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

Technical Report #UM-CS-2006-27

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 [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 [38]. Although many solutions (e.g. [19, 33, 39]) 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 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.

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) DTN 2.5, a delay tolerant transport protocol [10], and 3) Stitched-TCP, a hop-by-hop TCP,

- 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 7.4× and 3× showing that Hop degrades gracefully.
- Under high load, Hop achieves more than an order of magnitude benefit in median goodput (e.g., 39× 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 meshes and WLANs as well as disruption-tolerant networks. Hop coexists 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 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 routing [5], network coding [16], and concurrent transmissions [37], all of which have significant control overhead that needs to be 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 (in §5.1.1, we show that 802.11 ARQ has roughly 35% overhead), and 3) reduce overall throughput by disproportionately using the channel 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-

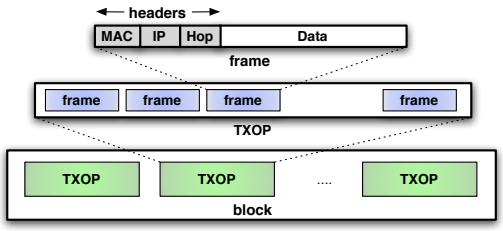


Figure 1: Structure of a block.

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: The use of hop-by-hop reliable block transfer also enables us to exploit caching at intermediate hops. Caching can help prevent wasted work by 1) enabling more efficient retransmissions by exploiting caching 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.

3 Design

This section describes the Hop protocol in detail. Hop's design consists of six main components: 1) reliable per-hop transfer, 2) virtual retransmissions for end-to-end reliability, 3) backpressure flow control, 4) handling 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.

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

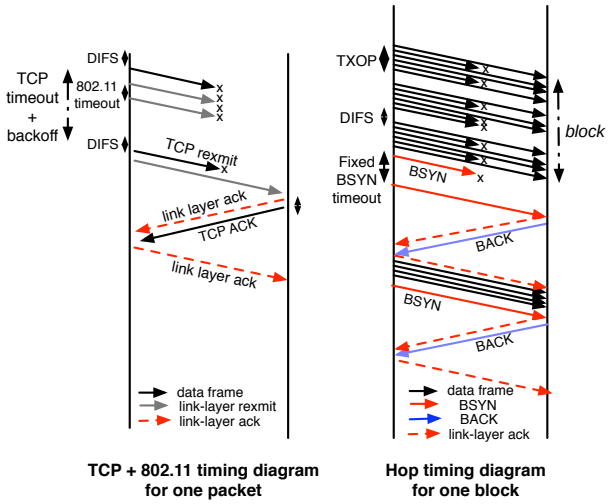


Figure 2: Timeline of TCP/802.11 vs. Hop

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 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 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 in secondary storage. 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 overhead packets transmitted on the old path. Hence, instead of retransmitting the entire block, the sender sends

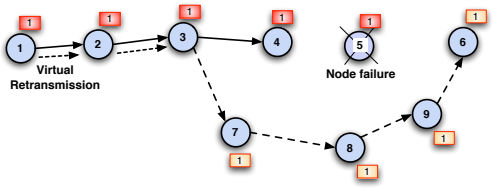


Figure 3: Virtual retransmission due to node failure.

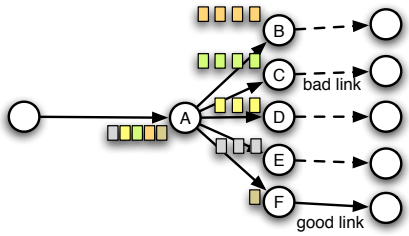


Figure 4: Example showing need for backpressure. Without backpressure, Node A would allocate $1/5$th of out-going capacity to each flow, resulting in queues increasing unbounded at nodes B through E. With backpressure, most data is sent to node F, thereby increasing utilization.

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.

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 and improve utilization. In wireless networks, congestion collapse can occur both due to increased packet loss due to contention [12], and increased loss due to buffer drops [11]. 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, 33].

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 flow control 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 equalizing the number of blocks buffered at the next-hops across all flows. Indeed, classical backpressure algorithms known to achieve optimal throughput (refer a seminal paper by Tassiulas [34]) 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 the next-hop to use. Unlike typical mesh routing protocols [26, 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 [9]). 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 [38]. Thus, TCP is highly unfair but has good aggregate throughput.

Hop uses a novel *ack witholding* 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 witholding 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 [37]. In contrast, ack witholding requires no additional control packets (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 two ways: (a) the initial BSYN/BACK exchange increases delay for micro-blocks, (b) a sender might be servicing multiple flows, in which case a micro-block may need to wait for multiple txops, and (b) ack-witholding 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 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-witholding delay. Second, the packet scheduler at the sender prioritizes micro-blocks over larger blocks. Finally, we use a block-size based ack-witholding policy, where smaller blocks are more likely to be acknowledged over larger blocks (with probability inversely proportional to block size). This lets micro-blocks propagate through the network faster than larger ones.

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[1].

