Block-switched Networks: A New Paradigm for Wireless Transport

Ming Li, Devesh Agrawal, Deepak Ganesan, and Arun Venkataramani
{*mingli, dagrawal, dganesan, arun*}*@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 better fairness, throughput, delay, and robustness to partitions over several alternate protocols, including gains of more than an order of magnitude in median throughput.

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 [13]. 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 [37]. Although many solutions (e.g. [16, 32, 38]) have been proposed to address parts of these problems, these have not gained much traction and TCP remains the dominant available alternative today.

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

Our transport protocol, Hop, uses *reliable per-hop block transfer* as a building block, in direct contrast to the above assumptions. Hop makes three fundamental changes to wireless transport. First, Hop replaces packets with *blocks*, i.e., large segments of contiguous data. Blocks amortize many sources of overhead including retransmissions, timeouts, and control packets over a larger unit of transfer, thereby increasing overall utilization. Second, Hop does not slow down in response to erroneous end-to-end feedback. Instead, it uses hop-by-hop backpressure, which provides more explicit and simple feedback that is robust to fluctuating loss and delay. Third, Hop uses hop-by-hop reliability in addition to end-to-end reliability. Thus, Hop is tolerant to intermittent disconnections and can make progress even when a contemporaneous end-to-end route is never available, i.e., the network is always partitioned [3].

Large blocks introduce two challenges that Hop con-

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

In summary, our main contribution is to show that reliable per-hop block transfer is fundamentally better than the traditional end-to-end packet stream abstraction through the design, implementation, and evaluation of Hop. The individual components of Hop's design are simple and perhaps right out of an undergraduate networking textbook, but they provide dramatic improvements in combination. In comparison to the best variant of 1) TCP, 2) Hop-by-hop TCP, and 3) DTN 2.5, a delay tolerant transport protocol [8],

- Hop achieves a median goodput benefit of 1.6× and 2.3× over single- and multi-hop paths respectively. The corresponding lower quartile gains are 28× and 2.7× showing that Hop degrades gracefully.
- Under high load, Hop achieves over an order of magnitude benefit in median goodput (e.g., 90× over TCP with 30 concurrent large flows), while achieving comparable or better aggregate goodput and transfer delay for large as well as small files.
- Hop is robust to partitions, and maintains its performance gains in well-connected WLANs and mesh networks as well as disruption-prone networks. Hop also co-exists well with delay-sensitive VoIP traffic.

2 Why reliable per-hop block transfer?

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

Block vs. packet: A major source of inefficiency is transport layer per-packet overhead for timeouts, acknowledgements and retransmissions. These overheads are low in networks with low contention and loss but increase significantly as wireless contention and loss rates increase. Transferring data in blocks as opposed to packets provides two key benefits. First, it amortizes the overhead of each control packet over larger number of data packets. This allows us to use additional control packets, for example, to exploit in-network caching, which would be prohibitively expensive at the granularity of a packet. Second, it enables transport to leverage link-layer techniques such as 802.11 burst transfer capability [1], whose benefits increase with large blocks.

Transport vs. link-layer reliability: Wireless channels can be lossy with extremely high raw channel loss rates in high interference conditions. In such networks, the end-to-end delivery rate decreases exponentially with the number of hops along the path, severely degrading TCP throughput. The state-of-the-art response today is to use a sufficiently large number of 802.11 link-layer acknowledgements (ARQ) to provide a reliable channel abstraction to TCP. However, 802.11 ARQ 1) interacts poorly with TCP end-to-end rate control as it increases RTT variance, 2) increases per-packet overhead due to more carrier sensing, backoffs, and acknowledgments, especially under high contention and loss (in §5.1.1, we show that 802.11b ARQ has 35% overhead). Note that TCP's woes cannot be addressed by just setting the 802.11 ARQ limit to a large value as it would reduce the overall throughput by disproportionately using the channel for transmitting packets over bad links. Unlike TCP, Hop relies solely on transport-layer reliability and avoids link-layer retransmissions for data, thereby avoiding negative interactions between the link and transport layers.

Hop-by-hop vs. end-to-end congestion control: Rate control in TCP occurs in response to end-to-end loss and delay feedback reported by each packet. However, end-to-end feedback is error-prone and has high variance in multi-hop wireless networks as each packet observes significantly different wireless interference across different contention domains along the route. This variance hurts TCP's utilization as: 1) its window size shrinks conservatively in response to loss, and 2) it experiences frequent retransmission timeouts when no data is sent.

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

In-network caching: The use of reliable per-hop block transfer enables us to exploit caching at intermediate hops for two benefits. First, caching obviates redundant retransmissions along previously traversed segments of a route. Second, caching is more robust to intermittent disconnections as it enables progress even when a contemporaneous end-to-end route is unavailable. Hop

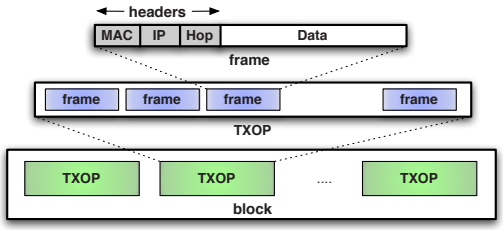


Figure 1: Structure of a block.

can also use secondary storage if needed in partitionable networks with long disconnections and reconnections.

3 Design

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

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, the receiver transmits a bitmap acknowledgement, BACK, with bits set for packets in B that have been correctly received. In response to the BACK, the sender transmits packets from B that are missing at the receiver. This procedure repeats until the block is correctly received at the receiver.

Control Overhead: Hop requires minimal control overhead to transmit a block. At the link layer, Hop disables acknowledgements for all data frames, and only enables them to send control packets: BSYN and BACK. At the transport layer, a BACK acknowledges data in large chunks rather than in single packets. The reduced number of acknowledgement packets is shown in Figure 2, which contrasts the timeline for a TCP packet transmission alongside a block transfer in Hop. For large blocks (e.g. 1 MB), Hop requires orders of magnitude fewer acknowledgements than for an equivalent number of packets using TCP with link-layer acknowledgements. In addition, Hop reduces idle time by ensuring that packets do not wait for link-layer ACKs, and at the transport layer by disabling rate control. Thus, Hop nearly always sends data at a rate close to the link capacity.

