Optimization Based Congestion Control Algorithms

Lecture 10

Subir Varma

Summary

Optimization based Congestion Control Algorithms are characterized by the fact that their window or rate adaptation rules are not pre-defined, but arise as a result of the optimization of an Utility Function.

Some recently proposed optimization based algorithms fall into the following classes:

- Offline Optimization: These algorithms use a network model which is used to define a Global Utility Function. The set of Congestion Control parameters are those that maximize this Utility Function.
- Online Optimization: These algorithms are Model Free. They do define a Local Utility Function, whose maximization leads to the window adaptation rules.
  - Deep Reinforcement Learning based algorithms fall into this category

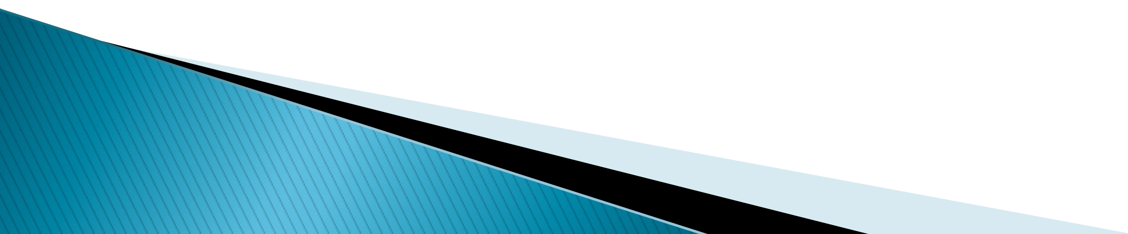
Optimization Based Algorithms

- Global Optimization
  - Remy: TCP ex Machina: Computer Generated Congestion Control, Winstein and Balakrishnan, (2013)
  - Indigo: Pantheon: The Training Ground for Internet Congestion Control Research, Yan et.al. (2018)

- Local Optimization
  - PCC Allegro: Re-Architecting Congestion Control for Consistent High Performance, Dong et.al. (2015)
  - PCC Vivace: Online Learning Congestion Control, Dong et.al. (2018)

- Deep RL Based Algorithms
  - TCP Aurora: A Deep RL Perspective on Internet Congestion Control: Jay et.al. (2019)
  - Symbolic Distillation for Learned TCP CC: Sharan et.al. (2022)

Global Optimization Remy and Indigo



Recall from Lecture 3

If each source performs local optimization by solving the problem

$$\max_{r_i}[U_i(r_i) - r_i q_i].$$

Where $U_i(r_i)$ is the utility that the source attains as a result of transmitting at rate r_i, and q_i is price per unit data that it is charged by the network.
Then this procedure also solves the following Global Optimization Problem

Find the rates r_i^{max} such that

$$\max_{r \geq 0} \sum_{i=1}^{N} U_i(r_i), \quad \textit{subject to}$$

$$Xr \leq C.$$

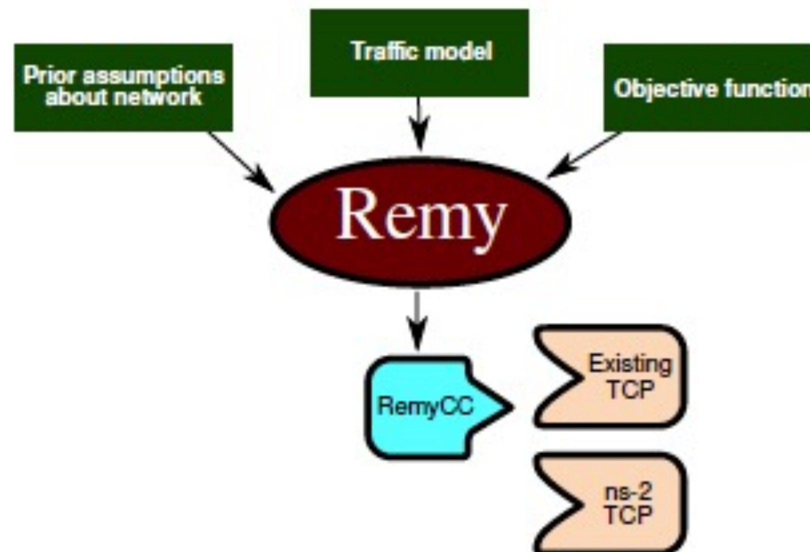
For TCP Reno, the Utility Function was derived using its rate control dynamics as

$$U_i(r_i) = \int \frac{\frac{1}{T_i^2}}{(\frac{1}{T_i^2} + \frac{2}{3}r_i^2)} dr_i = \frac{\sqrt{3/2}}{T_i} \tan^{-1}\left(\sqrt{\frac{2}{3}} r_i T_i\right) \approx -\frac{1.5}{T_i^2 r_i}$$

Can we reverse this procedure, i.e., start from a Utility Function and then find the rate control rule that maximizes it?

