TCP ex Machina: Computer-Generated Congestion Control

Keith Winstein and Hari Balakrishnan
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, Mass.
{keithw, hari}@mit.edu

ABSTRACT

This paper describes a new approach to end-to-end congestion control on a multi-user network. Rather than manually formulate each endpoint's reaction to congestion signals, as in traditional protocols, we developed a program called Remy that generates congestion-control algorithms to run at the endpoints.

In this approach, the protocol designer specifies their prior knowledge or assumptions about the network and an objective that the algorithm will try to achieve, e.g., high throughput and low queueing delay. Remy then produces a distributed algorithm—the control rules for the independent endpoints—that tries to achieve this objective.

In simulations with ns-2, Remy-generated algorithms outperformed human-designed end-to-end techniques, including TCP Cubic, Compound, and Vegas. In many cases, Remy's algorithms also outperformed methods that require intrusive in-network changes, including XCP and Cubic-over-sfqCoDel (stochastic fair queueing with CoDel for active queue management).

Remy can generate algorithms both for networks where some parameters are known tightly *a priori*, e.g. datacenters, and for networks where prior knowledge is less precise, such as cellular networks. We characterize the sensitivity of the resulting performance to the specificity of the prior knowledge, and the consequences when real-world conditions contradict the assumptions supplied at design-time.

CATEGORIES AND SUBJECT DESCRIPTORS

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design — Network communications

KEYWORDS

congestion control, computer-generated algorithms

1. INTRODUCTION

Is it possible for a computer to "discover" the right rules for congestion control in heterogeneous and dynamic networks? Should computers, rather than humans, be tasked with developing congestion control methods? And just how well can we make computers perform this task?

SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
 978-1-4503-2056-6/13/08 ...$15.00.

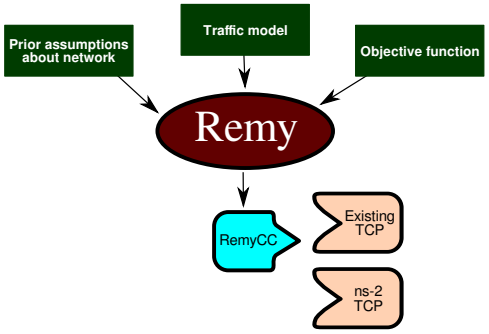

Figure 1: Remy designs congestion-control schemes automatically to achieve desired outcomes. The algorithms it produces may replace the congestion-control module of a TCP implementation, and fit into a network library or kernel module that implements congestion control (DCCP, SCTP, the congestion manager, application-layer transmission control libraries, ns-2 modules, etc.).

We investigated these questions and found that computers can design schemes that in some cases surpass the best human-designed methods to date, when supplied with the appropriate criteria by which to judge a congestion-control algorithm. We attempt to probe the limits of these machine-generated protocols, and discuss how this style of transport-layer protocol design can give more freedom to network architects and link-layer designers.

Congestion control, a fundamental problem in multi-user computer networks, addresses the question: when should an endpoint transmit each packet of data? An ideal scheme would transmit a packet whenever capacity to carry the packet was available, but because there are many concurrent senders and the network experiences variable delays, this question isn't an easy one to answer. On the Internet, the past thirty years have seen a number of innovative and influential answers to this question, with solutions embedded at the endpoints (mainly in TCP) aided occasionally by queue management and scheduling algorithms in bottleneck routers that provide signals to the endpoints.

This area has continued to draw research and engineering effort because new link technologies and subnetworks have proliferated and evolved. For example, the past few years have seen an increase in wireless networks with variable bottleneck rates; datacenter networks with high rates, short delays, and correlations in offered load; paths with excessive buffering (now called "bufferbloat"); cellular wireless networks with highly variable, self-inflicted packet delays; links with non-congestive stochastic loss; and networks with

large bandwidth-delay products. In these conditions, the classical congestion-control methods embedded in TCP can perform poorly, as many papers have shown (§2).

Without the ability to adapt its congestion-control algorithms to new scenarios, TCP's inflexibility constrains architectural evolution, as we noted in an earlier position paper [43]. Subnetworks and link layers are typically evaluated based on how well TCP performs over them. This scorecard can lead to perverse behavior, because TCP's network model is limited. For example, because TCP assumes that packet losses are due to congestion and reduces its transmission rate in response, some subnetwork designers have worked hard to hide losses. This often simply adds intolerably long packet delays. One may argue that such designs are misguided, but the difficulties presented by "too-reliable" link layers have been a perennial challenge for 25 years [12] and show no signs of abating. With the rise of widespread cellular connectivity, these behaviors are increasingly common and deeply embedded in deployed infrastructure.

The designers of a new subnetwork may well ask what they should do to make TCP perform well. This question is surprisingly hard to answer, because the so-called teleology of TCP is unknown: exactly what objective does TCP congestion control optimize? TCP's dynamic behavior, when competing flows enter and leave the network, remains challenging to explain [7]. In practice, the need to "make TCP perform well" is given as a number of loose guidelines, such as IETF RFC 3819 [23], which contains dozens of pages of qualitative best current practice. The challenging and subtle nature of this area means that the potential of new subnetworks and network architectures is often not realized.

Design overview

How should we design network protocols that free subnetworks and links to evolve freely, ensuring that the endpoints will adapt properly *no matter what* the lower layers do? We believe that the best way to approach this question is to take the design of specific algorithmic mechanisms out of the hands of human designers (no matter how sophisticated!), and make the end-to-end algorithm be a function of the desired overall behavior.

We start by explicitly stating an *objective* for congestion control; for example, given an unknown number of users, we may optimize some function of the per-user throughput and packet delay, or a summary statistic such as average flow completion time. Then, instead of writing down rules by hand for the endpoints to follow, we start from the desired objective and work backwards in three steps:

1. First, model the protocol's prior assumptions about the network; i.e., the "design range" of operation. This model may be different, and have different amounts of uncertainty, for a protocol that will be used exclusively within a data center, compared with one intended to be used over a wireless link or one for the broader Internet. A typical model specifies upper and lower limits on the bottleneck link speeds, non-queueing delays, queue sizes, and degrees of multiplexing.

2. Second, define a traffic model for the offered load given to endpoints. This may characterize typical Web traffic, video conferencing, batch processing, or some mixture of these. It may be synthetic or based on empirical measurements.

3. Third, use the modeled network scenarios and traffic to design a congestion-control algorithm that can later be executed on endpoints.

We have developed an optimization tool called Remy that takes these models as input, and designs a congestion-control algorithm that tries to maximize the total expected value of the objective function, measured over the set of network and traffic models. The resulting pre-calculated, optimized algorithm is then run on actual endpoints; no further learning happens after the offline optimization. The optimized algorithm is run as part of an existing TCP sender implementation, or within any congestion-control module. No receiver changes are necessary (as of now).

Summary of results

We have implemented Remy. Running on a 48-core server at MIT, Remy generally takes a few hours of wall-clock time (one or two CPU-weeks) to generate congestion-control algorithms offline that work on a wide range of network conditions.

Our main results from several simulation experiments with Remy are as follows:

1. For networks broadly consistent with the assumptions provided to Remy at design time, the machine-generated algorithms dramatically outperform existing methods, including TCP Cubic, Compound TCP, and TCP Vegas.

2. Comparing Remy's algorithms with schemes that require modifications to network gateways, including Cubic-over-sfqCoDel and XCP, Remy generally matched or surpassed these schemes, despite being entirely end-to-end.

3. We measured the tradeoffs that come from specificity in the assumptions supplied to Remy at design time. As expected, more-specific prior information turned out to be helpful when it was correct, but harmful when wrong. We found that RemyCC schemes performed well even when designed for an order-of-magnitude variation in the values of the underlying network parameters.

On a simulated 15 Mbps fixed-rate link with eight senders contending and an RTT of 150 ms, a computer-generated congestion-control algorithm achieved the following improvements in median throughput and reductions in median queueing delay over these existing protocols:

Protocol	Median speedup	Median delay reduction
Compound	2.1×	2.7×
NewReno	2.6×	2.2×
Cubic	1.7×	3.4×
Vegas	3.1×	1.2×
Cubic/sfqCoDel	1.4×	7.8×
XCP	1.4×	4.3×

In a trace-driven simulation of the Verizon LTE downlink with four senders contending, the *same* computer-generated protocol achieved these speedups and reductions in median queueing delay:

Protocol	Median speedup	Median delay reduction
Compound	1.3×	1.3×
NewReno	1.5×	1.2×
Cubic	1.2×	1.7×
Vegas	2.2×	0.44× ↓
Cubic/sfqCoDel	1.3×	1.3×
XCP	1.7×	0.78× ↓

The source code for Remy, our ns-2 models, and the algorithms that Remy designed are available from http://web.mit.edu/remy.

