Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Networks

Sukun Kim[†], Rodrigo Fonseca[†], Prabal Dutta[†], Arsalan Tavakoli[†]
David Culler[†], Philip Levis[⋆], Scott Shenker[†‡], and Ion Stoica[†]

[†]Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

[‡]ICSI
1947 Center Street
Berkeley, CA 94704

[⋆]Computer Systems Lab
Stanford University
Stanford, CA 94305

Abstract

We present Flush, a reliable, high goodput bulk data transport protocol for wireless sensor networks. Flush provides end-to-end reliability, reduces transfer time, and adapts to time-varying network conditions. It achieves these properties using end-to-end acknowledgments, implicit snooping of control information, and a rate-control algorithm that operates at each hop along a flow. Using several real network topologies, we show that Flush closely tracks or exceeds the maximum goodput achievable by a hand-tuned but fixed rate for each hop over a wide range of path lengths and varying network conditions. Flush is scalable; its effective bandwidth over a 48-hop wireless network is approximately one-third of the rate achievable over one hop. The design of Flush is simplified by assuming that different flows do not interfere with each other, a reasonable restriction for many sensornet applications that collect bulk data in a coordinated fashion, like structural health monitoring, volcanic activity monitoring, or protocol evaluation. We collected all of the performance data presented in this paper using Flush itself.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol architecture

General Terms

Design, Reliability

Keywords

Wireless Sensor Networks, Transport, Interference

1 Introduction

This paper presents the design and implementation of Flush, a reliable, high-goodput, bulk data transport protocol for wireless sensor networks. Flush is motivated by the data transfer needs of sensornet applications like structural health monitoring, volcanic activity monitoring, and bulk data collection [20, 30, 14]. Some of these applications cover large physical extents, measured in kilometers, and have network depths that range from a few to over forty hops. The Golden Gate Bridge structural health monitoring project [14] is one such example: 64 nodes (46 hops) span 4,200 feet and must operate continuously for weeks. Power concerns and challenging radio environments can make using smaller diameter networks built from higher-power radios unappealing. While delivery of bulk data to the network edge may sound simple, the nature of wireless communication brings several challenges for efficient and reliable delivery in multihop networks: links are lossy [28], inter-path interference is hard to cope with [12, 22], intra-path interference is hard to avoid [31], and transient rate mismatches can overflow queues. These challenges make naïve or greedy approaches to multihop wireless transport difficult.

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

Intra-path interference occurs when transmissions of the same packet by successor nodes prevent the reception of the following packet from a predecessor node. If the packet sending rate is set too high, persistent congestion occurs and goodput suffers – a condition that is potentially undetectable by single-hop medium access control algorithms due to hidden terminal effects. If the packet sending rate is set too low, then channel utilization suffers and data transfers take longer than necessary. The optimal rate also depends on the route length and link quality. Since in a dynamic environment, some of these factors may change during the course of a transfer, we show that no static rate is optimal at all hops and over all links. This suggests that a dynamic rate control algorithm is needed to estimate and track the optimal transmission rate.

Flush achieves its goals with a combination of mechanisms. It uses a simple, sink-initiated control protocol to coordinate transfers, with end-to-end selective negative acknowledgments and retransmissions to provide reliability. In the transfer phase, Flush finds the available bandwidth along a path using a combination of local measurements and a novel interference estimation algorithm. Flush efficiently

SenSys'07, November 6–9, 2007, Sydney, Australia.

communicates this rate to every node between the bottleneck and source, allowing the system to find and maintain the maximum rate that avoids intra-path interference. On long paths, Flush pipelines packets over multiple hops, maximizing spatial reuse.

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

We have implemented Flush in TinyOS [10] and evaluated it using a 100-node Mirage testbed [2, 5] as well as an ad hoc, 79-node outdoor network. The results show that Flush's rate control algorithm closely tracks or exceeds the maximum effective bandwidth sustainable using an optimized fixed rate. Furthermore, Flush's performance improvements scale to very long routes: our experimental results include results from a 48-hop network. On the MicaZ platform, our implementation of Flush requires just 629 bytes of RAM and 6,058 bytes of ROM.

We describe the Flush protocol in Section 2 and our implementation in Section 3. In Section 4 we compare Flush to standard routing protocols as well as to fixed rate algorithms, and distinguish the contributions that layer 3 and layer 4 congestion control have on goodput. In Section 5 we present and address some open concerns. Section 6 places Flush in the context of the large prior literature on transport reliability, rate optimization, congestion control, and flow control, and Section 7 we present our concluding thoughts.

2 Flush

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

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

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

The reverse delivery service need not route the packets; a simple flood or a data-driven virtual circuit is sufficient. The distinction between forward routing and reverse delivery exists because arbitrary, point-to-point routing in sensornets is uncommon and unnecessary for Flush.

2.1 Overview

To initiate a data transfer, the sink sends a request for a data object to a specific source in the network using the underlying delivery protocol. Naming of the data object is outside of the scope of Flush, and is left to an application running above it. After a request is made, Flush moves through four phases: topology query, data transfer, acknowledgment, and integrity check.

The topology query phase probes the depth of a target node to tune the RTT and compute a timeout at the receiver. During the data transfer phase, the source sends packets to the sink using the maximum rate that does not cause intra-path interference. Over long paths, this rate pipelines packets over multiple hops, spatially reusing the channel. Section 2.3 provides intuition on how this works, and describes how Flush actively estimates this rate. The initial request contains conservative estimates for Flush's runtime parameters, such as the transmit rate. When it receives the request, the data source starts sending the requested data packets, and nodes along the route begin their dynamic rate estimation. On subsequent requests or retransmissions, the sink uses estimated, rather than conservative, parameters.

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

This process repeats until the sink has received the requested data in total. When that occurs, the sink verifies the integrity of the data. If the integrity check fails, the sink discards the data and sends a fresh request. If the check succeeds, the sink may request the next data object, perhaps from another node. Integrity is checked at the level of both packets and data objects.

To minimize control traffic overhead, Flush bases its estimates on local data and snoops on control information in forwarded data packets. The only explicit control packets are those used to start a flow and request end-to-end retransmissions. The overhead of these control packets are amortized even further by the large data size of target applications. Flush is miserly with packet headers as well: three 1-byte fields are used for rate control and one field is used for the sequence number. The use of few control packets and small protocol headers helps to maximize data throughput, reducing transfer time. Section 3 describes a concrete implementation of the Flush protocol.

2.2 Reliability

Flush uses an end-to-end reliability protocol to be robust to node failures. Figure 1 shows a conceptual session of the protocol, where the data size is 9 packets, and a NACK packet can accommodate at most 3 sequence numbers. In the

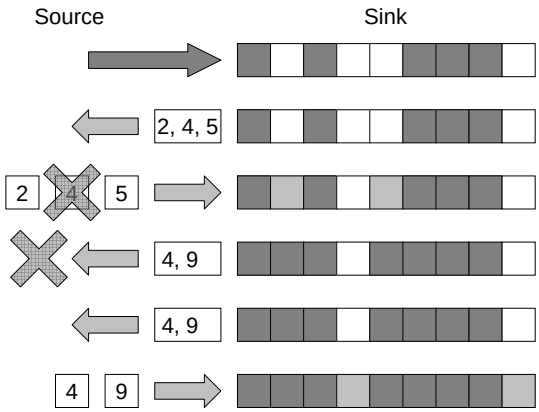


Figure 1. NACK transmission example. Flush has at most one NACK packet in flight at once.

data transfer stage, the source sends all of the data packets, of which some are lost (2, 4, 5, and 9 in the example), either due to retransmission failures or queue overflows. The sink keeps track of all received packets. When it believes that the source has finished sending data, the sink sends a single NACK packet, which can hold up to 3 sequence numbers, back to the source. This NACK contains the first 3 sequence numbers of lost packets, 2, 4, and 5. The source retransmits the requested packets. This process continues until the sink has received every packet. The sink uses an estimate of the round-trip time (RTT) to decide when to send NACK packets in the event that all of the retransmissions are lost.

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

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

2.3 Rate Control

The protocol described above achieves Flush's first goal: reliable delivery. Flush's second goal is to minimize transfer time. Sending packets as quickly as the data link layer will allow poses problems in the multihop case. First, nodes forwarding packets cannot receive and send at the same time. Second, retransmissions of the same packet by successive nodes may prevent the reception of the following packets, in what is called *intra-path interference* [28]. Single-hop medium access control algorithms cannot solve the problem because of the hidden terminal effect. Third, rate mismatches may cause queues further along the path to overflow, leading to packet loss, wasted energy, and additional end-to-end retransmissions.

