

Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Network

*Sukun Kim
Rodrigo Fonseca
Prabal Kumar Dutta
Arsalan Tavakoli
David E. Culler
Philip Levis
Scott Shenker
Ion Stoica*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-169

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-169.html>

December 12, 2006



Copyright © 2006, by the author(s).
All rights reserved.

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.

Acknowledgement

This work was supported by NSF grant #0435454 (NeTS-NOSS), NSF GRFP, Crossbow Technology, Microsoft Corporation, and Sharp Electronics.

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 Science Department*
Stanford University
Stanford, CA 94305

Abstract

We present Flush, a reliable, single-flow transport protocol for sensor networks, and show that it can efficiently transfer bulk data across a 48-hop wireless network. Flush provides end-to-end reliability, minimizes transfer time, is energy- and memory-efficient, and adapts robustly to changing network conditions. The protocol requires no special control packets to adjust its rate. Flush nodes propagate rate information against the direction of data flow by snooping on next hop traffic. We show that Flush closely tracks or exceeds the maximum achievable fixed rate over a wide range of path lengths. Flush is useful for many sensor network applications whose main requirement is to transmit all collected data to the edge, which include environmental monitoring, structural health monitoring, and protocol testing. Indeed, we collected the Flush performance data using Flush itself.

1 Introduction

Some sensor network applications, like data aggregation [11], synopsis diffusion [15], and target tracking [19], require in-network data processing. Another class of applications have much simpler requirements: they sample sensors at some low frequency and deliver these readings with best effort to the edge of the network without delay [12, 23]. For these applications, collection protocols like MintRoute [27] provide best effort service. A third class of applications sample sensors at high frequency and must *reliably* deliver these samples to the edge of the network for post processing with relatively loose latency requirements (e.g., minutes or hours) [16, 9, 26].

While delivery of bulk data to the network edge sounds simple, the vagaries of multihop wireless transmission make *efficient* and *reliable* delivery a formidable challenge [24]; intra-path interference is hard to avoid, inter-path interference is hard to cope with, and efficient end-to-end signaling along the reverse path is hard to

achieve. Despite the considerable progress made in addressing these issues in sensor networks [8, 3, 17], some have proposed largely sidestepping the problems of multihop by positioning relatively powerful nodes, called *microservers*, within one or two hops of less powerful, battery-operated nodes called *motes*; this approach works wonderfully well where feasible, but there are cases where microservers cannot be used.

The Golden Gate Bridge structural health monitoring project [9] provides one such example: 51 nodes span 4,200 feet and must operate continuously for weeks. The bridge's safety requirements, limited power availability, and linear topology prohibit the dense deployment of microservers. Since microservers are power-hungry devices, they would also need large batteries or solar cells, but mounting such bulky equipment on a public facility would pose a safety risk to pedestrians and vehicles. Because sink nodes can be placed at only the ends of the bridge, we need a transport protocol that can work well over a potentially large number of hops.

The Golden Gate Bridge application is representative of the class of problem we consider here. The task is to reliably deliver all data to the edge, without much concern for the latency between collection and delivery, at least not on small time scales. One way to accomplish this task is by retrieving the data from each relevant node sequentially, so there is never more than one active flow in the network, but what is the benefit in doing so?

Designing congestion control mechanisms for multihop wireless transport is difficult in the general case, but the problem can be greatly simplified if there are no interfering flows. In general, congestion control must address two issues: sharing the bandwidth between flows in some reasonably equitable manner, and adjusting the sending rate to match the available bandwidth. When only one flow is active at a time, the first issue disappears completely. This paper addresses the second issue. Ignoring inter-path interference allows us to focus on maximizing bandwidth through optimal use of pipelining.

While latency is not a pressing concern in this class of application, energy-efficiency is. Thus, the transport protocol should be designed to minimize *transfer time*. But, there are other reasons, beyond energy-efficiency, to minimize transfer time. For instance, some applications must alternate sensing and communications, either because the sensing rate exceeds the radio bandwidth or because radio operation adversely affects sensing due to electrical interference. In these cases, reducing transfer time increases sensing uptime.

Finally, to be viable in the sensor network regime, a protocol must have a small memory and code footprint. As of late 2006, typical motes have 4KB to 10KB of RAM, 48KB to 128KB of program memory, and 512KB to 1MB of non-volatile memory. This memory must be shared between the application and system software, limiting the amount available for message buffers and network protocol state.

Thus, our goal is to develop a protocol that delivers data reliably to the edge with minimal transfer time and has a small memory and code footprint. Our solution, called Flush, considers a single flow at a time, and uses rate-based hop-by-hop flow control to match the available bandwidth. Flush was implemented in TinyOS, the *de facto* operating system for sensor networks [6] and evaluated using a 48-node subset of the Mirage testbed [1] 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 a fixed rate, *even over a 48-hop wireless network*. Our implementation of Flush requires just 629 bytes of RAM and 6,058 bytes of ROM.

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. As we discussed in Section 1, this is a reasonable assumption for a large class of data collection applications, and makes the algorithm much simpler. The sink sends requests for a large data object, which is divided into packets and sent in a pipelined fashion in its entirety. Reliability is guaranteed by an end-to-end negative acknowledgment scheme: the sink successively requests missing packets from the source, until all packets have been successfully received. Within a transfer, Flush continually tries to estimate the bottleneck bandwidth by means of a simple and effective dynamic rate control algorithm. This algorithm requires no extra control packets: it obtains the necessary information by snooping.

