved by IFRC and RCRT in 30-node experiment

4.2.5 *Comparison*

In this section, we compare the performance of RCRT with that of IFRC [23], a recently proposed interference-aware distributed rate-control protocol.

Figure 15 shows the rate achieved by IFRC together with RCRT's rate allocation in a 30-node experiment. The two protocols are qualitatively different, since IFRC does not provide end-to-end reliable transmissions. However, we are interested in comparing their congestion control efficacy: IFRC does distributed congestion control, while RCRT performs congestion control at the sink, and we explore to what extent these approaches are quantitatively different. To compare these two protocols, we run them on the same set of nodes with the same radio power. RCRT was configured to use the fair allocation policy. However, IFRC has been evaluated only on static routing trees [23], so we ran IFRC on an empirically-determined "good" tree. RCRT runs with dynamic routing enabled.

In Figure 15, the solid line represents the rate allocated to all nodes in RCRT, and the dashed line represents the rate achieved by one of the nodes in IFRC. Since all nodes were allocated the same rate in RCRT, a single line is sufficient to show rate allocation of *all* nodes. In IFRC, all nodes adapt their rates in near-synchrony, and rate plots for various nodes overlap with each other. We only plot the rate adaptation plot for one node for clarity. The dotted horizontal lines show the average rate achieved by each protocol during the experiment.

The results show that RCRT achieves an average rate of 0.824 pkts/sec in this experiment, which is more than twice the rate achieved by IFRC: 0.293 pkts/sec.

We believe two main reasons account for this difference. The first, of course, is that much of RCRT's performance advantage comes from implementing its congestion control functionality at the sink, which has a more global view of network state. This results in less pronounced rate deviations in RCRT's rate allocation profile. A second reason is that IFRC aggressively *avoids congestion* whereas RCRT *detects congestion* after it has happened. To avoid dropping packets, IFRC detects incipient congestion and aggressively cuts its rate when queues exceed a small threshold. On the other hand, RCRT fully utilizes the network queues until packets are lost. RCRT can afford packet loss, since it has a built-in loss recovery mechanism. That said, it should be possible to improve IFRC performance by varying its parameters and making it less aggressive in avoiding congestion, at the possible expense of lower end-to-end goodput. In preliminary experiments, we have found that a hardware limitation of our current platform degrades the performance of IFRC. IFRC, by design, requires the radio and MAC to run in promiscuous mode. Our experimental platform does not permit the use of link-level acknowledgments along with promiscous mode, so the IFRC implementation uses software acknowledgments for link-level retransmission, which adds some software delays in the MAC layer and reduces IFRC throughput. When RCRT uses software acknowledgements, our preliminary experiments indicate that its average goodput is about 1.7 times that of IFRC. We have left a more extensive evaluation of this to future work.

We have left a more detailed comparison between these two protocols to future work, but believe that RCRT will always provide higher overall goodput because of its centralized design.

5 Conclusions and Future Work

RCRT is a reliable transport protocol for wireless sensor networks. RCRT places its congestion control functionality at the sink, whose perspective into the network enables better aggregate control of traffic, and affords flexibility in rate allocation. It supports multiple concurrent applications, and is robust to network dynamics. Finally, RCRT's rates are significantly higher than that of the state-of-the-art in sensor network congestion control. We envision several interesting directions for future work including coordinated rate allocation across sinks, differential treatment for flows unconstrained by the bottleneck region, and improved convergence time in networks with highly varing RTTs.

References

[1] TinyOS. http://www.tinyos.net/.

[2] A. M. Ali, K. Yao et al. An Empirical Study of Collaborative Acoustic Source Localization. In *Proc. of the IPSN/SPOTS*, 2007.

[3] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. *SIGCOMM Comput. Commun. Rev.*, 29(4):175–187, 1999.

[4] F. Bian, S. Rangwala, and R. Govindan. Quasi-static Centralized Rate Allocation for Sensor Networks. In *Proc. IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, 2007.

[5] J. Caffrey, R. Govindan et al. Networked Sensing for Structural Health Monitoring. In *Proceedings of the 4th International Workshop on Structural Control*, 2004.

[6] K. Chintalapudi, J. Paek et al. Structural Damage Detection and Localization Using NetSHM. In *Proc. IPSN/SPOTS'06*, 2006.

[7] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, 1989.

[8] L. Eggert, J. Heidemann, and J. Touch. Effects of ensemble-tcp. *ACM Computer Communication Review*, 30(1):15–29, 2000.

[9] S. Floyd. Congestion control principles. RFC2914, 2000.

[10] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[11] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[12] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. ACM SIGCOMM Conference*, 2000.

[13] O. Gnawali, B. Greenstein et al. The TENET Architecture for Tiered Sensor Networks. In *Proc. 4th ACM Conference on Embedded Networked Sensor Systems (SenSys'06)*, 2006.

[14] J. Hui and D. Culler. The dynamic behavior of a data dissemination algorithm at scale. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004.

[15] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating Congestion in Wireless Sensor Networks. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004.

[16] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *ACM/IEEE Transactions on Networking*, 11(1), 2002.

[17] Y. Iyer, S. Gandham, and S. Venkatesan. STCP: a generic transport layer protocol for wireless sensor networks. In *Proc. International Conference on Computer Communications and Networks*, 2005.

[18] V. Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM Conference*, 1988.

[19] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, 2002.

[20] S. Kim, R. Fonseca et al. Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Network. Technical Report UCB/EECS-2006-169, EECS, University of California, Berkeley, 2006.

[21] M. Rahimi, R. Baer et al. Cyclops: In situ image sensing and interpretation in wireless sensor networks. In *Proc. 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys'05)*, 2005.

[22] K. K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with Connectionless Network Layer. *ACM/IEEE Transactions on Networking*, 8(2):158–181, 1990.

[23] S. Rangwala, R. Gummadi, R. Govindan, and K. Psounis. Interference-Aware Fair Rate Control in Wireless Sensor Networks. In *Proc. ACM SIGCOMM Symposium on Network Architectures and Protocols*, 2006.

[24] Y. Sankarasubramaniam, O. B. Akan, and I. F. Akyildiz. ESRT: Event-to-sink Reliable Transport in Wireless Sensor Networks. In *Proc. 4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc '03)*, 2003.

[25] F. Stann and J. Heidemann. RMST: Reliable Data Transport in Sensor Networks. In *Proc. 1st IEEE Workshop on Sensor Network Protocols and Applications (SNPA)*, 2003.

[26] T. Stathopoulos, L. Girod, J. Heidemann, and D. Estrin. Mote herding for tiered wireless sensor networks. Technical Report 58, CENS, Dec. 7 2005.

[27] USC/ISI. Transmission control protocol. RFC793, 1981.

[28] C. Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks. In *Proc. 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[29] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell. CODA: Congestion Detection and Avoidance in Sensor Networks. In *Proc. 1st ACM Conference on Embedded Networked Sensor Systems (SenSys'03)*, 2003.

[30] N. Xu, S. Rangwala et al. A Wireless Sensor Network for Structural Monitoring. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004.

[31] H. Zhang, A. Arora, Y. Choi, and M. Gouda. Reliable bursty convergecast in wireless sensor networks. In *Proc. 4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc '03)*, 2003.