2. Related Work

Starting with Ramakrishnan and Jain's DECBit scheme [36] and Jacobson's TCP Tahoe (and Reno) algorithms [21], congestion control over heterogeneous packet-switched networks has been an active area of research. End-to-end algorithms typically compute a congestion window (or, in some cases, a transmission rate) as well

as the round-trip time (RTT) using the stream of acknowledgments (ACKs) arriving from the receiver. In response to congestion, inferred from packet loss or, in some cases, rising delays, the sender reduces its window; conversely, when no congestion is perceived, the sender increases its window.

There are many different ways to vary the window. Chiu and Jain [10] showed that among linear methods, additive increase / multiplicative decrease (AIMD) converges to high utilization and a fair allocation of throughputs, under some simplifying assumptions (long-running connections with synchronized and instantaneous feedback). Our work relaxes these assumptions to handle flows that enter and leave the network, and users who care about latency as well as throughput. Remy's algorithms are not necessarily linear, and can use both a window and a rate pacer to regulate transmissions.

In this paper, we compare Remy's generated algorithms with several end-to-end schemes, including NewReno [19], Vegas [9], Compound TCP [39], Cubic [18], and DCTCP for datacenters [2]. NewReno has the same congestion-control strategy as Reno—slow start at the beginning, on a timeout, or after an idle period of about one retransmission timeout (RTO), additive increase every RTT when there is no congestion, and a one-half reduction in the window on receiving three duplicate ACKs (signaling packet loss). We compare against NewReno rather than Reno because NewReno's loss recovery is better.

Brakmo and Peterson's Vegas is a delay-based algorithm, motivated by the insight from Jain's CARD scheme [22] and Wang and Crowcroft's DUAL scheme [41] that increasing RTTs may be a congestion signal. Vegas computes a BaseRTT, defined as the RTT in the absence of congestion, and usually estimated as the first RTT on the connection before the windows grow. The expected throughput of the connection is the ratio of the current window size and BaseRTT, if there is no congestion; Vegas compares the *actual* sending rate, and considers the difference, *diff*, between the expected and actual rates. Depending on this difference, Vegas either increases the congestion window linearly ($diff < \alpha$), reduces it linearly ($diff > \beta$), or leaves it unchanged.

Compound TCP [39] combines ideas from Reno and Vegas: when packet losses occur, it uses Reno's adaptation, while reacting to delay variations using ideas from Vegas. Compound TCP is more complicated than a straightforward hybrid of Reno and Vegas; for example, the delay-based window adjustment uses a binomial algorithm [6]. Compound TCP uses the delay-based window to identify the absence of congestion rather than its onset, which is a key difference from Vegas.

Rhee and Xu's Cubic algorithm is an improvement over their previous work on BIC [45]. Cubic's growth is independent of the RTT (like H-TCP [29]), and depends only on the packet loss rate, incrementing as a cubic function of "real" time. Cubic is known to achieve high throughput and fairness independent of RTT, but it also aggressively increases its window size, inflating queues and bloating RTTs (see §5).

Other schemes developed in the literature include equation-based congestion control [16], binomial control [6], FastTCP [42], HSTCP, and TCP Westwood [30].

End-to-end control may be improved with explicit router participation, as in Explicit Congestion Notification (ECN) [15], VCP [44], active queue management schemes like RED [17], BLUE [14], CHOKe [35], AVQ [27], and CoDel [33] fair queueing, and explicit methods such as XCP [24] and RCP [38]. AQM schemes aim to prevent persistent queues, and have largely focused on reacting to growing queues by marking packets with ECN or dropping them even before the queue is full. CoDel changes the model from reacting to specific average queue lengths to reacting when the delays measured over some duration are too long, suggesting a persistent queue. Scheduling algorithms isolate flows or groups of flows from each other, and provide weighted fairness between them. In XCP and RCP, routers place information in packet headers to help the senders determine their window (or rate). One limitation of XCP is that it needs to know the bandwidth of the outgoing link, which is difficult to obtain accurately for a time-varying wireless channel.

In §5, we compare Remy's generated algorithm with XCP and with end-to-end schemes running through a gateway with the CoDel AQM and stochastic fair queueing (sfqCoDel).

TCP congestion control was not designed with an explicit optimization goal in mind, but instead allows overall network behavior to emerge from its rules. Kelly et al. present an interpretation of various TCP congestion-control variants in terms of the implicit goals they attempt to optimize [25]. This line of work has become known as Network Utility Maximization (NUM); more recent work has modeled stochastic NUM problems [46], in which flows enter and leave the network. Remy may be viewed as combining the desire for practical distributed endpoint algorithms with the explicit utility-maximization ethos of stochastic NUM.

We note that TCP stacks have adapted in some respects to the changing Internet; for example, increasing bandwidth-delay products have produced efforts to increase the initial congestion window [13, 11], including recent proposals [3, 40] for this quantity to automatically increase on the timescale of months or years. What we propose in this paper is an automated means by which TCP's entire congestion-control algorithm, not just its initial window, could adapt in response to empirical variations in underlying networks.

3. Modeling the Congestion-Control Problem

We treat congestion control as a problem of distributed decision-making under uncertainty. Each endpoint that has pending data must decide for itself at every instant: send a packet, or don't send a packet.

If all nodes knew in advance the network topology and capacity, and the schedule of each node's present and future offered load, such decisions could in principle be made perfectly, to achieve a desired allocation of throughput on shared links.

In practice, however, endpoints receive observations that only hint at this information. These include feedback from receivers concerning the timing of packets that arrived and detection of packets that didn't, and sometimes signals, such as ECN marks, from within the network itself. Nodes then make sending decisions based on this partial information about the network.

Our approach hinges on being able to evaluate quantitatively the merit of any particular congestion control algorithm, and search for the best algorithm for a given network model and objective function. We discuss here our models of the network and cross traffic, and how we ultimately calculate a figure of merit for an arbitrary congestion control algorithm.

3.1 Expressing prior assumptions about the network

From a node's perspective, we treat the network as having been drawn from a stochastic generative process. We assume the network is Markovian, meaning that it is described by some state (e.g. the packets in each queue) and its future evolution will depend only on the current state.

Currently, we typically parametrize networks on three axes: the speed of bottleneck links, the propagation delay of the network paths, and the degree of multiplexing, i.e., the number of senders

contending for each bottleneck link. We assume that senders have no control over the paths taken by their packets to the receiver.

Depending on the range of networks over which the protocol is intended to be used, a node may have more or less uncertainty about the network's key parameters. For example, in a data center, the topology, link speeds, and minimum round-trip times may be known in advance, but the degree of multiplexing could vary over a large range. A virtual private network between "clouds" may have more uncertainty about the link speed. A wireless network path may experience less multiplexing, but a large range of transmission rates and round-trip times.

As one might expect, we have observed a tradeoff between generality and performance; a protocol designed for a broad range of networks may be beaten by a protocol that has been supplied with more specific and accurate prior knowledge. Our approach allows protocol designers to measure this tradeoff and choose an appropriate design range for their applications.

3.2 Traffic model

Remy models the offered load as a stochastic process that switches unicast flows between sender-receivers pairs on or off. In a simple model, each endpoint has traffic independent of the other endpoints. The sender is "off" for some number of seconds, drawn from an exponential distribution. Then it switches on for some number of bytes to be transmitted, drawn from an empirical distribution of flow sizes or a closed-form distribution (e.g. heavy-tailed Pareto). While "on," we assume that the sender will not stall until it completes its transfer.

In traffic models characteristic of data center usage, the off-to-on switches of contending flows may cluster near one another in time, leading to incast. We also model the case where senders are "on" for some amount of time (as opposed to bytes) and seek maximum throughput, as in the case of videoconferences or similar real-time traffic.

3.3 Objective function

Resource-allocation theories of congestion control have traditionally employed the alpha-fairness metric to evaluate allocations of throughput on shared links [37]. A flow that receives steady-state throughput of x is assigned a score of $U_\alpha(x) = \frac{x^{1-\alpha}}{1-\alpha}$. As $\alpha \to 1$, in the limit $U_1(x)$ becomes $\log x$.