Spatial Pipelining: The use of large blocks and hop-by-hop reliability can hurt spatial pipelining since each

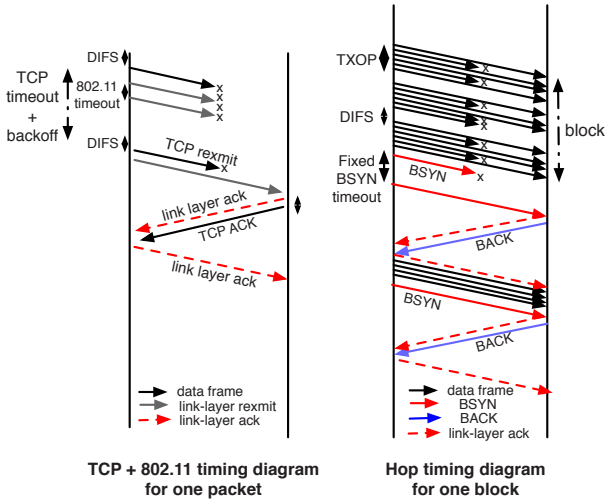


Figure 2: Timeline of TCP/802.11 vs. Hop

node waits for the successful reception of a block before forwarding it. To improve pipelining, an intermediate hop forwards packets as soon as it receives at least a txop worth of new packets instead of waiting for an entire block. Thus, Hop leverages spatial pipelining as well as the benefits of burst transfer at the link layer.

3.2 Ensuring end-to-end reliability

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

Hop uses virtual retransmissions together with in-network caching to limit the overhead of retransmitting large blocks. Hop routers store all packets that they overhear. Thus, a re-transmitted block is likely cached at nodes along the original route until the point of failure or drop, and might be partially cached at a node that is along a new path to the destination but overheard packets transmitted on the old path. Hence, instead of retransmitting the entire block, the sender sends a *virtual retransmission*, i.e., a special BSYN packet, using the same hop-by-hop reliable transfer mechanism as for a block. Virtual retransmissions exploit caching at intermediate nodes by only transmitting the block (or parts of the block) when the next hop along the route does not already have the block cached as shown in Figure 3.

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

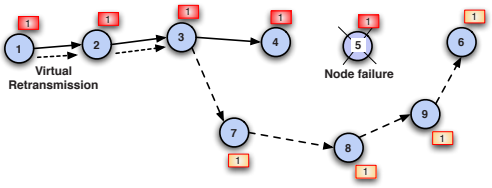


Figure 3: Virtual retransmission due to node failure.

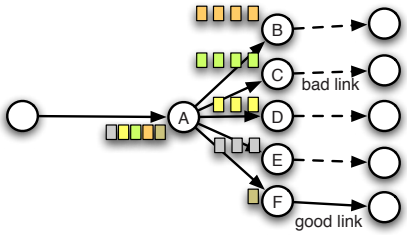


Figure 4: Example showing need for backpressure. Without backpressure, Node A would allocate $1/5$th of out-going capacity to each flow, resulting in queues increasing unbounded at nodes B through E. With backpressure, most data is sent to node F, thereby increasing utilization.

3.3 Backpressure congestion control

Rate control in response to congestion is critical in TCP to prevent congestion collapse and improve utilization. In wireless networks, congestion collapse can occur both due to increased packet loss due to contention [11], and increased loss due to buffer drops [9]. Both cases result in wasted work, where a packet traverses several hops only to be dropped before reaching the destination. Prior work has observed that end-to-end loss and delay feedback has high variance and is difficult to interpret unambiguously in wireless networks, which complicates the design of congestion control [2, 32].

Hop relies only on hop-by-hop backpressure to avoid congestion. For each flow, a Hop node monitors the difference between the number of blocks received and the number reliably transmitted to its next-hop as shown in Figure 4. Hop limits this difference to a small fixed value, H, and implements it with no additional overhead to the BSYN/BACK exchange. After receiving H complete blocks, a Hop node does not respond to further BSYN requests from an upstream node until it has moved at least one more block to its downstream node. The default value of H is set to 1 block.

Backpressure in Hop significantly improves utilization. To appreciate why, consider the following scenario where flows $1, \ldots, k$ all share the first link with a low loss rate. Assume that the rest of flow 1's route has a similar low loss rate, while flows $2, \ldots, (k-1)$ traverse a poor route or are partitioned from their destinations. Let C be the link capacity, p_1 be the end-to-end loss observed by the first flow, and p_2 be the end-to-end loss rate observed by other flows ($p_1 \ll p_2$). Without backpressure, Hop would allocate a $1/k$ fraction of link capacity to each flow, yielding a total goodput of $C\frac{((1-p_1)+(1-p_2)\cdot(k-1))}{k}$. And the number of buffered blocks at the next-hops of the latter $k-1$ flows grows unbounded. On the other hand, limiting the number of buffered blocks for each flow yields a goodput close to $C \cdot (1-p_1)$ in this example.

Why does Hop limit the number of buffered blocks, H, to a small default value? Note that the example above can be addressed simply by choosing the block corresponding to the flow with the largest differential backlog (along A-F). Indeed, classical backpressure algorithms known to achieve optimal throughput [33] work similarly. Hop limits the number of buffered blocks to a small value in order to ensure small transfer delay for finite-sized files, as well as to limit intra-path contention.

3.4 Robustness to partitions

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

The ability to make progress during partitions relies on knowing which next-hop to use. Unlike typical mesh routing protocols [23, 4], routing protocols designed for disruption-tolerance expose next-hop information even if an end-to-end route is unavailable (e.g. RAPID [3], DTLSR [7]). In conjunction with such a disruption-tolerant routing protocol, Hop can accomplish data transfer even if a contemporaneous end-to-end route is never available, i.e., the network is always partitioned.

In disruption-prone networks, a Hop node may need to cache blocks for a longer duration in order to make progress upon reconnection. In this case, the backpressure limit needs to be set taking into account the fraction of time a node is partitioned and the expected length of a connection opportunity with a next-hop node along a route to the destination (see §5.7 for an example).

