Block-switched Networks: A New Paradigm for Wireless Transport

Ming Li, Devesh Agrawal, Deepak Ganesan, Arun Venkataramani, and Himanshu Agrawal
{*mingli, dagrawal, dganesan, arun, himanshu*}*@cs.umass.edu*
University of Massachusetts Amherst

Abstract

TCP has well-known problems over multi-hop wireless networks as it conflates congestion and loss, performs poorly over time-varying and lossy links, and is fragile in the presence of route changes and disconnections.

Our contribution is a clean-slate design and implementation of a wireless transport protocol, Hop, that uses *reliable per-hop block transfer* as a building block. Hop is 1) fast, because it eliminates many sources of overhead as well as noisy end-to-end rate control, 2) robust to partitions and route changes because of hop-by-hop control as well as in-network caching, and 3) simple, because it obviates complex end-to-end rate control as well as complex interactions between the transport and link layers. Our experiments over a 20-node multi-hop mesh network show that Hop is dramatically more efficient achieving 1.5x to 10x goodput over several alternate protocols across a wide range of operating conditions, while achieving better fairness.

1 Introduction

Wireless networks are ubiquitous, but traditional transport protocols perform poorly in wireless environments, especially in multi-hop scenarios. Many studies have shown that TCP, the universal transport protocol for reliable transport, is ill-suited for multi-hop 802.11 networks. There are three key reasons for this mismatch. First, multi-hop wireless networks exhibit a range of loss characteristics depending on node separation, channel characteristics, external interference, and traffic load, whereas TCP performs well only under low loss conditions. Second, many emerging multi-hop wireless networks such as long-distance wireless mesh networks, and delay-tolerant networks exhibit intermittent disconnections or persistent partitions. TCP assumes a contemporaneous end-to-end route to be available and breaks down in partitioned environments [14]. Third, TCP has well-known fairness issues due to interactions between its rate control mechanism and CSMA in 802.11, e.g., it is common for some flows to get completely shut out when many TCP/802.11 flows contend simultaneously [35]. Although many solutions (e.g. [18, 32, 36]) have been proposed to address parts of these problems, these have not gained much traction and TCP remains the dominant available alternative today.

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

Our transport protocol, Hop, uses *reliable per-hop block transfer* as a building block, in direct contrast to the above assumptions. Hop makes three fundamental changes to wireless transport. First, Hop replaces packets with *blocks*, i.e., large segments of contiguous data. Blocks amortize many sources of overhead including retransmissions, timeouts, and control packets over a larger unit of transfer, thereby increasing overall utilization. Second, Hop does not slow down in response to erroneous end-to-end feedback. Instead, it uses hop-by-hop backpressure, which provides more explicit and simple feedback (stop/send) that is considerably more robust to fluctuating loss and delay. Third, Hop uses hop-by-hop reliability in addition to end-to-end reliability. Thus, Hop is tolerant to intermittent disconnections and makes

progress even during partitions when a contemporaneous end-to-end route is unavailable.

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

A fundamental benefit of Hop is its ability to gracefully degrade with increasing loss in a wireless network, and its tolerance to disruptions in end-to-end connectivity. In the latter respect, Hop is inspired by work in delay-tolerant networking (DTN). However, unlike DTN transport protocols that are designed solely for intermittently connected networks [10], Hop is designed to maximize throughput both in well-connected mesh networks as well as challenged wireless edge networks.

Hop also co-exists easily with delay-sensitive applications like voice-over-IP and video using link-layer prioritization. These packets continue to use a traditional datagram abstraction, but traverse a separate high priority queue at the link layer, which the 802.11 MAC enables using smaller contention windows and backoff times. This feature already exists in widely used 802.11 implementations and commodity chipsets [1]. Thus, Hop requires no modification to 802.11 beyond disabling link layer acknowledgments for reliable traffic.

In summary, our main contribution is to show that reliable per-hop block transfer is fundamentally better than the traditional reliable end-to-end packet stream abstraction. We design, implement, and evaluate Hop: a fast, robust, and simple protocol for wireless transport. The individual components of Hop's design are simple and perhaps right out of an undergraduate networking textbook, but they provide dramatic improvements in combination. We compare Hop against 1) legacy TCP, 2) DTN 2.5, a delay tolerant transport protocol [10], 3) Stitched-TCP, a hop-by-hop TCP implementation, and 4) TCP-Westwood, a wireless TCP protocol [21], and show that:

- Hop is 50% better than alternate schemes over one hop paths and 7x better over five hop paths, across a range of 802.11 settings.
- Hop scales well with load, and offers 2x improvement in aggregate goodput while being fairer under heavy load comprising many large flows. Hop outperforms alternate schemes including TCP by upto an order of magnitude for Web traffic workloads.
- Hop is robust to route changes and partitions: it provides at least 50% better goodput than TCP in networks with high route flux, and achieves nonzero end-to-end goodput even in partitioned networks.

2 Why reliable per-hop block transfer?

In this section, we give some elementary arguments for why the use of reliable per-hop block transfer with hop-by-hop flow control is better than TCP's end-to-end packet stream with end-to-end rate control under high loss and high variability in loss and delay.

Block vs Packet: A major source of inefficiency is transport layer per-packet overhead for timeouts, acknowledgements and retransmissions. These sources of overhead are low in networks with low contention and loss but increase significantly as wireless contention and loss rates increase. Transferring data in blocks as opposed to packets provides two key benefits. First, it amortizes the overhead of each control packet over larger number of data packets. This allows us to use additional control packets, for example, to exploit in-network caching, which would be prohibitively expensive at the granularity of a packet. Second, it enables transport to leverage link-layer techniques such as 802.11 burst transfer capability [1], whose benefits increase with large blocks. In addition, we believe that a block-based protocol is better suited to take advantage of wireless link-layer optimizations such as opportunistic broadcast (e.g. ExOR [4]), network coding (e.g. MORE [15]), and concurrent transfers (e.g. CMAP [34]), all of which have significant control overhead and perform better when amortized over large blocks.

Transport vs Link-layer Reliability: Wireless channels can be lossy, with extremely high raw channel loss rates in high interference conditions. In such networks, the end-to-end loss rate along a multi-hop path increases exponentially in the number of hops, hence TCP throughput severely degrades due to prohibitive number of retransmissions. The state-of-the-art in wireless networks is to use sufficiently large number of 802.11 link-layer acknowledgements to provide a reliable channel abstraction to TCP. However, 802.11 retransmissions 1) interact poorly with TCP end-to-end rate control since it increases RTT variance, 2) increase per-packet overhead due to more carrier sense, backoffs, and acks, especially under high contention and loss conditions, and 3) reduce overall throughput by disproportionately using the chan-

nel for packets transmitted over bad links. Our experiments show that TCP's woes cannot be addressed by just setting the maximum number of 802.11 retransmissions to a large value. Unlike TCP, Hop relies solely on transport-layer reliability and avoids link-layer retransmissions for data, thereby avoiding negative interactions between the link and transport layers.

End-to-end rate control: Rate control in TCP happens in response to end-to-end loss and delay feedback obtained from each packet. However, end-to-end feedback is fundamentally error-prone and has high variance in multi-hop wireless networks since each packet can observe significantly different wireless interference across different contention domains as it is transmitted across a network. This variability impacts TCP's ability to: 1) utilize spatial pipelining since TCP window size is often small due to its conservative response to loss, and 2) fully utilize channel capacity since TCP experiences more frequent retransmission timeouts, during which no data is transmitted.