In this section we provide an overview of Flush, and discuss its two main components in detail: the end-to-end reliability protocol, and the dynamic rate estimation.

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 reasonably reliable *delivery* mechanism that can forward packets from the data sink to the data source.

The reverse delivery service need not route the packets; a simple flood or a data-driven virtual circuit is sufficient. The distinction between forward routing and reverse delivery exists because arbitrary, point-to-point routing in sensor networks is uncommon so we do not wish to depend on it.

2.1 Flush 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 transfers move through three phases: data transfer, acknowledgment, and integrity check.

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 stage completes, the acknowledgment phase begins. The sink sends the sequence numbers of packets it did not receive back to the data source. Flush uses negative rather than positive acknowledgments because it assumes the end-to-end reception rate exceeds

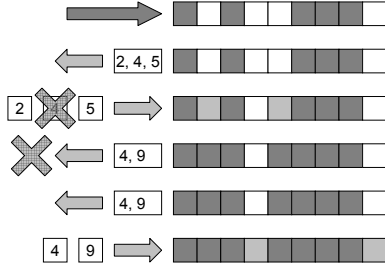


Figure 1: NACK transmission example. Flush has at most one NACK packet in flight at once.

50%. When it receives a NACK packet, the source re-enters the data transfer stage.

This process repeats until the sink has received the requested data *in toto*. 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 for the data. If the check succeeds, the sink can request the next data object, perhaps from another node. Integrity is checked at the level of both packets and data items.

To minimize 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. Flush is miserly with packet headers as well: three 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, minimizing transfer time. Section 3 describes a concrete implementation of the Flush protocol.

2.2 Reliability Protocol

Flush uses an end-to-end reliability protocol in order to be robust to node failures. Figure 1 shows an example session of the protocol, where the data size is 9 packets, and a NACK packet can accommodate at most 3 sequence numbers. In the data transfer stage, the source sends all of the data packets, some of which are lost (2, 4, 5, and 9 in the example), either due to retransmission failures or queue overflows. The sink keeps a bitmask of all received packets. When it believes that the source has finished sending data, the sink sends a single NACK packet, which can hold up to N sequence numbers, back to the source. This NACK contains the first N (where $N = 3$ in this case) sequence numbers of lost packets, 2, 4, and 5. The source retransmits the requested packets. This process continues until the sink has received every packet. The sink uses an estimate of the round-trip time (RTT) to decide when to send NACK packets in the event that all of the retransmissions are lost. A portion of NACK protocol code is borrowed from [9].

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.

In one experiment along a 48-hop path, deployed at the Richmond Field Station (RFS), Flush had an end-to-end loss rate of 3.9%. For a 760 packet data item 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 serious 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* [24]. One-hop medium access control algorithms will not solve the problem, due to hidden-terminal problems. Lastly, rate mismatches may cause queues further along the path to overflow, leading to packet loss, wasted energy, and more 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, maximizing spatial reuse. 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.

2.3.1 A very simple model

In this simplified model there are N nodes arranged linearly plus a basestation B . Node N sends packets to the basestation through $N - 1, \dots, 1$. 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 1 packet per slot, and cannot both send and receive in the same slot. Nodes can only send and hear packets from neighbors 1 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.

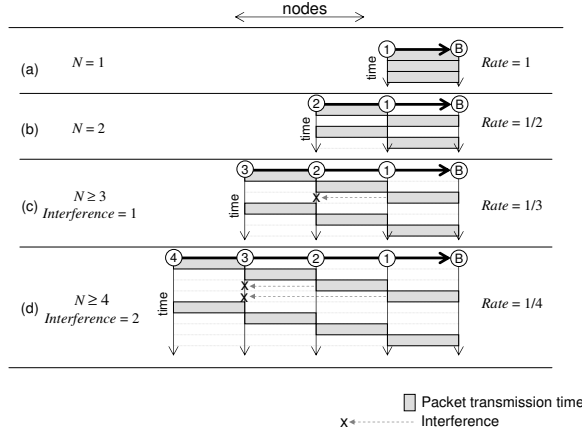


Figure 2: Maximum sending rate without collision in the simplified pipelining model, for different number of nodes (N) and interference ranges (I).

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

Figure 2 shows the maximum rate we can have 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 than the maximum possible.

2.3.2 Dynamic Rate Control

We now describe how Flush dynamically estimates the maximum sending rate that maximizes the pipeline utilization. The algorithm is agile in reacting to increases and decreases in per-hop throughput and interference

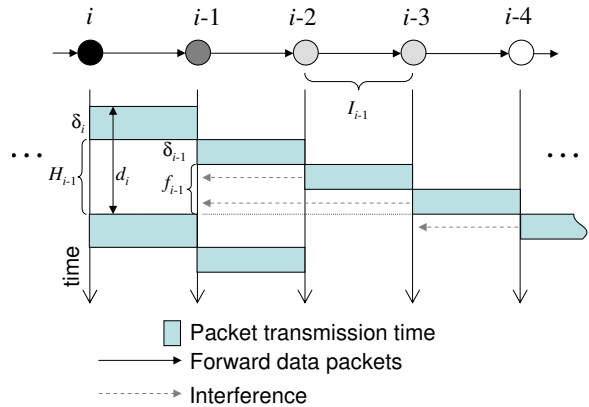


Figure 3: A detailed look at pipelining from the perspective of node i , with the quantities relevant to the algorithm shown.

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. It 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 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 δ s of all successors that node i can hear. As the values δ_{i-1} and f_{i-1} of its successor may change over time and space due to environmental effects such as path and noise, Flush continually estimates and updates δ_i and f_i .