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 [37]. Thus, TCP is highly unfair but has good aggregate throughput.

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

Although ack withholding does not address hidden terminals caused by flows to different receivers, it offers a lightweight alternative to expensive and conservative techniques like RTS/CTS for the common single-terminal hidden terminal case. The high overhead of RTS/CTS arises from the additional control packets, especially since these are broadcast packets that are transmitted at the lowest bit-rate. The use of broadcast also makes RTS/CTS more conservative since a larger contention region is cleared than typically required [39]. In contrast, ack withholding requires no additional control packets (as BSYNs and BACKs are already in place for block transfer).

3.6 Packet scheduling

Hop's unit of link layer transmission is a txop, which is the maximum duration for which the network interface card (NIC) is permitted to send packets in a burst without contending for access [1]. Hop's scheduler leverages the burst mode and sends a txop's worth of data from each concurrent flow at a time in a round-robin manner.

Hop traffic is isolated from delay-sensitive traffic such as VoIP or video by using link-layer prioritization. 802.11 chipsets support four priority queues—voice, video, best-effort, and background in decreasing order of priority—with the higher priority queues also having smaller contention windows [1]. Hop traffic is sent using the lowest priority background queue to minimize impact on delay-sensitive datagrams.

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

4 Implementation

We have implemented a prototype of Hop with all the features described in §3. Hop is implemented in Linux 2.6 as an event-based user-space daemon in roughly 5100 lines of C code. Hop is currently implemented on top of UDP (i.e., there is a UDP header in between the IP and Hop headers in each frame in Figure 1). Below, we describe important aspects of Hop's implementation.

4.1 MAC parameters

Our implementation uses the Atheros-based wireless chipset and the Madwifi open source 802.11 device driver [18], a popular commodity implementation. By default, the MadWifi driver (as well as other commodity implementations) supports the 802.11e QoS extension. However, MadWiFi supports the extension only in the access point mode, so we modify the driver to enable it in the ad-hoc mode as well. Hop uses default 802.11 settings, except for the following. The transmission opportunity (txop) for the background queue is set to the maximum value permitted by the MadWifi driver (8160 μs or roughly 8KB of data). Link-layer ARQ is disabled for all data frames sent via Hop but enabled for control packets (BSYN, BACK, etc).

4.2 Hop implementation

Parameters A large block size increases batching benefits [15], so we set the default maximum block size to 1MB. Note that this means that a Hop block is allowed to be up to 1MB in size, but may be any smaller size. Hop never waits idly in anticipation of more application data in order to obtain batching benefits. The BSYN batch threshold for micro-blocks is set to a default value of 16KB, and the backpressure limit, H, is set to 1. The virtual retransmission timeout is set to an initial value of 60 seconds and simply reset to the round-trip block delay reported by the most recent block. The TTL limit for a virtual retransmissions is set to 50 hops. In the current implementation, an intermediate Hop node keeps all the blocks that it has received in memory.

Header format: The Hop header consists of the following fields. All frames contain the `msg_type` that identifies if the frame is a data, BSYN, BACK, virtual retransmission BSYN, or an end-to-end BACK frame;

the `flow_id` that uniquely identifies an end-to-end Hop connection; and the `block_num` identifies the current block. Data frames also contain the `packet_num` that is the offset of the packet in the current block. The `packet_num` is also used to index into the bitmap returned in a BACK frame.

End-to-end connection management: Because Hop is designed to work in partitionable networks, it does not use a three-way handshake like TCP to initiate a connection. A destination node sets up connection state upon receiving the first block. The loss of the first block due to a node failure or expiry or the loss of the first end-to-end BACK is handled naturally by virtual retransmissions. In our current implementation, a Hop node tears down a connection simply by sending a FIN message and recovering state; we have not yet implemented optimizations to handle complex failure scenarios.

5 Evaluation

We evaluate the performance of Hop in a 20-node wireless mesh testbed. Each node is an Apple Mac Mini computer running Linux 2.6 with a 1.6 Ghz CPU, 2 GB RAM and a built-in 802.11a/b/g Atheros/MadWiFi wireless card. Each node is also connected via an Ethernet port to a wired backplane for debugging, testing, and data collection. The nodes are spread across a single floor of the UMass CS building as shown in Figure 5.

All experiments, except those in §5.9 and §5.10, were run in 802.11b mode with bit-rate locked at 11 Mbps. There is significant inherent variability in wireless conditions, so in order to perform a meaningful comparison, a single graph is generated by running the corresponding experiments back-to-back interspersed with a short random delay. The compared protocols are run in sequence, and each sequence is repeated many times to obtain confidence bounds.

We compare Hop against two classes of protocols: *end-to-end* and *hop-by-hop*. The former consists of 1) UDP, and 2) the default TCP implementation in Linux 2.6 with CUBIC congestion control [10]; we did not use the Westwood+ congestion control algorithm since it performed roughly 10% worse. The latter consists of 3) Hop-by-Hop TCP, and 4) DTN2.5 [8]. Hop-by-Hop TCP is our implementation of TCP with backpressure. It splits a multi-hop TCP connection into multiple one-hop TCP connections, and leverages TCP flow control to achieve hop-by-hop backpressure. Each node maintains one outgoing TCP socket and one incoming TCP socket for each flow. When the outgoing socket is full, Hop-by-Hop TCP stops reading from the incoming socket, thereby forcing TCP's flow control to pause the previous hop's outgoing socket. This "backpressure" propagates up to the source and forces the source to slow down. DTN2.5 is a reference implementation of the IEEE RFC 4838 and 5050

Figure 5: Experimental testbed with dots representing nodes.

from the Delay Tolerant Networking Research Group [8] that reliably transfers a bundle using TCP at each hop. Hop and UDP were set to use the same default packet size as TCP (1.5KB). In all our experiments, the delay and goodput of TCP are measured after subtracting connection setup time.