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

In-network Caching: Our case for hop-by-hop reliable block transfer is bolstered by technology trends in storage. Falling prices of storage devices make a compelling case for equipping each wireless router with fast secondary storage, and using storage to cache blocks that are in-transit. Caching can help prevent wasted work by 1) enabling more efficient retransmissions by exploiting cached blocks at intermediate hops, and 2) providing greater robustness to intermittent disconnections by enabling progress to be made even though a contemporaneous end-to-end route is unavailable. Hop exploits storage trends to fully utilize in-network storage for caching data blocks, and thereby minimizes the overhead of block retransmissions.

3 Design

This section explains the Hop protocol in detail. Hop is designed with the following four design goals in mind:

- Utilization: Hop should maximize channel utilization by minimizing control overhead, and should transmit data at close to 802.11 CSMA capacity when there is data to send.

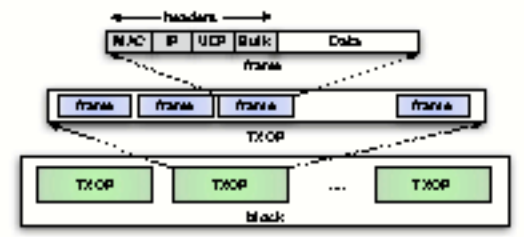


Figure 1: Structure of a block.

- Robustness: Hop should operate efficiently under a range of loss, contention, and disconnection settings, and gracefully degrade as network quality worsens.
- Co-existence: Hop should co-exist with delay-sensitive VoIP and video traffic without adversely impacting their performance.
- Caching: Hop should leverage in-network storage to cache data, thereby improving throughput by minimizing wasted transmissions and improving resilience to disconnections.

Hop's design consists of six main components: 1) reliable per-hop transfer, 2) ensuring end-to-end reliability, 3) backpressure flow control, 4) dealing with route changes and partitions, 5) ack witholding to handle hidden terminals, and 6) a per-node packet scheduler.

3.1 Reliable per-hop block transfer

The unit of reliable transmission in Hop is a *block*, i.e., a large segment of contiguous data. A block comprises a number of txops (the unit of a link layer burst), which in turn consists of a number of frames (Figure 1). The protocol proceeds in rounds until a block B is successfully transmitted. In round i, the transport layer sends a BSYN packet to the next-hop requesting an acknowledgment for B. Upon receipt of the BSYN packet, the receiver transmits a bitmap acknowledgement, BACK, with bits set for packets in the B that have been correctly received. In response to the BACK, the sender transmits packets from B that are missing at the receiver. This procedure repeats until the block is correctly received at the receiver.

The reason a BSYN is sent before the block in the first round is to check if the next-hop already has the block cached, e.g., because of a routing loop or an end-to-end retransmission. To keep a BSYN from looping indefinitely in the event of a routing loop, Hop uses TTL-scoping to drop the BSYN after 30 hops.

Control Overhead: Hop requires minimal control overhead to transmit a block. At the link layer, Hop disables acknowledgements for all data frames, and only enables them to send control packets: BSYN and BACK.

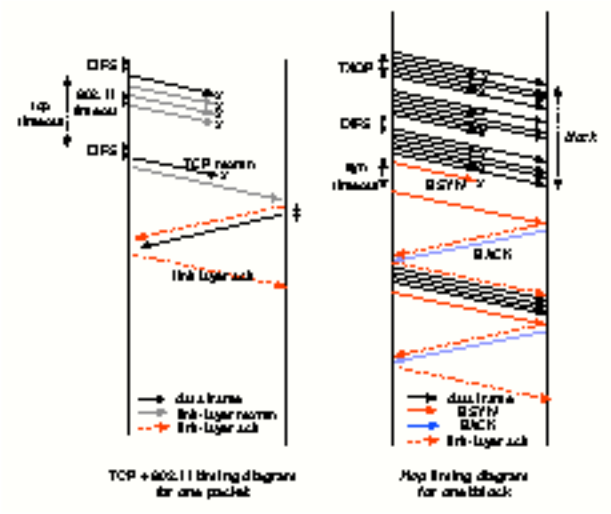


Figure 2: Timeline of TCP/802.11 vs. Hop

At the transport layer, a BACK acknowledges data in large chunks rather than in single packets. The reduced number of acknowledgement packets is shown in Figure 2, which contrasts the timeline for a TCP packet transmission alongside a block transfer in Hop. The cumulative number of Hop and 802.11 acknowledgements for the transfer of a block using Hop is two or more orders of magnitude less than for an equivalent number of packets using TCP for the default block size of Hop(1MB). In addition, Hop reduces idle time at the link layer by ensuring that packets do not wait for link-layer ACKs, and at the transport layer by disabling rate control. Thus, Hop nearly always sends data at a rate close to the link capacity.

Spatial Pipelining: The use of large blocks and hop-by-hop reliability can be a detriment for spatial pipelining since each node waits for the successful reception of a block before forwarding it. To improve pipelining, an intermediate hop forwards packets as soon as it receives at least a txop worth of new packets instead of waiting for an entire block. Thus, our protocol takes full advantage of spatial pipelining while simultaneously exploiting the benefits of burst transfer at the link layer.

3.2 Ensuring end-to-end reliability

Hop-by-hop reliability is insufficient to ensure reliable end-to-end transmission. A block may be dropped if 1) an intermediate node fails in the middle of transmitting a block to the next-hop, or 2) it exceeds its TTL limit.

Hop uses virtual retransmissions to limit the overhead of retransmitting large blocks. A re-transmitted block is likely cached at nodes along the original route until the point of failure or drop. Hence, instead of retransmitting

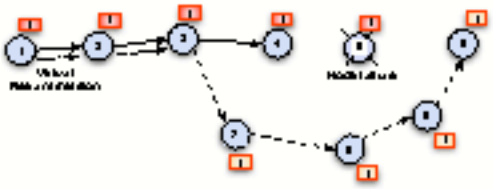


Figure 3: Virtual retransmission due to node failure.

the entire block, the sender sends a *virtual retransmission*, i.e., a special BSYN packet, using the same hop-by-hop reliable transfer mechanism as for a block. Virtual retransmissions exploit caching at intermediate nodes by only transmitting the block when the next hop along the route does not already have the block cached.

A premature timeout in TCP incurs a high cost both due to redundant transmission as well as its detrimental rate control consequence, so a careful estimation of timeout is necessary. In contrast, virtual retransmissions triggered by premature timeouts do little harm, so Hop simply uses a fixed short value for end-to-end timeouts.

3.3 Backpressure flow control

Rate control in response to congestion is critical in TCP to prevent congestion collapse. In wireless networks, congestion collapse can occur both due to increased packet loss rates due to contention [12], and increased loss due to buffer drops [11]. Both these cases result in wasted work, where a packet traverses several hops only to be dropped before reaching the destination.

Much work has observed that end-to-end feedback is highly noisy and difficult to interpret in wireless networks, which greatly complicates the design of a transport protocol [2, 32]. Hop is designed to rely solely on hop-by-hop signaling instead of end-to-end signaling to avoid congestion. For each flow, a Hop node monitors the difference between the number of blocks received and the number reliably transmitted to its next-hop as shown in Figure 4. Hop limits this difference to a small fixed value, H. Upon reaching that limit, a Hop node does not send any more BACKs until the difference drops to a smaller fixed value L. In practice, H and L are set to 1 and 0 respectively, except in highly partitioned scenarios (e.g. DTNs) where communication opportunities between nodes are infrequent.