Let us look at an example. In Figure 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.

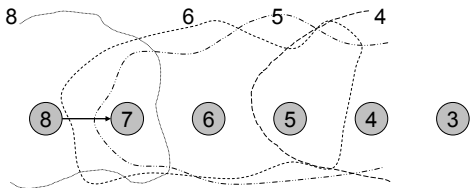


Figure 4: Packet transfer from node 8 to node 7 interferes with transfer from node 5 to node 4. However it does not interfere with transfer from node 4 to node 3

As described above, each node can determine its own fastest sending rate. This is not enough on its own to ensure the optimal path 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 rate 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 may be beneficial to impose a rate limit of D_i for each node i in the path, and not only for the source. 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 sim-

The Flush rate control algorithm	
(1)	δ_i : 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 .

ple 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 slow down. We use a simple mechanism to do this, which has proved efficient. While a node's queue size is past a specified threshold, it temporarily increases the delay it advertises by doubling δ_i .

3 Implementation

To empirically evaluate the Flush algorithms, we implemented them in the nesC programming language [5] and the TinyOS [7] 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 materializing the requested data. After the data has been materialized, the protocol engine writes the data and identifying sequence numbers into a packet and then submits the packet to the routing layer at the rate specified by the packet delay estimator, which is discussed in the next section.

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.

3.2 Routing Layer

MintRoute [27] 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 [10], TinyAODV, or BVR [4]. 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 [14]. It is neither obvious that our interference estimation algorithm would work with such protocols nor clear that a high rate could be achieved or sustained because Flush would be unable to coordinate the transmissions of the distinct next hops.

Flush uses the TinyOS flooding protocol, `Bcast`, to send packets from the receiver to the source for both initiating a transfer and sending end-to-end selective NACKs. `Bcast` implements a simple flood: each node rebroadcasts each unique packet exactly once, assuming there is room in the queue to do so. A packet is rebroadcast with a small, random delay. We chose a flood rather than an epidemic protocol because flooding is fast and simple. We did not use a point-to-point routable protocol because the protocol overhead from headers would decrease Flush’s efficiency.

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 next hops and previous hop along the path of a flow, for predicting the set of interferers. The δ , f , and D fields, used by the estimator, are extracted from the next hop’s intercepted transmissions.

The estimator also extracts the received signal strength indicator (RSSI) of packets received from the previous hop and snooped from all successor hops along the routing path. These RSSI values are smoothed using an exponentially-weighted moving average to smooth out transients on single-packet timescales. History is weighted more heavily because RSSI is typically quite stable and outliers are rare, so a single outlier should have little influence on the RSSI estimate.

A node i considers a successor node $(i - j)$ an interferer of node $i + 1$ at time t if for any $j > 1$ if

$$rssi_{i+1}(t) - rssi_{i-j}(t) < 10 \text{ dBm} \quad (1)$$

This inequality checks if an adequate signal-to-interference-plus-noise ratio (SINR) exists to successfully receive packets. The SINR threshold of 10 dBm was chosen after 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 have found that this interference estimator works reasonably well in practice and we further note that any better estimator would not change the algorithm and would only improve performance. We defer to Section 5 a deeper discussion of the challenges in estimating interference.

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 two to three 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. 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 of 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. We chose a queue depth of 5, about one-third of the queue size, as our congestion threshold.

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 next hop’s delay information and the previous hop and successor hops’ RSSI values. Unfortunately, these two requirements are at odds with each other for the CC2420 radio used in the MicaZ mote, as the radio does not simultaneously support both acknowledgments and snooping in hardware, 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 [17].

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 follows: MintRoute (7 bytes), sequence numbers (2 bytes), Flush rate control fields (6 bytes), and application payload (14 bytes). Since in the default implementation, only 14 bytes are available for the application payload, Flush’s effective data throughput suffers. The initial implementation of Flush used two bytes for each of the δ , f , D fields, with the least significant bit representing 1 ms. Through experimentation, we discovered that the distribution of delay values take on a narrow range that very rarely exceeds 255 ms, so we changed these three fields to 1 byte each, and the application payload to 35-bytes, during subsequent experiments. 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. First, we establish a baseline using fixed rates against which we compare Flush’s performance.

This baseline also allows us to factor out overhead common to all protocols and avoid concerning ourselves with questions like, “why is there a large disparity between the raw radio rate of 250 kbps and Flush?” Next, we compare two variants of Flush against the baseline. One variant, *Flush Source*, adjusts the sending rate only at the source while the other variant, *Flush Hop-by-Hop*, adjusts the sending rate all along the path. Then, we consider the effects of abrupt link quality changes on Flush’s performance and analyze Flush’s response to a parent change in the middle of a transfer. The preceding experiments are carried out on the Mirage testbed [1]. Next, 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.

4.1 Testbed Methodology

We evaluate the effectiveness of Flush through a series of experiments on the Intel Research Berkeley sensor-net testbed, Mirage, as well as a 79-node, ad hoc, outdoor testbed. The Mirage testbed consists of 100 MicaZ nodes, although we use the 48 node subset depicted in Figure 6. This subset was sufficient to construct the set of longest possible paths in the testbed (using the underlying tree-building protocol). We used node 0, in the bottom left corner, as the sink, or basestation. Setting the MicaZ node’s CC2420 radio power level to -15 dBm, the diameter of the resulting network varied between 6 and 8 hops in our experiments, with a typical path shown in Figure 6. The end-to-end quality of the paths was generally good, but in Section 4.4 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.