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

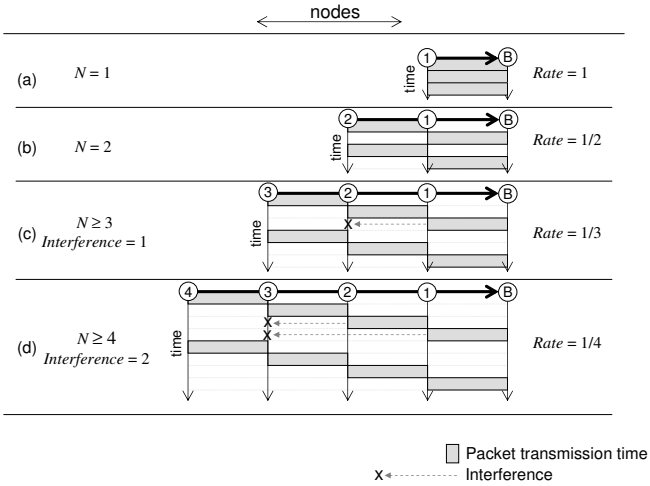


Figure 2. Maximum sending rate without collision in the simplified pipelining model, for different number of nodes (N) and interference ranges (I).

2.3.1 A Conceptual Model

In this simplified model, there are N nodes arranged linearly plus a basestation B. Node N sends packets to the basestation through nodes $N-1,...,1$. Nodes forward a packet as soon as possible after receiving it. Time is divided in slots of length 1s, and nodes are synchronized. They can send exactly one packet per slot, and cannot both send and receive in the same slot. Nodes can only send and hear packets from neighbors one hop away, and there is no loss. There is however a variable range of interference, I: a node's transmission interferes with the reception of all nodes that are I hops away.

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

Figure 2 shows the maximum rate we can achieve in the simplified pipeline model for key values of N and I. If there is only one node, as in Figure 2(a), it can send to the basestation at the maximum rate of 1 pkt/s. There is no contention, as no other nodes transmit. For two nodes (b), the maximum rate falls to $1/2$, because node 1 cannot send and receive at the same time. The interference range starts to play a role if $N \geq 3$. In (c), node 3 has to wait for node 2's transmission to finish, and for node 1's, because node 1's transmission prevents node 2 from receiving. This is true for any node beyond 3 if we keep I constant, and the stable maximum rate is $1/3$. Finally, in (d) we set I to 2. Any node past node 3 has to wait for its successor to send, and for its successor's *two* successors to send. Generalizing, the maximum transmission rate in this model for a node N hops away with interference range I is given by

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

Thus, the maximum rate at which nodes can send depends on the interference range at each node, and on the path length (for short paths). If nodes send faster than this rate, there will be collisions and loss, and the goodput can greatly suffer. If nodes send slower than this rate, throughput will be lower

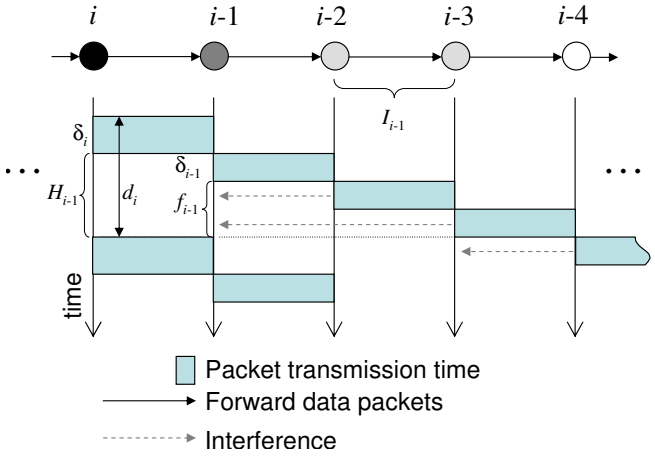


Figure 3. A detailed look at pipelining from the perspective of node *i*, with the quantities relevant to the algorithm shown.

than the maximum possible. The challenge is to efficiently discover and communicate this rate, which will change with the environment.

2.3.2 Dynamic Rate Control

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

- **Rule 1:** A node should only transmit when its successor is free from interference.
- **Rule 2:** A node's sending rate cannot exceed the sending rate of its successor.

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

Establishing the best rate requires each node i to determine the smallest safe inter-packet delay d_i (from start to start) that maintains Rule 1. As shown in Figure 3, d_i comprises the time node i takes to send a packet, δ_i, plus the time it takes for its successor to be free from interference, H_{i-1}. δ_i is measured between the start of the first attempt at transmitting a packet and the first successfully acknowledged transmission. H_{i-1} is defined for ease of explanation, and has two components: the successor's own transmission time δ_{i-1} and the time f_{i-1} during which its interfering successors are transmitting. We call the set of these interfering nodes I_{i-1}. In summary, for node i, $d_i = \delta_i + (\delta_{i-1} + f_{i-1})$: the minimum delay is the sum of the time it takes a node to transmit a packet, the time it takes the next hop to transmit the packet, and the time it takes that packet to move out of the next hop's interference range.

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

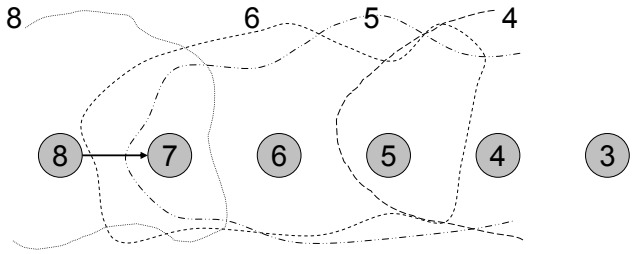


Figure 4. Packet transfer from node 8 to node 7 interferes with transfer from node 5 to node 4. However it does not interfere with transfer from node 4 to node 3

Let us look at an example. In Figure 4, node 7 determines, by overhearing traffic, that the transmissions of node 6 and 5 (but not node 4) can interfere with reception of traffic from node 8. This means that node 7 can not hear a new packet from node 8 until node 5 finishes forwarding the previous packet. Thus, $f_7 = \delta_6 + \delta_5$. Node 7 can not receive a packet while sending, and $H_7 = \delta_7 + f_7$. Considering node 8's own transmission time, $d_8 = \delta_8 + H_7 = \delta_8 + \delta_7 + f_7 = \delta_8 + \delta_7 + \delta_6 + \delta_5$. So the interval between two packets should be separated by at least that time.

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

When applied recursively, Rule 2 leads to the correct sending interval at node i: $D_i = \max(d_i, D_{i-1})$. Most importantly, this determines the sending interval at the source, which is the maximum d_i over all nodes. This rate is easy to determine at each node: all nodes simply include D_i in their data packets, so that the previous node can learn this value by snooping. To achieve the best rate it is necessary and sufficient that the source send at this rate, but as we show in Section 4, it is beneficial to impose a rate limit of D_i for each node i in the path, and not only for the source. Flush take an advantage of in-network rate control on a hop-by-hop basis. Figure 5 presents a concise specification of the rate control algorithm and how it embodies the two simple rules described above.

Finally, while the above formulation works in a steady state, environmental effects and dynamics as well as simple probability can cause a node's D_i to increase. Because it takes n packets for a change in a delay estimate to propagate back n hops, for a period of time there will be a rate mismatch between incoming and outgoing rates. In these cases, queues will begin to fill. In order to allow the queues to drain, a node needs to temporarily tell its previous hop to

The Flush rate control algorithm

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

Figure 5. The Flush rate control algorithm. D_i determines the smallest sending interval at node i.

slow down. We use a simple mechanism to do this, which has proved efficient: while a node's queue occupancy exceeds a specified threshold, it temporarily increases the delay it advertises by doubling δ_i.

2.4 Identifying the Interference Set

A node can interfere with transmissions beyond its own packet delivery range. The fact that Flush assumes it can hear its interferers raises the question of how common this effect is in the routes it chooses. Packet reception rates follow a curve with respect to the signal-to-noise ratio, but for simplicity's sake we consider the case where it follows a simple threshold, such that reception is worse than reality.

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

Using the 100-node Mirage testbed, we examined the topology Flush's underlying routing algorithm, MintRoute [32], establishes. We measured the noise floor of each node by sampling the CC2420 RSSI register and the signal strength of its predecessor using TinyOS packet metadata. Figure 6 shows the results. Fewer than 20% of the links chosen are within the range $[N_i + T, N_i + 2T]$. For there to be a jammer, it must be within T of these links, or approximately 3.5dBm. While Flush's interference estimation is not perfect, its window of inaccuracy is narrow.

3 Implementation

To empirically evaluate the Flush algorithms, we implemented them in the nesC programming language [8] and the TinyOS [11] operating system for sensor networks. Our implementation runs on the Crossbow MicaZ platform but we believe porting it to other platforms like the Mica2 or Telos would be straightforward.