Because $U_\alpha(x)$ is concave for $\alpha > 0$ and monotonically increasing, an allocation that maximizes the total score will prefer to divide the throughput of a bottleneck link equally between flows. When this is impossible, the parameter α sets the tradeoff between fairness and efficiency. For example, $\alpha = 0$ assigns no value to fairness and simply measures total throughput. $\alpha = 1$ is known as proportional fairness, because it will cut one user's allocation in half as long as another user's can be more than doubled. $\alpha = 2$ corresponds to minimum potential delay fairness, where the score goes as the negative inverse of throughput; this metric seeks to minimize the total time of fixed-length file transfers. As $\alpha \to \infty$, maximizing the total $U_\alpha(x)$ achieves max-min fairness, where all that matters is the minimum resource allocations in bottom-up order [37].

Because the overall score is simply a sum of monotonically increasing functions of throughput, an algorithm that maximizes this total is Pareto-efficient for any value of α; i.e., the metric will always prefer an allocation that helps one user and leaves all other users the same or better. Tan et al. [28] proved that, subject to the requirement of Pareto-efficiency, alpha-fairness is *the* metric that places the greatest emphasis on fairness for a particular α.

Kelly et al. [25] and further analyses showed that TCP approximately maximizes minimum potential delay fairness asymptotically in steady state, if all losses are congestive and link speeds are fixed.

We extend this model to cover dynamic traffic and network conditions. Given a network trace, we calculate the average throughput x of each flow, defined as the total number of bytes received divided by the time that the sender was "on." We calculate the average round-trip delay y of the connection.

The flow's score is then

$$U_\alpha(\mathrm{x}) - \delta \cdot U_\beta(\mathrm{y}), \tag{1}$$

where α and β express the fairness-vs.-efficiency tradeoffs in throughput and delay, respectively, and δ expresses the relative importance of delay vs. throughput.

We emphasize that the purpose of the objective function is to supply a quantitative goal from a protocol-design perspective. It need not (indeed, does not) precisely represent users' "true" preferences or utilities. In real usage, different users may have different objectives; a videoconference may not benefit from more throughput, or some packets may be more important than others. We have not yet addressed the problem of how to accommodate diverse objectives or how endpoints might learn about the differing preferences of other endpoints.

4. How Remy Produces a Congestion-Control Algorithm

The above model may be viewed as a cooperative game that endpoints play. Given packets to transmit (offered load) at an endpoint, the endpoint must decide when to send packets in order to maximize its own objective function. With a particular congestion-control algorithm running on each endpoint, we can calculate each endpoint's expected score.

In the traditional game-theoretic framework, an endpoint's decision to send or abstain can be evaluated after fixing the behavior of all other endpoints. An endpoint makes a "rational" decision to send if doing so would improve its expected score, compared with abstaining.

Unfortunately, when greater individual throughput is the desired objective, on a best-effort packet-switched network like the Internet, it is always advantageous to send a packet. In this setting, if every endpoint acted rationally in its own self-interest, the resulting Nash equilibrium would be congestion collapse![1] This answer is unsatisfactory from a protocol-design perspective, when endpoints have the freedom to send packets when they choose, but the designer wishes to achieve an efficient and equitable allocation of network capacity.

Instead, we believe the appropriate framework is that of *superrationality* [20]. Instead of fixing the other endpoints' actions before deciding how to maximize one endpoint's expected score, what is fixed is the common (but as-yet unknown) algorithm run by all endpoints. As in traditional game theory, the endpoint's goal remains maximizing its own self-interest, but *with the knowledge* that other endpoints are reasoning the same way and will therefore arrive at the same algorithm.

Remy's job is to find what that algorithm should be. We refer to a particular Remy-designed congestion-control algorithm as a "RemyCC," which we then implant into an existing sender as part of TCP, DCCP [26], congestion manager [5], or another module

[1]Other researchers have grappled with this problem; for example, Akella et al. [1] studied a restricted game, in which players are forced to obey the same particular flavor of TCP, but with the freedom to choose their additive-increase and multiplicative-decrease coefficients. Even with this constraint, the authors found that the Nash equilibrium is inefficient, unless the endpoints are restricted to run TCP Reno over a drop-tail buffer, in which case the equilibrium is unfair but not inefficient.

running congestion control. The receiver is unchanged (as of now; this may change in the future), but is expected to send periodic ACK feedback.

Formally, we treat the problem of finding the best RemyCC under uncertain network conditions as a search for the best policy for a decentralized partially-observable Markov decision process, or Dec-POMDP [34]. This model originated from operations research and artificial intelligence, in settings where independent agents work cooperatively to achieve some goal. In the case of end-to-end congestion control, endpoints are connected to a shared network that evolves in Markovian fashion. At every time step, the agents must choose between the actions of "sending" or "abstaining," using observables from their receiver or from network infrastructure.

4.1 Compactly representing the sender's state

In principle, for any given network, there is an *optimal* congestion-control scheme that maximizes the expected total of the endpoints' objective functions. Such an algorithm would relate (1) the entire history of observations seen thus far (e.g. the contents and timing of every ACK) and (2) the entire history of packets already sent, to the best action at any given moment between sending a new packet or abstaining. However, the search for such an algorithm is likely intractable; on a general Dec-POMDP it is NEXP-complete [8].

Instead, we approximate the solution by greatly abridging the sender's state. A RemyCC tracks just three state variables, which it updates each time it receives a new acknowledgment:

1. An exponentially-weighted moving average (EWMA) of the interarrival time between new acknowledgments received (`ack_ewma`).
2. An exponentially-weighted moving average of the time between TCP sender timestamps reflected in those acknowledgments (`send_ewma`). A weight of $1/8$ is given to the new sample in both EWMAs.
3. The ratio between the most recent RTT and the minimum RTT seen during the current connection (`rtt_ratio`) .

Together, we call these three variables the *RemyCC memory*. It is worth reflecting on these variables, which are the "congestion signals" used by any RemyCC. We narrowed the memory to this set after examining and discarding quantities like the most-recent RTT sample, the smoothed RTT estimate, and the difference between the long-term EWMA and short-term EWMA of the observed packet rate or RTT. In our experiments, adding extra state variables didn't improve the performance of the resulting protocol, and each additional dimension slows down the design procedure considerably. But we don't claim that Remy's three state variables are the only set that works, or that they are necessarily optimal for all situations a protocol might encounter. We expect that any group of estimates that roughly summarizes the recent history could form the basis of a workable congestion-control scheme.

We note that a RemyCC's memory does not include the two factors that traditional TCP congestion-control schemes use: packet loss and RTT. This omission is intentional: a RemyCC that functions well will see few congestive losses, because its objective function will discourage building up queues (bloating buffers will decrease a flow's score). Moreover, avoiding packet loss as a congestion signal allows the protocol to robustly handle stochastic (non-congestive) packet losses without adversely reducing performance. We avoid giving the sender access to the RTT (as opposed to the RTT ratio), because we do not want it to learn different behaviors for different RTTs.

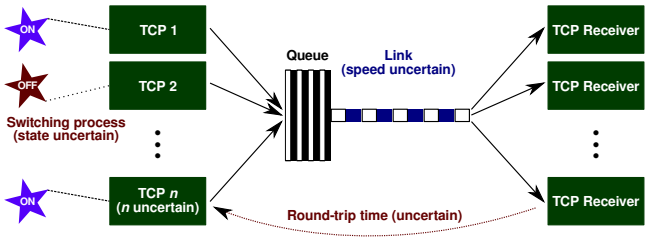

Figure 2: Dumbbell network with uncertainty.

At the start of each flow, before any ACKs have been received, the memory starts in a well-known all-zeroes initial state. RemyCCs do not keep state from one "on" period to the next, mimicking TCP's behavior in beginning with slow start every time a new connection is established (it is possible that caching congestion state is a good idea on some paths, but we don't consider this here). Although RemyCCs do not depend on loss as a congestion signal, they do inherit the loss-recovery behavior of whatever TCP sender they are added to.