4.1 MAC parameters

Our implementation uses the Atheros-based wireless chipset and the Madwifi open source 802.11 device driver [21], 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 The size of a Hop block is limited by default to 1MB. (This limit was chosen because much

[1] A more detailed draft RFC including a full description of the protocol state machine, header format, etc is available at `http://www.cs.umass.edu/~mingli/hop/`

smaller values lose out on batching benefits.) 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 small files) 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. An intermediate Hop node keeps all the blocks that it has received in secondary storage, and keeps the blocks being actively exchanged cached in memory. To prevent storage capacity from being exhausted, a default expiry time of one hour is used for cached blocks after which the block is discarded.

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, BSYN, and BACK frames contain the `round_num` that is incremented every round of the per-hop BSYN/BACK protocol. 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: Hop as described thus far described the per-hop protocol and mechanisms for end-to-end reliability. 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 802.11b/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 run in 802.11b mode with bit-rate locked at 11 Mbps. (We discuss results with 802.11g briefly in §5.9). 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. The experiments were performed over roughly a three month period, and the total running time of all experiments reported in this paper is almost a month.

Figure 5: Experimental testbed with dots representing nodes.

We compare Hop against two classes of protocols: *end-to-end* and *hop-by-hop*. The former consists of 1) default TCP implementation in Linux 2.6 with CUBIC congestion control [41] and FRTO [42] enabled (for detecting spurious retransmissions)[2]; and 2) UDP. The latter consists of 3) Stitched-TCP [18], and 4) DTN2.5 [10]. DTN2.5 is a reference implementation of the IEEE RFC 4838 and 5050 from the Delay Tolerant Networking Research Group [10] that reliably transfers a bundle using TCP at each hop. Stitched-TCP splits an end-to-end TCP connection into hop-by-hop TCP segments [18] and operates over packets. We did not find a reference implementation of Stitched-TCP, so we implemented our own version. No mechanism for flow- or rate-control is used between the TCP segments in either DTN2.5 or Stitched-TCP. 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?

[2]We did not use the TCP Westwood+ congestion control optimization [22] since it performed consistently worse under a range of wireless topologies (by roughly 10%).

Sec.	Experiment setup	Experiment	Result: Median (Mean)
§5.1	One single-hop flow	Hop vs. TCP	1.6× (1.8×)
§5.2	One multi-hop flow	Hop vs. TCP Hop vs. Stitched-TCP Hop vs. DTN2.5	2.3× (2×) 2.55× (2.02×) 2.92× (3.9×)
§5.3	Many multi-hop flows	Hop vs. TCP	39.6× (1.13×)
§5.4	Performance breakdown	Base Hop + only Ack withholding + Ack witholding + Backpressure	1× (2.3×) (2.77×)
§5.5	WLAN AP mode	Hop vs. TCP Hop vs. TCP + RTS/CTS	2.67× (1.12×) 1.95× (1.43×)
§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× (2.87×)
§5.8	Impact on VoIP traffic	Hop vs TCP/*/*	Marginally lower MoS score but significantly higher throughput

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

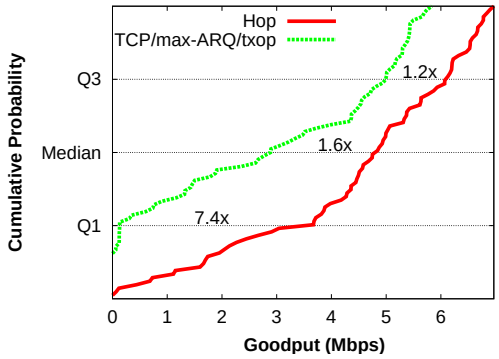


Figure 6: Experiment with one-hop flows. Hop improves lower quartile goodput by 7.4×, median goodput by 1.6×, and mean goodput by 1.8×.

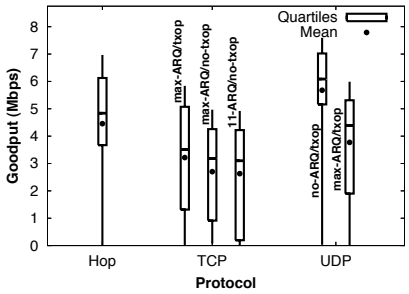


Figure 7: Experiment with one-hop flows. Box shows lower/median/upper quartile, lines show max/min, and dot shows mean. Increasing 802.11 ARQ limit and using txops helps TCP but Hop is still considerably better. UDP results show that ARQs incur significant performance overhead (35%). Hop is within 24%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

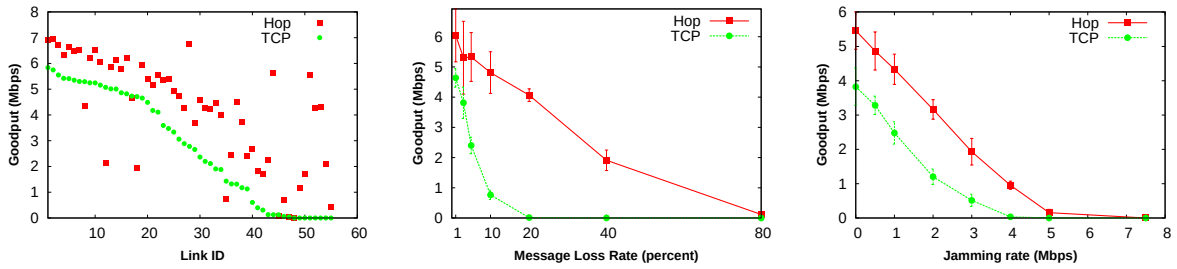
5.1.1 Randomly picked links

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 10mins, 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 Appendix A).