Project Remy

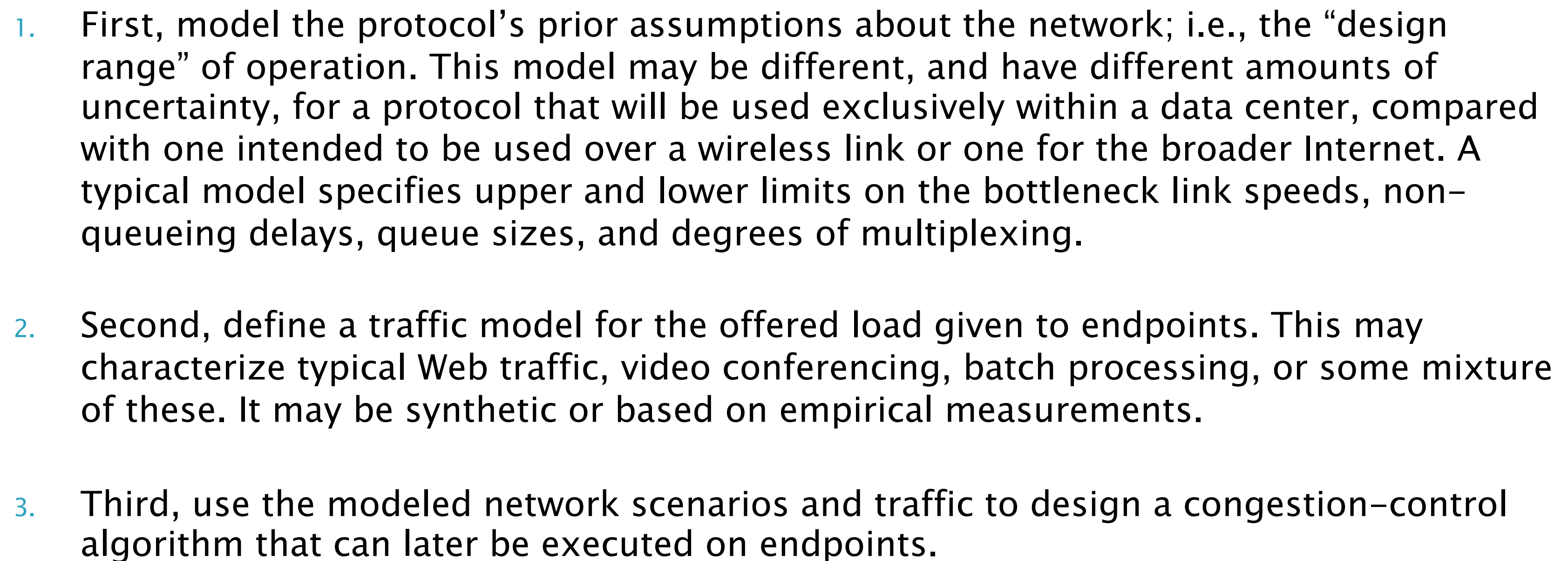
- Is it possible for a computer to “discover” the right rules for congestion control in heterogeneous and dynamic networks?
- Rather than manually formulate each endpoint’s reaction to congestion signals, as in traditional protocols, the protocol designer specifies their prior knowledge or assumptions 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.



“TCP ex Machina: Computer-Generated Congestion Control”, Winstein and Balakrishnan

Project Remy: Methodology

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 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.

Project Remy: Traffic Model and Objective Function

- Traffic Model: Remy models the offered load as a stochastic process that switches unicast flows between sender-receivers pairs on or off. 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).
- Objective Function: 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

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

Where

$$U_\alpha(x) = \frac{x^{1-\alpha}}{1-\alpha}.$$

the parameters α and β set the tradeoff between fairness and efficiency and and δ expresses the relative importance of delay vs. throughput.