4.2 RemyCC: Mapping the memory to an action

A RemyCC is defined by how it maps values of the memory to output *actions*. Operationally, a RemyCC runs as a sequence of lookups triggered by incoming ACKs. (The triggering by ACKs is inspired by TCP's ACK clocking.) Each time a RemyCC sender receives an ACK, it updates its memory and then looks up the corresponding action. It is Remy's job to pre-compute this lookup table during the design phase, by finding the mapping that maximizes the expected value of the objective function, with the expectation taken over the network model.

Currently, a Remy action has three components:

1. A multiple $m \geq 0$ to the current congestion window (`cwnd`).
2. An increment b to the congestion window (b could be negative).
3. A lower bound $r > 0$ milliseconds on the time between successive sends.

If the number of outstanding packets is greater than `cwnd`, the sender will transmit segments to close the window, but no faster than one segment every r milliseconds.

A RemyCC is defined by a set of piecewise-constant *rules*, each one mapping a three-dimensional rectangular region of the three-dimensional memory space to a three-dimensional action:

$$\langle \texttt{ack\_ewma}, \texttt{send\_ewma}, \texttt{rtt\_ratio} \rangle \rightarrow \langle m, b, r \rangle.$$

4.3 Remy's automated design procedure

The design phase of Remy is an optimization procedure to efficiently construct this state-to-action mapping, or *rule table*. Remy uses simulation of the senders on various sample networks drawn from the network model, with parameters drawn within the ranges of the supplied prior assumptions. These parameters include the link rates, delays, the number of sources, and the on-off distributions of the sources. Offline, Remy evaluates candidate algorithms on millions of randomly generated network configurations. Because of the high speed of current computers and the "embarrassingly parallel" nature of the task, Remy is able to generate congestion-control algorithms within a few hours.

A single evaluation step, the innermost loop of Remy's design process, consists of drawing 16 or more network specimens from

the network model, then simulating the RemyCC algorithm at each sender for 100 seconds on each network specimen. At the end of the simulation, the objective function for each sender, given by Equation 1, is totaled to produce an overall figure of merit for the RemyCC. We explore two cases, $\alpha = \beta = 1$ and $\alpha = 2, \delta = 0$. The first case corresponds to proportional throughput and delay fairness, maximizing

$$U = \log(\text{throughput}) - \delta \cdot \log(\text{delay}),$$

with δ specifying the importance placed on delay vs. throughput. The second case corresponds to minimizing the potential delay of a fixed-length transfer, by maximizing

$$U = -\frac{1}{\text{throughput}}.$$

Remy initializes a RemyCC with only a single rule. Any values of the three state variables (between 0 and 16,384) are mapped to a default action where $m = 1$, $b = 1$, $r = 0.01$.

Each entry in the rule table has an "epoch." Remy maintains a global epoch number, initialized to 0. Remy's search for the "best" RemyCC given a network model is a series of greedy steps to build and improve the rule table:

1. **Set all rules to the current epoch.**
2. **Find the most-used rule in this epoch.** Simulate the current RemyCC and see which rule in the current epoch receives the most use. If no such rules were used, go to step 4.
3. **Improve that action until we can't anymore.** Focus on this rule and find the best action for it. Draw at least 16 network specimens from the model, and then evaluate roughly 100 candidate increments to the current action, increasing geometrically in granularity as they get further from the current value. For example, evaluate $r \pm 0.01$, $r \pm 0.08$, $r \pm 0.64$, ..., taking the Cartesian product with the alternatives for m and b.

   The modified action is evaluated by substituting it *into all senders* and repeating the simulation in parallel. We use the same random seed and the same set of specimen networks in the simulation of each candidate action to reduce the effects of random variation.

   If any of the candidates is an improvement, replace the action with the best new action and repeat the search, still with the same specimen networks and random seed. Otherwise, increment the epoch number of the current rule and go back to step 2.
4. **If we run out of rules in this epoch.** Increment the global epoch. If the new epoch is a multiple of a parameter, K, continue to step 5. Otherwise go back to step 1. We use $K = 4$ to balance structural improvements vs. honing the existing structure.
5. **Subdivide the most-used rule.** Recall that each rule represents a mapping from a three-dimensional rectangular region of memory space to a single action. In this step, find the most-used rule, and the median memory value that triggers it. Split the rule at this point, producing eight new rules (one per dimension of the memory-space), each with the same action as before. Then return to step 1.

By repeating this procedure, the structure of a RemyCC's rule table becomes an octree [32] of memory regions. Areas of the memory space more likely to occur receive correspondingly more attention from the optimizer, and are subdivided into smaller bins that yield a more granular function relating memory to action. *Which* rules are more often triggered depends on every endpoint's behavior as well as the network's parameters, so the task of finding the right structure for the rule table is best run alongside the process of optimizing existing rules.

To the best of our knowledge, this dynamic partitioning approach is novel in the context of multi-agent optimization. The "greedy" approach in step 2 is key to the computational tractability and efficiency of the search because it allows us to prune the search space. Dividing the memory space into cells of different size proportional to their activity produces a rule table whose granularity is finer in regions of higher use. An improvement to consider in the future is to divide a cell only if the actions at its boundaries markedly disagree.[2]

5. EVALUATION

We used ns-2 to evaluate the algorithms generated by Remy and compare them with several other congestion-control methods, including both end-to-end schemes and schemes with router assistance. This section describes the network and workload scenarios and our findings.

5.1 Simulation setup and metrics

Congestion-control protocols. The end-to-end schemes we compared with are NewReno, Vegas, Cubic, and Compound. In addition, we compared against two schemes that depend on router assistance: XCP, and Cubic over stochastic fair queueing [31] with each queue running CoDel [33]. We use Nichols's published sfqCoDel implementation (version released in March 2013) for ns-2.[3] The Cubic, Compound, and Vegas codes are from the Linux implementations ported to ns-2 and available in ns-2.35. For the datacenter simulation, we also compare with the DCTCP ns-2.35 patch.[4]

RemyCCs. We used Remy to construct three general-purpose RemyCCs. Each one was designed for an uncertain network model with the dumbbell topology of Figure 2, but with three different values of δ (the relative importance of delay): 0.1, 1, and 10. The parameters of the network and traffic model used at design time were:

Quantity	Design range	Distribution
n max senders	1–16	uniform
"on" process	mean 5 s	exponential
"off" process	mean 5 s	exponential
link speed	10–20 Mbps	uniform
round-trip time	100–200 ms	uniform
queue capacity	unlimited	

The model captures a 64-fold range of bandwidth-delay product per user. Each RemyCC took about 3–5 CPU-days to optimize. Calculations were run on Amazon EC2 and on an 80-core and 48-core server at MIT. In wall-clock time, each RemyCC took a few hours to be constructed. The RemyCCs contain between 162 and 204 rules each.

We also used Remy to assess how performance varies based on the specificity of the assumptions used at design time, by building one RemyCC for a link speed known exactly *a priori*, and one that assumes only that the link speed will lie within a tenfold range:

[2] We thank Leslie Kaelbling for this suggestion.
[3] http://www.pollere.net/Txtdocs/sfqcodel.cc
[4] http://www.stanford.edu/~alizade/Site/DCTCP.html

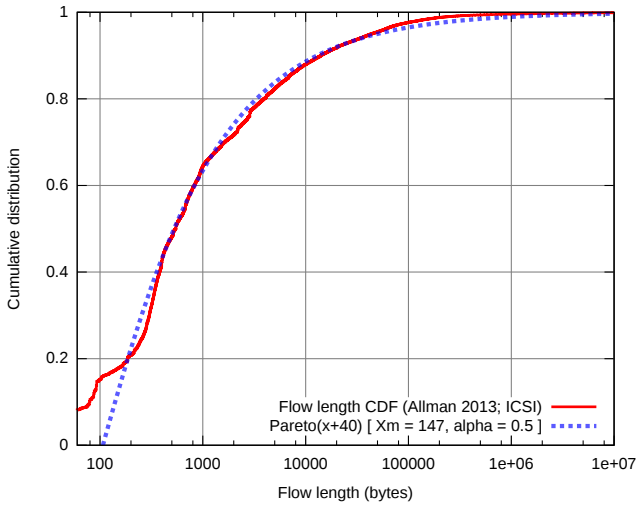

Figure 3: Observed Internet flow length distribution matches a Pareto ($\alpha = 0.5$) distribution, suggesting mean is not well-defined.