Backpressure flow control in Hop serves two purposes: (a) better utilization, and (b) better management of storage. Hop uses backpressure to improve utilization. Consider the following scenario where flows $1, \ldots, k$ all share the first link with a low loss rate. Assume that the rest of flow 1's route has a similar low loss rate, while flows $2, \ldots, (k-1)$ traverse a poor route or are parti-

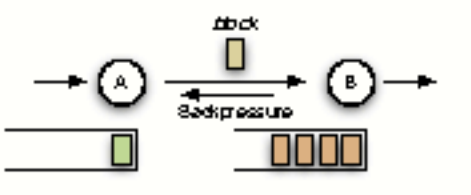


Figure 4: Backpressure from a node.

tioned from their destinations. Let C be the link capacity, p_1 be the end-to-end loss observed by the first flow, and p_2 be the end-to-end loss rate observed by other flows ($p_1 \ll p_2$). Without backpressure flow control, Hop would allocate a $1/k$ fraction of link capacity to flow 1, yielding a total goodput of $C\frac{((1-p_1)+(1-p_2)\cdot(k-1))}{k}$. On the other hand, a protocol that devotes all of the link capacity to flow 1 yields a goodput close to $C \cdot (1 - p_1)$. Thus, backpressure flow control achieves a balance between utilization and fairness.

The second benefit of backpressure flow control is to manage storage. Consider the case where a good link on a route is followed by an extremely slow wireless link. In this case, the disparity in draining rates can cause the amount of stored data to build up over time. Backpressure can avoid this problem by limiting the maximum number of blocks that accumulate at a node.

3.4 Robustness to Partitions

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

The ability to make progress during partitions relies on knowing which next-hop to use. Routing protocols designed for delay- or disruption-tolerance expose next-hop information even if an end-to-end route is unavailable (e.g. DTLSR [9]). However, mesh routing protocols [26, 3] do not expose this information. Hop requires a routing protocol that exposes next hop information even under partitioned settings. In conjunction with a delay-tolerant routing protocol, Hop can accomplish data transfer even if a contemporaneous end-to-end route is never available, i.e., the network is always partitioned.

3.5 Handling hidden terminals

The elimination of control overhead for block transfer improves efficiency but has an undesirable side-effect — it exacerbates loss in hidden terminal situations. Hop transmits blocks without rate control or link-layer retransmissions, which can result in a continuous stream of collisions at a receiver if the senders are hidden from each other. While hidden terminals are a problem even for TCP, rate control mitigates its impact on overall throughput. Flows that collide at a receiver observe increased loss and throttle their rate. Since different flows get different perceptions of loss, some reduce their rate more aggressively than others, resulting in most flows being completely shut out and bandwidth being devoted to one or few flows [35]. Thus, TCP is highly unfair but has good aggregate throughput.

Hop uses a novel *ack witholding* technique to achieve high throughput and fairness under hidden terminal situations. Here, a receiver acknowledges only one BSYN packet at any time, and withholds acknowledgement to other concurrent BSYN packets until the outstanding block has completed. In this manner, the receiver ensures that it is only receiving one block from any sender at a given time, and other senders wait their turn. Once the block has completed, the receiver transmits the BACK to one of the other transmitters, which starts transmitting its block. This approach allows us to balance utilization and fairness by using different policies for determining the order in which BSYNs are acknowledged. The receiver can preferentially ack senders on good links to improve throughput at the cost of fairness, or ack senders in the order of received blocks. On the other hand, TCP shuts off some flows and offers a fixed, unpredictable throughput/fairness tradeoff.

Ack witholding provides a lightweight alternative to more expensive and conservative techniques like RTS/CTS to deal with hidden terminals. The high overhead of RTS/CTS arises from the additional control packets, especially since these are broadcast packets that are transmitted at the lowest bit-rate. The use of broadcast also makes RTS/CTS more conservative since a larger contention region is cleared than typically required [34]. In contrast, ack witholding requires no additional control packets (BSYNs and BACKs are already in place for block transfer), and only targets hidden terminal scenarios involving multiple flows to a single receiver. While our scheme does not handle hidden terminal situations caused by flows to different receivers, we find that, in practice, a lightweight approach significantly alleviates hidden terminals at low cost.

3.6 Packet scheduling

Hop's unit of link layer transmission is a txop, which is the maximum duration for which an 802.11 card can send packets in a burst without contending for access [1]. A txop assumes and leverages the burst mode functionality at the link layer. Hop's scheduler sends a txop's worth of data from each flow at a time, thereby ensuring that the link layer can transmit it efficiently in burst mode.

Hop supports delay-sensitive traffic such as VoIP or

video using link layer prioritization. 802.11e enables differentiation between the traffic types by using smaller CSMA contention windows and backoff times for frames in the higher priority queues. There are four priority queues supported by 802.11e — Voice, Video, Best Effort, and Background, listed in order of reducing priority. Hop is designed primarily for reliable transfer, so delay-sensitive datagrams are scheduled to use the high priority Voice queue at the link layer. Link layer acknowledgments are turned on for delay-sensitive traffic. Hop effectively isolates bulk and delay-sensitive traffic by exploiting link-layer prioritization. Hop uses link layer prioritization also to separate control and data packets for reliable transport. The scheduler sends all control packets to the high priority Voice queue and all data packets to the Background queue. This helps avoid premature timeouts and stalled transfers by ensuring that control packets do not wait behind large data blocks in the 802.11 hardware buffer.

Finally, Hop has a flexible scheduler for data packets to enable different trade-offs between utilization and fairness. Hop's default scheduling policy is round-robin across all active flows; it also supports a per-hop proportionally fair scheduler that gives more time to packets over good links. Although we have not implemented them, it seems straightforward to support other per-hop scheduling policies as desired.

4 Implementation

We have implemented in Linux, all the features of Hop as described in Section 4.2. Our prototype implementation takes the form of a user-space library written in C, comprising about 5100 lines of code. The protocol runs as a single thread to avoid linux thread scheduling delay and jitter from impacting performance. Hop is completely implemented atop UDP, and has two UDP sockets open at all times, one for data packets and one for control packets. This section describes salient features of the implementation in detail.

4.1 Setting MAC Parameters

We use Atheros-based wireless chipsets and the Madwifi open source 802.11 device driver [20] in our implementation. By default, the MadWifi driver supports 802.11e QoS extensions only in the Access point mode. We modify the driver to enable the functionality in the Ad-hoc demo mode as well. We use the following 802.11 MAC layer settings in our protocol:

- We set the transmission opportunity (txop) for the background queue to the maximum setting permitted by the MadWifi driver (8160 μs or roughly 8KB of data).