3.1 Protocol Engine

The Flush *protocol engine* implements the reliable block transfer service. This module receives and processes data read requests and selective NACKs from the receiver. The requests are forwarded to the application, which is responsible for returning the requested data. After the requested data have been returned, the protocol engine writes the data and identifying sequence numbers into a packet and then sub-

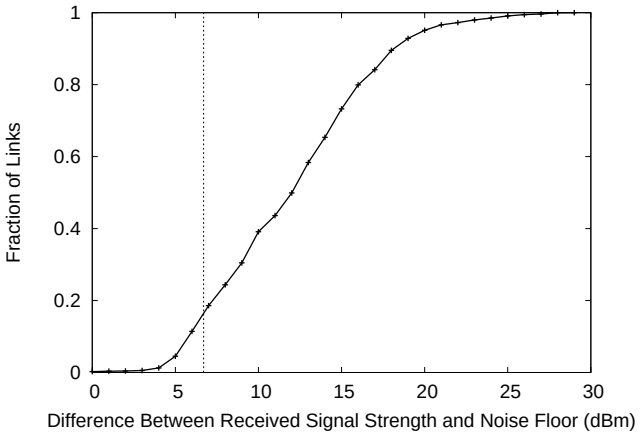


Figure 6. CDF of the difference between the received signal strength from a predecessor and the local noise floor. The dotted line indicated twice the SNR threshold. Links with an SNR exceeding this threshold will not be undetectably affected by interferers. A large fraction of interferers are detectable and avoidable.

mits the packet to the routing layer at the rate specified by the packet delay estimator, which is discussed below.

Although the Flush interface supports 32-bit offsets, our current implementation only supports a limited range of data object sizes: 917,504 bytes (64K x 14 bytes/packet), 1,114,112 bytes (64K x 17 bytes/packet), or 2,293,760 bytes (64K x 35 bytes/packet), depending on the number of bytes available for the data payload in each packet. This restriction comes from the use of 16-bit sequence numbers, which we use in part to conserve data payload and in part because the largest flash memory available on today's sensor nodes is 1 MB. Individual packets are protected using link layer CRCs and the entire data objects is protected using a simple checksum, although more robust methods could be used.

3.2 Routing Layer

MintRoute [32] is used to convergecast packets from the source to the sink, which in our case is the root of a collection tree. Flush does not place many restrictions on the path other than it be formed by reasonably stable bidirectional links. Therefore, we believe Flush should work over most multihop routing protocols like CLDP [15], TinyAODV [1], or BVR [7]. However, we do foresee some difficulty using routing protocols that do not support reasonably stable paths. Some routing protocols, for example, dynamically choose distinct next hops for packets with identical destinations [19]. It is neither obvious that our interference estimation algorithm would work with such protocols nor clear that a high rate could be achieved or sustained because Flush would be unable to coordinate the transmissions of the distinct next hops.

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

the reverse traffic would be infrequent. For example, the deployment at the Golden gate Bridge [13] indicates that about 1126 times more packets are received than sent at the basestation. The control overhead of more sophisticated routing layers is likely to exceed the traffic generated by infrequent flooding, so we do not focus our efforts on optimizing the data transfer in the reverse direction.

3.3 Packet Delay Estimator

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

The Flush estimator extracts the received signal strength indicator (RSSI) of packets received from the predecessor hop and snooped from all successor hops along the routing path. Flush assumes some type of received signal strength indicator (RSSI) is provided by the radio hardware. These RSSI values are smoothed using an exponentially-weighted moving average to filter out transients on single-packet timescales. History is weighted more heavily because RSSI is typically quite stable [25] and outliers are rare, so a single outlier should have little influence on the RSSI estimate. A node i considers an successor node $(i-j)$ an interferer of node $i+1$ at time t if, for any $j > 1$, $rssi_{i+1}(t) - rssi_{i-j}(t) <$ 10 dBm. The threshold of 10 dBm was chosen after consulting the literature [21] and empirically evaluating a range of values.

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

3.4 Queuing

Queues provide buffer space during transient rate mismatches which are typically due to changes in link quality. In Flush, these mismatches can occur over short time scales because rate estimates are based on averaged interval values, so unexpected losses or retransmissions can occur. Also, control information can take longer to propagate than data: at a node i along the path of a flow, data packets are forwarded with a rate $\frac{1}{\delta_i}$ while control information propagates in the reverse direction with a rate $\frac{1}{\delta_i + f_i}$. The forwarding interference time f_i is typically two to three times larger than the packet sending delay δ_i, so control information flows three to four times slower than data. Since it can take some time for the control information to propagate to the source, queues provide buffering during this time.

Our implementation of Flush uses a 16-deep *rate limited queue*. Our queue is a modified version of `QueuedSend`, the standard TinyOS packet queue. Our version, called `RatedQueuedSend`, implements several functions that are not available in the standard component. First, our version measures the local forwarding delay, δ, and keeps an exponentially-weighted moving average over it. This smoothed version of δ is provided to the packet estimator. Second, `RatedQueuedSend` enforces the queue departure delay D_i specified by the packet delay estimator. Third, when a node becomes congested, it doubles the delay advertised to its descendants but continues to drain its own queue with the smaller delay until it is no longer congested. We chose a queue depth of 5, about one-third of the queue size, as our congestion threshold. Fourth, the queue inserts the then-current local delay information into a packet immediately preceding transmission. Fifth, `RatedQueuedSend` retransmits a packet up to four times (for a total for five transmissions) before dropping it and attempting to send the next packet. Finally, the maximum queuing delay is bounded, which ensures the queue will be drained eventually, even if a node finds itself neighborless.

3.5 Link Layer

Flush employs link-layer retransmissions to reduce the number of expensive end-to-end transmissions that are needed. Flush also snoops on the channel to overhear the next hop's delay information and the predecessor hop and successor hops' RSSI values. Unfortunately, these two requirements – hardware-based link layer retransmission and snooping – are at odds with each other on the MicaZ mote. The CC2420 radio used in the MicaZ does not simultaneously support hardware acknowledgments and snooping, and the default TinyOS distribution does not provide software acknowledgments. Our implementation enables the snooping feature of the CC2420 and disables hardware acknowledgments. We use a modified version of the TinyOS MAC, `CC2420RadioM`, which provides software acknowledgments [22]. We configure the radio to perform a clear channel assessment and employ CSMA/CA for medium access. Since Flush performs rate control at the network layer, and does not schedule packets at the link layer, CSMA decreases collision due to retransmissions during transient periods.

3.6 Protocol Overhead

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

4 Evaluation

We perform a series of experiments to evaluate Flush's performance. We first establish a baseline using fixed rates

against which we compare Flush's performance. This baseline also allows us to factor out overhead common to all protocols and avoid concerning ourselves with questions like, "why is there a large disparity between the raw radio rate of 250 kbps and Flush?" We also look at where time is spent, and where packets are lost for the different algorithms. Next, we take a more detailed look at Flush's performance: we explore the benefits of Flush's hop-by-hop rate control compared to controlling the rate only at the source. We also consider the effects of abrupt link quality changes on Flush's performance and analyze Flush's response to a parent change in the middle of a transfer. The preceding experiments are carried out on the Mirage testbed [2, 5]. We consider Flush's scalability by evaluating its performance over a 48-hop, ad hoc, outdoor wireless network. To the best of our knowledge, this is the longest multihop path used in evaluating a protocol in the wireless literature. Finally, we present Flush's code and memory footprint.

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

4.1 Testbed Methodology

We evaluate the effectiveness of Flush through a series of experiments on the Intel Research Berkeley sensornet testbed, Mirage, as well as a 79-node, ad hoc, outdoor testbed. The Mirage testbed consists of 100 MicaZ nodes. We used node 0, in the south west corner, as the sink, or basestation. Setting the MicaZ node's CC2420 radio power level to -11 dBm, the diameter of the resulting network varied between 6 and 7 hops in our experiments. The end-to-end quality of the paths was generally good, but in Section 4.5 we present the results of an experiment in which the quality of a link was artificially degraded. The outdoor testbed consisted of 79 nodes deployed in a linear fashion with 3ft spacing in an open area, creating an approximately 48-hop network.

We use the MintRoute [32] protocol to form a collection tree with the sink located at the root. MintRoute uses periodic beacons to update link quality estimates. Prior to starting Flush, we allow MintRoute to form a routing tree, and then freeze this tree for the duration of the Flush transfer. In Section 5, we discuss Flush's interactions with other protocols and applications.

We look at the following metrics when analyzing Flush's performance:

- **Overall Throughput:** the number of unique data packets or bytes received at the sink divided by the total transfer time. This metric considers all of Flush's phases and overhead.
- **Transfer Phase Throughput:** the number of unique data packets or bytes received at the sink *during Flush's transfer phase*, divided by the transfer phase duration.