Quantity	Design range	Distribution
n max senders	2	uniform
"on" process	mean 5 sec	exponential
"off" process	mean 5 sec	exponential
link speed	15 Mbps ("1×")	exact
link speed	4.7–47 Mbps ("10×")	uniform
round-trip time	150 ms	exact
queue capacity	unlimited	

In most experiments, all the sources run the same protocol; in some, we pick different protocols for different sources to investigate how well they co-exist. Each simulation run is generally 100 seconds long, with each scenario run at least 128 times to collect summary statistics.

Workloads. Each source is either "on" or "off" at any point in time. In the evaluation, we modeled the "off" times as exponentially distributed, and the "on" distribution in one of three different ways:

- *by time*, where the source sends as many bytes as the congestion-control protocol allows, for a duration of time picked from an exponential distribution,
- *by bytes*, where the connection sends as many bytes as given by an exponential distribution of a given average and shape, and
- *by empirical distribution*, using the flow-length CDF from a large trace captured in March 2012 and published recently [4]. The flow-length CDF matches a Pareto distribution with the parameters given in Figure 3, suggesting that the underlying distribution does not have finite mean. In our evaluation, we add 16 kilobytes to each sampled value to ensure that the network is loaded.

Topologies. We used these topologies in our experiments:

1. **Single bottleneck ("dumbbell"):** The situation in Figure 2, with a 1,000-packet buffer, as might be seen in a shared cable-modem uplink. We tested a configuration whose link speed and delay were within the RemyCC design ranges:

Quantity	Range	Distribution
link speed	15 Mbps	exact
round-trip time	150 ms	exact
queue capacity	1000 pkts (tail drop)	

2. **Cellular wireless:** We measured the downlink capacity of the Verizon and AT&T LTE cellular services while mobile, by carefully saturating the downlink (without causing buffer overflow) and recording when packets made it to the user device. We recreate this link within ns-2, queueing packets until they are released to the receiver at the same time they were released in the trace. This setup probes the RemyCC's resilience to "model mismatch" — in both the Verizon and AT&T traces, throughput and round-trip time were outside the limits of the RemyCC design range.

Quantity	Range	Distribution
link speed	varied 0–50 Mbps	empirical
round-trip time	50 ms	exact
queue capacity	1000 pkts (tail drop)	

3. **Differing RTTs:** Cases where different RemyCCs, contending for the same link, had different RTTs to their corresponding receiver. We analyzed these cases for throughput and delay fairness and compared with existing congestion-control schemes.

Quantity	Range	Distribution
n max senders	4	
"on" process	16×10^3–3.3×10^9 bytes	Fig. 3
"off" process	mean 0.2 sec	exponential
link speed	10 Mbps	exact
queue capacity	1000 pkts (tail drop)	

4. **Datacenter:** We compared a RemyCC against DCTCP in a simulated datacenter topology.

Quantity	Range	Distribution
n max senders	64	exact
"on" process	mean 20 megabytes	exponential
"off" process	mean 0.1 sec	exponential
link speed	10 Gbps	exact
round-trip time	4 ms	exact
queue capacity	1000 pkts (tail drop)	(for RemyCC)
queue capacity	modified RED	(for DCTCP)

In addition, we investigate:

5. **Competing protocols:** We assessed how a RemyCC "played with" existing congestion-control schemes (Cubic and Compound) when contending for the same bottleneck link.
6. **Sensitivity of design range:** We investigated how helpful prior knowledge of the network is to the performance of Remy's generated algorithms.

Metrics. We measure the throughput and average queueing delay observed for each source-destination pair. With an on-off source, measuring throughput takes some care. We define the throughput of a pair as follows. Suppose the pair is active during (non-overlapping) time intervals of length $t_1, t_2, \ldots$ during the entire simulation run of T seconds. If in each interval the protocol successfully receives s_i bytes, we define the throughput for this connection as $\sum s_i / \sum t_i$.

We are interested in the end-to-end delay as well; the reasoning behind Remy's objective function and the δ parameter is that protocols that fill up buffers to maximize throughput are not as desirable as ones that achieve high throughput *and* low delay — both for their

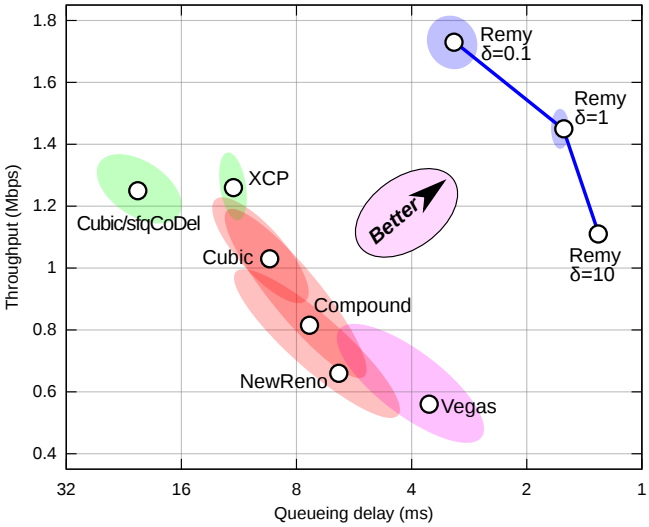

Figure 4: Results for each of the schemes over a 15 Mbps dumbbell topology with $n = 8$ senders, each alternating between flows of exponentially-distributed byte length (mean 100 kilobytes) and exponentially-distributed off time (mean 0.5 s). Medians and 1-σ ellipses are shown. The blue line represents the efficient frontier, which here is defined entirely by the RemyCCs.

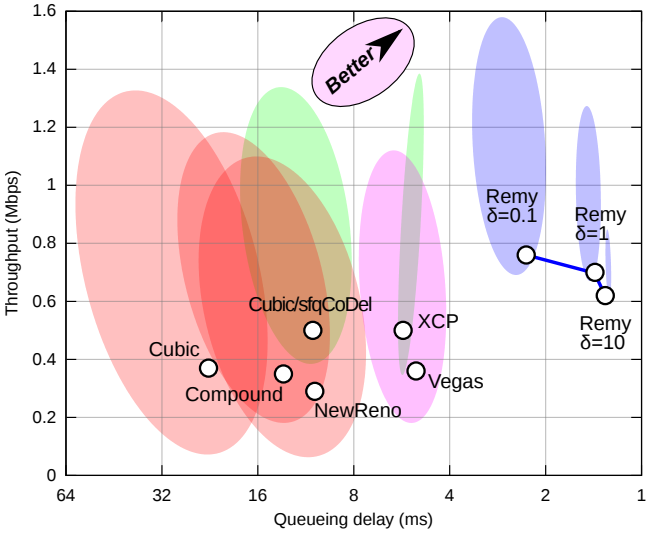

Figure 5: Results for the dumbbell topology with $n = 12$ senders, each alternating between flows whose length is drawn from the ICSI trace (Fig. 3) and exponentially-distributed off time (mean = 0.2 s). Because of the high variance of the sending distribution, $\frac{1}{2}$-σ ellipses are down. The RemyCCs again mark the efficient frontier.

effect on the user, who may prefer to get his packets to the receiver sooner, as well as any other users who share the same FIFO queue.

We present the results for the different protocols as **throughput-delay plots**, where the log-scale x-axis is the queueing delay (average per-packet delay in excess of minimum RTT). Lower, better, delays are to the right. The y-axis is the throughput. Protocols on the "top right" are the best on such plots. We take each individual 100-second run from a simulation as one point, and then compute the 1-σ elliptic contour of the maximum-likelihood 2D Gaussian distribution that explains the points. To summarize the whole scheme, we plot the median per-sender throughput and queueing delay as a circle.

Ellipses that are narrower in the throughput or delay axis correspond to protocols that are fairer and more consistent in allocating those quantities. Protocols with large ellipses — where identically-positioned users differ widely in experience based on the luck of the draw or the timing of their entry to the network — are less fair. The orientation of an ellipse represents the *covariance between the throughput and delay* measured for the protocol; if the throughput were uncorrelated with the queueing delay (note that we show the queueing delay, not the RTT), the ellipse's axes would be parallel to the graph's. Because of the variability and correlations between these quantities in practice, we believe that such throughput-delay plots are an instructive way to evaluate congestion-control protocols; they provide more information than simply reporting mean throughput and delay values.

5.2 Single Bottleneck Results

We start by investigating performance over the simple, classic single-bottleneck "dumbbell" topology. Although it does not model the richness of real-world network paths, the dumbbell is a valuable topology to investigate because in practice there are many single-bottleneck paths experienced by Internet flows.