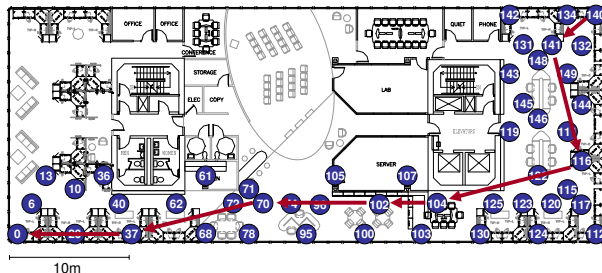


Figure 6: The Mirage indoor testbed used for all experiments except scalability. A path we used in one of the experiments is highlighted. A subset of nodes are selected such that a longer path can be obtained.

We use the MintRoute [27] protocol to form a collec-

tion 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.

In our experiments the basestation issues a request for data from a specific node. All the nodes are time synchronized prior to each trial, and they log to flash memory the following information for each packet sent: the Flush sequence number, timestamp, the values of δ , f , and D , and the instantaneous queue length. After each run we collect the data from all of the nodes *using Flush itself*. We compare two variants of Flush with a static algorithm that fixes the sending rate:

- **Flush Source:** Uses the full rate estimation algorithm, but only *limits* the rate at the *source*. The intermediary nodes still estimate the delays and propagate them as described in Section 2.3.
- **Flush Hop-by-Hop:** Adds rate limiting at each node.

To better appreciate the choice of evaluation metrics presented in this section, we revisit Flush’s design goals. First, Flush requires complete reliability, which the protocol design itself provides (and our experiments validate, across all trials, networks, and data sizes). The remaining goals are to maximize throughput, minimize energy consumption, and adapt gracefully to dynamic changes in the network.

4.2 High Level Performance

We examine the effective packet throughput and effective bandwidth by comparing the two variants of Flush with various values of the static fixed-rate algorithm.

To establish a baseline for comparison, we first look at the packet throughput achieved by the fixed rate algorithm. For each sending interval of 10, 15, 20, 30, and 40ms, we transferred 715 packets along a fixed 6-hop path. The smallest inter-packet interval our hardware platform could physically sustain was 8ms, which we empirically measured over one hop.

We began by initiating a multihop transfer from a source node six hops away. Subsequently, after this transfer completed, we performed a different transfer from the 5th hop, and continued this process up to, and including, a transfer in which the source node was only one hop away. Figure 7 shows the results of these trials. 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

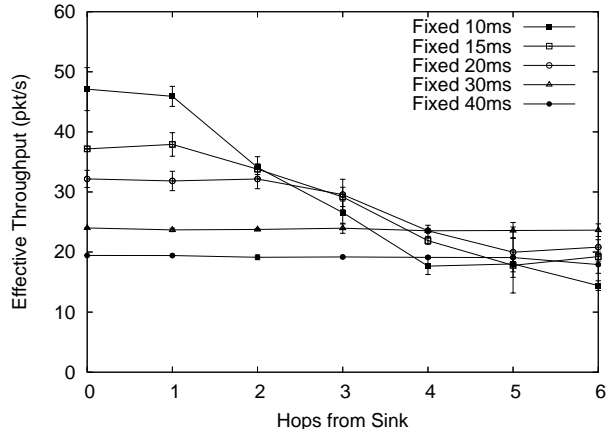


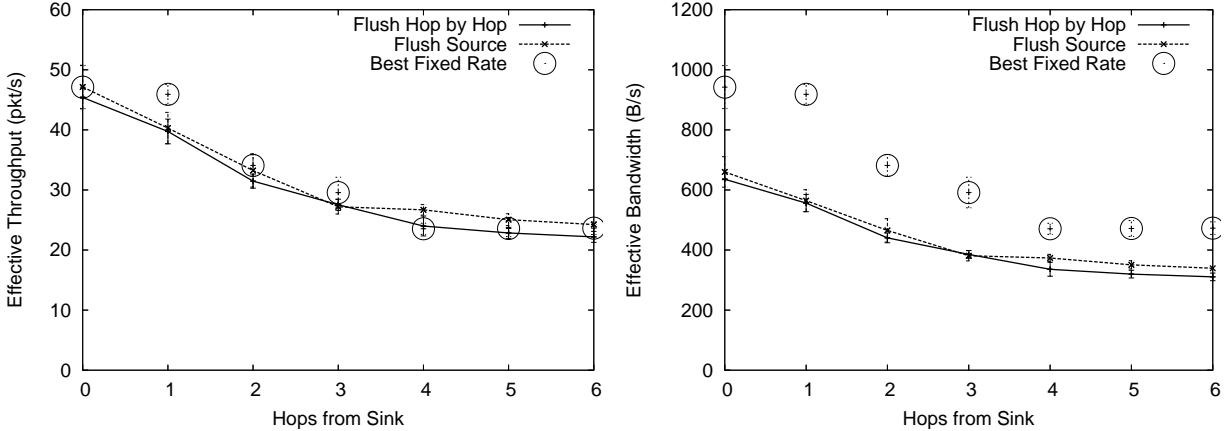
Figure 7: Packet goodput of fixed rate streams over different hop counts. The optimal fixed rate depends on the distance from the sink.

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 30ms provides the best packet throughput over 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 the two variations of Flush. The circles in the figure show the performance of the best fixed rate at the specific path length. Flush performs very close, or better, than this envelope, on a packets/second basis. These results suggest that Flush’s rate control algorithm is automatically adapting to select the best possible sending rate along the path, optimizing for changing link qualities and forward interference.