In our experiments the sink issues a request for data from a specific node. All the nodes are time-synchronized prior to each trial, and they log to flash memory the following information for each packet sent: the Flush sequence number, timestamp, the values of δ, f, and D, and the instantaneous queue length. After each run we collect the data from all nodes *using Flush itself*. We compare Flush with a static algorithm that fixes the sending rate. Then, to evaluate the benefits of using hop-by-hop in-network rate control, we compare Flush with a variation which only adjusts the sending rate at the *source*, even though the intermediate nodes still estimate and propagate the delays as described in Section 2.3.

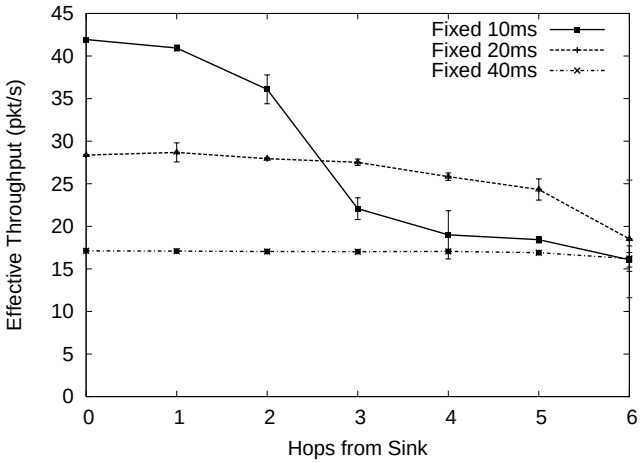


Figure 7. Overall packet throughput of fixed rate streams over different hop counts. The optimal fixed rate varies with the path length.

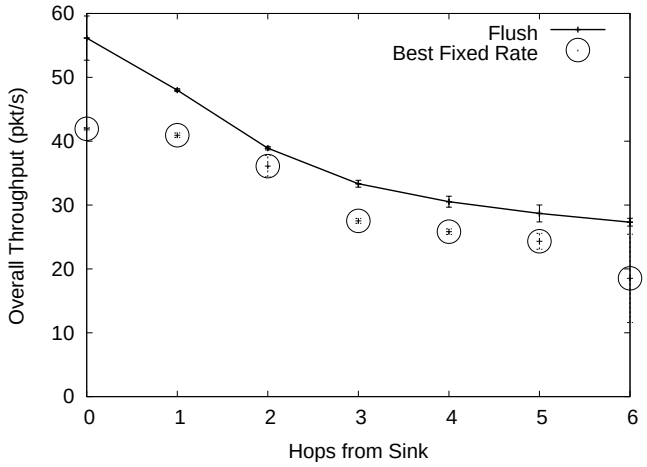
4.2 High Level Performance

In this section, we examine the overall throughput by comparing Flush to various values of the fixed-rate algorithm. To establish a baseline, we first consider the overall packet throughput achieved by the fixed rate algorithm. For each sending interval of 10, 20, and 40ms, we reliably transfer 17,000 bytes along a fixed path of length ranging from zero to six hops. The smallest inter-packet interval our hardware platform physically sustains is 8ms, which we empirically discover using a one hop throughput test.

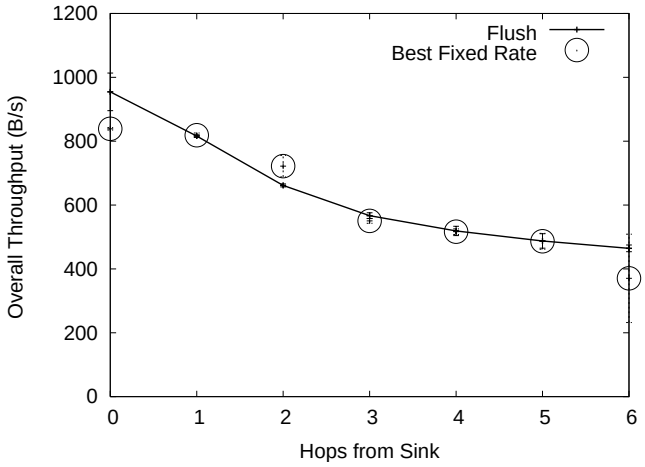
We begin by initiating a multihop transfer from a source node six hops away. After this transfer completes, we perform a different transfer from the 5th hop, and continue this process up to, and including, a transfer in which the source node is only one hop away. The data is also transferred from a basestation. Figure 7 shows the results of these trials. The 0th hop indicates the basestation. Each point in the graph is the average of four runs, with the vertical bars indicating the standard deviation.

Each path length has a fixed sending rate which performs best. When transferring over one hop there is no forward interference, and a node can send packets as fast as the hardware itself can handle. As the path length increases, the rate has to be throttled down, as packets further along in the pipeline interfere with subsequent ones. For example, sending with an interval of 20ms provides the best packet throughput over 3, 4, 5, and 6 hop transfers, as there are too many losses due to queue overflows when sending faster. Slower rates do not cause interference, but also do not achieve the full packet throughput as the network is underutilized.

Figure 8(a) shows the results of the same experiments with Flush. The circles in the figure show the performance of the best fixed rate at the specific path length. Flush performs very close, or better, than this envelope, on a packets/second basis. These results suggest that Flush's rate control algo-



(a) Overall packet throughput



(b) Overall byte throughput

Figure 8. Performance of Flush compared to the best fixed rate at each hop, taken from Figure 7. (a) Flush tracks the best fixed packet rate. (b) Flush's protocol overhead reduces the effective data rate.

rithm is automatically adapting to select the best sustainable sending rate along the path and optimizing for changing link qualities and forward interference.

Figure 8(b) shows the overall throughput on a bytes/second basis. The overall throughput of Flush is sometimes lower than the best fixed rate because we adjust for protocol overhead. In this figure, Flush's rate control header fields account for 3 bytes (δ, f, and D each require 1 byte), leaving only 17 bytes for the payload – a 15% protocol overhead penalty. These figures show that fixing a sending interval may work best for a specific environment, but that no single fixed rate performs well across different path lengths, topologies, and link qualities. We could fine tune the rate for a specific deployment and perhaps get slightly better performance than Flush, but that process is cumbersome, because it requires tuning the rate for every node, and brittle because it does not handle changes in the network topology or variations in link quality gracefully.

At this point it is useful to take a closer look at the gap between the overall throughput achieved by Flush and the hardware nominal capacity. The Chipcon CC2420 radio on the MicaZ mote has a maximum data rate of 250kbps, while the serial interface with the PC runs with a data rate of 115.2kbps. Clearly, if we are to transfer all bytes out of the network through the serial interface, we will be limited by the lower serial rate. To measure what packet sending rates our hardware and OS configuration can achieve, we ran an experiment sending packets to the sink from a node one hop away, varying the packet interval, i.e., the time between the start of consecutive packets. The results showed that with any packet interval lower than 10ms we experienced a high loss rate at the sink. With packets with 29-byte payloads, this interval translates to at most 23,200bps of throughput above the MAC layer. This is lower than the maximum hardware throughput, but it is outside of the scope of this paper to change the underlying MAC and OS to improve this bound.

There are other important overheads above the MAC layer. The routing header in MintRoute is 7-bytes long; for reliable data collection, 2 bytes are further used to include a sequence number. Lastly, Flush adds 3 bytes for rate control information. This reduces the available bandwidth for transfers by 12 bytes, or 41%, for a maximum achievable 1-hop transfer throughput of 13,600bps (1,700B/s). Lastly, Flush has the additional overhead of a topology query, end-to-end retransmissions, and an integrity check. For the 1-hop case, 68.9% of the time is spent in the actual data transfer phase, and the possible overall bandwidth is 9,370bps (1,171 B/s). As will be shown later, Flush exploits most of the capacity that the routing layer can provide. Flush is trying to work on top of a basic CSMA layer, rather than designed from an optimal cross-protocol design outlook. Our evaluation of Flush is conservative, in that we can use a larger packet size to decrease the header overhead, but there is still a large performance gap to the raw radio bandwidth that would require a cross-layer design and integration with the MAC and the packet processing in the OS.

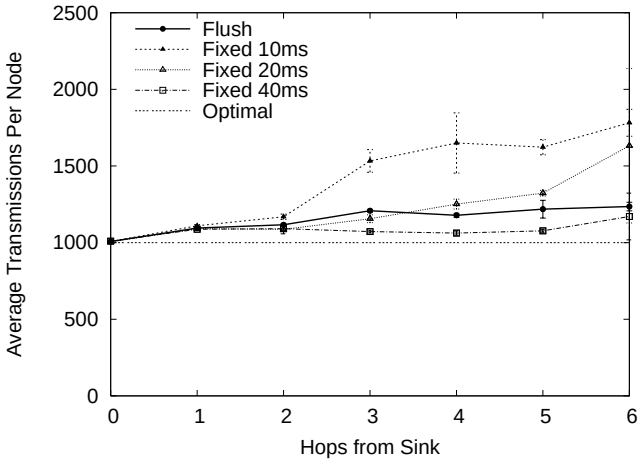


Figure 9. Average number of transmissions per node for sending an object of 1000 packets. The optimal algorithm assumes no retransmissions.