Recall that this particular dumbbell link had most of its parameters found inside the limits of the design range of the RemyCCs tested. As desired, this test demonstrates that Remy was successful in producing a family of congestion-control algorithms for this type of network.

Results from the 8-sender and 12-sender cases are shown in Figures 4 and 5. RemyCCs are shown in light blue; the results demonstrate the effect of the δ parameter in weighting the cost of delay. When $\delta = 0.1$, RemyCC senders achieve greater median throughput than those of any other scheme, and the lowest delay (other than the two other RemyCCs). As δ increases, the RemyCCs trace out an achievability frontier of the compromise between throughput and delay. In this experiment, the computer-generated algorithms outperformed all the human-designed ones.

From right to left and bottom to top, the end-to-end TCP congestion-control schemes trace out a path from most delay-conscious (Vegas) to most throughput-conscious (Cubic), with NewReno and Compound falling in between.

The schemes that require in-network assistance (XCP and Cubic-over-sfqCoDel, shown in green) achieve higher throughput than the TCPs, but less than the two more throughput-conscious RemyCCs.[5] This result is encouraging, because it suggests that even a purely end-to-end scheme can outperform well-designed algorithms that involve active router participation. This demonstrates that distributed congestion-control algorithms that explicitly maximize well-chosen objective functions can achieve gains over existing schemes. As we will see later, however, this substantially better performance will not hold when the design assumptions of a RemyCC are contradicted at runtime.

[5]It may seem surprising that sfqCoDel, compared with DropTail, *increased* the median RTT of TCP Cubic. CoDel drops a packet at the front of the queue if all packets in the past 100 ms experienced a queueing delay (sojourn time) of at least 5 ms. For this experiment, the transfer lengths are only 100 kilobytes; with a 500 ms "off" time, such a persistent queue is less common even though the mean queueing delay is a lot more than 5 ms. DropTail experiences more losses, so has lower delays (the maximum queue size is $\approx 4\times$ the bandwidth-delay product), but also lower throughput than CoDel. In other experiments with longer transfers, Cubic did experience lower delays when run over sfqCoDel instead of DropTail.

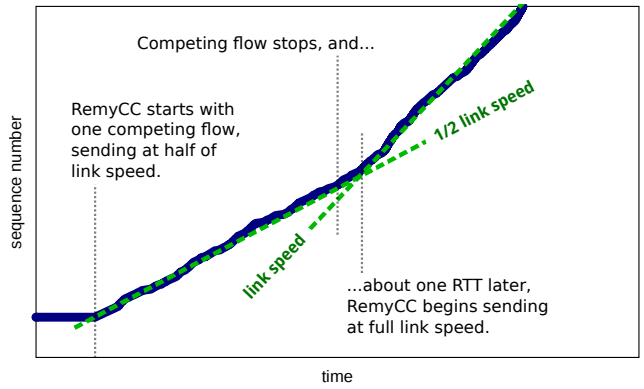

Figure 6: Sequence plot of a RemyCC flow in contention with varying cross traffic. The flow responds quickly to the departure of a competing flow by doubling its sending rate.

In Figures 4 and 5, the RemyCCs do not simply have better median performance — they are also more fair to individual flows, in that the performance of an individual sender (indicated by the size of the ellipses) is more consistent in both throughput and delay.

To explain this result, we investigated how multiple RemyCC flows share the network. We found that when a new flow starts, the system converges to an equitable allocation quickly, generally after little more than one RTT. Figure 6 shows the sequence of transmissions of a new RemyCC flow that begins while sharing the link. Midway through the flow, the competing traffic departs, allowing the flow to start consuming the whole bottleneck rate.

5.3 Cellular Wireless Links

Cellular wireless links are tricky for congestion-control algorithms because their link rates vary with time.[6]

By running a program that attempts to keep a cellular link backlogged but without causing buffer overflows, we measured the variation in download speed on Verizon's and AT&T's LTE service while mobile. We then ran simulations over these pre-recorded traces, with the assumption that packets are enqueued by the network until they can be dequeued and delivered at the same instants seen in the trace.

As discussed above, we did not design the RemyCCs to accommodate such a wide variety of throughputs. Running the algorithm over this link illustrated some of the limits of a RemyCC's generalizability beyond situations encountered during the design phase.

Somewhat to our surprise, for moderate numbers of concurrent flows, $n \leq 8$, the RemyCCs continued to surpass (albeit narrowly) the best human-designed algorithms, even ones benefiting from in-network assistance. See Figures 7 and 8.

5.4 Differing RTTs

We investigated how the RemyCCs allocate throughput on a contested bottleneck link when the competing flows have different RTTs. At the design stage, all contending flows had the same RTT (which was drawn randomly for each network specimen from between 100 ms and 200 ms), so the RemyCCs were not designed to exhibit RTT fairness explicitly.

We compared the RemyCCs with Cubic-over-sfqCoDel by running 128 realizations of a four-sender simulation where one sender-receiver pair had RTT of 50 ms, one had 100 ms, one 150 ms, and

[6]XCP, in particular, depends on knowing the speed of the link exactly; in our tests on cellular traces we supplied XCP with the long-term average link speed for this value.

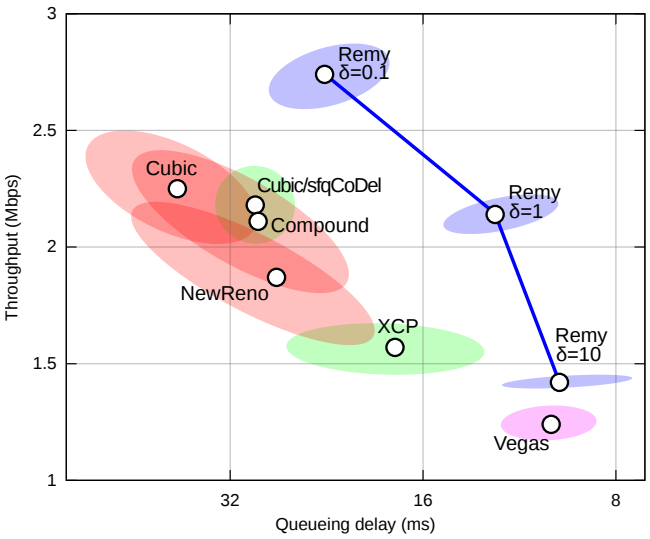

Figure 7: Verizon LTE downlink trace, $n = 4$. 1-σ ellipses are shown. The RemyCCs define the efficient frontier. Senders alternated between exponentially-distributed file transfers (mean 100 kilobytes) and exponentially-distributed pause times (mean 0.5 s).

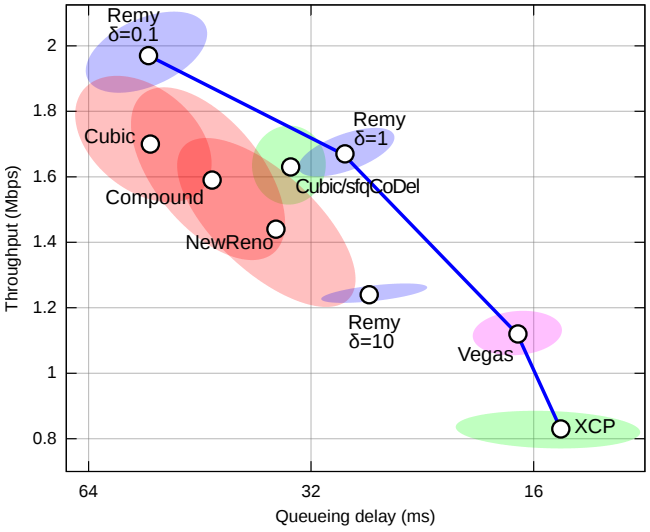

Figure 8: Verizon LTE downlink trace, $n = 8$. 1-σ ellipses are shown. As the degree of multiplexing increases, the schemes move closer together in performance and router-assisted schemes begin to perform better. Two of the three RemyCCs are on the efficient frontier.