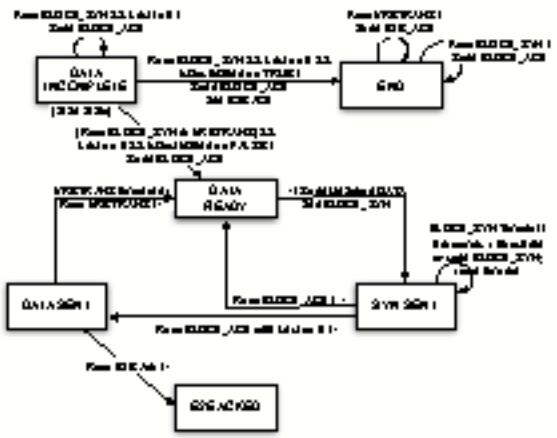


Figure 5: State machine for each block in Hop

- The contention window parameters (cw_min and cw_max) for all queues are set to their default settings.
- We disable link layer retransmissions and automatic ACKs for all packets being transmitted out through the Background queue, and enable it for the Voice queue.
- We disable auto rate control for packets sent through the background queue and set the data rate to 11 Mbps. All control packets in our system are transmitted at 1Mbps to ensure greater reliability.

4.2 Protocol Implementation

We now discuss the most important components of the implementation of our protocol.

State machine: The state machine for each block in Hop is shown in Figure 5. Briefly, when a node starts receiving a block the state of the block is set to INCOMPLETE. The block remains in INCOMPLETE state until the node receives the complete block and either a BSYN or virtual retransmission packet. If the node is the final destination, it sends an end-to-end acknowledgement to the sender acknowledging the receipt of the block. If the node is not the destination, the state of the block is then set to READY, and the node looks up the cache of recent routes to determine the most recent route to the destination. If the next node along this source-route path is not present in the routing table, the node issues a new probe to the destination, and uses the route returned by the new probe. The node then sends the BSYN to the next hop, and once the BACK is received, the data is transmitted and the block state is set to DATA SENT. The process of sending BSYNs, getting BACKs, and sending data repeats until the block is compeletely transmitted, after which the state is set to SEND DONE. The state of the

block remains in SEND DONE unless a virtual retransmission message is received. In this case, the state of the block is set back to READY, and the process repeats.

Serialized BACK: Our implementation uses the Received Signal Strength Indicator (RSSI) for different links to determine the order in which to transmit BACKs. Links are assigned probabilities in proportion to their RSSI values, therefore, BSYNs from senders that have higher signal strength are preferentially acknowledged. The RSSI values for the links is obtained from the 802.11 link layer to decide which BSYN to acknowledge. In practice, we find that this policy provides a good balance between utilization and fairness.

Hop socket: The basic Hop protocol provides a one-way reliable block transfer service and a block transfer API for applications to send and receive blocks. To enable existing applications to easily use Hop, we also provide a stream socket API like that of TCP, and a datagram socket API like that of UDP. When applications use the stream or datagram API to transfer data over Hop, a simple timeout-based batching policy is used to determine the size of a block. Data is enqueued in a Hop buffer until the timeout (set to 0.5 secs in our implementation), after which it is packaged into a single block and transmitted. An application can also request to send "urgent" data, which causes the buffer to be immediately transmitted.

Connection establishment/teardown: Connection establishment and teardown in Hop uses a two-way handshake. If node A wants to connect to node B, it sends a SYN message to B. If B is willing to accept a connection, it responds with a SYN-ACK, else it response with a RST message. All connection setup messages are send reliably, hence a two way handshake is sufficient to establish a connection (unlike TCP which requires a three-way handshake). To reduce the overhead of connection setup, we piggyback a small block with the SYN packet (set to a max of 8KB in our implementation). Connection teardown occurs by sending a FIN message to B. Since the FIN's are send reliably over Hop, we don't need an explicit FIN-ACK message as in the case of TCP.

Block size optimization: The transmission of a BSYN and BACK at the beginning of each block transmission has high overhead when a block is small in size. To improve efficiency for small blocks (less than 8KB), we piggyback the block together with the BSYN packet without waiting for a BACK. While this approach can result in wasted work if the block is already cached at the next hop, the utilization benefits by avoiding waiting for a BACK are considerably greater.

Storage: Hop may need to store blocks for long durations to handle long-lived network disconnections. Our current implementation caches blocks in RAM and only uses secondary storage when necessary. In practice, we find that we do not need to use secondary storage to store blocks except in partitioned settings where queues can build up over time. If storage is filled up, blocks can be aged using a simple LRU policy.

4.3 Routing

Our evaluation of Hop over a dynamic topology uses the Optimized Link State Routing (OLSR v0.5.5) routing protocol [26]. A major challenge that we face is that OLSR routes fluctuate many times a second, even in the absence of any traffic in the network. The reason for this behavior is because there are often multiple paths with similar ETX, and frequent, small changes in the ETX metric for paths trigger route changes. This fluctuation causes frequent transient loops to appear as the next hop of a node keeps changing. We modify OLSR to damp the rate at which a node calculates its routes from the link state database. Instead of the default settings which result in route re-computation many times a second, we wait at least 3 seconds between successive route re-computations.

4.4 Hop to TCP Proxy

We have implemented a simple application-level Hop-TCP proxy that enables a wireless node using Hop to connect to a TCP server in a wired network. The proxy divides the connection into two halves: a Hop connection from the wireless host to the proxy, and a TCP connection from the proxy to the Internet host. The proxy uses Hop's stream socket API, hence it does not need to perform any explicit batching of data received over the TCP connection before transmitting it over Hop. One limitation of our current implementation is that it breaks the end to end semantics between the client and the Internet server. However, this is an artifact of our userspace implementation. The effect on end-to-end semantics can be mitigated by a kernel implementation that uses the approach proposed in [6], where the proxy delays the FIN packet for the TCP connection until all the data has been acknowledged for the Hop connection.

5 Evaluation

Our goals in this section are to answer the following questions: 1) how does a single Hop flow perform for different path lengths, file sizes, and loss rates, 2) how much throughput and fairness benefits does Hop provide in heavily loaded networks, 3) what are the contributions of individual components of Hop towards overall throughput, 4) how does Hop perform for web traffic traces, 5) can Hop be used for access point networks and for internet access, 6) how robust is Hop under route flux and route partitions, and 7) how does Hop interact with delay-sensitive traffic like VoIP.

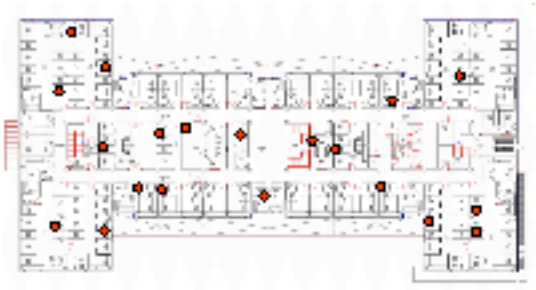


Figure 6: Experimental testbed with dots representing nodes.

We evaluate the performance of Hop in a multi-hop wireless testbed comprising 20 Apple Mac Mini computers. Each node is connected via its built-in 802.11 b/g wireless card to the multi-hop wireless network, and via its Ethernet port to a debugging and testing backplane that is used for delay and throughput measurements. All nodes are set to operate in 802.11b mode. The nodes are spread across a single floor of our Computer Science department as shown in Figure 6. A majority of our experiments were performed with static routing (except Section 5.7). The static routing topology was obtained by running the OLSR wireless routing protocol over our testbed until the network topology stabilizes. The resulting topology is used to configure routing tables.