Figure 6(a) 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 7.4×, 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



(a) Sorted Single-hop Flows (b) Impact of Loss. (c) Impact of Interference

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

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.

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

5.1.2 Graceful performance degradation

A key benefit of Hop is its 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 links in the testbed where TCP's throughput 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 conduct two additional experiments. First, we evaluate the impact of loss rate observed 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 degrades much more gracefully and is operational until the loss rate is about 80%.

Second, we evaluate the impact of interference. We start with a mediocre link that has 28% loss rate, and introduce external interference by using an 802.11 jamming node. The jammer sends 1.5KB UDP packets at a fixed rate. Carrier sense is disabled at the jammer to ensure the fixed rate. The jammer was placed such that 1) the transmitter could not detect its signal (i.e. the three nodes formed a hidden terminal), and 2) there was no capture effect at the receiver. Figure 8(c) shows that TCP degrades faster than Hop under jamming. The gains over TCP increase from 1.5× when there is no jamming to 2.6× when the jamming rate increases to 2 Mbps, to 29× at 4 Mbps jamming rate.

5.2 Multi-hop microbenchmarks

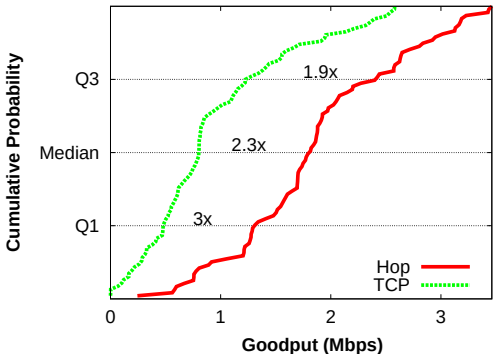


Figure 9: Experiment with multi-hop flows. Hop improves lower quartile goodput by 3×, median goodput by 2.3×, and mean goodput by 2×.

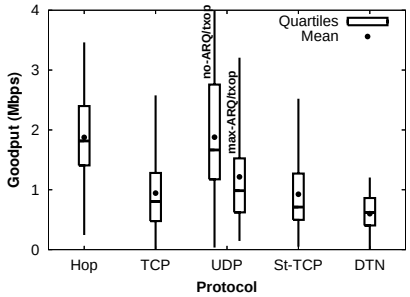


Figure 10: Boxplot of multi-hop single-flow benchmarks. Hop has 2-3× median, and 2-4× mean improvements over other reliable transport protocols. Hop is comparable to UDP/no-ARQ/txop in terms of median/mean — the latter is extremely fast since it has no overhead, but experiences more loss.

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 until the routing topology stabilized and then used to configure routing tables at the beginning of the experiment. We compare the multi-hop goodput of Hop to TCP, Stitched-TCP, DTN2.5, and UDP.

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 TCP—the Q1, Q2, and Q3 gains are 3×, 2.3× and 1.9× respectively. We also find that hop-by-hop TCP variants such as DTN2.5 and Stitched-TCP perform much worse than TCP as they are impacted more by self-interference. 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 . For example, the lower quartile gains grow from about 2.7× for two hops to more than 4× for five and six hops.(details in Appendix B)

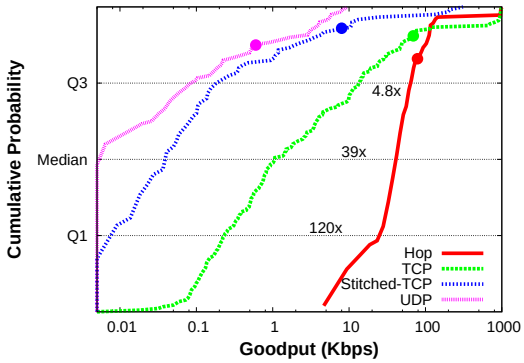


Figure 11: Hop for 30 concurrent flows. Dots on each line shows mean goodput. Median gains by Hop are massive (e.g. 39× better than TCP), mean gains are less (13%) since TCP has very unfair allocation. Stitched TCP and UDP perform very poorly.

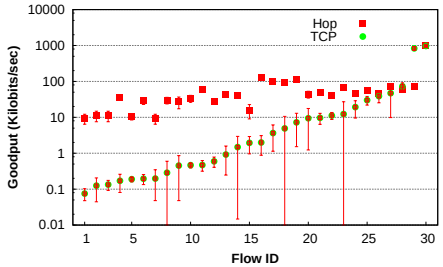


Figure 12: Hop for 30 concurrent flows: Per-flow throughput aggregates across five runs. Plot shows that Hop rate allocation is considerably fairer than TCP.