Figure 9 compares the efficiency of the different alternatives from the experiment above. We use the average number of packets sent *per hop* in our transfer of 1000 packets as an indicator for the efficiency at each sending rate. Overall throughput is negatively correlated with the number of messages transmitted, as the transfers with a small fixed interval lose many packets due to queue overflows. As in the previous graphs, Flush performs close to the best fixed rate at each path length. Note that the extra packets transmitted by Flush

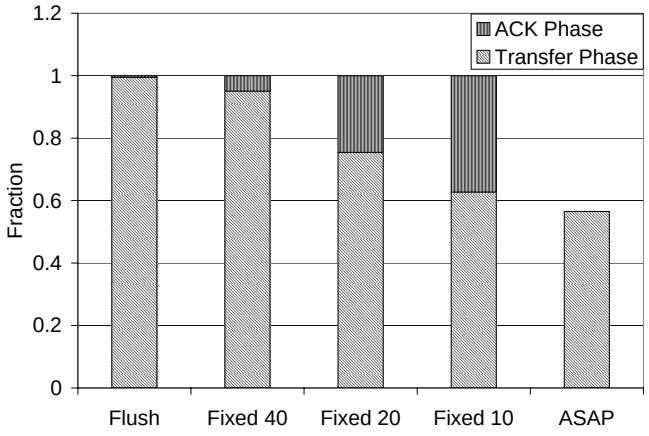


Figure 10. Fraction of data transferred from the 6th hop during the transfer phase and acknowledgment phase. Greedy best-effort routing is *unreliable*, and exhibits a loss rate of 43.5%. A higher than sustainable rate leads to a high loss rate.

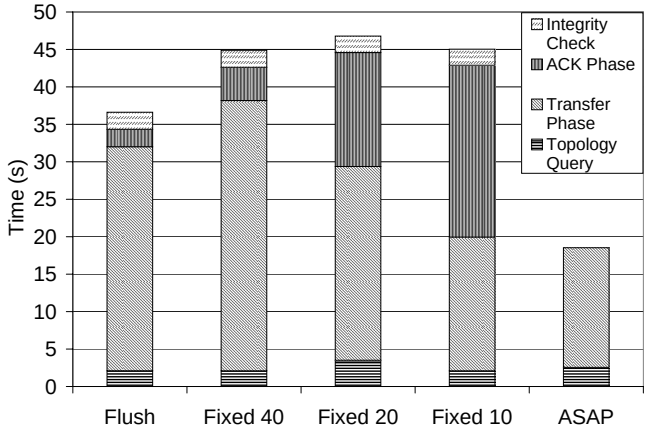


Figure 11. Fraction of time spent in different stages. A retransmission during the acknowledgment phase is expensive, and leads to poor throughput.

and by the "Fixed 40ms" flow are mostly due to link level retransmissions, which depend on the link qualities along the path. Flush and "Fixed 40ms" flow *experienced no losses due to queue overflows*. In contrast, the retransmissions of the "Fixed 10ms" and "Fixed 20ms" curves include both the link level retransmissions and end-to-end retransmissions for packet losses due to queue overflows at intermediate nodes.

4.3 Performance Breakdown

We now examine how Flush compares to the fixed rate algorithms in more detail, examining where time is spent and where packets are lost for each case. To recall, Flush first sends the entire data object once during the *transfer phase*. After this completes, in the *acknowledgment phase*, the receiver identifies missing packets and repeatedly requests retransmissions until all packets are received successfully. Sending a packet during this phase is more expensive than sending a packet during the transfer phase both in terms of the number of packets needed and the total transfer time required. Therefore, an important design goal is to maximize the data received during the transfer phase. The higher the throughput during the transfer phase, the greater the overall throughput becomes.

Figure 10 shows the fraction of packets received during

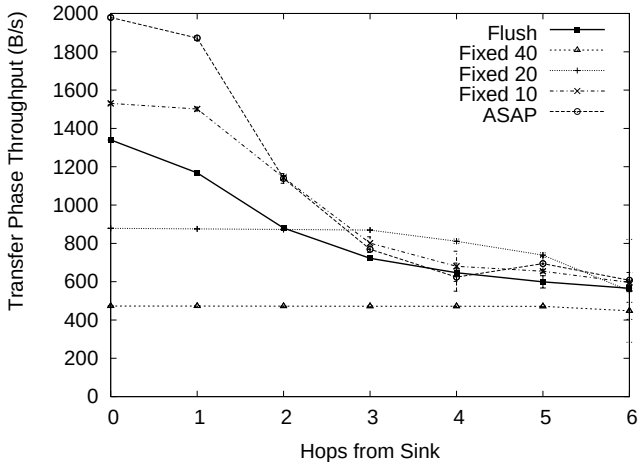


Figure 12. Transfer phase byte throughput. Flush results take into account the extra 3-byte rate control header. This metric does not take loss into account. Flush achieves a good fraction of the throughput of "ASAP", with a 65% lower loss rate.

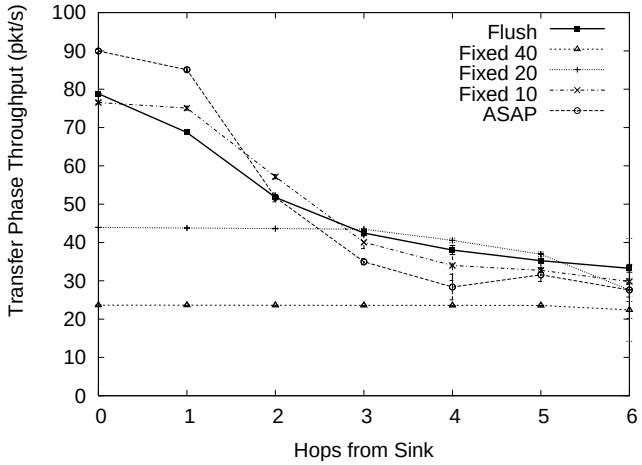


Figure 13. Transfer phase packet throughput. Flush provides comparable throughput with a lower loss rate.

the transfer phase from a node that is 6 hops away from the basestation. The data set is the same one as in Subsection 4.2. Flush collects 99.5% of packets during the transfer phase. In contrast, the "Fixed 10ms" rate flow collects only 62.7% during the transfer phase, due to severe loss from intra-path interference. We also compare with a naïve transfer algorithm that sends a packet as soon as the last packet transmission is done, which we call "ASAP (As Soon As Possible)". ASAP has no acknowledgment phase, and is included just to assess how quickly the underlying routing layer can send packets with no rate control. ASAP can transmit faster than Fixed 10ms in the transfer phase, but exhibits an even higher loss rate of 43.5%.

Figure 11 shows a breakdown of how time is spent during the transfer. The "Topology Query" probes the depth of a target node before the "Transfer Phase". The topology query is needed to tune the RTT and compute a timeout at the receiver. The "Transfer Phase" and "Acknowledgment Phase" were explained above. The "Integrity Check" tests the checksum of the data computed at a target node with the checksum computed at the sink after the "Acknowledgment Phase". As we argued previously, the "Acknowledgment Phase" is ex-

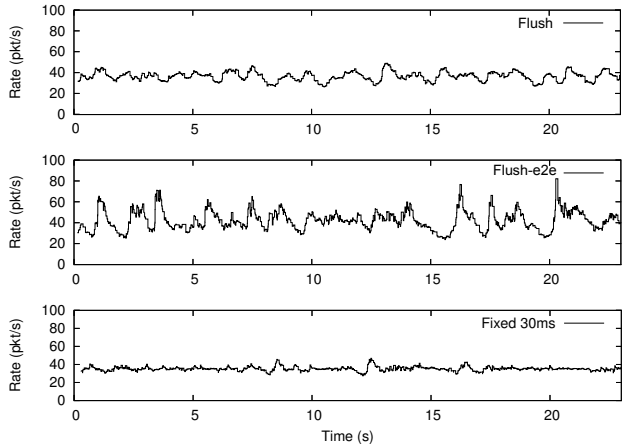


Figure 14. Packet rate over time for a source node 7 hops away from the base station. Packet rate averaged over 16 values, which is the max size of the queue. Flush approximates the best fixed rate with the least variance.