5.1 Single-hop microbenchmarks

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

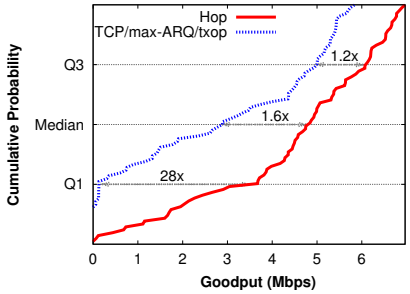


Figure 6: Experiment with one-hop flows. Hop improves lower quartile goodput by 28×, median goodput by 1.6×, and mean goodput by 1.6× over TCP with the best link layer settings.

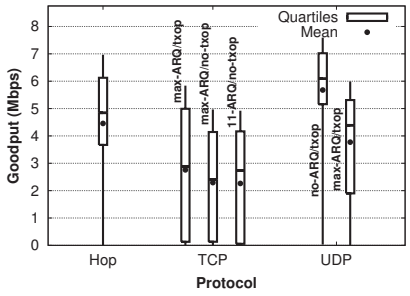


Figure 7: Experiment with one-hop flows. Box shows lower/median/upper quartile, lines show max/min, and dot shows mean. Increasing 802.11 ARQ limit and using txops helps TCP but Hop is still considerably better. UDP results show that ARQs incur significant performance overhead (35%). Hop is within 24% of UDP without ARQ (achievable goodput).

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

Table 1: Summary of evaluation results. All protocols above are given the benefit of burst-mode (txop) and the maximum number of link-layer retransmissions (max-ARQ) supported by the hardware.

5.1.1 Randomly picked links

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

We compare Hop against TCP for three 802.11 settings: 1) 11 link layer retries (ARQ) with no txop, the default settings of the MadWifi driver, 2) 11 ARQ + txop, and 3) maximum permitted ARQ setting (18 for the Atheros card) + txop. We do not consider TCP with no ARQ since it (expectedly) performs poorly without 802.11 retransmissions on lossy links. We also compare against UDP under different 802.11 settings. Since UDP has no transport-layer control overhead, and transmits as fast as the card can transmit packets, it provides an upper bound on the achievable capacity on the link. For clarity of presentation, we show cumulative distributions (CDFs) for Hop and the best TCP combination and summary statistics for the other combinations (for which full distributions are available in [15]).

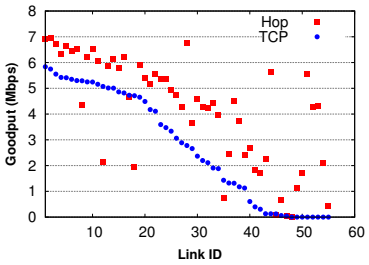
Figure 6 shows that Hop significantly outperforms TCP/max-ARQ/txop, the best TCP combination. The Q1, Q2, and Q3 gains over TCP/max-ARQ/txop TCP combination are 28×, 1.6×, and 1.2× respectively. The Q1 gain is notable and shows Hop's robust performance on poor links compared to TCP.

Figure 7 shows the summary statistics for Hop and two best TCP and UDP schemes using a box plot representation. The "box" shows the upper quartile (Q3), median (Q2) and lower quartile (Q1), and the "whiskers" show the maximum and minimum goodput. UDP/no-ARQ/txop is the best UDP combination and provides an upper bound on the achievable rate. The median Hop is about 24% lower than the achievable rate. Interestingly, turning on ARQ degrades UDP by 35% showing that ARQ in 802.11 comes at a high overhead and ARQ alone is not sufficient to fix TCP's problems.

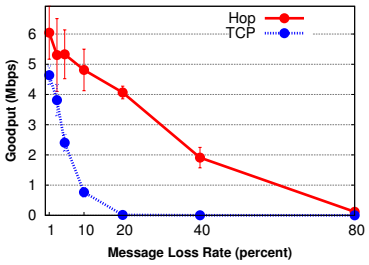
As we find that TCP performance consistently improves by using txops and ARQ with the maximum possible limit, we give TCP and its variants the benefit of txop/max-ARQ in the rest of our evaluation.

5.1.2 Graceful performance degradation

A key benefit of Hop is robustness, i.e., its performance gracefully degrades with increasing link losses and interference. To confirm this, we further analyze the data from the experiment in §5.1.1. Figure 8(a) shows the per-link throughput across the 56 links in the testbed (with multiple runs over the same link averaged) sorted by TCP goodput. Hop degrades gracefully to some of the poorest



(a) Sorted Single-hop Flows



(b) Impact of Loss.

Figure 8: Graceful degradation to adverse channel conditions. First plot shows per-link goodputs from one-hop experiment sorted in TCP order. Second plot shows controlled experiments demonstrating impact of loss. In both cases, Hop is more robust and degrades far more gracefully than TCP.

links in the testbed where TCP's throughput is near-zero. The average goodput for the worst 20 TCP flows is 334 Kbps, whereas Hop's goodput for the same flows is 2.37 Mbps, a difference of 7×.

To understand the cause of TCP's fragile behavior, we evaluate the impact of loss perceived at the transport layer on the performance of Hop and TCP. We start with a perfect link that has a near-zero loss rate and introduce loss by modifying the MadWifi device driver to randomly drop a specified fraction of incoming packets. Figure 8(b) shows that, unsurprisingly, TCP goodput drops to near-zero when loss rate is roughly 20%. Hop shows graceful near-linear degradation and is operational until the loss rate is about 80%.

5.2 Multi-hop microbenchmarks

How does Hop perform on multi-hop paths compared to existing alternatives? To study this question, we pick a sequence of 100 node pairs randomly with repetition from the testbed. Static routes are set up a priori between all node pairs to isolate the impact of route flux (considered in §5.3). The static routes were obtained by running OLSR with the default ETX metric until the routing topology stabilized at the beginning of the experiment. Among the 100 randomly chosen flows, 30% are two-hop, 30% are three-hop, 10% are four-hop, 20% are five-hop, and the remaining 10% are seven-hop flows. We compare the multi-hop goodput of Hop to TCP, Hop-by-Hop TCP, DTN2.5, and UDP.

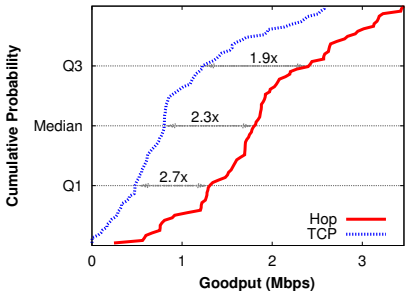


Figure 9: Experiment with multi-hop flows. Hop improves lower quartile goodput by 2.7×, median goodput by 2.3×, and mean goodput by 2×.