We compare Hop against two classes of protocols, 1) two variants of end-to-end TCP protocols — the default TCP implementation in Linux and TCP Westwood+ [21], and 2) two variants of hop-by-hop TCP — DTN 2.5 [10] and Stitched-TCP [17]. TCP Westwood+ is a sender side modification of TCP that optimizes the performance of TCP congestion control over wireless networks. TCP Westwood is implemented in the Linux kernel and can be turned on using *sysctl*. DTN 2.5 is a reference implementation of the IRFC RFC 4838 and 5050 from the Delay Tolerant Networking Research Group [10]. It transfers data in a hop-by-hop manner using TCP to the destination. Stitched-TCP is a hop-by-hop TCP implementation that splits an end-to-end TCP connection into smaller hop-by-hop TCP segments [17]. No mechanism for flow control/rate control is used between the TCP segments. Since we did not find a reference implementation of the protocol, we implemented our own version. Each of the protocols are evaluated under different 802.11 settings for txop and retransmissions.

Since there is significant variability in wireless interference conditions, all experiments for a single graph were done back-to-back to obtain consistent results.

5.1 Single Flow Microbenchmarks

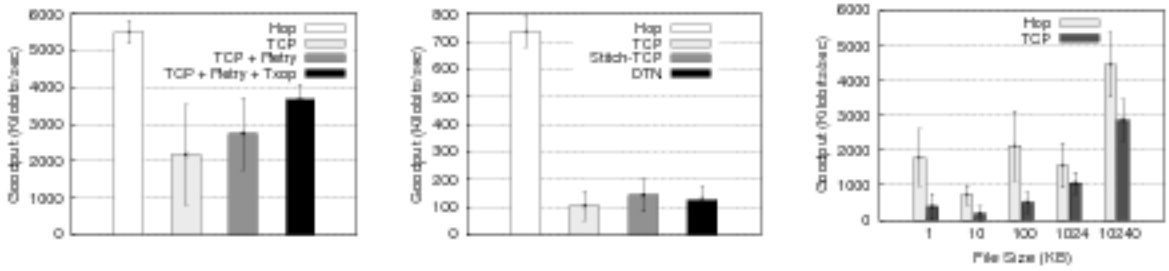
Our first set of experiments is a series of microbenchmarks of the performance of a single Hop flow under a wide range of topologies, loss rates, and parameters.

Single Hop Microbenchmark Our first microbenchmark validates our claim that Hop is faster than TCP under a range of 802.11 MAC layer parameter settings. In this experiment, we compare a single Hop flow against TCP with three different 802.11 settings: (a) default 802.11 settings (no txop, 11 retransmission limit), (b) 802.11 with txop and 11 retransmission limit, and (c) 802.11 with txop and 100 retransmission limit. Each flow sends 10 MB of data, and results are averaged over ten runs. Figure 7(a) shows that Hop outperforms TCP under all three settings. The fastest TCP combination is TCP + 802.11 txops + 100 retransmission limit, and Hop is 40% faster than this combination. While not shown for lack of space, similar improvements in throughput are obtained over 802.11g as well. For example, in the one hop case, Hop over 802.11g out-performs TCP over 802.11g by upto 2x in the case of a lossy link, and by roughly 30% in the case of a good link. These results suggest that the gains of Hop cannot be obtained by using legacy TCP and just changing link-layer parameters.

We find that turning on 802.11 txops and using large limit for retransmissions always gives the best throughput for all TCP-based schemes. Therefore, our experiments using these protocols use these settings for the rest of our evaluation.

Multi-hop Microbenchmark We now compare the performance of a single Hop flow against TCP, Stitched-TCP, and DTN over a five hop path (from the upper left corner node to the lower right corner node in our testbed). Our results in Figure 7(b) show that Hop over the five hop path has even larger benefits than in the one hop case. Hop is roughly 4x better than all other protocols. These benefits are because 1) TCP does not fully exploit 802.11 txops since it introduces delays between packets and does not send packets in a batch, 2) high per-packet overhead is introduced due to numerous 802.11 retransmissions, and 3) interactions between TCP rate control and 802.11 retransmission-induced variable delays result in more timeouts.

Impact of File Size Our next set of microbenchmarks evaluates whether the benefits provided by Hop is restricted to large file transfers, or whether there are benefits to be had for small files as well. Figure 7(c) compares Hop against TCP for different file sizes in a single hop case, and presents the aggregate results from ten runs. The results show that Hop's benefits are significant across a spectrum of file sizes from 1KB to 1MB, in fact, the benefits are highest for small files. Hop's benefits for small files is because of piggybacking data with connec-



(a) Single-Hop Benchmark (b) Multi-Hop Benchmark (c) Impact of File Size

Figure 7: Single flow microbenchmarks on performance of Hop.

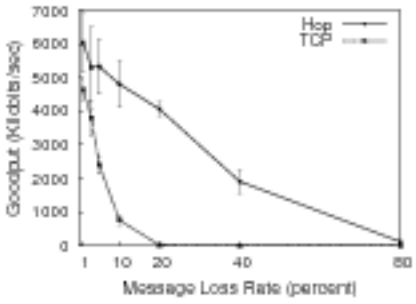


Figure 8: Impact of Loss on TCP and Hop.

tion establishment packets when the amount of data is less than 8KB (as described in Section 4.2). This considerably reduces the overhead of connection setup for small flows. The overall benefits of Hop over TCP range from 5x for files smaller than 100KB to about 50% for 1MB and 10MB files. The graph also seems to suggest that the average goodput does not follow a monotonic behavior as flow size increases. This is an artifact of changing external interference during our experiment. These results show that Hop can be expected to provide gains across a wide range of traffic patterns in wireless networks, ranging from web traffic with small flows, to large file transfers or peer-to-peer downloads.

Impact of Loss We now evaluate the impact of loss rate observed at the transport layer on the performance of Hop and TCP. For this experiment, we start with a one hop path that has extremely low loss. Loss is introduced by modifying the MadWifi device driver to randomly drop a specified fraction of incoming packets. Figure 8 shows the result of increasing loss rates. As seen, Hop is considerably more robust to packet loss than other protocols. Unsurprisingly, TCP goodput drops dramatically with increasing loss rate, and decreases to zero when loss rate is roughly 20%. Hop degrades much more gracefully and operates until the loss rate is about 80%. This shows that Hop is resilient to lossy network conditions, which bodes well for its usefulness in a wide range of wireless networks with different loss rates.

5.2 Hop under High Load

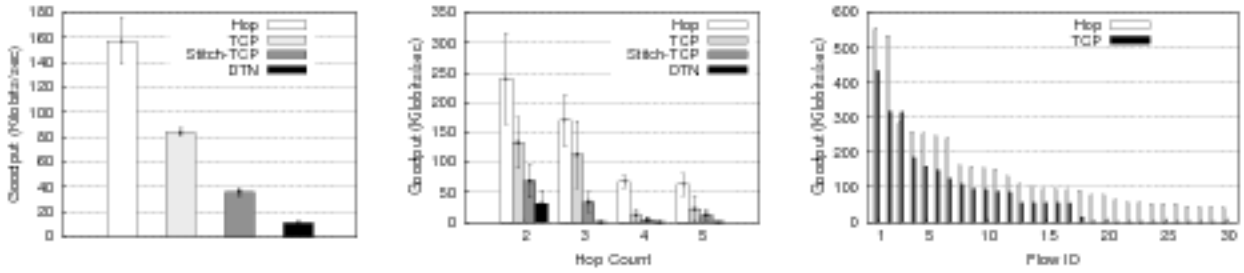
We now evaluate Hop in a heavily loaded network with large flows. Our goal in these experiments is to gain insight into the effect of increased inter-flow contention and collisions on the performance and fairness of Hop. In this experiment, we have thirty flows, each of transmits 10MB of data from a random sender to a random receiver in the network. To demonstrate multihop benefits, we only pick flows that are two hops or more. Each result is averaged over 10 runs.