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. The experiment consists of thirty concurrent flows that transfer data continually between randomly chosen node pairs in the testbed. All protocols are run over the OLSR routing protocol. To focus on multihop benefits, we pick src-dst pairs 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 the CDF of the goodputs across all runs of the experiment (with the y-axis is on a log scale). Hop achieves a median goodput of 42.9 Kbps whereas all the other protocols achieve less than 1.6 Kbps—an improvement of over an order of magnitude! The Q1 gain over TCP is more than two orders of magnitude, and upper quartile gain is 6×. UDP, Stitched-TCP suffer more than TCP because of the lack of congestion control, which reduces overall utilization in a heavily loaded network.

Hop's mean gain over TCP is just 13%, 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. To illustrate this point, Figure 12 shows the average per-flow goodput of the 30 flows across all runs, sorted in descending order of TCP goodput (with the y-axis on a log scale) The results show that TCP is highly unfair — the top 3 out of 30 flows get 89% of the total goodput. In contrast, Hop is significantly fairer and has higher throughput than TCP for 27 of 30 flows.

	Fairness Index
Hop	0.24 (0.07)
TCP	0.1 (0.01)
Stitched-TCP	0.06 (0.03)

Table 2: Fairness index for 30 flow experiment. 95% confidence in parenthesis

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 [29]). When there are n flows, x_1 through x_n, with 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, although far from perfect, is significantly fairer than all other protocols. It is noteworthy that while TCP sacrifices fairness for goodput, Hop is superior on both metrics. The results show that TCP, Stitched-TCP are all

much more unfair than Hop. Stitched-TCP have similar fairness problems, as they lack a congestion control mechanism.

5.4 Hop performance breakdown

How much do components of Hop individually contribute to its overall performance? To answer this question, we compare three versions of Hop: 1) the basic Hop protocol that only uses hop-by-hop block transfer, 2) Hop with only ack witholding turned on, and 3) Hop with both 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 concurrent flows between randomly picked sender-receiver node pairs. Each flow transmits a large amount of data, and we take a snapshot of the measurements after an hour.

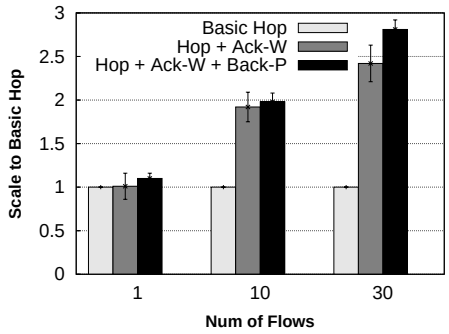


Figure 13: Hop performance breakdown showing contribution of Ack Witholding, and Backpressure.

Figure 13 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 1.5× 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 1.2× 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.

5.5 Hop in Access Point Networks

Next, we evaluate how the ack witholding 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 mins when the flow rates had 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 more than 2.6×. As before TCP has a crafty way of maintaining high aggregate goodput amidst hidden terminals by squelching all but one or two flows. 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 for micro-block 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) Hop piggybacks micro-blocks less than 16KB in size with the initial BSYN to reduce connection setup overhead, 2) Hop's ack witholding mechanism prioritizes micro-blocks over large ones.

5.6.1 Delay in many-to-one topology

First, we evaluate the benefits of Hop's size-aware ack witholding 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 continously 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 14 shows that the delay observed by the micro-block for Hop is always lower than for TCP (with or

without RTS/CTS). In many cases, the delay gains are significant, for example, for file sizes less than 16 KB, 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 a many-to-one AP setting.

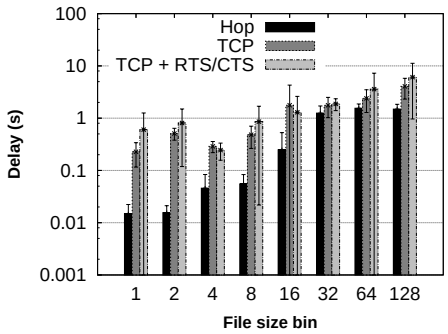


Figure 14: Hop for WLAN: Hop improves delay for all file sizes with improvements between 3-15×

5.6.2 Hop for concurrent micro-block transfers

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 almost always has introduces more delay.) In particular, we consider a Web traffic pattern where most files are small web pages [6]. 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 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.

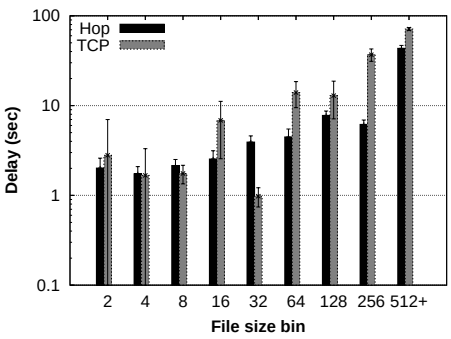


Figure 15: Performance for web traffic: Except the 32KB bin, Hop has comparable or better delay, with gains upto 6×

Figure 15 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 optimal. For other 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, but are deferred to Appendix C for lack of space.