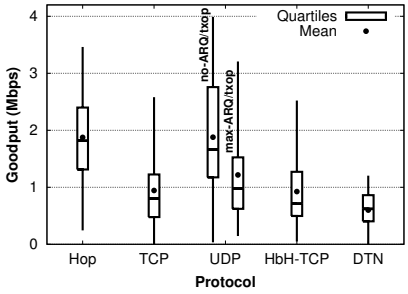


Figure 10: Boxplot of multi-hop single-flow benchmarks. Hop has 2-3× median, and 2-4× mean improvements over other reliable transport protocols. Hop is comparable to UDP/no-ARQ/txop in terms of median/mean — the latter is extremely fast since it has no overhead, but experiences more loss.

Figure 9 shows the CDF of goodput for just Hop and TCP, while Figure 10 shows the summary statistics for all the protocols. Hop consistently outperforms all other protocols. The Q1, Q2, and Q3 gains over TCP are 2.7×, 2.3× and 1.9× respectively. The Q1 gain over TCP is lower than for the single-hop experiment because only good links selected by OLSR are used in this experiment (as evidenced by the better performance of UDP/no-ARQ/txop compared to UDP/max-ARQ/txop). Over lossier paths, Hop's gains are much higher. We also find that the gains also grow with increasing number of hops (refer technical report [15]). For example, the lower quartile gains grow from about 2.7× for two hops to more than 4× for five and six hops.

5.3 Hop under high load

The experiments so far considered one flow in isolation. Next, we evaluate Hop in a heavily loaded network to understand the effect of increased contention and collisions on Hop's performance and fairness. We compare Hop, TCP, and Hop-by-Hop TCP. The experiment consists of thirty concurrent flows that transfer data continually between randomly chosen node pairs in the testbed. All protocols are run over a static mesh topology identical to §5.2. To focus on multihop benefits, we pick src-dst pairs

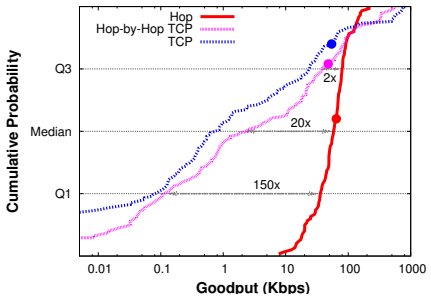


Figure 11: Hop for 30 concurrent flows. Dots on each line shows mean goodput. Median gains of Hop over Hop-by-Hop TCP and regular TCP are huge (20× and 90× respectively) while mean gains are modest (roughly 25% improvement).

that are not immediate neighbors of each other. We run the experiment five times, and for each run, we measure the goodputs of flows half an hour into the experiment, since the network reaches a steady state at this time.

5.3.1 Goodput

Figure 11 shows that Hop achieves a huge improvement in median goodput over TCP and Hop-by-Hop TCP. Hop achieves a median goodput of 54.9 Kbps whereas all the other protocols achieve less than 2.8 Kbps—an improvement of over an order of magnitude! Hop also improves the Q1 goodput by more than two orders of magnitude and upper quartile goodput by 2× over the other protocols. The exact numbers of Hop's median and Q1 gains over other protocols are sensitive to environmental conditions, but we consistently observe them to be large under different conditions. The figure also shows that Hop-by-Hop TCP achieves more than 4× improvement over TCP's median goodput. This shows that end-to-end rate control hurts TCP utilization and using hop-by-hop backpressure with TCP improves its performance. We also run UDP (not shown for clarity), but due to lack of congestion control, around 67% flows get zero goodput (i.e., the median is zero) and the mean goodput is 0.32Kbps.

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

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

Table 2: Fairness indexes for the 30 flow experiment. Parentheses show 95% confidence intervals.

5.3.2 Fairness

Table 2 shows the fairness index for different protocols. The fairness metric that we use is hop-weighted Jain's fairness index (JFI [28]). When there are n flows, with throughput x_1 through x_n and hop lengths h_1 through h_n, it is computed as follows: JFI $= \frac{(\sum_{i=1}^{n} x_i \cdot h_i)^2}{n \sum_{i=1}^{n} (x_i \cdot h_i)^2}$.

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

5.4 Hop performance breakdown

How much do components of Hop individually contribute to its overall performance? To answer this question, we compare four versions of Hop: 1) the basic Hop protocol that only uses hop-by-hop block transfer, 2) Hop with ack withholding turned on, 3) Hop with backpressure turned on, and 4) Hop with both ack withholding and backpressure turned on. Since the impact of these mechanisms depends on the load in the network, we consider 10, 20 and 30 concurrent flows between randomly picked sender-receiver node pairs. A static mesh topology identical to §5.2 was used. The length of the randomly picked paths are between three and seven hops. The average path length is 3.9 hops in the 10 flow case, 4 hops in the 20 flow case, and 3.9 hops in the 30 flow case. Each flow transmits a large amount of data, and we take a snapshot of the measurements after half an hour.

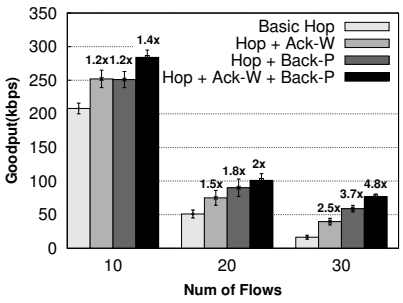


Figure 12: Hop performance breakdown showing contribution of ack withholding and backpressure. Ack withholding and backpressure improve Hop's performance by more than 4.8x under high load.

Figure 12 shows the performance of the different schemes. The benefit of ack withholding and backpressure increases with network load. In the 10 flow case, both ack withholding and backpressure increase goodput by around 20%. With greater network load, congestion increases dramatically, hence the gains due to backpressure is more than due to ack withholding. For example, in the 30 flow case, Hop with backpressure yields 3.7× improvement over basic Hop, whereas Hop with ack withholding yields 2.5× improvement. Furthermore, the benefits of using both backpressure and ack withholding are considerably more than using either one of them.