Throughput Figure 9(a) shows the aggregate throughput obtained by Hop in comparison with other schemes. As shown, Hop out-performs all other schemes by significant margins. It is roughly 2x better than TCP, 4x better than Stitched TCP, and roughly 8x better than DTN. Stitched TCP and DTN suffer more than legacy TCP because of the lack of rate control or backpressure, which reduces overall utilization in a heavily loaded network. DTN also transmits the entire 10MB file one hop at a time, thereby not fully exploiting pipelining opportunities. A breakdown of the goodput for flows of different path lengths is provided in Figure 9(b). The results show that the gains increase with increasing number of hops. Hop's benefits over TCP increase from 80% to 180% as the path length increases from two to five hops.

	Fairness Index
Hop	0.61 (0.07)
TCP	0.35 (0.05)
Stitched-TCP	0.32 (0.07)
DTN	0.06 (0.006)

Table 1: Fairness index for high load setting.

Fairness In this experiment, we evaluate the fairness properties of Hop. The fairness metric that we use is hop-weighted Jain's fairness index (JFI [30]). When there are



(a) Aggregate Throughput (b) Throughput Breakdown (c) Per-flow Throughput

Figure 9: Hop under High load (30 flows)

n flows, x_1 through x_n, with hop lengths h_1 through h_n, JFI is computed as follows: $JFI = \frac{(\sum_{i=1}^{n} x_i \cdot h_i)^2}{n \sum_{i=1}^{n} (x_i \cdot h_i)^2}$

The fairness index for different protocols is shown in Table 1. Hop's fairness index is about 75% greater than that of TCP, and achieves even more gains for other protocols. This is notable since TCP sacrifices fairness for throughput, whereas Hop is significantly better than TCP on both these metrics. Figure 9(c) shows the individual goodput for different flows in a typical 30 flow experiment. The results show that TCP, Stitched-TCP, and DTN are all significantly less fair than Hop. In fact, almost 30% of the TCP flows are starved. Stitched TCP and DTN have similar fairness problems, primarily since they lack a rate control or backpressure-like mechanism. DTN, in particular, performs extremely poorly as the number of hops increase, resulting in long multi-hop flows being completely starved.

5.3 Hop Performance Breakdown

In this experiment, we answer the following question: how much does backpressure and ack witholding impact the overall performance of Hop? To answer this question, we compare three versions of Hop: (a) the basic Hop protocol that only uses hop-by-hop block transfer, (b) Hop with only ack witholding turned on, and (c) Hop with ack witholding and backpressure turned on. Since the impact of these mechanisms depend on the load in the network, we consider 1, 10 and 30 flow cases between random sender/receiver pairs in this experiment. Each flow transmits 10 MB of data.

Figure 10 shows the performance of different schemes normalized to the performance of Basic Hop. In the case of a single flow, ack witholding has no benefit since there are no hidden terminals. Backpressure has a small impact on throughput since it improves pipelining. As the number of flows increases, the benefits of the two schemes increase dramatically. In particular, ack witholding has a tremendous impact on overall throughput, and almost doubles the throughput in the 10 flow case, and increases throughput by almost 150% in the 30 flow case. The impact of backpressure increases with higher traffic load as well, since there is greater contention and loss in the network. For the 30 flow case, backpressure increases throughput by about 20% over just using ack witholding. Thus, both ack witholding and backpressure have significant impact on overall throughput of Hop, with ack witholding being the main contributor.

5.4 Hop for Web Traffic

Until now, our multi-flow experiments have considered a workload comprising solely of large files. Our next set of experiments evaluate the performance of Hop for a traffic pattern that comprises predominantly of small files. In particular, we consider a web traffic pattern where most files are webpages that are small in size [5]. The flow sizes used in this experiment were obtained from a HTTP proxy server trace obtained from the IRCache project [13]. The CDF obtained was sampled to obtain the 30 representative flow sizes used in this experiment. The distribution of file sizes is as follows: roughly 35% of the files are less than 10KB, 60% are between 10KB-100KB, and 5% are greater than 100KB. The sender and receiver for each flow are chosen randomly among the nodes in the network.

The results for this experiment are shown in Figure 11. Hop is 6-16x faster than TCP for files less than 10KB in size, and 4-7x faster for files between 10-20KB in size. They show that Hop has benefits across diverse traffic patterns from small to large files.

5.5 Hop in Access Point Networks

In this experiment, we evaluate the gains that Hop can achieve in a typical one-hop WiFi network where a number of terminals connect to a single access point. We setup a seven to one topology for this experiment, by selecting a node in the center of our testbed to act as the "AP node", and transmitting data to this node from all its seven neighbors. Among the seven nodes that were transmitting, five pairs were hidden terminals (i.e. they could not reach each other but could reach the AP). We

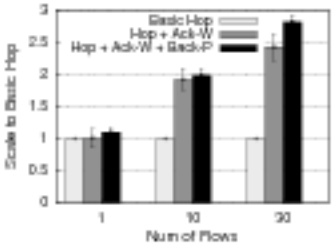


Figure 10: Hop breakdown

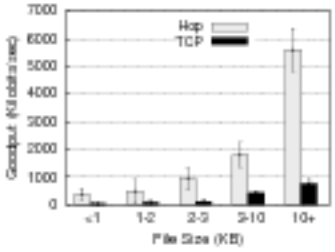


Figure 11: Performance for web traffic.

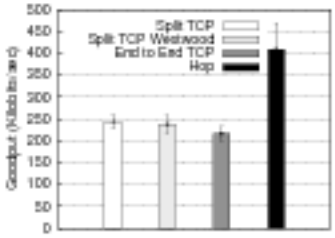


Figure 12: Hop for Internet access

verified this by checking to see if they could transmit simultaneously. All flows transmit 10MB of data.

We use two policies for ack-witholding: 1) Best link first policy where the BSYNs from the best links are preferentially acked (the default policy in Hop), and 2) Weighted RSSI policy where BSYNs are acked in proportion to their received signal strength. We compare these two variants of Hop against TCP without RTS/CTS enabled, and TCP with RTS/CTS to handle hidden terminals. The results are presented in Table 2. The two policies enable Hop to tradeoff utilization against fairness: the Best Link first policy has 70% higher utilization but has 13% lower fairness than the Weighted RSSI policy. Both variants of Hop outperform the two TCP variants. Hop with Best Link policy achieves more than 2x improvement over TCP without RTS/CTS while achieving comparable fairness, and Hop with Weighted RSSI policy gets 70% higher throughput than TCP with RTS/CTS while achieving equivalent fairness. These results show that Hop can be used to achieve high throughput and high fairness in an AP setting.