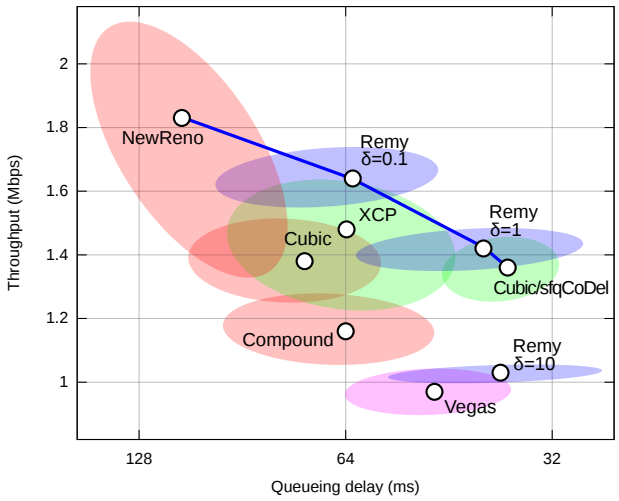

Figure 9: AT&T LTE downlink trace, $n = 4$. Two of the RemyCCs are on the efficient frontier.

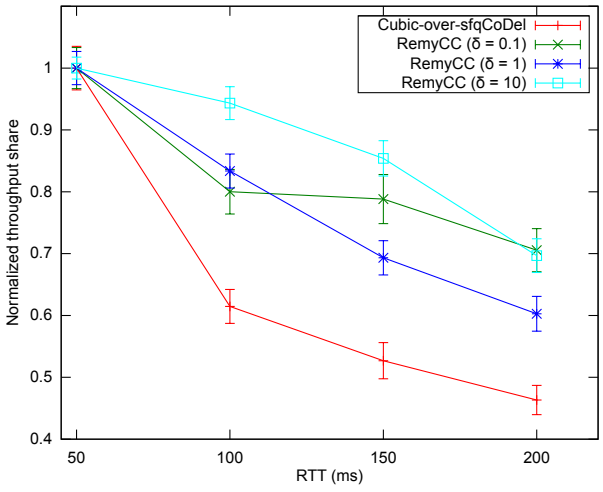

Figure 10: Remy's RTT unfairness compares favorably to Cubic-over-sfqCoDel. Error bar represents standard error of the mean over 128 100-second simulations.

one 200 ms. The RemyCCs did exhibit RTT unfairness, but more modestly than Cubic-over-sfqCoDel (Fig. 10).

5.5 Datacenter-like topology

We simulated 64 connections sharing a 10 Gbps datacenter link, and compared DCTCP [2] (using AQM inside the network) against a RemyCC with a 1000-packet tail-drop queue. The RTT of the path in the absence of queueing was 4 ms. Each sender sent 20 megabytes on average (exponentially distributed) with an "off" time between its connections exponentially distributed with mean 100 milliseconds.

We used Remy to design a congestion-control algorithm to maximize $-1/\text{throughput}$ (minimum potential delay) over these network parameters, with the degree of multiplexing assumed to have been drawn uniformly between 1 and 64.

The results for the mean and median throughput (tput) for the 20 megabyte transfers are shown in the following table:

	tput: mean, med	**rtt: mean, med**
DCTCP (ECN)	179, 144 Mbps	7.5, 6.4 ms
RemyCC (DropTail)	175, 158 Mbps	34, 39 ms

These results show that a RemyCC trained for the datacenter-network parameter range achieves comparable throughput at lower variance than DCTCP, a published and deployed protocol for similar scenarios. The per-packet latencies (and loss rates, not shown) are higher, because in this experiment RemyCC operates over a DropTail bottleneck router, whereas DCTCP runs over an ECN-enabled RED gateway that marks packets when the instantaneous queue exceeds a certain threshold. Developing RemyCC schemes for networks with ECN and AQM is an area for future work.

5.6 Competing protocols

We investigated the possibility of incremental deployment of a RemyCC, by simulating a single bottleneck link with one RemyCC flow contending with one flow from either Compound or Cubic, with no active queue management. The RemyCC was designed for round-trip-times between 100 ms and 10 s, in order to accommodate a "buffer-filling" competitor on the same bottleneck link.

We used the same observed traffic distribution from Figure 3 and varied the mean "off" time (exponentially distributed) of the senders. The bottleneck link speed was 15 Mbps and baseline RTT was 150 ms. We also experimented with flows of mean sizes 100 kilobytes and 1 megabyte, with an exponentially distributed mean "off" time of 0.5 seconds between successive flows.

The results, shown in the two tables below, depended on the duty cycle of the senders dictated by the mean off time (numbers in parentheses are standard deviations).

Mean off time	**RemyCC tput**	**Compound tput**
200 ms	2.12 (.11) Mbps	1.79 (.18) Mbps
100	2.18 (.08)	2.75 (.27)
10	2.28 (.10)	3.9 (.13)
Mean size	**RemyCC tput**	**Cubic tput**
100 KBytes	2.04 (.14)	1.31 (.16)
1 MByte	2.09 (.11)	1.28 (.11)

We observe that this RemyCC does well at low duty cycles because it is able to grab spare bandwidth more quickly. At higher duty cycles (with low mean off time), Cubic and Compound tend to grab a higher share of the bandwidth. The results, however, are close enough that we believe a RemyCC designed for competing with more aggressive protocols may close the gap, while retaining high performance when competing only with like-minded RemyCCs.

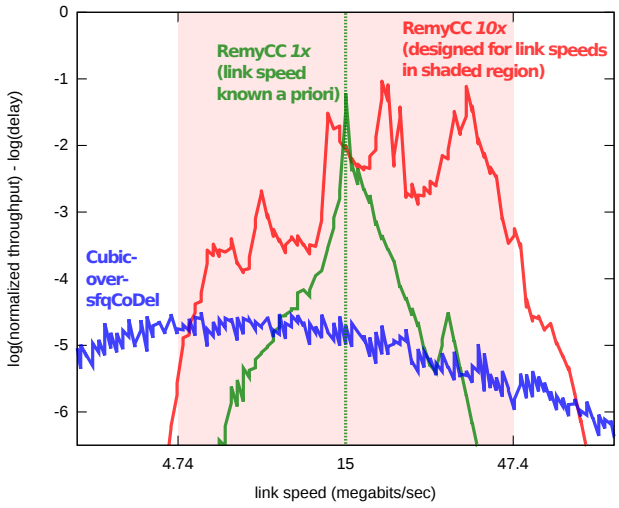

Figure 11: Performance of two end-to-end RemyCCs that were designed with different prior information about the network, compared with Cubic-over-sfqCoDel as the link speed varies. Despite running only at the sender, the RemyCCs each outperform Cubic-over-sfqCoDel over almost their entire design ranges. But when a RemyCC's assumptions aren't met, performance deteriorates.

5.7 How helpful is prior knowledge about the network?

We investigated the performance benefit conferred by having more-specific prior information about the network, and what happens when that prior information is incorrect.

We used Remy to construct two additional RemyCCs, each for a network with a known minimum RTT of 150 ms. For one RemyCC, the link speed was assumed to be 15 Mbps exactly. A second RemyCC was designed to span a 10× range of link speeds, from 4.7 Mbps to 47 Mbps. We also compared against Cubic-over-sfqCoDel over this range.

The results are shown in Figure 11. On the particular link for which the "1×" RemyCC was designed, it performs the best, but its performance trails off quickly around that value. Within the range of the "10×" RemyCC, it beats Cubic-over-sfqCoDel, but again deteriorates when the true network violates its design assumptions. The results show that more-specific prior knowledge is helpful and improves performance — when it happens to be correct.

5.8 Summary of results

Using a few CPU-weeks of computation, Remy produced several computer-generated congestion-control algorithms, which we then evaluated on a variety of simulated network conditions of varying similarity to the prior assumptions supplied at design-time.

On networks whose parameters mostly obeyed the prior knowledge supplied at design range — such as the dumbbell network with the 15 Mbps link — Remy's end-to-end algorithms outperformed all of the human-generated congestion-control algorithms, even algorithms that receive help from network infrastructure.

RemyCC ($\delta = 0.1$) achieved $> 1.7\times$ gains in median throughput and $> 2.7\times$ reductions in median queueing delay against Cubic and Compound, generally thought to be excellent general-purpose congestion-control algorithms. Against Cubic-over-sfqCoDel, which has the benefit of code running on network infrastructure, RemyCC achieved a 40% increase in median throughput and a 7.8× decrease in median queueing delay.

On the cellular link traces, which are variable and were not designed for, Remy's schemes outperformed the existing congestion-control algorithms (end-to-end or otherwise) when the maximum degree of multiplexing was 4 or less, and outperformed the end-to-end schemes and sfqCoDel when it was 8 or less. However, as the network conditions grew farther afield from the supplied prior assumptions, Remy's performance declined, although the algorithms were still competitive with traditional TCP congestion control on the networks we examined.