Figure 8(b) shows the effective data bandwidth on a bytes/second basis. The effective bandwidth of Flush is lower than the best fixed rate because we adjust for protocol overhead. In this figure, Flush’s rate control header fields account for 6 bytes (δ , f , and D each require 2 bytes), leaving only 14 bytes for the payload. We discovered, however, that the δ , f , and D values never exceeded 255, so these fields were reduced to a single byte each in the later scalability experiments.

These figures show that fixing a sending interval may work best for a specific environment, but 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



(a) Effective packet throughput of the two variants of Flush rate control compared to the best fixed rate at each hop, taken from Figure 7 (b) Effective bandwidth of the two variants of Flush rate control compared to the best fixed rate at each hop, taken from Figure 7.

Figure 8: (a) Flush tracks the best fixed packet rate. (b) Flush’s protocol overhead reduces the effective data rate.

brittle because it does not handle changes in the network topology or variations in link quality gracefully.

Figure 9 compares the energy efficiency of the different alternatives from the experiment above. We use the average number of packets sent *per hop* in our transfer of 715 packets as an indicator for how much energy each node consumed, as radio communication is the most power-intensive operation of a sensor node. Effective bandwidth is negatively correlated with the number of messages transmitted, as the transfers with a small fixed interval lose many packets due to queue overflows. As in the previous graphs, Flush performs close to the best fixed rate at each path length. Note that the extra packets transmitted by Flush and by the “Fixed 40ms” flow are mostly due to link level retransmissions, which depend on the link qualities along the path. The Flush Source, Flush Hop-by-Hop, 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 packets losses due to queue overflows at intermediate nodes.

4.3 A More Detailed Look

A more detailed analysis provides additional insight into why Flush performs well, and how the two variants behave differently. 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 10 shows the sending rate of one node over a particular interval, where the rates are averaged over the

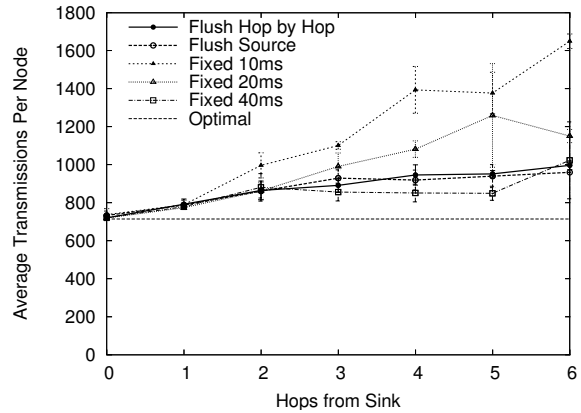


Figure 9: Average number of transmissions per node for sending a page of 715 packets. The optimal algorithm assumes no retransmissions.

last k packets received. We set k to 16, which is the maximum queue length. Other nodes had very similar curves. We compare the two variants of Flush, Source and Hop-by-Hop, 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 the two Flush variants showed very similar high-level performance in terms of throughput and bandwidth, we see here that the Hop-by-Hop variant is much more stable, although not to the same extent as the fixed interval transfer.

Another benefit of the hop-by-hop rate limiting, as opposed to source-only limiting, can be seen in Figure 11. This plot shows the maximum queue occupancy for all nodes in the path, versus the packet sequence num-

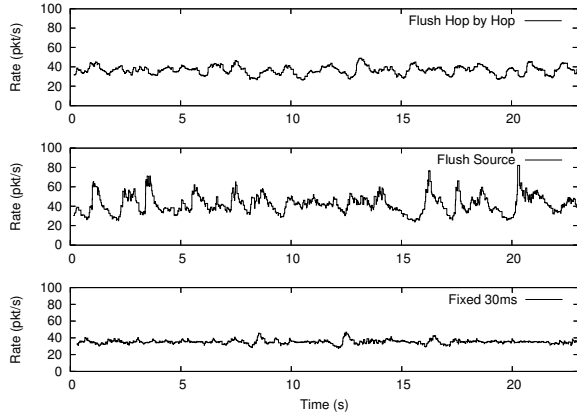


Figure 10: Packet rate over time for a source node 7 hops away from the base station. Packet rate averaged over 16 values, which is the max size of the queue. Flush Hop-by-Hop approximates the best fixed rate with the least variance.

ber. 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 the hop-by-hop case 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, because it allows the node to quickly increase its rate if there is a sudden increase in available bandwidth.

In contrast, Flush Source produces highly variable queue lengths. The lack of rate limiting at intermediary nodes induces a cascading effect in queue lengths, as shown in Figure 12. 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 10.

4.4 Adapting to Network Changes

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

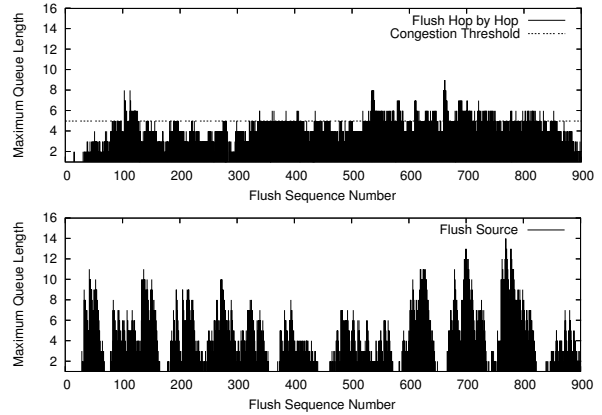


Figure 11: Maximum queue occupancy across all nodes for each packet. Flush Hop-By-Hop exhibits more stable queue occupancies than Flush Source.