	Goodput (Kbps)	Fairness Index
Hop (Best link)	2177 (43)	0.59 (0.01)
Hop (Weighted RSSI)	1279 (211)	0.67 (0.11)
TCP	955 (68)	0.58 (0.16)
TCP + RTS/CTS	743 (49)	0.65 (0.06)

Table 2: Goodput and Fairness for a many-to-one "AP" setting. 95% confidence intervals shown in parenthesis

5.6 Hop for Internet Download

Now we evaluate Hop's performance in the context of downloading a file from the Internet using Hop. In this experiment, three nodes downloaded a 10MB file from from a webserver at MIT. Each of the nodes was five hops away from a gateway node, that was configured as the Hop to TCP proxy.

We compare Hop against three TCP variants: 1) end-to-end TCP from the wireless node to the Internet server, 2) Split TCP where the end-to-end connection is split into two segments — TCP from the wireless to the proxy, and TCP from the proxy to the Internet server, and 3) Split TCP where TCP Westwood+ is used from the wireless node to the proxy instead of legacy TCP. Fig. 12 shows that Hop achieves roughly 60% improvement over other schemes. We believe that a kernel implementation of Hop and a more efficient Hop-TCP proxy implementation can further improve our benefits. Our results show that Hop is a better solution than TCP for wireless to Internet communication in multi-hop mesh networks.

5.7 Robustness

In this section, we ask whether the performance benefits of Hop translate to dynamic topologies with frequent route changes as well as topologies with partitions.

	Goodput (Kbps)	Fairness
Hop	201.5 (28)	0.74 (0.07)
TCP	132.6 (14.9)	0.64 (0.13)

Table 3: Goodput and Fairness of Hop and TCP over OLSR. 95% confidence intervals shown in parenthesis

Route Flux In this experiment, we run Hop over the OLSR routing protocol. Each run of our experiment has ten flows, which transmit 10MB of data. Sender/receiver pairs were chosen such that they would be roughly two hops or more apart to stress multi-hop performance.

Table 3 shows the results. Hop achieves roughly 50% improvement in throughput over TCP, and achieves 16% higher fairness. The benefits of Hop falls short of the considerably better performance that we observed in the static topology experiments. The reason is because OLSR has frequent routing changes that are triggered by loss of its probe packets due to collision with data packets. During this period, Hop blocks that are being transmitted to a neighbor may be forwarded over multiple hops by OLSR. In our experiments, we found that roughly 50% of one-hop transmissions by Hop were being forwarded through multiple hops to a neighbor, resulting in at least 50-100% overhead for Hop.

This problem is an artifact of our current implementa-

tion. One solution is to bypass the routing table while a block is in transit by transmitting raw packets to the next hop. A second method is to use broadcast packets for data. Since broadcast packets are not forwarded through IP forwarding, this bypasses the routing table and prevents multi-hop forwarding of the packets. While we have not implemented these fixes currently, we believe that with such changes, Hop will exhibit the same benefits with dynamic topologies as with static topologies.

	Goodput (Kbps)
Hop w/ Back-P=1	40 (20)
Hop w/ Back-P=100	281 (17)

Table 4: Goodput achieved by Hop in a partitioned network without an end-to-end path.

Route Partitions Our second experiment evaluates a major benefit of Hop — its ability to keep operating even during partitions. To evaluate our partition tolerance, we pick a five hop path and simulate a partition scenario by bringing down the third, fourth and fifth nodes along the path for one minute each in a round robin manner. Table 4 shows the goodput obtained by Hop averaged over three runs under two different backpressure settings: 1) backpressure limit (H) is set to 1 and 2) backpressure limit is set to 100. The results show that Hop continues to operate under such a partitioned setting, and a large backpressure limit improves throughput six-fold. This is intuitive since having a larger threshold enables maximal use of periods of connectivity between nodes. The ability to easily tune the backpressure threshold to adapt Hop to disconnected settings is a significant property of our protocol. In contrast to Hop, TCP achieves zero throughput since a contemporaneous end-to-end path is never available. We were unable to get DTN 2.5 to work for this experiment, but we expect its performance to be worse than Hop given its high overhead as observed in previous one-hop microbenchmarks.

5.8 Hop with VoIP

In this experiment, we quantify the impact of Hop and TCP on Voice over IP traffic. We use the mean opinion score (MoS) to evaluate the quality of a voice call. The MoS value can range from 1-5, where above 4 is considered good, and below 3 is considered bad. The MoS score for a VoIP call is estimated from the R-factor [8] as: $MoS = 1 + 0.035R + 7 * 10^{-6} R(R-60)(100-R)$. We then calculate the R-factor for the G.729 codec, which is used on many VoIP devices. The R-factor for this codec is: $R = 94.2 - 0.024d - 0.11(d-177.3) \cdot H(d-177.3) - 11 - 4\log(1+10e)$, where d is the total ear-to-mouth delay and e is the loss rate. We assume that the coding delay is 25 ms, jitter buffer delay is 60 ms, and the delay over the wired segment of the end-to-end path is 30ms.

A VoIP flow is generated as a stream of 20 byte packets every 20 ms, which are transmitted over UDP. We have five Hop/TCP flows, each of which transmits over three hops, and one VoIP flow transmitting over three hops in this experiment. The five Hop/TCP flows are distributed across the network such that there is significant interference between them and the VoIP flow.

Table 5 shows that Hop achieves twice the goodput of TCP, but only has small impact on delay and loss for VoIP. As a result, the MoS score remains good, and only reduces by 0.1.

	Goodput (Kbps)	Delay (ms)	Loss	MoS
Hop	266 (148)	76.4	0.1	4.27 (0.25)
TCP	136 (119)	59.3	0.07	4.36 (0.12)

Table 5: Interaction between VoIP and Hop/TCP flows. Result shows the avg. goodput, delay, loss rate, and Mean opinion score for VoIP. Standard deviation is shown in parenthesis.

6 Related work

The performance and fairness problems in TCP over 802.11 have been well studied. Our primary contribution is to draw upon this work and show that reliable per-hop block transfer may be a better building block for reliable transport in wireless networks, and to design a practical, full-fledged implementation based on this idea.

6.1 Proposed Schemes

TCP in wireless: Much work on TCP in multi-hop wireless networks addresses performance drawbacks in wireless networks resulting from its difficulty in distinguishing between packet losses due to wireless packet corruption, congestion, and route changes [2], and its negative interactions with the routing layer and CSMA link layer. Proposed solutions include: (a) end-to-end approaches that try to distinguish between the different loss events [28, 37], attempt to estimate the rate to recover quickly after a loss event [21], or reduce TCP congestion window increments to be fractional [23] in-order to reduce interactions between TCP and the routing layer, (b) network-assisted approaches that utilize feedback from intermediate nodes, either for ECN notification [36], failure notification [19] or rate estimation [32], and (c) link-layer solutions that use a fixed window TCP in conjunction with link-layer techniques such as neighborhood-based Random Early Detection ([11]) or backpressure flow control (RAIN [18]) to prevent losses due to link queues filling up. Hop avoids reliance on noisy end-to-end feedback, and bypasses complex problems of distinguishing

between loss events as well as rate estimation. Instead, it relies solely on simple hop-by-hop signaling for dealing with congestion and loss. In addition, it requires no link-layer modifications unlike link-layer solutions.