For instance, the full-fledged Hop yields 4.8× improvement over basic Hop for the 30 flow case.

5.5 Hop with WLAN access points

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

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

Table 3: Mean/median goodput and Fairness for a many-to-one "AP" setting. 95% confidence intervals shown in parenthesis

We compare Hop against TCP both with and without 802.11 RTS/CTS enabled. The results are presented in Table 3, and show that Hop beats TCP with or without RTS/CTS both in throughput and fairness. While the mean gains over TCP without RTS/CTS are only 12%, the median improvement is about 2.7×. TCP has a crafty way of maintaining high aggregate goodput amidst hidden terminals by squelching all but one of the flows and in effect serializing them. In contrast, Hop achieves almost perfectly fair allocation across the different flows. The addition of RTS/CTS to TCP hurts aggregate throughput but improves median throughput and fairness. However, Hop achieves 1.4× the aggregate throughput, 1.96× the median throughput, in addition to hugely improving fairness over TCP with RTS/CTS.

5.6 Hop delay for small file transfers

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

5.6.1 Single-hop transfer delay for small files

First, we evaluate the benefits of Hop's size-aware ack withholding policy. To evaluate this, we pick a one-hop Wifi network where five nodes are connected to an AP (similar setup as our WLAN experiments). In each experiment, one of the five nodes (randomly chosen), transmits a micro-block to the AP at a random time, whereas the other four nodes continually transfer large amounts of data. Each experiment runs until the micro-block completes, at which point we compute the delay for the transfer. We compare against TCP with and without RTS/CTS, and report aggregate numbers over five runs. Figure 13 shows that the transfer delay of the micro-block with Hop is always lower than for TCP (with or without RTS/CTS). In many cases, the delay gains are significant, e.g., for file sizes less than 16KB, the gains range from 3× to 15×. This experiment shows that Hop can be used for delay-sensitive transfers like web transfers, ssh, and SMS in many-to-one AP settings.

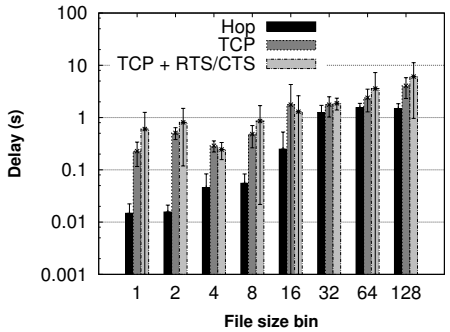


Figure 13: Hop for WLAN: Hop improves delay for all file sizes with improvements between 3-15×

5.6.2 Multi-hop transfer delay for Web file sizes

Next, we evaluate Hop and TCP over a larger workload that comprises predominantly of micro-blocks. (We do not consider TCP with RTS/CTS enabled, since it consistently introduces more delay.) In particular, we consider a Web traffic pattern where most files are small web pages [5]. The flow sizes used in this experiment were obtained from a HTTP proxy server trace obtained from the IRCache project [12]. The CDF obtained was sampled to obtain the representative flow sizes used in this experiment. The distribution of file sizes is as follows: roughly 63% of the files are less than 10KB, 25% are between 10KB-100KB, and remaining are greater than 100KB. To stress multi-hop performance, the sender and receiver for each flow are chosen randomly among the node-pairs that were multiple hops away in our mesh network. Flows followed a Poisson arrival pattern with $\lambda = 2$ flows per second. We present results from 100 flows aggregated in bins of size $[2^{n-1}, 2^n]$ except the bins at the edge, *i.e.* $\leq$2KB, and $\geq$512.

Figure 14 shows that Hop has less or comparable delay to TCP for almost all file sizes except those between 16K-32K. This dip occurs because 16KB is our threshold for piggybacking data with BSYNs. This suggests that a slightly larger threshold might be more effective, but we leave the optimization for future work. For other

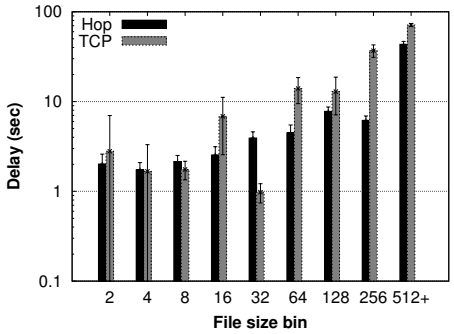


Figure 14: Performance for web traffic: Except the 32KB bin, Hop has comparable or better delay, with gains upto 6×

bins, delay with Hop is mostly lower than TCP (between 19% higher to 6× lower than TCP), demonstrating its benefits for micro-block transfer. Detailed file size microbenchmarks in isolation (i.e., without concurrent transfers) show a similar behavior (detailed in [15]).

5.7 Robustness to partitions

A key strength of Hop is its ability to operate even under disruptions unlike end-to-end protocols such as TCP. We now evaluate how, in a partitioned scenario, Hop compares to hop-by-hop schemes such as DTN2.5 that are designed primarily for disruption-tolerance. In this experiment, we pick a seven hop path and simulate a partition scenario by bringing down the third node and fifth node in succession along the path for one minute each in an alternating manner. Table 4 shows the goodput obtained by Hop averaged over five runs under two different backpressure settings: 1) backpressure limit (H) is set to 1 and 2) backpressure limit is set to 100. Hop outperforms DTN2.5, a protocol specifically designed for partitioned settings, by 2× when $H = 1$, and 3× when $H = 100$. The results show that Hop achieves excellent throughput under partitioned settings, and a large backpressure limit improves throughput by about 15%. This result is intuitive as having a larger threshold enables maximal use of periods of connectivity between nodes. In contrast to Hop, TCP achieves zero throughput since a contemporaneous end-to-end path is never available.

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

Table 4: Goodput achieved by Hop and DTN2.5 in a partitioned network without an end-to-end path.