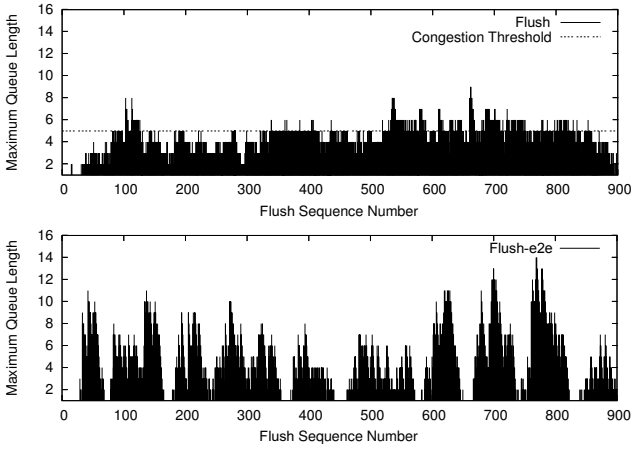


Figure 15. Maximum queue occupancy across all nodes for each packet. Flush exhibits more stable queue occupancies than Flush-e2e.

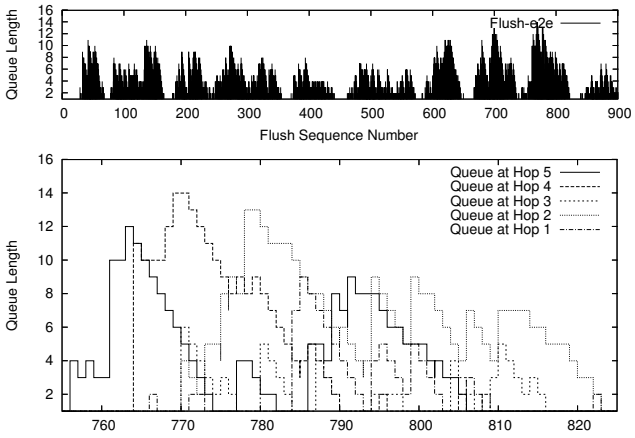


Figure 16. Detailed view of instantaneous queue length for Flush-e2e. Queue fluctuations ripple through nodes along a flow.

pensive. In the case of the Fixed 10ms rate, a modest fraction of packets are transferred during the transfer phase, and more time is spent in the acknowledgment phase. While both Fixed 10ms and ASAP spend less time than Flush int the transfer phase, they also deliver less data than Flush in these phases.

Figures 12 and 13 show the transfer phase byte throughput of the different algorithms. The byte throughput measurements include an adjustment factor for payload size, because of the additional 3-byte rate control header of Flush. When looking at the packet throughput, we see that starting from the 2nd hop, and continuing for all greater hop counts, Flush provides a similar throughput to the best case among fixed rates and the greedy best-effort. For a basestation (0th hop) and the 1st hop, "Fixed 10ms" and "ASAP" provide a higher throughput. However, they suffer higher loss rates and pay a high price during the acknowledgment phase. Overall, Flush provides competitive throughput during the transfer phase.

In summary, Flush achieves comparable transfer phase throughput to the fixed rate algorithms as Figure 13 shows, but with very low loss rates (Figure 10). Flush also spends much less time in the expensive acknowledgment phase as Figure 11 shows. This combination makes Flush's overall transfer time relatively short, and explains Flush's good overall throughput.

4.4 A More Detailed Look

We now take a more detailed look at Flush's operation. In the following two subsections, Flush's rate control header fields account for 6 bytes (δ, f, and D each require 2 bytes). leaving 14 bytes for the payload. At this initial stage, 2 bytes are used just in case where a value exceeds 255. We discovered, however, that the δ, f, and D values never exceeded 255, so these fields were reduced to a single byte each in other bandwidth and scalability experiments. "Flush-e2e" is a variation of Flush which only *limits* the rate at the *source*, even though the intermediary nodes still estimate the delays and propagate them as described in Section 2.3. Using the detailed logs collected for a sample transfer of 900 packets (12600 bytes) over a 7 hop path, we are able to look at the real sending rate at each node, as well as the instantaneous queue length at each node as each packet is transmitted.

Figure 14 shows the sending rate of one node over a particular interval, where the rates are averaged over the last k packets received. We set k to 16, which is the maximum queue length. Other nodes had very similar curves. We compare Flush and Flush-e2e, with the best performing fixed-rate sending interval at this path length, 30ms. Sending at this interval did not congest the network. As expected, under stable network conditions, the fixed-rate algorithm maintains a stable rate. Although Flush and Flush-e2e showed very similar high-level performance in terms of throughput and bandwidth, we see here that the Flush is much more stable, although not to the same extent as the fixed interval transfer.

Another benefit of the in-network rate limiting, as opposed to source-only limiting, can be seen in Figure 15. This plot shows the maximum queue occupancy for all nodes in the path, versus the packet sequence number. Note that we use sequence number here instead of time because two of the nodes were not properly time-synchronized due to errors in the timesync protocol. The results are very similar, though, as the rates do not vary much. The queue length in Flush is always close to 5, which is the congestion threshold we set for increasing the advertised delay (c.f. Section 2.3). Our simple scheme of advertising our delay as doubled when the queue is above the threshold seems to work well in practice. It is actually good to have some packets in the queue, be-

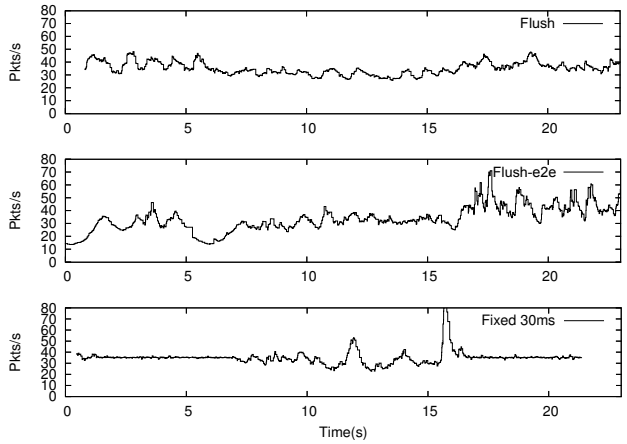


Figure 17. Sending rates at the lossy node for the forced loss experiment. Packets were dropped with 50% probability between 7 and 17 seconds. Both Flush and Flush-e2e adapt while the fixed rate overflows its queue.

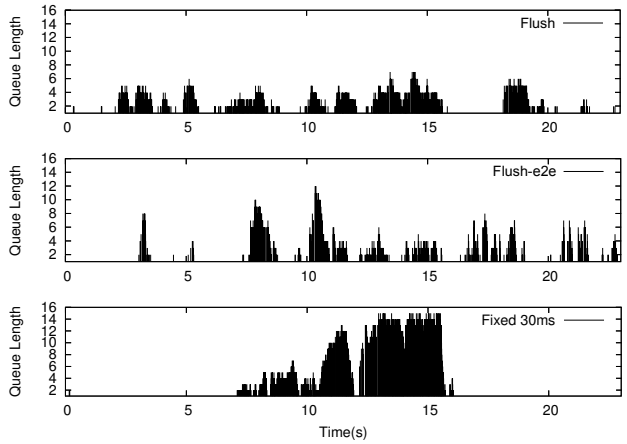


Figure 18. Queue length at the lossy node for the forced loss experiment. Packets were dropped with 50% probability between 7 and 17 seconds. Flush and Flush-e2e adapt while the fixed rate overflows its queue.

cause it allows the node to quickly increase its rate if there is a sudden increase in available bandwidth.

In contrast, Flush-e2e produces highly variable queue lengths. The lack of rate limiting at intermediary nodes induces a cascading effect in queue lengths, as shown in Figure 16. The bottom graph provides a closer look at the queue lengths for 5 out of the 7 nodes in the transfer during a small subset of the entire period. The queue is drained as fast as possible when bandwidth increases, thus increasing the queue length at the next hop. This fast draining of queues also explains the less stable rate shown in Figure 14.

4.5 Adapting to Network Changes

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

Figure 17 provides the instantaneous sending rate over the link with the forced losses for Flush, Flush-e2e, and Fixed

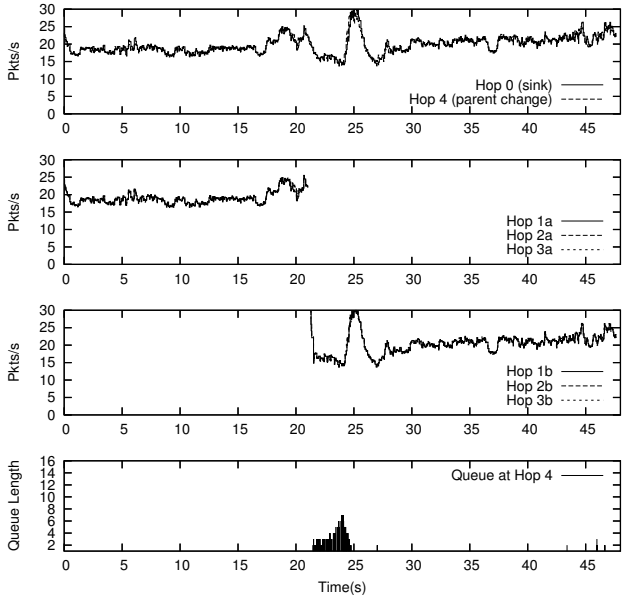


Figure 19. Detailed look at the route change experiment. Node 4's next hop is changed, changing all nodes in the subpath to the root. No packets were lost, and Flush adapted quickly to the change. The only noticeable queue increase was at node 4, shown in the bottom graph. This figure shows Flush adapts when the next hop changes suddenly.