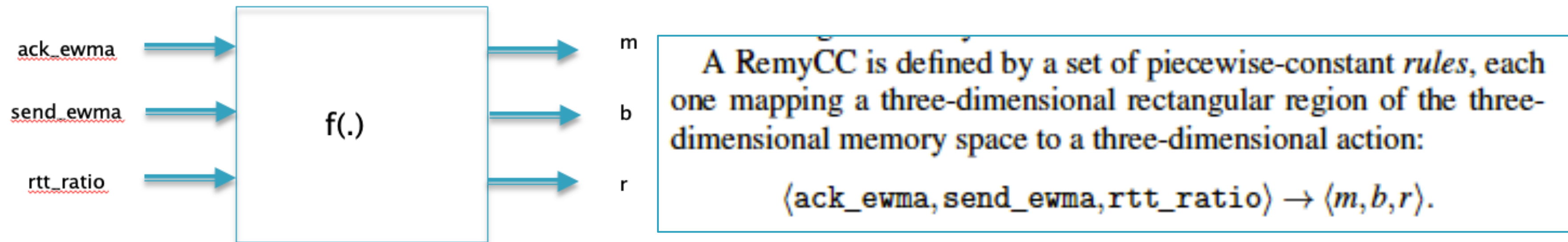
RemyCC State Variables

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).

- 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.

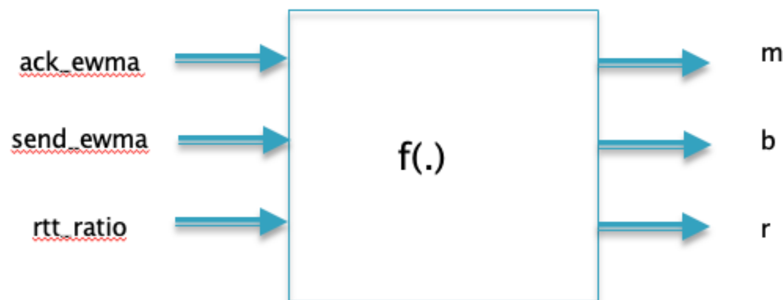
RemyCC Mapping Function



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.

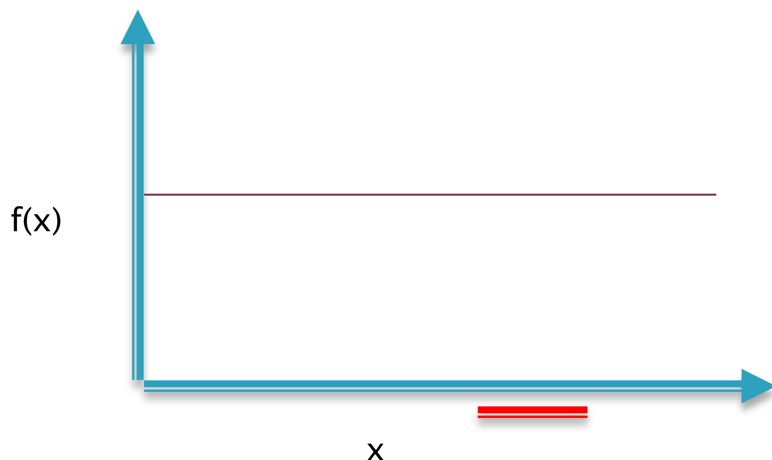
- Each time a RemyCC sender receives an ACK, it updates its memory and then looks up the corresponding action.
- Remy pre-computes this lookup table during the design phase, by finding the mapping f(.) that maximizes the expected value of the objective function, with the expectation taken over the network model.
- 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.

Remy Automated Design Procedure



- Find the function f(.) that maximizes the expected network utility function.
- This problem falls within the framework of a machine learning model (regression)
- However, we don't know have the ground truth data to train the network

Remy Solution: Use simulations



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.

Simulate the current RemyCC and see which rule in the current epoch receives the most use (this is a function of the network and traffic model)

Focus on this rule and find the best action for it using brute force search. The modified action is evaluated by substituting it into all senders and repeating the simulation in parallel.