5.8 Hop with VoIP

In this experiment, we quantify the impact of Hop and TCP on Voice-over-IP (VoIP) traffic. We use two metrics: 1) the mean opinion score (MoS) to evaluate the quality of a voice call, and 2) the conditional loss probability (CLP) to measure loss burstiness. The MoS value can range from 1-5, where above 4 is considered good, and below 3 is considered bad. The MOS score for a VoIP call is estimated as in [6]. The CLP is calculated as the conditional probability that a packet is lost given that the previous packet was also lost.

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

Table 5 shows that Hop achieves significantly better throughput than TCP (in terms of median/mean) but has more impact on the quality of VoIP calls. This is to be expected as TCP starves most of its flows as evidenced by the abysmal median throughput (1-2 Kbps), and therefore has lower impact on the VoIP flow. In contrast, Hop obtains median throughput of a few hundreds of Kbps, while sacrificing a little VoIP quality. We believe that even this discrepancy can be reduced by exploiting 802.11e to set larger contention window parameters to the background queue (e.g. higher backoff), but have not experimented with this so far.

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

Table 5: Impact of Hop and TCP on VoIP flows. Result shows the median/mean goodput, conditional loss probability, and MOS for VoIP with 95% confidence intervals in parentheses.

5.9 Network and link layer dynamics

Our experiments so far were run with static routes and with a fixed wireless bit-rate. Now, we evaluate the impact of dynamic routing using OLSR and auto bit-rate control using the default Madwifi *Sample* algorithm. We run TCP under all four combinations of static/dynamic routes and fixed/auto bit-rate selection. We compare these to Hop with a fixed bit-rate and static/dynamic routes. We are unable to evaluate Hop with auto-rate control as the current implementation of Hop disables link-layer ARQs that auto-rate control requires to estimates link quality. As in §5.3, we consider thirty concurrent long-lived flows between randomly chosen node pairs, and run the experiment five times.

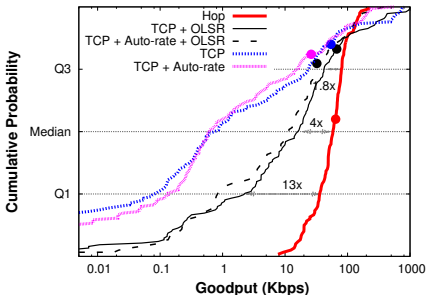


Figure 15: Hop for 30 concurrent flows under dynamic routing and auto bit-rate. Dots on each line shows mean goodput. Median gains by Hop with fixed bit-rate are around 4× over TCP with OLSR and more than 90× over TCP with static routing.

Figure 15 shows that Hop is better than TCP across all combinations, with median gains of 4× over the best of them. (Hop behaves almost identically with dynamic or static routes, therefore we only show the static case in the figure.) Surprisingly, we see that the best combination for TCP is with OLSR and fixed bit-rate. OLSR significantly improves TCP's median goodput or fairness, thereby reducing Hop's gain over TCP in comparison to the static case (§5.3). OLSR benefits TCP as it constantly changes the routing topology with concurrent TCP flows, which makes high goodput flows backoff and yield transmission opportunities to the previously low goodput flows. While the constant shuffling of flows increases TCP's median goodput, OLSR's impact on TCP's mean goodput is small (25%) because the links in the network are already heavily loaded. Auto-rate control makes almost no improvement to TCP since the testbed remains well-connected at 11 Mbps, and hence OLSR choses good links at this bit-rate.

5.10 Hop under 802.11g

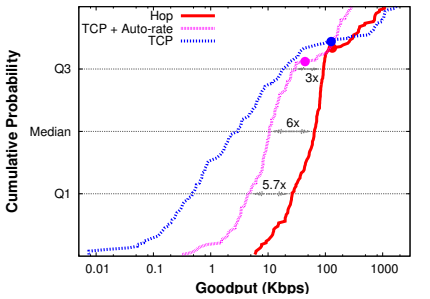


Figure 16: Hop for 30 concurrent flows under 802.11g. Dots on each line shows mean goodput. Hop's median gain is 22× over TCP with bit-rate fixed at 24Mbps, and is 6× over TCP with auto-rate control. Hop's mean gain is 3× over TCP with auto-rate control.

All of our experiments so far were done with 802.11b. How does Hop perform under higher bit-rates obtained using 802.11g? To answer this question, we consider an experiment similar to that in §5.3 with thirty long-lived concurrent flows between randomly chosen node pairs. We use a subset of our testbed (15 nodes) for this experiment as many nodes get disconnected under 802.11g. We ran this experiment with a static routing topology obtained by running OLSR under 802.11g. We consider Hop and TCP with a fixed 802.11g bit-rate of 24 Mbps that yields a reasonably connected topology, as well as TCP with auto-rate control.

Figure 16 shows that Hop improves median goodput by 6× over TCP with auto-rate control and by 22× over TCP with fixed bit-rate. The gains over TCP with auto-rate are lower than in the case of our 802.11b experiments in §5.3 because the maximum bit-rate in 802.11g is higher than the selected fixed bit-rate of 24 Mbps. Thus, TCP with auto-rate control can take advantage of the fact that the maximum bit-rate on 802.11g links is 54 Mbps, whereas Hop's bit-rate is fixed at 24 Mbps. As a result, the highest goodput achieved by a flow that uses TCP with auto-rate control is 23 Mbps, which is higher than Hop's maximum goodput of 16 Mbps. The fact that Hop shows considerable benefits despite using a static best bit-rate suggests that Hop with a good bit-rate selection scheme can benefit even more.

Figure 16 also shows that auto-rate control improves TCP's fairness (median goodput increases by 3.2×) but hurts network utilization (mean goodput decreases by 65%). This is because auto-rate improves the low goodput flows over lossy links by reducing the bit-rate (and thereby the loss rate), but impacts high goodput flows as flows over low bit-rate links are slow and consume a large portion of transmission opportunities.