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

Figure 18 compares the queue lengths for the same experiment for all three algorithms, and the reasons for the rate instability become apparent, especially for the fixed rate case. The queue at the lossy node becomes full as its effective rate increases, and is rapidly drained once the link quality is reestablished.

The last experiment looks at the effect of a route change *during a transfer* on the performance of Flush. We started a transfer over a 5 hop path, and approximately 21 seconds into the experiment forced the node 4 hops from the sink to switch its next hop. Consequently, the entire subpath from the node to the sink changed. Note that this scenario does not simulate node failure, but rather a change in the next hop, so packets should not be lost. The high level result is that the change had a negligible effect on performance. Figure 19 presents a detailed look at the rates for all nodes, and the queue length at the node that had its next hop changed. There was no packet loss, and the rate control algorithm was able to quickly reestablish a stable rate. Right after the change there was a small increase in the affected node's queue, but that was rapidly drained once the delays were adjusted.

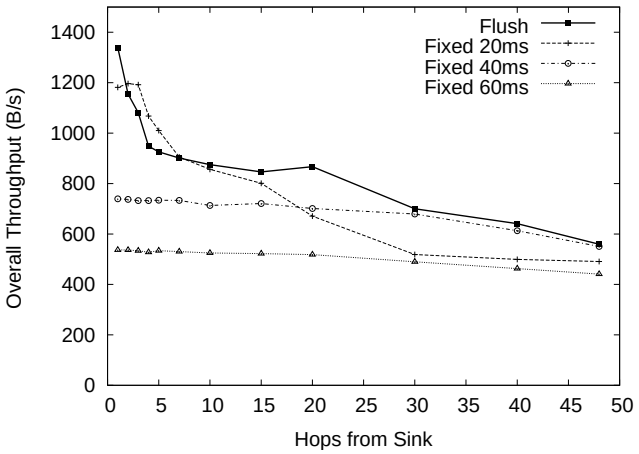


Figure 21. Effective bandwidth from the real-world scalability test where 79 nodes formed 48 hop network. The Flush header is 3 bytes and the Flush payload is 35-bytes (versus a 38 byte payload for the fixed rates). Flush closely tracks or exceeds the best possible fixed rate across at all hop distances that we tested.

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

4.6 Scalability

Finally, to evaluate the scalability of Flush, we deployed an outdoor network consisting of 79 MicaZ nodes in an outdoor setting. These nodes were placed in a line on the ground, with neighboring nodes separated by 3ft. The physical extent of the network spanned 243ft. The radio transmission power was lowered to decrease range, but not so much so that the network would be void of interference. The resulting topology is shown in Figure 20, where the rightmost node is 48 hops from the root, which is the leftmost node.

For the following experiments, we increased the data payload size to 38 bytes (from 20 bytes used previously) for the fixed rate and 35 bytes (from 17 bytes used previously) for Flush. The size of the Flush rate control header was 3 bytes, leaving us with a protocol overhead of about 8%. We transfer a 26,600 byte data object from the node with a depth of 48 (node 79), and then perform similar transfers from nodes at depths 40, 30, 20, 15, 10, 7, 5, 4, 3, 2, and 1. The experiment is repeated for Flush, and fixed rates of 20ms, 40ms, and 60ms. Each experiment is performed twice and the results are averaged. We omit error bars for clarity. Figure 21 shows the results of this experiment. The results indicate that Flush efficiently transfers data over very long networks – 48 hops in this case.

4.7 Memory and Code Footprint

We round out our evaluation of Flush by reviewing its footprint. Flush uses 629 bytes of RAM and 6,058 bytes of code, including the routines used to debug, record performance statistics, and log traces. These constitute 15.4% of RAM and 4.62% of program ROM space on the MicaZ platform. Table 1 shows a detailed breakdown of memory footprint and code size. The Protocol Engine accounts for 301 out of 629 bytes of RAM, or 47.9% of Flush's memory usage. A significant fraction of this memory (180 bytes) is used for message buffers, which are used to hold prefetched data.

Table 1. Memory and code footprint for key Flush components compared with the regular TinyOS distribution of these components (where applicable). Flush increases RAM by 629 bytes and ROM by 6,058 bytes.

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

5 Discussion

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

5.1 Is Multihop Necessary?

We claimed that wireless multihop delivery of bulk data to the network edge is complicated by lossy links, inter-path interference, intra-path interference, and transient rate mismatches. One compelling solution to these challenges might be to simply sidestep them. By judiciously placing high-power radios within a low-power sensornet, a network administrator might be able to reduce it to a single-hop problem, remove the complexities that multihop introduces, and allow simple, robust solutions.

Unfortunately, it is not always possible to convert a dense deployment of low-power, short-range nodes into an equivalent network of high-power, long-range nodes. Some applications are deployed in challenging radio environments that do not provide a clear line-of-sight over multiple hops. For example, signals do not propagate well in steel-framed buildings, steel truss bridges, or dense foliage [14]. When physically large networks are deployed in such environments, multihop delivery is often the enabler, so eliminating it eliminates the application as well.

5.2 Interactions with Routing

We freeze the MintRoute collection tree immediately prior to a Flush transfer and then let the tree thaw after the transfer. This freeze-thaw cycle prevents collisions between routing beacons and Flush traffic. MintRoute [32] generates periodic routing beacons but these beacons use a separate, unregulated data path to the radio. With no rate control at the link layer, the beacons are transmitted at inopportune times, collide with Flush traffic, and are lost. Since MintRoute depends on these beacons for route updates, and it evicts stale routes aggressively, Flush freezes the MintRoute state during a transfer to avoid stale route evictions.

Our freeze-thaw approach sidesteps the issue and works well in practice. Over small time scales on the order of a Flush session, routing paths are generally stable, even if instantaneous link qualities vary somewhat. Our results show that Flush can easily adapt to these changes. In Figure 19, we also showed that Flush adapts robustly to a sudden change in

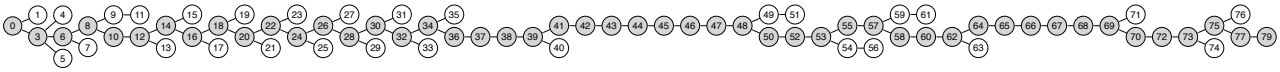


Figure 20. The network used for the scalability experiment. Of the 79 total nodes, the 48 nodes shown in gray were on the test path. This test is a demonstration that Flush works over a long path, and Flush is *not* limited to a linear topology, as shown in previous tests.

the next hop. If the underlying routing protocol can find an alternate route, then Flush will adapt to it. But if the physical topology changes, and routing cannot adapt, then new routes will need to be rebuilt and the session needs to be restarted.

It may seem that forcing all traffic to pass through a single (rate-limited) queue would address the issue, but it does not. Nodes located on the flow path *would* be able to fairly queue routing beacons and Flush traffic. However, nodes located off the flow path, but within interference range of the flow, would not be able to contend for bandwidth and successfully transmit beacons. Hence, if the physical topology changes along the flow path *during* a transfer, the nodes along the path may not be able to find an alternate route since beacons from these alternate routes may have been lost. A solution may be an interference-aware fair MAC. Many pieces are already in place [12, 6, 22] but the complete solution would require rate-controlling all protocols at the MAC layer across all nodes within interference range of the path.

6 Related Work

Our work is heavily influenced by earlier work in congestion mitigation, congestion control, and reliable transfer in wired, wireless, and sensor networks. Architecturally, our work was influenced by Mishra's hop-by-hop rate control [18], which established analytically that a hop-by-hop scheme reacts faster to changes in the traffic intensity and thus, utilizes resources at the bottleneck better and loses fewer packets than an end-to-end scheme. Kung et al's work on credit-based flow control for ATM networks [4] also influenced our work. Their approach of using flow-controlled virtual circuits (FCVC) with guaranteed per-hop buffer space is similar to our design. We adapted ideas of in-network processing in high-speed wired networks to wireless networks in which transmissions interfere due to the nature of the broadcast medium.

A complicating factor that distinguishes wired and wireless communication is that the radio interference range often exceeds transmission range. Therefore, a major element of our work is adapting the ideas of these earlier papers to an environment that has forward interferers but supports broadcast snooping. Li et al. [16] studied the theoretical capacity of a chain of nodes limited by interference using 802.11, which is related to our work in finding a capacity and rate. Indeed, our results generally appear to agree with Li's models but our work also demonstrates how real-world factors can cause significant variance from ideal performance models.