Comparison with other Protocols

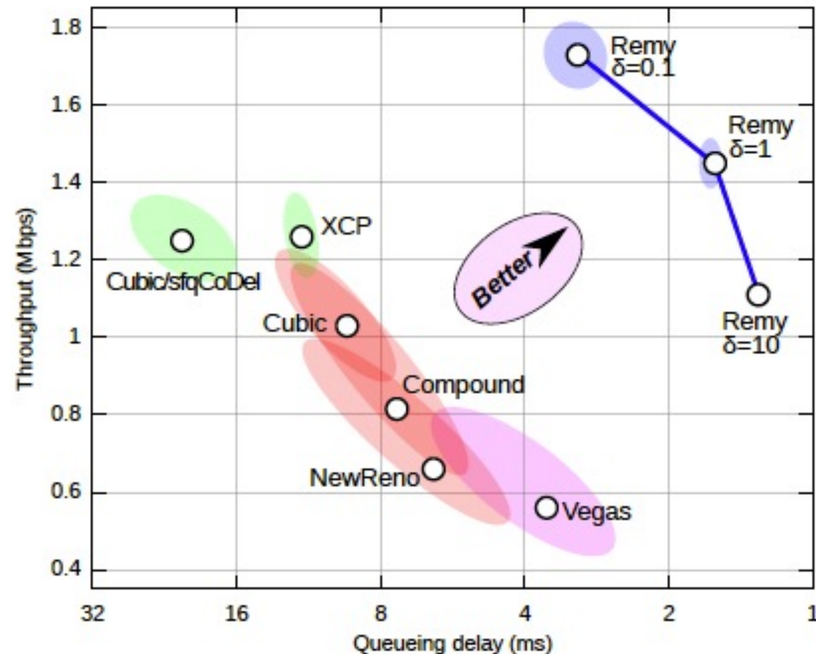


Figure 4: Results for each of the schemes over a 15 Mbps dumbbell topology with $n = 8$ senders, each alternating between flows of exponentially-distributed byte length (mean 100 kilobytes) and exponentially-distributed off time (mean 0.5 s). Medians and 1-σ ellipses are shown. The blue line represents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

Indigo

- RemyCC did not employ Supervised Learning as a way to learn the Mapping Function, even though it seems to be a natural fit to the problem.
- Keith Winstein (the author of Remy) led another team a few years later, which came up with a protocol called Indigo that indeed employed Supervised Learning.
- Indigo was trained on a network model that was an emulation of real world data gathered on a measurement tool called Pantheon.

“Pantheon: the training ground for Internet congestion-control research,” Yan et.al.

Indigo

- Just like Remy, Indigo does two things: it observes the network state, and it adjusts its congestion window every 10 ms, i.e., the allowable number of in-flight packets. Observations occur each time an ACK is received, and their effect is to update Indigo's internal state.

The State Vector is:

- An exponentially-weighted moving average (EWMA) of the queuing delay, measured as the difference between the current RTT and the minimum RTT observed during the current connection.
- An EWMA of the sending rate, defined as the number of bytes sent since the last ACK'ed packet was sent, divided by the RTT.
- An EWMA of the receiving rate, defined as the number of bytes received since the ACK preceding the transmission of the most recently ACK'ed packet, divided by the corresponding duration (similar to and inspired by TCP BBR's delivery rate).
- The current congestion window size.
- The previous action taken.

The Actions are the congestion window size

"Pantheon: the training ground for Internet congestion-control research," Yan et.al.

Indigo

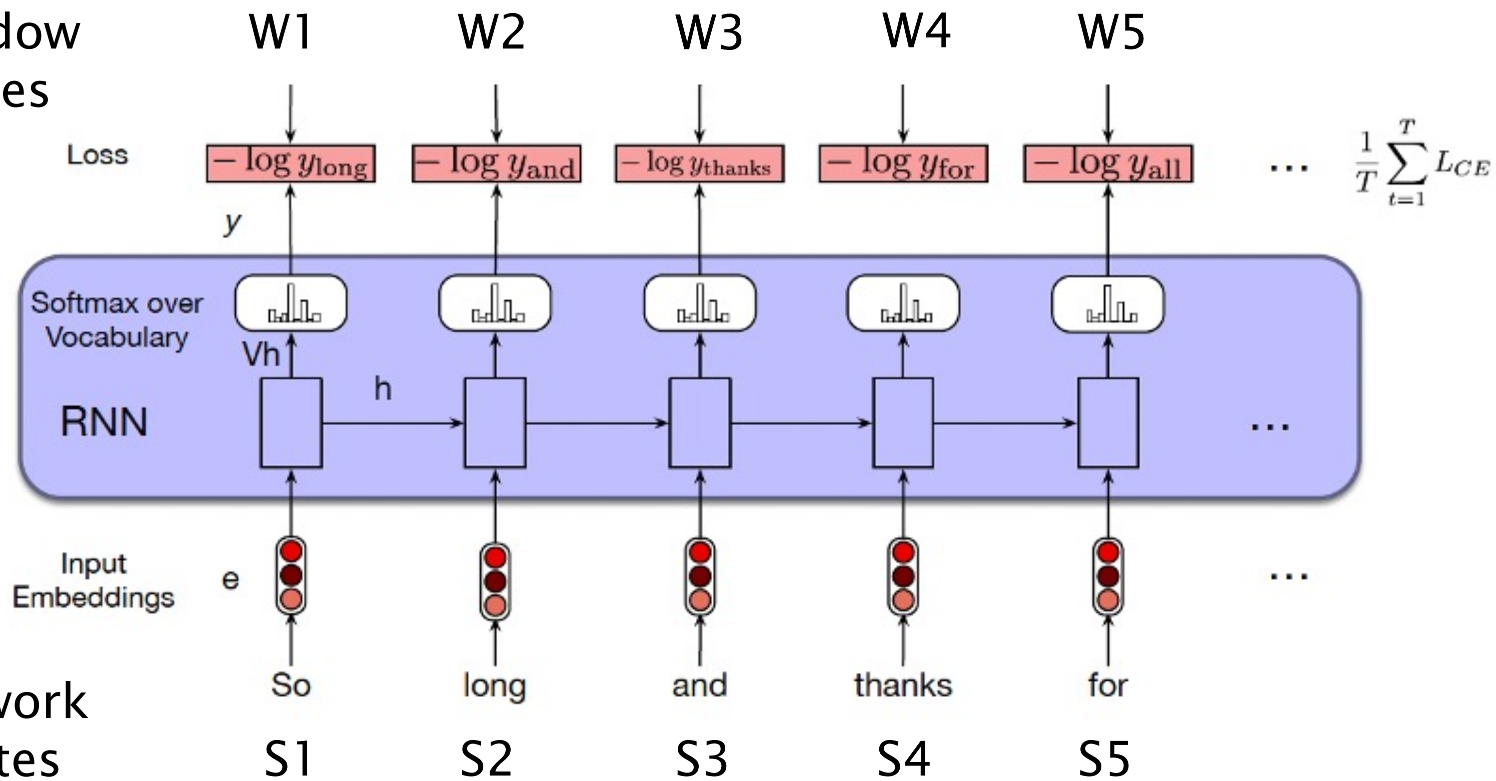
- Indigo stores the mapping from states to actions in a Long Short-Term Memory (LSTM) recurrent neural network with 1 layer of 32 hidden units.