5.11 Discussion: Hop vs. TCP

Although the above results show Hop's benefits across a wide range of scenarios, our evaluation has some limitations. First, our results are based on a 20-node indoor testbed, so we can not claim that they will hold in other wireless mesh networks. For example, it is conceivable that the benefits due to ack withholding are because of hidden terminals specific to our testbed's topology. Nevertheless, our experience with Hop has been encouraging. Over the last few months, we have experimented with different node placements, static topology configurations, and diurnal as well as seasonal variations in cross traffic and channel conditions, and have seen results consistent with those described in this paper. Second, we have not compared Hop to a large number of proposed TCP modifications for multi-hop wireless networks for which implementations are not available (refer §6.1). We present Hop as a simple and robust alternative to end-to-end rate control schemes, but do not claim that end-to-end rate control can not be fixed to obtain comparable benefits at least in well-connected environments.

TCP's strengths are undeniable. Under high load, it is

difficult to outperform TCP significantly in terms of aggregate throughput (refer Figures 11 and 16). TCP backs off aggressively on bad paths reducing contention for flows on good paths resulting in an efficient but unfair allocation. TCP has a similar effect on hidden terminals—by squelching most of the colliding flows, TCP in effect unfairly serializes them but ensures high throughput. Finally, despite its many woes in wireless environments, TCP enjoys the luxury of experience through widespread deployment, setting a high bar for alternate proposals.

Hop is not designed to be TCP-friendly. For example, in the 30 flow scenario, if we convert just 7 of the 30 TCP flows to use Hop instead of TCP, the median goodput of the remaining 23 drops by an order of magnitude [15]. This is unsurprising as Hop's bursty traffic increases the loss and contention perceived by TCP flows causing them to aggressively back off.

6 Related work

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

6.1 Proposed alternatives to TCP

TCP performance: TCP's drawbacks in wireless networks include its inability to disambiguate between congestion and loss [2], and its negative interactions with the CSMA link layer. Proposed solutions include: 1) end-to-end approaches that try to distinguish between the different loss events [25], attempt to estimate the rate to recover quickly after a loss event [19], or reduce TCP congestion window increments to be fractional [21], 2) network-assisted approaches that utilize feedback from intermediate nodes, either for ECN notification [38], failure notification [17] or for rate estimation [32], and 3) link-layer solutions that use a fixed window TCP in conjunction with link-layer techniques such as neighborhood-based Random Early Detection ([9]) or backpressure flow control (RAIN [16]) to prevent losses due to link queues filling up.

TCP fairness: TCP unfairness over 802.11 stems primarily from: 1) excess time spent in TCP slow-start, which is addressed in [32] by use of better rate estimation, and 2) interactions between spatially proximate interfering flows [37, 29] by using neighborhood-based random early detection and rate control techniques.

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

6.2 Implemented alternatives to TCP

Few implemented alternatives to TCP are available for reliable transport in 802.11 networks today. At the time of writing, we found only two such implementations—TCP Westwood+ and DTN2.5—both of which we compare against Hop. Hop's use of hop-by-hop reliability and backpressure is similar to a recent proposal, CXCC [31], but differs in its use of burst-mode, ack withholding, virtual retransmissions, etc. We could not compare Hop against CXCC as it is not implemented for 802.11.

Two recent systems, WCP [30] and Horizon [27], also address TCP's performance and fairness problems over 802.11. WCP, similar in spirit to NRED [37], augments TCP's end-to-end rate control with network-assisted feedback about contention along the path. WCP shows significant gains in median throughput (or fairness) under load, but often reduces the mean throughput considerably. Horizon uses backpressure scheduling with multi-path routing as a shim between unmodified TCP and 802.11 layers, and shows improved fairness under load in a majority of experimental runs at the cost of mean throughput. In comparison, Hop consistently shows significant improvement in fairness and mild improvement in mean throughput under load. Although we have not performed a head-to-head comparison to Hop, we note that both WCP and Horizon rely on link-layer ARQ per frame that our experiments (Figures 7 and 10) suggest are inefficient for lossy wireless links.

6.3 Other related work

Backpressure: Backpressure was first investigated in ATM [24] and high-speed networks [20] to handle data bursts. A seminal paper by Tassiulas and Ephremides [33] showed that backpressure scheduling can achieve the stable capacity region of a wireless network. This paper sparked off a large body of theoretical work [34] on optimal scheduling, routing, and flow control in wireless networks. However, backpressure scheduling is NP-hard, incurs a high signaling overhead per transmission, and is difficult to implement with the 802.11 MAC layer, so few practical implementations exist.

In recent times, backpressure-like ideas have been adapted for congestion control as an alternative to TCP [31] or underneath TCP [16, 27]; for unreliable hierarchical data aggregation in sensor networks [11]; for reliable bulk transport in linear sensor networks and a single flow [14], etc. In comparison, Hop performs backpressure over blocks to amortize the signaling overhead, uses ack withholding to to alleviate hidden terminal losses, and uses per-hop reliability with virtual retransmissions

to efficiently deal with in-network losses.

Batching: Ng et al. [22] show that adapting the burst size of txop's in 802.11e to the load can improve TCP fairness in WLAN settings. WildNet [26] leverages batching with FEC and bulk acknowledgments at the link layer over long-distance unidirectional 802.11 links. Kim et al. [35] aggregate TCP frames using the 802.11n burst mode to amortize the MAC protocol overhead. In comparison, Hop jointly leverages batching both at the link and transport layers.

7 Conclusions

The last decade has seen a huge body of research on TCP's problems over wireless networks, but TCP for good reasons continues to to be the dominant real-world alternative today. One reason may be that TCP is good enough in the common case of wireless LANs, and solutions proposed for more challenged environments do not perform well in the common case. A natural question is if we can have one simple transport protocol that yields robust performance across diverse networks such as WLANs, meshes, MANETs, sensornets, and DTNs. Our work on Hop suggests that this goal is achievable. Hop achieves significant throughput, fairness, and delay gains both in well-connected WLANs and mesh networks as well as disruption-prone networks.

References

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

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

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

[4] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *MobiCom*, 2005.

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

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

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

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

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

[10] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. In *SIGOPS*, 2008.

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

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

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

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

[15] M. Li, D. Agrawal, D. Ganesan, and A. Venkataramani. Block-switched networks: A new paradigm for wireless transport. Technical Report TR UM-CS-2008-27, UMass Amherst, 2008.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] L. Tassiulas, and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *IEEE Transactions on Automatic Control*, 37(12), 1992.

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

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

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

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

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

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