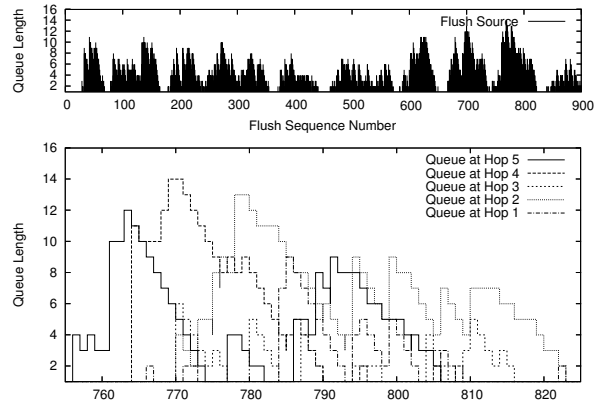


Figure 12: Detailed view of instantaneous queue length for Flush Source. Queue fluctuations ripple through nodes along a flow.

Figure 13 provides the instantaneous sending rate over the link with the forced losses for Flush Hop-by-Hop, Flush Source, and Fixed 30ms. Again, 30ms was the best fixed rate for this path before the link quality change was initiated. In the test, the link between two nodes, 3 and 2 hops from the sink, respectively, has its quality halved between the 7 and 17 second marks, relative to the start of the experiment. We see that the static algorithm rate becomes unstable during this period; due to the required retransmissions, the link can no longer sustain the fixed rate. Flush Hop-by-Hop adapts gracefully to the change, with a slight decrease in the sending rate. The variability remains constant during the entire experiment. Flush Source is not very stable when we introduce the extra losses, and is also less stable after the link quality is restored.

Figure 14 compares the queue lengths for the same

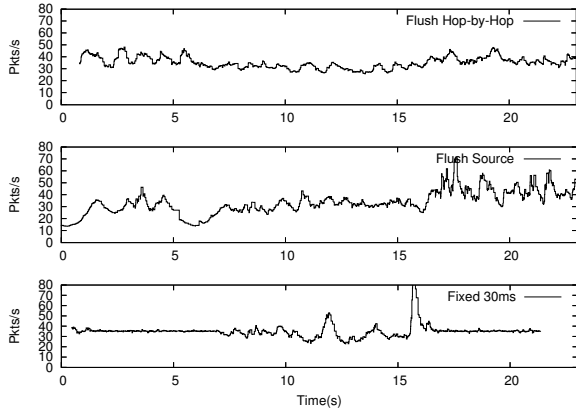


Figure 13: Sending rates at the lossy node for the forced loss experiment. Packets were dropped with 50% probability from between 7 and 17 seconds. Both Flush variants adapt while the fixed rate overflows its queue.

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.

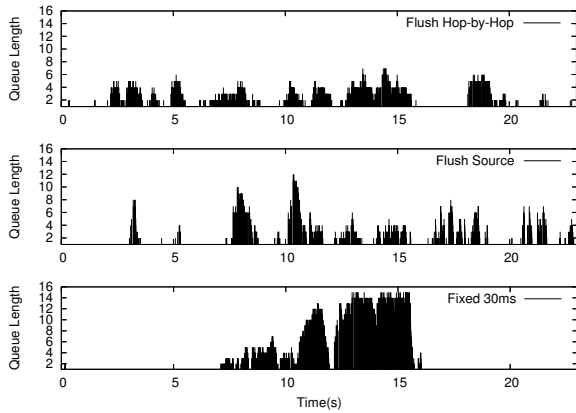


Figure 14: Queue length at the lossy node for the forced loss experiment. Packets were dropped with 50% probability from between 7 and 17 seconds. Both Flush variants adapt while the fixed rate overflows its queue.

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

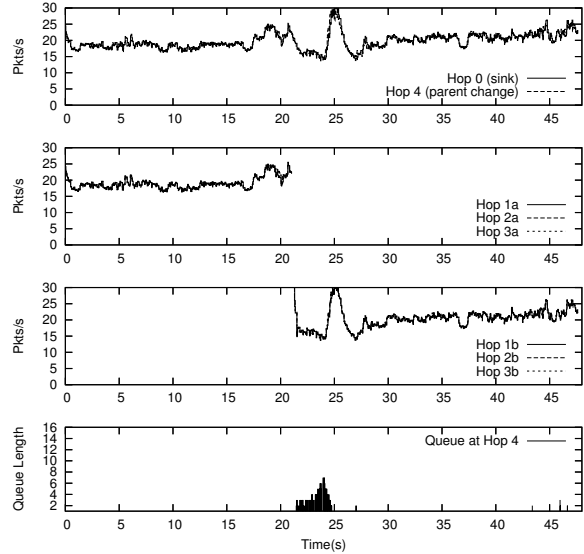


Figure 15: Detailed look at the route change experiment. Node 4's next hop is changed, changing all nodes in the subpath to the root. No packets were lost, and Flush adapted quickly to the change. The only noticeable queue increase was at node 4, shown in the bottom graph. This figure shows Flush adapts when the next hop changes suddenly.

a negligible effect on performance. Figure 15 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.

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.

4.5 Scalability

Finally, to evaluate the scalability of Flush, we deployed an outdoor network consisting of 79 MicaZ nodes. 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 16, where the rightmost node is 48 hops from the root, which is the leftmost node.