- Indigo requires a training phase in which, it learns a mapping from states to actions. Once trained and deployed, this mapping is fixed.

- Indigo uses an imitation learning to train its neural network, called DAgger.
  - Generate one or more congestion-control oracles, idealized algorithms that perfectly map states to correct actions, corresponding to links on which Indigo is to be trained.
  - Then apply a standard imitation learning algorithm that use these oracles to generate training data.
  - The best congestion window is usually given by a link's bandwidth delay product per flow.
  - The best window for a given state was obtained by experimenting with an emulator with various window sizes. Start from the BDP and then search near this value to find the best window.

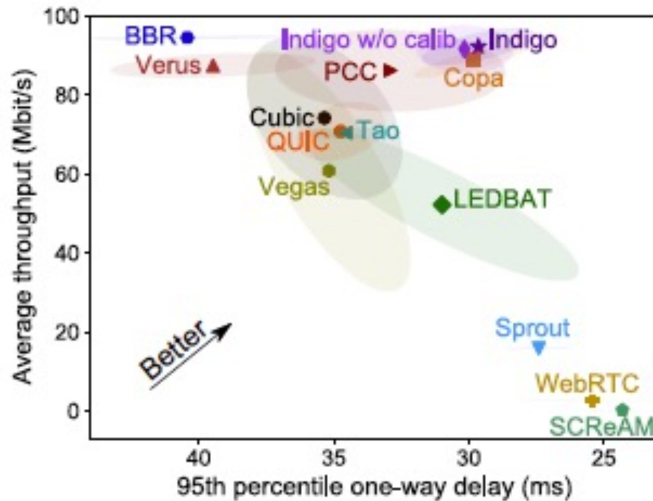
- Dagger operation:
  - First, it allows the neural network to make a sequence of congestion-control decisions on the training link's emulator, recording the state vector that led to each decision.
  - Next, it uses the congestion-control oracle to label the correct action corresponding to each recorded state vector.
  - Finally, it updates the neural network by using the resulting state-action mapping as training data. This process is repeated until further training does not change the neural network.

Training an LSTM Model

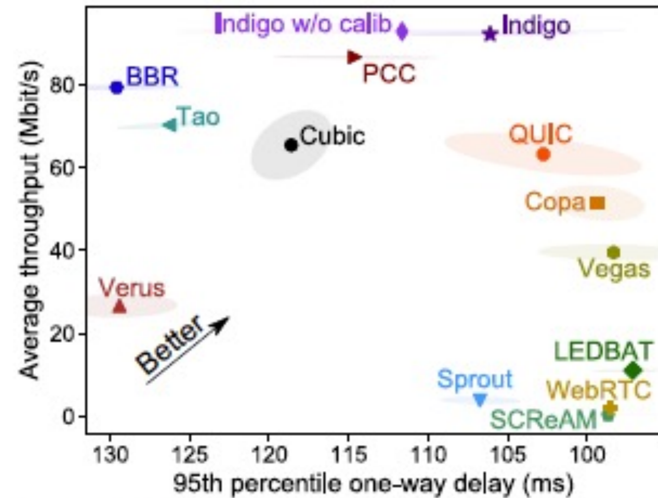


New Idea: The window size is now a function of the history of the states vs the current state only.