Bulk transfer: Hop is similar in spirit to bulk transfer protocols such as NETBLT [7], that are designed to reduce the impact of unreliable links and variable delays on throughput. NETBLT relies an end-to-end reliability and window-less rate estimation, whereas Hop avoids noisy end-to-end feedback and the complex problem of rate estimation, and uses simple hop-by-hop backpressure flow control. Bulk transfer in sensor networks is addressed in Flush [16], which is designed for transferring data across a long linear chain of nodes. Flush uses a hop-by-hop rate control approach that relies on observed intra-flow interference to adjust flow rate at each hop. This protocol assumes that there is only one flow in the network, and does not extend easily to multi-flow settings that we address in this work.

802.11 enhancements: The 802.11e extensions [1] are crucial to the link layer optimizations in Hop. 802.11e enables adjustment of three key MAC layer parameters: txop or Transmission Opportunity that controls the maximum duration of a packet burst; CWmin and CWmax, which control the minimum and maximum sizes of the contention window; and AIFS, which controls the minimum duration between two packets in a burst. Much work has evaluated the benefits of 802.11e enhancements (e.g.: [33]). However, there has been limited work on understanding the impact of 802.11e on TCP. One exception is [25] which explores how TCP fairness over wireless networks can be improved by 802.11e prioritization. We show that 802.11e enhancements have limited impact on TCP throughput. Hop is designed to fully exploit 802.11e features such as txops and prioritization to maximize throughput.

Backpressure: Hop by hop flow control or backpressure was first used in ATM and high-speed networks to handle data bursts [22, 27]. More recently, they have been used for congestion adaptation in wireless and sensor networks (RAIN [18], Fusion [12]). While Hop is similar to these techniques, it differs in the use of block-level backpressure, which requires far fewer control packets than packet-level backpressure.

Transport Fairness: TCP unfairness over 802.11 stems from three key problems: (a) short-term unfairness in CSMA/CA behavior, which is analyzed in [24], (b) excess time spent in TCP slow-start, which is addressed in [32] by use of better rate estimation, and (c) interactions between spatially proximate interfering flows that may not traverse a single link, which is addressed in [35] and [31] by using neighborhood-based random early detection and rate control techniques. We eliminate complex interactions between TCP and 802.11 that are the source of these fairness issues, and instead provide simple tuning knobs — backpressure flow control, and delayed block-acknowledgements — to tradeoff between fairness and throughput.

Batching: Hop relies heavily on batching of packets for its throughput gains. Several recent tangentially-related work on wireless networks such as ExOR [4], MORE [15], CMAP [34], and WildNet [29] use batching as well. Among these, ExOR, and MORE use batching to reduce the number of control packets to obtain link-layer information from neighbors, whereas WildNet uses batching with FEC encoding to eliminate the long delay in waiting for an ACK when the receiver is many kilometers away. While batching is used mostly as an optimization in the above approaches, we investigate fundamental benefit of such batching and design a transport protocol to exploit batching benefits at both the transport and link layers.

6.2 Implemented Systems

While numerous schemes have been proposed for addressing various aspects of wireless transport, there are few fully implemented alternatives to TCP for 802.11 mesh networks. In fact, we found only two such implementations — DTN 2.5 and TCP Westwood — both of which we evaluate in our experiments. We believe that the paucity of available implementations is due to 1) the difficulty in implementing a full-blown wireless protocol that shows consistent gains across a range of network conditions, and 2) the need to operate under the constraints of existing 802.11 hardware. We provide a full design and implementation of a high-performance transport protocol that consistently outperforms existing techniques across a range of scenarios.

7 Conclusions

In this paper, we presented the design and implementation of Hop, a fast, robust and simple wireless transport protocol. Hop is fast because it reduces sources of overhead for reliable per-hop block transfer, eliminates noisy end-to-end rate control, and nearly always send at link capacity. Hop is robust because it operates under high route flux and makes progress even in partitioned topologies. Hop is simple because it eliminates many complex interactions between the transport layer and 802.11. Our results show that Hop improves over TCP in terms of goodput, delay, and fairness in almost all experiments that we have conducted, with the largest improvements (upto 10x improvement in throughput) for flows traversing multiple hops.

Hop opens up new directions, primarily because it is simple and considerably more predictable than the combination of TCP and 802.11. One direction is to exploit opportunism in the protocol, which is trivial to en-

able since data blocks can be transferred over unicast or broadcast modes to reduce wasted transmissions during route changes. A second direction is showing that Hop can be used in mobile networks (MANETs) in addition to mesh networks and DTNs. We are currently evaluating Hop over MANETs and a real partitionable network environment, DieselNet, a mobile bus-based testbed. A third direction is moving towards high data rate wireless standards such as 802.11n. Based on our preliminary studies, we believe that Hop will perform even better with 802.11n since we can transmit more data within a txop (64KB), giving us more batching benefits. All of these are part of ongoing work.

The Hop source code and a draft RFC is available at: `http://www.cs.umass.edu/~mingli/hop/`.

References

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

[2] Balakrishnan, H., Padmanabhan, V. N., Seshan, S., and Katz, R. H. A comparison of mechanisms for improving tcp performance over wireless links. In *SIGCOMM* (1996).

[3] Bicket, J., Aguayo, D., Biswas, S., and Morris, R. In *MobiCom* (2005).

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

[5] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: evidence and implications. *INFOCOM* (1999).

[6] Chakravorty, R., Katti, S., Crowcroft, J., and I.Pratt. Proxy-based flow aggregation for enhanced tcp over gprs. *INFOCOM* (2003).

[7] Clark, D. L., Lambert, M. M., and Zhang, L. RFC 998: Netblt: A bulk data transfer protocol, Mar. 1987.

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

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

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

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

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

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

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

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

[16] Kim, S., Fonseca, R., Dutta, P., Tavakoli, A., Culler, D., Levis, P., Shenker, S., and Stoica, I. Flush: a reliable bulk transport protocol for multihop wireless networks. In *SenSys* (2007).

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

[18] Lim, C., Luo, H., and Choi, C.-H. RAIN: A reliable wireless network architecture. In *Proceedings of IEEE ICNP* (2006).

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

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

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

[22] Mishra, P. P., and Kanakia, H. A hop by hop rate-based congestion control scheme. *SIGCOMM* (1992).

[23] Nahm, K., Helmy, A., and Kuo, C.-C. J. Tcp over multihop 802.11 networks: issues and performance enhancement. In *MobiHoc* (2005).

[24] Nandagopal, T., Kim, T.-E., Gao, X., and Bharghavan, V. Achieving mac layer fairness in wireless packet networks. In *MobiCom* (2000).

[25] Ng, A. C. H., Malone, D., and Leith, D. J. Experimental evaluation of tcp performance and fairness in an 802.11e test-bed. In *E-WIND* (2005).

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

[27] Özveren, C., Simcoe, R., and Varghese, G. Reliable and efficient hop-by-hop flow control. *SIGCOMM* (1994).

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

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

[30] Raj Jain, Arian Durresi, G. B. Throughput fairness index: An explanation. atm forum/99-0045, february 1999.

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

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

[33] Tinnirello, I. S. C. Efficiency analysis of burst transmissions with block ack in contention-based 802.11e wlans. *ICC* (2005).

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

[35] Xu, K., Gerla, M., Qi, L., and Shu, Y. Enhancing tcp fairness in ad hoc wireless networks using neighborhood red. In *MobiCom* (2003).

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

[37] Zhenghua Fu; Greenstein, B. X. M. S. L. Design and implementation of a tcp-friendly transport protocol for ad hoc wireless networks. *Proceedings of ICNP* (12-15 Nov. 2002), 216–225.