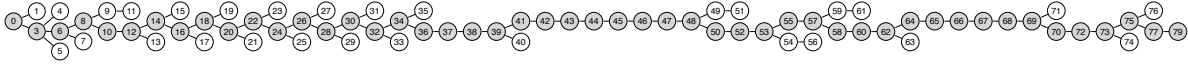


Figure 16: The network used for the scalability experiment. Of the 79 total nodes, the 48 nodes shown in gray were on the test path.

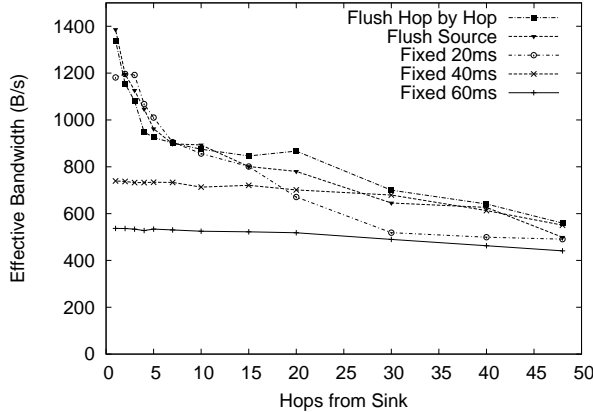


Figure 17: Effective bandwidth from the real-world scalability test where 79 nodes formed 48 hop network. The Flush header is 3 bytes and the Flush payload is 35-bytes (versus a 38 byte payload for the fixed rates). Flush closely tracks or exceeds the best possible fixed rate across at all hop distances that we tested.

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 14 bytes used previously) for Flush. The size of the Flush rate control header was reduced to 3 bytes (from 6 bytes used previously), leaving us with a protocol overhead of about 8%. We transfer a 26,600 byte data item 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 Source, Flush Hop-by-Hop, 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 17 shows the results of this experiment. The results indicate that *Flush efficiently transfers data over very long networks – 48 hops in this case.*

4.6 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

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.

Component	Memory Footprint		Code Size	
	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	

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.

5 Discussion

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 [27] 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 15, we also showed that Flush adapts robustly to a sudden change in the next hop. If the underlying routing protocol can find an alternate route, then Flush will adapt to it. But if the physical topology changes, and routing cannot adapt, then new routes will need to be rebuilt and the Flush session will have to be restarted.

It may seem that forcing all traffic to pass through a

single (rate-limited) queue would address the issue, but it does not. Nodes located on the flow path *would* be able to fairly queue routing and Flush traffic. However, nodes located off the flow path, but within interference range of the flow, would not be able to contend for bandwidth and successfully transmit beacons. Hence, if the physical topology changes along the flow path *during* a transfer, the nodes along the path may not be able to find an alternate route since beacons from these alternate routes may have been lost. A solution may be an interference-aware fair MAC. Many pieces are already in place [8, 3, 17] but the complete solution would require rate-controlling all protocols at the MAC layer across all nodes within interference range of the path.

One potential issue with the interference estimation algorithm presented in Section 3.3 is that it only works for packets actually received from the successors. Since the interference range of radios may exceed the reception range, a successor that is within interference range but beyond reception range might undetectably interfere with the previous hop's transmissions. Our current implementation does not address this issue in part because we have not seen any evidence that this condition occurs and in part because we are unclear how it might be detected and addressed without some tradeoffs.

If such interference were found to cause problems, it may be possible to address it in at least two ways. The network layer could prune the set of next hop candidates to only those links with at least 10 dBm of receive link margin. Alternately, the estimator could add some padding to the estimated forwarding time. Unfortunately, both of these approaches have some drawbacks. Choosing links with at least 10 dBm of link margin may increase the network diameter as short links that are at least 10 dBm above the noise floor will be chosen over longer links with lower link margins. In the extreme case, there may be no links with at least 10 dBm margin. Still, there is some hope that this approach would not appreciably reduce the throughput even though it would increase RTT slightly. In contrast, artificially padding the estimated forwarding time would decrease throughput. We defer for future work a careful analysis of these tradeoffs.

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 [13], 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 [2] 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 where transmissions interfere due to the nature of the broadcast medium.

ATP [22] and W-TCP [20], two wireless transport protocols that use rate-based transmission due to the high rates on wireless links, have also influenced our work. In ATP, each node in a path keeps track of its local delay and puts it in 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. In contrast, 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. We apply rate control to the problem of optimizing pipelining and interference, neither of which is addressed in ATP and W-TCP.

Wisden [16], like Flush, is a reliable data collection protocol. Nodes send data concurrently at a static rate over a collection routing 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 using a packet size of 80 bytes (the same as Wisden) and 68 byte payload. This experiment, repeated four times, shows that Flush achieved 2,226 bytes per second from a single hop compared with Wisden's 782 bytes per second. This difference can be explain 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 happen, leading to inefficiency.

A number of protocols have been proposed in the sensor network space which investigate aspects of this problem. RMST [21] outlines many of the theoretical and design considerations that influenced our thinking including architectural choices, link layer retransmission poli-

cies, end-to-end vs hop-by-hop semantics, and choice of ACKs, NACKs, SACKs, and SNACKs, even though their work was focused on analytical results and ns-2 simulations of 802.11 traffic. Fusion [8], IFRC [17], and the work in [3] address the problems of rate and congestion control for collection, but are focused on a fair allocation of bandwidth among several senders, rather than efficient and reliable end-to-end delivery. Fusion [8] 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 the sawtooth pattern of rate fluctuations makes for less overall efficiency than Flush’s estimated rate. Event-to-Sink Reliable Transport (ESRT) [18] 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 [25] is a transport protocol for sensor networks aimed at node reprogramming. This is a different problem than ours, as the data moves from the base station to a large number of nodes.