Indigo Performance



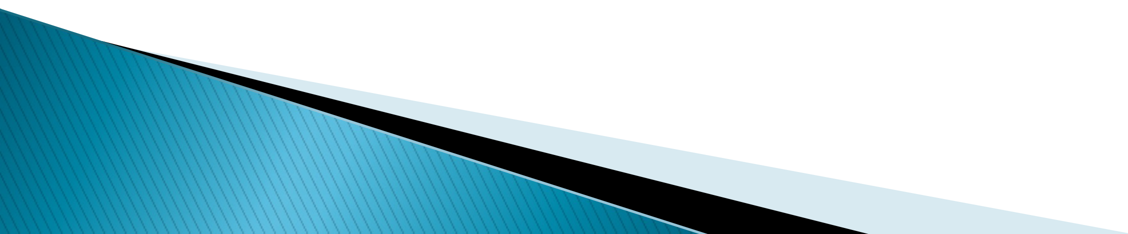
(a) Mexico to AWS California, 10 trials. P1272.



(b) AWS Brazil to Colombia, 10 trials. P1439.

Local Optimization

PCC Allegro



Critique of Existing Algorithms

- Existing rate control algorithms do what is referred to as hardwired mapping: certain predefined packet-level events are hardwired to certain predefined control responses.
- A hardwired mapping has to make assumptions about the network.
- It is fundamentally hard to formulate an "always optimal" hardwired mapping in a complex real-world network because the actual optimal response to an event like a loss (i.e. decrease rate or increase? by how much?) is sensitive to network conditions.
- Moreover if the hardwired control actions are indeed harming performance, the performance of the protocols can potentially "jump off the cliff", because they do not notice the control action's actual effect on performance.

  Congestion Control Algo does not monitor its own performance and take action based on that
- Modern networks have an immense diversity of conditions that add complexity far beyond what can be summarized by the relatively simplistic assumptions embedded in a hardwired mapping.

Example: Reno makes the assumption that a packet loss is due to congestion rather than link error OR
BBR makes the assumption that a packet loss is due to link error not congestion

"PCC: Re-architecting Congestion Control for Consistent High Performance", Dong et.al

PCC Allegro: Performance Oriented Congestion Control

- PCC's goal is to understand what rate control actions improve performance based on live experimental evidence, avoiding TCP's assumptions about the network.
- PCC sends at a rate r for a short period of time, and observes the results (e.g. SACKs indicating delivery, loss, and latency of each packet).
- It aggregates these packet-level events into a utility function that describes an objective like "high throughput and low loss rate". The result is a single numerical performance utility u.
- At this point, PCC has run a single "micro-experiment" that showed sending at rate r produced utility u.

Rate based control

PCC Allegro: Performance Oriented Congestion Control

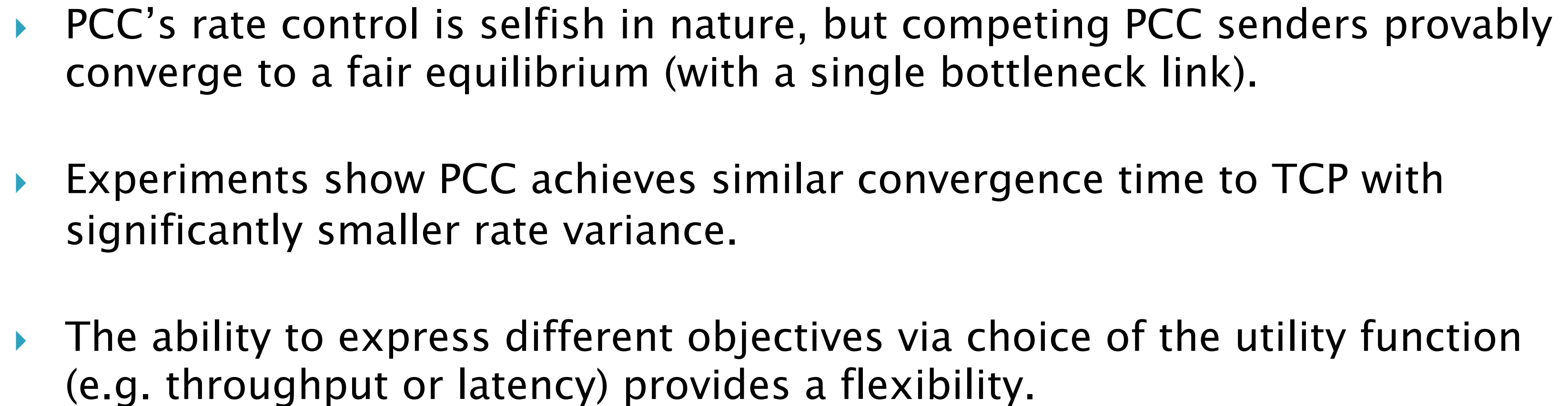
- To make a rate control decision, PCC runs multiple such micro-experiments: it tries sending at two different rates, and moves in the direction that empirically results in greater performance utility.