6. Discussion

Much remains unknown about the capabilities and limits of computer-generated algorithms, much less decentralized algorithms that cooperate indirectly across a network to achieve a common goal. Although the RemyCCs appear to work well on networks whose parameters fall within or near the limits of what they were prepared for — even beating in-network schemes at their own game and even when the design range spans an order of magnitude variation in network parameters — we do not yet understand clearly *why* they work, other than the observation that they seem to optimize their intended objective well.

We have attempted to make algorithms ourselves that surpass the generated RemyCCs, without success. That suggests to us that Remy may have accomplished something substantive. But digging through the dozens of rules in a RemyCC and figuring out their purpose and function is a challenging job in reverse-engineering. RemyCCs designed for broader classes of networks will likely be even more complex, compounding the problem.

Our approach *increases* endpoint complexity in order to *reduce* the complexity of overall network behavior. Traditional TCP congestion control specifies simpler behavior for each endpoint, but the resulting emergent behavior of a multiuser network is not easily specified and is often suboptimal and variable, and even unstable.

By contrast, our approach focuses on maximizing a well-specified overall objective at the cost of complex endpoint algorithms. We think this tradeoff is advisable: today's endpoints can execute complex algorithms almost as easily as simple ones (and with Remy, the bulk of the intelligence is computed offline). What users and system designers ultimately care about, we believe, is the quality and consistency of overall behavior.

Our *synthesis-by-simulation* approach also makes it easier to *discuss* competing proposals for congestion control. Today, it is not easy to say why one flavor of TCP or tweak may be preferred over another. But if two computer-generated algorithms differ, there is a reason: either they make different assumptions about the expected networks they will encounter, or they have different goals in mind, or one is better optimized than the other. This formulation allows the implementer to choose rationally among competing options.

All that said, we have much to learn before computer-generated algorithms will have proven themselves trustworthy:

- Other than by exhaustive testing, we don't know how to predict the robustness of RemyCCs to unexpected inputs. Do they break catastrophically in such situations?
- How would a RemyCC designed for a 10,000-fold range of throughputs and RTTs perform?
- Although we are somewhat robust against a RemyCC's latching on to the peculiarities of a simulator implementation (because RemyCCs are designed within Remy but then evaluated within ns-2), we can't be certain how well RemyCCs will perform on real networks without trying them.

We believe that making congestion control a *function* of the desired ends, and the assumptions we make about the network, is the solution to allow the Internet and its subnetworks to evolve without tiptoeing around TCP's assumptions about how networks behave. But many dots need to be connected before the the Internet at large — as opposed to internal networks — might agree on a model that could be used to prepare a "one-size-fits-all" RemyCC.

7. CONCLUSION

This paper asks whether the design of distributed congestion-control algorithms for heterogeneous and dynamic networks can be done by specifying the assumptions that such algorithms are entitled to have and the policy they ought to achieve, and letting computers work out the details of the per-endpoint mechanisms.

Much future work remains before this question can be answered for the real-world Internet, but our findings suggest that this approach has considerable potential.

We developed and evaluated Remy, a program that designs end-to-end congestion-control algorithms to human-supplied specifications. Remy's outputs handily outperform the best-known techniques, including ones that require intrusive in-network changes, in scenarios where network parameters varied over one or two orders of magnitude.

Our results, and many others in the literature, indicate that there is no existing single congestion-control method that is the best in all situations. Moreover, the set of "all situations" is rapidly growing as new subnetworks and link technologies proliferate. A computer-generated approach that maximizes an explicit function of the throughput and delay to generate algorithms may be the right way forward for the networking community. Today's informal approach of hampering lower layers or providing vague advice on how best to accommodate TCP should be replaced by end-to-end algorithms (in TCP and elsewhere) that adapt to *whatever* the lower layers are doing. Remy provides a way to achieve this goal.

8. ACKNOWLEDGMENTS

We are grateful to Anirudh Sivaraman for several contributions to the simulator and for helpful discussions. We thank Leslie Kaelbling, Christopher Amato, Scott Shenker, and our shepherd, Ranjita Bhagwan. We thank Frans Kaashoek and Nickolai Zeldovich for the use of multicore machines at MIT. KW was supported by the Claude E. Shannon Research Assistantship. We thank the members of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT), including Amazon.com, Cisco, Google, Intel, Mediatek, Microsoft, ST Microelectronics, and Telefonica, for their support. This work was also supported in part by NSF grant CNS-1040072.

REFERENCES

[1] A. Akella, S. Seshan, R. Karp, S. Shenker, and C. Papadimitriou. Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP. In *SIGCOMM*, 2002.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] M. Allman. Initial Congestion Window Specification. http://tools.ietf.org/html/draft-allman-tcpm-bump-initcwnd-00, 2010.

[4] M. Allman. Comments on Bufferbloat. *ACM SIGCOMM Computer Communication Review*, 43(1), Jan. 2013.

[5] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, 1999.

[6] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *INFOCOM*, 2001.

[7] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms. In *SIGCOMM*, 2001.

[8] D. S. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840, Nov. 2002.

[9] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

[10] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[11] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. http://tools.ietf.org/html/draft-ietf-tcpm-initcwnd-08, 2013.

[12] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM*, 1988.

[13] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *ACM SIGCOMM Computer Communication Review*, 40(3):27–33, 2010.

[14] W. Feng, K. Shin, D. Kandlur, and D. Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Trans. on Networking*, Aug. 2002.

[15] S. Floyd. TCP and Explicit Congestion Notification. *CCR*, 24(5), Oct. 1994.

[16] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.

[17] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking*, 1(4), Aug. 1993.

[18] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.

[19] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996.

[20] D. Hofstadter. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Basic books, 1985.

[21] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[22] R. Jain. A Delay-based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. In *SIGCOMM*, 1989.

[23] P. Karn, C. Bormann, G. Fairhurst, D. Grossman, R. Ludwig, J. Mahdavi, G. Montenegro, J. Touch, and L. Wood. Advice for Internet Subnetwork Designers, 2004. RFC 3819, IETF.

[24] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[25] F. P. Kelly, A. Maulloo, and D. Tan. Rate Control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[26] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control Without Reliability. In *SIGCOMM*, 2006.

[27] S. Kunniyur and R. Srikant. Analysis and Design of an Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. In *SIGCOMM*, 2001.

[28] T. Lan, D. Kao, M. Chiang, and A. Sabharwal. An Axiomatic Theory of Fairness. In *INFOCOM*, 2010.

[29] D. Leith and R. Shorten. H-TCP Protocol for High-Speed Long Distance Networks. In *PFLDNet*, 2004.

[30] S. Mascolo, C. Casetti, M. Gerla, M. Sanadidi, and R. Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *MobiCom*, 2001.

[31] P. E. McKenney. Stochastic Fairness Queueing. In *INFOCOM*, 1990.

[32] D. Meagher. Geometric Modeling Using Octree Encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[33] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[34] F. A. Oliehoek. Decentralized POMDPs. In *In Reinforcement Learning: State of the Art, Adaptation, Learning, and Optimization*, pages 471–503, 2012.

[35] R. Pan, B. Prabhakar, and K. Psounis. CHOKe—A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation. In *INFOCOM*, 2000.

[36] K. K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks. *ACM Trans. on Comp. Sys.*, 8(2):158–181, May 1990.

[37] R. Srikant. *The Mathematics of Internet Congestion Control*. Birkhauser, 2004.

[38] C. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.

[39] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[40] J. Touch. Automating the Initial Window in TCP. http://tools.ietf.org/html/draft-touch-tcpm-automatic-iw-03, 2012.

[41] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). In *SIGCOMM*, 1991.

[42] D. Wei, C. Jin, S. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking*, 14(6):1246–1259, 2006.

[43] K. Winstein and H. Balakrishnan. End-to-End Transmission Control by Modeling Uncertainty about the Network State . In *HotNets-X*, 2011.

[44] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One More Bit is Enough. *IEEE/ACM Trans. on Networking*, 16(6):1281–1294, 2008.

[45] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *INFOCOM*, 2004.

[46] Y. Yi and M. Chiang. Stochastic Network Utility Maximisation. *European Transactions on Telecommunications*, 19(4):421–442, 2008.