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 2x when $H = 1$, and 3x 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 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, e is the loss rate, and $H(.)$ is the heavyside step function. The coding delay is 25 ms, the jitter buffer delay is 60 ms, and the delay over the wired segment of the end-to-end path is 30ms. The CLP is calculated as the conditional probability that a packet is lost given that the previous packet was also lost.

A VoIP flow is generated as a stream of 20 byte packets every 20 ms, which are transmitted over UDP. We run two experiments, one with five flows five Hop/TCP flows, and the second with ten flows. Each flow which transmits over three hops, and we measure one VoIP flow transmitting over three hops in this experiment. The 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 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: Interaction between VoIP and Hop/TCP flows. Result shows the median/mean goodput, conditional loss probability, and Mean opinion score for VoIP. 95% confidence is shown in parenthesis.

5.9 Other Results

Due to space constraints, we briefly summarize results from other experiments that we performed (details available in Appendix): 1) experiments with 802.11g show consistent gains for Hop - for example, we obtain aggregate and median improvements of 1.4x for single-hop flows over 28 links on our testbed(Appendix D), 2) experiments showing Hop's performance for wired + wireless communication show aggregate goodput gains of more than 60%(Appendix E), and 3) experiments with a variant of the ack witholding scheme to address multi-receiver hidden terminals in addition to single-receiver hidden terminals showed no gains in our testbed(Appendix F).

5.10 Hop's limitations and TCP's strengths

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, Hop significantly outperforms existing alternatives, but how close is it to optimal? This is a difficult and broad research question that we do not attempt to address here. Third, there is an enormous number of proposals for reliable wireless transport for which implementations are not available. We were able to implement and compare against only a few of the proposed schemes.

TCP's strengths are undeniable. Under high load, it is difficult to outperform TCP significantly in terms of aggregate throughput, and we were humbled in our attempts to do so. TCP backs off aggressively on bad paths reducing contention for flows on good paths resulting in an efficient but unfair allocation. In hidden terminal scenarios, TCP ingeniously squelches all but one flows effectively serializing them but ensuring high utilization. Finally, despite its many woes in wireless environments, TCP enjoys the luxury of experience through widespread deployment, setting a high bar for alternate proposals.

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 schemes and related systems

TCP in wireless: TCP's drawbacks in wireless networks include its inability to disambiguate between congestion and loss [2], and its negative interactions with the routing layer and CSMA link layer. Proposed solutions include: 1) end-to-end approaches that try to distinguish between the different loss events [27, 40], attempt to estimate the rate to recover quickly after a loss event [22], or reduce TCP congestion window increments to be frac-

tional [24], 2) network-assisted approaches that utilize feedback from intermediate nodes, either for ECN notification [39], failure notification [20] or rate estimation [33], and 3) 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 [19]) to prevent losses due to link queues filling up. Hop avoids reliance on fragile end-to-end rate control. 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.

Transport Fairness: TCP unfairness over 802.11 stems from primarily from: 1) excess time spent in TCP slow-start, which is addressed in [33] by use of better rate estimation, and 2) interactions between spatially proximate interfering flows that may not traverse a single link, which is addressed in [38] and [30, 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 ack withholding—to balance fairness and throughput.

Bulk transfer: Hop shares similarities with bulk transfer protocols such as NETBLT for wired networks [7] that transmit large buffers in burst mode. However, NETBLT lacked congestion control as it was designed before Jacobson's work on TCP. Bulk transfer in sensor networks is addressed in Flush [17], which uses hop-by-hop rate control to adjust flow rate at each hop based on observed intra-flow interference. 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, in particular the ability to send packets in burst using txop or Transmission Opportunity, and the availability of link-layer prioritization. Much work has evaluated the benefits of 802.11e enhancements on channel utilization (e.g.: [35]), however its interaction with transport-layer protocols is not well understood. One excepton of [25] which explores how 802.11e prioritization impacts TCP fairness. We show that 802.11e enhancements have limited impact on TCP throughput since TCP is not designed to exploit them. In contrast, Hop leverages these enhancements to dramatically improve throughput.

Backpressure: Hop by hop flow control or backpressure was first used in ATM and high-speed networks to handle data bursts (e.g. [23]) with foundational work by Tassiulas and others [34]. More recently, backpresssure has been used for congestion adaptation in wireless and sensor networks (RAIN [19], Fusion [12]). While Hop is similar to these techniques, it uses block-level rather than packet-level backpressure for greater efficiency, and incorporates mechanisms such as ack witholding and virtual retransmissions for dealing with contention and loss.