- This is effectively A/B testing for rate control and is the core of PCC's decisions. PCC runs these micro-experiments continuously (on every byte of data, not on occasional probes), driven by an online learning algorithm that tracks the empirically-optimal sending rate.

- Thus, rather than making assumptions about the potentially-complex network, PCC adopts the actions that empirically achieve consistent high performance.

This process has some similarities to choosing the parameters of a Neural Network, with the utility u serving as the Loss Function for the network. However note that each sender is separately optimizing its own rate.

PCC Allegro



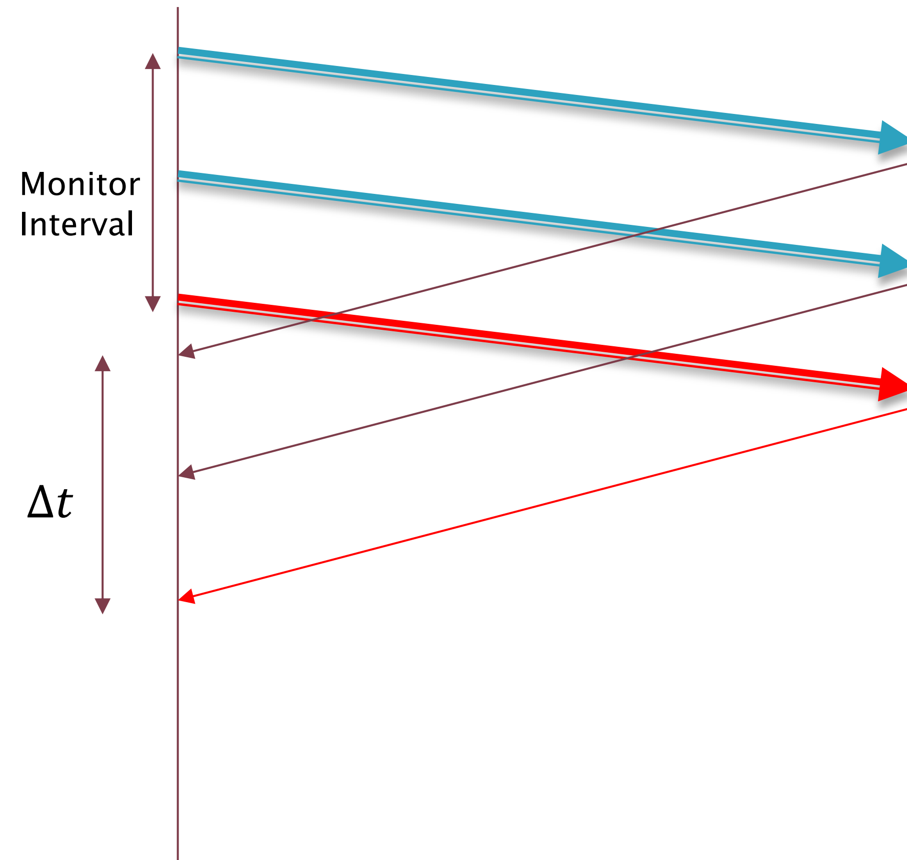
- PCC’s rate control is selfish in nature, but competing PCC senders provably converge to a fair equilibrium (with a single bottleneck link).
- Experiments show PCC achieves similar convergence time to TCP with significantly smaller rate variance.
- The ability to express different objectives via choice of the utility function (e.g. throughput or latency) provides a flexibility.