7 Conclusion

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

8 Acknowledgments

This work was supported by NSF grant #0435454 (“NeTS-NOSS”), NSF GRFP, Crossbow Technology, Microsoft Corporation, and Sharp Electronics.

References

- [1] <http://mirage.berkeley.intel-research.net/>.
- [2] BLACKWELL, T., CHANG, K., KUNG, H., AND LIN, D. Credit-based flow control for ATM networks. In *Proc. of the First Annual Conference on Telecommunications R&D in Massachusetts* (1994).
- [3] EE, C. T., AND BAJCSY, R. 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* (2004), ACM Press, pp. 148–161.
- [4] FONSECA, R., RATNASAMY, S., ZHAO, J., EE, C.-T., CULLER, D., SHENKER, S., AND STOICA, I. Beacon-vector routing: Scalable point-to-point routing in wireless sensor networks. In *Proceedings of the Second USENIX/ACM NSDI* (May 2005).
- [5] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)* (June 2003).
- [6] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems* (2000), pp. 93–104. TinyOS is available at <http://webs.cs.berkeley.edu>.
- [7] HILL, J., SZEWCZYK, R., WOO, A., LEVIS, P., WHITEHOUSE, K., POLASTRE, J., GAY, D., MADDEN, S., WELSH, M., CULLER, D., AND BREWER, E. Tinyos: An operating system for sensor networks, 2003.
- [8] HULL, B., JAMIESON, K., AND BALAKRISHNAN, H. Mitigating congestion in wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (Nov. 2004).
- [9] KIM, S., PAKZAD, S., CULLER, D. E., DEMMEL, J., FENVES, G., GLASER, S., AND TURON, M. Health monitoring of civil infrastructures using wireless sensor networks. Tech. Rep. UCB/ECS-2006-121, EECS Department, University of California, Berkeley, October 2 2006.
- [10] KIM, Y., GOVINDAN, R., KARP, B., AND SHENKER, S. Geographic routing made practical. In *Proceedings of the Second USENIX/ACM Symposium on Networked System Design and Implementation (NSDI 2005)* (Boston, MA, May 2005).
- [11] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI* (2002).
- [12] MAINWARING, A., POLASTRE, J., SZEWCZYK, R., CULLER, D., AND ANDERSON, J. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications* (Sept. 2002).
- [13] MISHRA, P. P., KANAKIA, H., AND TRIPATHI, S. K. On hop-by-hop rate-based congestion control. *IEEE/ACM Trans. Netw.* 4, 2 (1996), 224–239.
- [14] NAIK, V., ARORA, A., SINHA, P., AND ZHANG, H. 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)*.

- [15] NATH, S., GIBBONS, P. B., SESHAN, S., AND ANDERSON, Z. R. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (Sensys)* (New York, NY, USA, 2004), ACM Press, pp. 250–262.
- [16] PAEK, J., CHINTALAPUDI, K., CAFFEREY, J., GOVINDAN, R., AND MASRI, S. A wireless sensor network for structural health monitoring: Performance and experience. In *Proceedings of the Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*.
- [17] RANGWALA, S., GUMMADI, R., GOVINDAN, R., AND PSOUNIS, K. Interference-aware fair rate control in wireless sensor networks. In *SIGCOMM 2006* (Pisa, Italy, August 2006).
- [18] SANKARASUBRAMANIAM, Y., AKAN, O., AND AKYILDIZ, I. ESRT: Event-to-sink reliable transport in wireless sensor networks. In *In Proceedings of MobiHoc* (June 2003).
- [19] SHARP, C., SCHAFFERT, S., WOO, A., SASTRY, N., KARLOF, C., SASTRY, S., AND CULLER, D. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Second European Workshop on Wireless Sensor Networks* (January – February 2005).
- [20] SINHA, P., NANDAGOPAL, T., VENKITARAMAN, N., SIVAKUMAR, R., AND BHARGHAVAN, V. WTCP: A reliable transport protocol for wireless wide-area networks. *Wireless Networks* 8, 2-3 (2002), 301–316.
- [21] STANN, F., AND HEIDEMANN, J. RMST: Reliable data transport in sensor networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications* (Apr. 2003), IEEE, pp. 102–112.
- [22] SUNDARESAN, K., ANANTHARAMAN, V., HSIEH, H.-Y., AND SIVAKUMAR, R. 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)* (2003), pp. 64–75.
- [23] TOLLE, G., POLASTRE, J., SZEWCZYK, R., TURNER, N., TU, K., BUONADONNA, P., BURGESS, S., GAY, D., HONG, W., DAWSON, T., AND CULLER, D. A microscope in the redwoods. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (Sensys 05), San Diego* (November 2005), ACM Press.
- [24] VYAS, A. K., AND TOBAGI, F. A. 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)*.
- [25] WAN, C.-Y., CAMPBELL, A. T., AND KRISHNAMURTHY, L. 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).
- [26] WERNER-ALLEN, G., JOHNSON, J., RUIZ, M., LEES, J., AND WELSH, M. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)* (2005).
- [27] WOO, A., TONG, T., AND CULLER, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems* (2003), ACM Press, pp. 14–27.