Hop's use of hop-by-hop reliability and backpressure is similar to a recent proposal by Scheurmann et al. [32] implemented on a custom hardware platform, but differs in its use of burst-mode, ack withholding, virtual retransmissions, etc. MIXIT [15], a radically different network architecture implemented on a software radio platform, enables end-to-end reliable transport. However, MIXIT as implemented does not do congestion control ("congestion-aware forwarding" in MIXIT does not reduce the load injected into the network). Congestion control is critical for high utilization under load with or without end-to-end reliability (refer §5.3). We could not compare Hop against the above two proposals as Hop is implemented on commodity 802.11 hardware.

Batching: Hop relies heavily on batching of packets for its throughput gains. Several recent network and link layer protocols such as ExOR [5], MORE [16], CMAP [37], and WildNet [28] 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 alternatives to TCP/802.11

Few implemented alternatives to TCP are available for reliable transport in 802.11 mesh networks today. In fact, we found only two such implementations—TCP Westwood+ and DTN2.5—both of which we evaluate. We believe that the paucity of available implementations is because it is difficult to demonstrate consistent gains over TCP across a range of network conditions, especially under high load. Hop consistently improves aggregate throughput (mildly) as well as fairness (significantly) and delay (moderately) compared to TCP. A very recent proposal, WCP, improves fairness under light load compared to TCP but does so by reducing aggregate throughput (refer [31]). We have recently acquired a pre-release implementation of WCP from the authors, comparing against which is part of ongoing work.

7 Conclusions

The last decade has seen a huge body of research on TCP's problems over wireless networks, but TCP for good reason 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 solu-

tions proposed for more challenged environments do not perform well in the common case. Can we have a single 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 performance, fairness, and delay gains both in well-connected mesh networks and disruption-tolerant networks.

Hop opens up new directions, primarily because it is simple and more predictable than the combination of TCP and 802.11. One direction is to design a conflict-map based Hop that offers a complete transport-layer and link-layer alternative to TCP over CSMA. By eliminating CSMA overhead, Hop's performance can be improved even further. A second direction is showing that Hop can be used in mobile networks (MANETs) and sensor networks in addition to mesh networks and DTNs. We plan to evaluate Hop over a real partitionable network environment, DieselNet [36], a mobile bus-based testbed. The disruption-tolerance aspect of Hop is also useful for energy-efficiency since it can be adapted for use over duty-cycled sensor networks. Hop reduces inter-node contention and transfers blocks fast, thereby reducing the time for which the radio needs to be active. We plan to design an energy-aware Hop in TinyOS for Motes. 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/`.

APPENDIX

A Single-hop microbenchmarks

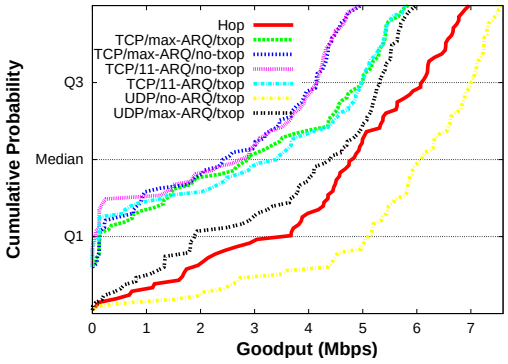


Figure 16: experiment with one-hop flow

In this section We conduct a microbenchmark to evaluate the performance of Hop, TCP and UDP over a single hop. We evaluate each protocol with different link layer settings to find out the best combination. For TCP we consider four variants: i) TCP with max ARQ and 8000 TXOP, ii) TCP with max ARQ and no TXOP, iii) TCP with 11 ARQ and 8000 TXOP, iv) TCP with 11 ARQ and no TXOP. For UDP we consider two variants: i) UDP with zero ARQ and 8000 TXOP, ii) UDP with max ARQ and 8000 TXOP. For Hop we only consider Hop with no ARQ and 8000 TXOP. We randomly chose 100 links with repetition from about 59 active links in our testbed and transferred a 10 MB file over each of them. Figure 16 shows the cumulative distributions(CDFs) of the goodput. As we can see from the graph, UDP/0 ARQ/8000 TXOP achieves the highest goodput that is close to the link capacity. Hop/0 ARQ/8000 TXOP achieves goodput close to UDP and is much higher than the TCP variants. The graph shows that Hop is especially robust on lossy links. For example Hop's first quartile goodput is 7.4x of the best TCP variant. It is interesting to see the four TCP variants are clustered into two groups: TCP/11 ARQ/8000 TXOP and TCP/max ARQ/8000 TXOP are very close to each other, the other group consists of TCP/11 ARQ/0 TXOP and TCP/max ARQ/0 TXOP. This suggests that merely increasing the number of link layer retries does not increase TCP's goodput.

B Multi-hop microbenchmarks

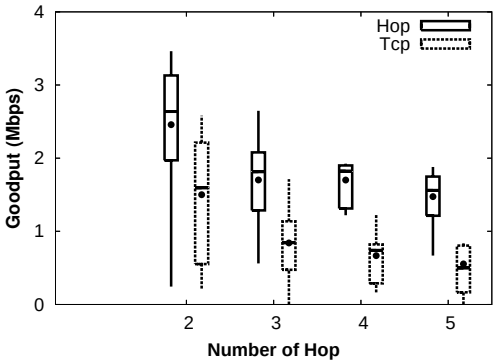


Figure 17: Boxplot of multi-hop single-flow benchmarks.