ATP [27] and W-TCP [24], two wireless transport protocols that use rate-based transmission, have also influenced our work. In ATP, each node in a path keeps track of its local delay and inserts this value into the data packet. Intermediate nodes inspect the delay information embedded in the packet, and compare it with its own delay, and then insert the larger of the two. This way, the receiver learns the largest delay experienced by a node on the path. The receiver reports this delay in each epoch, and the sender uses this delay to set its sending rate. W-TCP uses purely end-to-end mechanisms. In particular, it uses the ratio of the inter-packet separation at the receiver and the inter-packet separation at the sender as the primary metric for rate control. As a result, ATP and W-TCP reduce the effect of non-congestion related packet losses on the computation of transmission rate. Flush, in contrast, computes fine-grained estimates of the bottleneck capacity in real-time and communicates this during a transfer, allowing our approach to react as congestion occurs. Flush also applies rate control to the problem of optimizing pipelining and interference, neither of which is addressed by ATP or W-TCP.

A number of protocols have been proposed in the sensor network space which investigate aspects of this problem. RMST [26] outlines many of the theoretical and design considerations that influenced our thinking including architectural choices, link layer retransmission policies, end-to-end vs hop-by-hop semantics, and choice of selective/cumulative or positive/negative acknowledgments, even though their work was focused on analytical results and ns-2 simulations of 802.11 traffic.

Fusion [12], IFRC [22], and the work in [6] address the problems of rate and congestion control for collection, but are focused on a fair allocation of bandwidth among several competing senders, rather than efficient and reliable end-to-end delivery. Fusion [12] uses only buffer occupancy to measure congestion and does not try to directly estimate forward path interference. IFRC estimates the set of interferers on a collection tree with multiple senders, and searches for a fair rate among these with an AIMD scheme. It does not focus on reliability, and we conjecture that the sawtooth pattern of rate fluctuations makes for less overall efficiency than Flush's more stable rate estimates.

Fetch [30] is a reliable bulk-transfer protocol used to collect volcanic activity monitoring data. Fetch's unit of transfer is a 256-byte block, which fits in 8 packets. Similar to Flush, Fetch requests a block, and then issues a repair request for any missing packets in the block. Fetch was used to collect data from a six hop network in an extremely hazardous environment. In collecting 52,736 bytes of data, the median bandwidth for 1st hop was 561 bytes per second, and median bandwidth for 6th hop was 129 bytes per second, a fraction of the bandwidth that Flush can achieve over the same path length.

Wisden [20], like Flush, is a reliable data collection protocol. Nodes send data concurrently at a static rate over a collection tree and use local repair and end-to-end negative acknowledgments. The paper reports on data collected from 14 nodes in a tree with a maximum depth of 4 hops. Of the entire dataset, 41.3% was transferred over a single hop, and it took over 700 seconds to collect 39,096 bytes from each node. To compare with Flush, we assume the same distribution of path lengths. Based on the data from our experiments, it would take Flush 465 seconds for an equivalent transfer. We ran a microbenchmark in which we collected 51,680 bytes us-

ing a packet size of 80 bytes (the same as Wisden) and 68 byte payload. This experiment, repeated four times, shows that Flush achieved 2,226 bytes per second from a single hop compared with Wisden's 782 bytes per second. This difference can be explained by the static rate at every node in Wisden. Incorrectly tuned rates or network dynamics can cause buffer overflows and congestion collapse at one extreme and poor utilization at the other extreme. Since in Wisden, nodes are sending without avoidance and adjustment to interference, cascading losses can occur, leading to inefficiency.

We also wanted to compare Flush with Tenet's reliable stream transport service with end-to-end retransmission [9]. However, since comparable performance numbers for Tenet is not available, we attempted to use and characterize Tenet's performance on the Mirage testbed. Unfortunately, despite the assistance of Tenet's authors, we were unable to instrument and evaluate Tenet's performance on the Mirage testbed we used for the majority of our experiments.

Event-to-Sink Reliable Transport (ESRT) [23] defines reliability as "the number of data packets required for reliable event detection" collectively received from all nodes experiencing an event and without identifying individual nodes. This does not satisfy our more stringent definition of reliability. PSFQ [29] is a transport protocol for sensor networks aimed at node reprogramming. PSFQ addresses a dissemination problem that is distinct from our collection problem, since the data move from the basestation to a large number of nodes. ExOR [3] is a data transport protocol that takes advantages of the broadcast medium. However, it requires that each packet header contain considerable protocol data (e.g. forward list, batch map). This protocol data would introduce an additional overhead on our already quite-limited payload. Recent attempts to apply network utility maximization techniques to wireless networks [17] may be promising to understand Flush's performance from a more theoretical perspective.

7 Conclusion

Rate-based flow control algorithms are known to work better than window-based ones for multihop wireless flows over unscheduled links. The challenge lies in deciding the rate: too high a rate will cause self-interference while too low a rate will cause poor capacity utilization. In contrast with earlier work, this paper advocates directly measuring intra-path interference at each hop and using this information to compute an upper bound on the rate (the actual rate could be lower due to lossy links). This paper demonstrates that such measurements are feasible and practical, and that they can form the basis of transport protocol that can scale to dozens of hops. Indeed, we used Flush to collect the performance data presented in this paper.

8 Acknowledgments

This material is based upon work supported by the National Science Foundation under grants #0435454 ("NeTS-NR"), #0454432 ("CNS-CRI"), and #0615308 ("CSR-EHS"). A National Science Foundation Graduate Research Fellowship and a Stanford Terman Fellowship, as well as generous gifts from Intel Research, DoCoMo Capital, Foundation Capital, Crossbow Technology, Microsoft Corporation, and Sharp Electronics, also supported this work.

9 References

[1] `http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/hsn/README_TinyAODV`.

[2] `http://mirage.berkeley.intel-research.net/`.

[3] S. Biswas and R. Morris. ExOR: opportunistic multi-hop routing for wireless networks. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 133–144, New York, NY, USA, 2005. ACM Press.

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

[5] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, May 2005.

[6] C. T. Ee and R. Bajcsy. Congestion control and fairness for many-to-one routing in sensor networks. In *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 148–161. ACM Press, 2004.

[7] R. Fonseca, S. Ratnasamy, J. Zhao, C.-T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon-vector routing: Scalable point-to-point routing in wireless sensor networks. In *Proceedings of the 2nd USENIX Symposium on Networked System Design and Implementation (NSDI'05)*, May 2005.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[9] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM Press.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[11] J. Hill, R. Szewczyk, A. Woo, P. Levis, K. Whitehouse, J. Polastre, D. Gay, S. Madden, M. Welsh, D. Culler, and E. Brewer. Tinyos: An operating system for sensor networks, 2003.

[12] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, Nov. 2004.

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

[14] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 254–263, New York, NY, USA, 2007. ACM Press.

[15] Y. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In *Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2005)*, Boston, MA, May 2005.

[16] J. Li, C. Blake, D. S. D. Couto, H. I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 61–69, New York, NY, USA, 2001. ACM Press.

[17] X. Lin and N. Shroff. Utility maximization for communication networks with multi-path routing. 51(5):766–781, May 2006.

[18] P. P. Mishra, H. Kanakia, and S. K. Tripathi. On hop-by-hop rate-based congestion control. *IEEE/ACM Trans. Netw.*, 4(2):224–239, 1996.

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

[20] J. Paek, K. Chintalapudi, J. Cafferey, R. Govindan, and S. Masri. A wireless sensor network for structural health monitoring: Performance and experience. In *Proceedings of the Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, 2005.

[21] M. Petrova, J. Riihijarvi, P. Mahonen, and S. Labella. Performance study of ieee 802.15.4 using measurements and simulations. In *Wireless Communications and Networking Conference 2006 (WCNC 2006)*, volume 1, pages 487–492, April 2006.

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

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

[24] P. Sinha, T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. WTCP: A reliable transport protocol for wireless wide-area networks. *Wireless Networks*, 8(2-3):301–316, 2002.

[25] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis. Some implications of low-power wireless to ip routing. In *Proceedings of the Fifth Workshop on Hot Topics in Networks (HotNets V)*, November 2006.

[26] F. Stann and J. Heidemann. RMST: Reliable data transport in sensor networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, pages 102–112. IEEE, Apr. 2003.

[27] K. Sundaresan, V. Anantharaman, H.-Y. Hsieh, and R. Sivakumar. ATP: a reliable transport protocol for ad-hoc networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc '03)*, pages 64–75, 2003.

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

[29] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: a reliable transport protocol for wireless sensor networks. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, 2002.

[30] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, pages 381–396, 2006.

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

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