In Section 5.2 we evaluate Hop, UDP and TCP over multi-hop path. The aggregated results are presented in Figure 910. Now we show a detailed evaluation of Hop and Tcp as the number of hops increases. We sort the results by path length. Figure 17 shows the sorted results. As we can see both Hop's and Tcp's goodput decreases with increasing number of hops, but Hop degrades much slower than Tcp. Hop's median gain over Tcp grows from 1.7× for two hops to 3× for five hops, and Hop's Q1 gain grows from 3.6× for two hops to 7.3× for five hops. Thus, Hop is more robust to increasing number of hops than Tcp.

C File size microbenchmarks

In this section we evaluate Hop and Tcp's performance in transferring files of different sizes. We pick an average link in our testbed and send different files of different sizes using Hop and TCP. The x-axis of Figure 18 shows the data sizes used in the experiment. For TCP both the

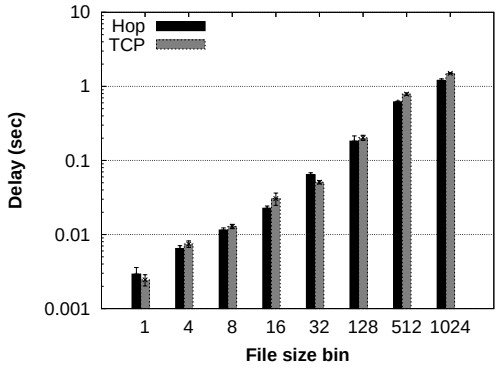


Figure 18: Single flow single-hop file size microbenchmark. The y-axis is in log-scale. TCP's connection build-up time is removed from its transmission delay.

ARQ and the TXOP are set to maximum. We also remove the connection build-up time(as measured by the time taken by the socket *connect call) from TCP's transmission delay to make a fair comparison. Hop's BSYN batching threshold is set at 16KB, i.e. data smaller than 16KB are sent together with BSYN with link layer retransmissions.*

Figure 18 shows the mean delay versus the file size. As we can see Hop has lower delay over all file sizes except 1KB and 32KB. For instance, Hop's delay is 25 percent lower than TCP's at 16KB and 20 percent lower at 512KB. TCP's long delay is due to its slow start policy. At the first round TCP only sends about 2KB of data to probe the network. If the 2KB data is acked promptly 4KB will be sent in next round. Thus when sending data larger than 2KB TCP needs at least two rounds. On the other hand Hop will send all the data in one round with BSYN if the data is smaller than the batching threshold. Therefore Hop has lower transmission delay. The performance degradation of Hop at 32KB suggests that a larger BSYN batching threshold like 32KB might help.

D Hop over 802.11g

Protocol	Hop	UDP	TCP	Split-TCP
Goodput(Kbps)	206(15)	85(63)	164(16)	93(104)

Table 6: Microbenchmark over 802.11g. All protocols except Hop use maximum ARQ and 8000 TXOP. Results show mean goodput. 95% confidence is shown in parenthesis.

Till now all the experiments are run over 802.11b network where the highest data rate is 11Mbps. Can Hop scale up to even higher data rate? In this section we evaluate Hop's performance over 802.11g network where the highest data rate is 54Mbps, 4× of in 802.11b network. We picked a 6-hop path from the testbed and put all the nodes along the path in 802.11g mode with auto-rate control enabled. Ten 10MB files are transmitted through the path simultaneously, which stresses Hop even further.

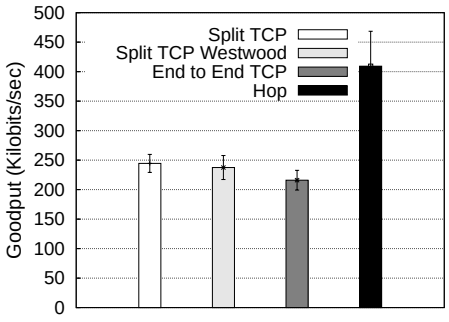


Figure 19: Mean goodput of Hopwhen downloading a file from the Internet, over a 5 hop wireless path, compared to three other TCP variants.

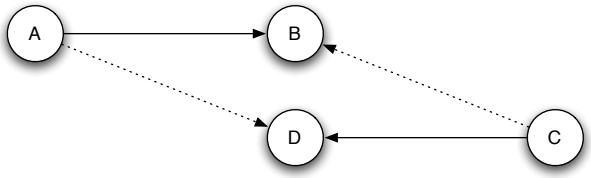


Figure 20: A multi-receiver hidden terminal case. Node A sends to node B. Node C sends to node D. Node A and node C can not hear each other. Node D can hear node A. Node B can hear node C.

We evaluate four protocols: Hop, UDP, TCP and Split-TCP. All protocol except Hop use maximum ARQ and TXOP. Table 6 shows the mean goodputs. Hop achieves the highest goodput, 1.25× of Tcp's, and 2.4× of UDP's. This shows that Hop is scalable to higher data rate.

E Hop for Internet Download

Next, we evaluate Hop's performance when downloading a file from the Internet to a mesh node. In this experiment, three nodes download a 10MB file from a web server at MIT. Each node is five hops away from a gateway node that was configured as the Hopto TCP proxy.

We compare Hopagainst 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. Figure 19 shows that Hopachieves roughly 1.6× improvement over other schemes. We believe that a kernel implementation of Hopand a more efficient Hop-TCP proxy implementation can further improve our benefits.

F Combating Hidden Terminals in the Multi-receiver Case

In Section 3.5 we present an ack-withholding scheme that solves the hidden terminal problem when multiple hid-

den senders send to the same receiver. However, hidden terminal can also happen when multiple hidden senders send to different receivers. Consider the case shown in Figure 20 where node A sends to node B and node C sends to node D. Node A and C are hidden to each other. Node B can hear node C, and node D and hear node A. Then the two transmissions will collide at the receivers. The ack-withholding scheme presented in Section 3.5 can not handle this case since node B can not withhold node C's BACK, neither can node D withhold node A's BACK. To solve this problem we extend the existing ack-withholding scheme to a promiscuous ack-withholding scheme as described below.

With the promiscuous ack-withholding scheme each node monitors all on-going transmissions in its reception range. This is achieved by putting the node in promiscuous listening mode in which each node sniffs all ongoing transmission from the channel. Each node maintains a list of on-going flows in its reception range. Each time it overhears a flow's data packets it inserts that flow into the list. A sender node sends an End-of-Block(EOB) message in the end of a block transmission as the sign of the end of the block. A node overhearing an EOB message checks its on-going flow list and removes the flow matching the EOB message. Each time a node needs to release a BACK it checks the on-going flow list, and withholds the BACK if the list is not empty.

We evaluate the performance of the promiscuous ack-withholding scheme in a multi-flow experiment. We randomly picked 30 pairs of nodes from the testbed. Each pair of nodes are at least 3 hops away from each other. Thirty 10MB files are transmitted by Hopconcurrently, one file on each pair of nodes. We run the experiment with both the single-receiver ack-withholding scheme and the promiscuous ack-withholding scheme. The results show that the promiscuous ack-withholding scheme message loss rate is 30 percent lower than the non-promiscuous scheme. Thus the promiscuous scheme does decrease message collisions. But the goodputs of the two schemes are roughly the same. This might be because of that the new scheme is too conservative and backs off too much. In certain cases multiple hidden transmissions can go simultaneously without hurting goodput. For instance, if the node placement is changed in Figure 20 so that node D can not hear node A and node A's signal strength on node B is much higher than node C's, node A's signal might be able to capture node C's signal. Then the two transmissions can go simultaneously. In the light of this observation, we modify the ack-withholding policy to that a node does not withhold a sender's BACK if the sender's signal strength(RSSI) is XdBm higher than any sender in the on-going traffic list. We tried $X = 10$ but did not see any improvement in goodput. In future we will explore the best tradeoff that minimizes collisions without hurting goodput too much.

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

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

[7] *David L. Clark, Mark M. Lambert, and Lixia Zhang. RFC 998: Netblt: A bulk data transfer protocol, March 1987.*

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

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

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

[11] *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.*

[12] *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.*

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

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

[15] *S. Katti, D. Katabi, H. Balakrishnan, and M. Medard. Symbol-level network coding for wireless mesh networks.* SIGCOMM, *2008.*

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

[17] *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.*

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

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

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

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

[22] *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.*

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

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

[25] *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.*

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

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

[28] *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.*

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

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

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

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

[33] 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.*

[34] L. Tassiulas. Adaptive back-pressure congestion control based on local information. In IEEE Transactions on Automatic Control*, Feb 1995.*

[35] I.; Sunghyun Choi Tinnirello. Efficiency analysis of burst transmissions with block ack in contention-based 802.11e wlans. ICC*, 2005.*

[36] UMassDieselNet: A Bus-based Disruption Tolerant Network. http://prisms.cs.umass.edu/diesel/.

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

[38] Kaixin Xu, Mario Gerla, Lantao Qi, and Yantai Shu. Enhancing tcp fairness in ad hoc wireless networks using neighborhood red. In MobiCom*, 2003.*

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

[40] B.; Xiaoqiao Meng; Songwu Lu Zhenghua Fu; Greenstein. Design and implementation of a tcp-friendly transport protocol for ad hoc wireless networks. Proceedings of ICNP*, pages 216–225, 12-15 Nov. 2002.*

[41] S. Ha and I. Rhee and L. Xu CUBIC: a new TCP-friendly high-speed TCP variant Proceedings of SIGOPS*, pages 64–74 Nov. 2008.*

[42] P. Sarolahti and M. Kojo and K. Raatikainen F-RTO: an enhanced recovery algorithm for TCP retransmission timeouts SIGCOMM Comput. Commun. Rev.*, pages 51-63 2003.*