PCC Allegro Micro Experiment

- PCC divides time into continuous time periods, called monitor intervals (MIs), whose length is normally one to two RTTs.
- In each MI, PCC tests an action: it picks a sending rate, say r, and sends data at rate r through the interval.
- After about an RTT, the sender will see selective ACKs (SACK) from the receiver, just like TCP. However, it does not trigger any predefined control response. Instead, PCC aggregates these SACKs into meaningful performance metrics including throughput, loss rate and latency.
- These performance metrics are combined to a numerical utility value u.
- It changes its rate in the direction that has higher utility
- It continues in this direction as long as the utility continues increasing.
- If utility falls, it returns to a decision making state where it again tests both higher and and lower rates to find which produces higher utility.

PCC runs these micro-experiments continuously

PCC Allegro Micro Experiment



$$x_i = \frac{\Delta delivered}{\Delta t}$$

- Send data at rate r during the monitor interval
- Monitor the resulting delivery rate x
- Monitor the fraction L of packets that were lost

Compute Utility Function

$$u_i(x_i) = T_i \cdot Sigmoid_\alpha(L_i - 0.05) - x_i \cdot L_i$$

where

$$T_i = x_i(1 - L_i)$$

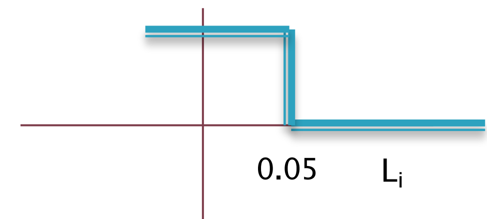
Allegro Utility Function

- We assume n PCC senders 1, . . . ,n send traffic across a bottleneck link of capacity $C > 0$.
- Each sender i chooses its sending rate x_i to optimize its utility function u_i.
- We choose a utility function expressing the common application-level goal of “high throughput and low loss”:

$$u_i(x_i) = T_i \cdot Sigmoid_\alpha(L_i - 0.05) - x_i \cdot L_i$$

Fraction of lost data rate

x_i: Sender i’s sending rate
L_i: Observed data loss rate
$T_i = x_i(1 - L_i)$: Sender i’s throughput
$Sigmoid_\alpha(y) = \frac{1}{1+e^{\alpha y}}$ for some $\alpha > 0$



- What about end-to-end latency?
- This is a matter of just changing the utility function.
- Different sources can have different utility functions and co-exist in the same network.

Allegro Utility Function

- The sigmoid function makes the utility < 0 for $L_i > 0.05$, thus preventing the rate x_i from increasing further.
- This prevents a congestion collapse when the losses are due to buffer overflows.
- If the losses are due to random link related events, then as long as they are under 5% The protocol does not react to them.
- If $> 5\%$ then it clamps down on the data rate.

Potential Issues:
- Buffferbloat is still happening
- Leads to high packet loss upon convergence
- Un-Fairness with Loss Based Variants
- u_i does not take delay into account

PCC Allegro: Handling Noise in the Measurement Process

- The network might be changing over time for reasons unrelated to the sender's action. This adds noise to the decision process.

- To improve PCC's decisions use multiple randomized controlled trials (RCTs):
  Rather than running two tests (one each at 100 and 105 Mbps), we conduct four in randomized order—e.g. perhaps (100,105,105,100). PCC only picks a particular rate as the winner if utility is higher in both trials with that rate.

- If results are inconclusive, so each rate "wins" in one test, then PCC maintains its current rate, and may have reached a local optimum

Comparison with BBR

- Both are rate based protocols.
- Both BBR and PCC Allegro carry out experiments to probe for more bandwidth. BBR changes its rate directly, while PCC Allegro uses its utility function to make this decision.
- BBR reduces its rate based on its current measurement of delivery rate, while PCC Allegro uses its micro-experiments where it explicitly reduces sending rate and then uses uses the resulting utility function to decide whether to reduce rate.
- BBR does not react to packet losses, while PCC Allegro uses this information to set its sending rate.
- BBR tries to limit the queueing in the network, while PCC Allegro does not have a mechanism for doing so. Allegro keeps increasing its sending rate until congestion losses exceed 5% and then it clamps down.
- They both use MIMD rate control. PCC Allegro researchers claim that Allegro does not suffer from unfairness inspite of this, since the senders make independent rate change decisions (vs loss based systems where the the window control gets synchronized).
- Unlike BBR, PCC Allegro does not incorporate a window to limit the in-flight data.

Other Properties

- What about end-to-end latency?
  - Bufferbloat is still a problem
- What about fairness in the presence of loss based protocols?
  - The utility function $u_i(x_i) = T_i \cdot Sigmoid_\alpha(L_i - 0.05) - x_i \cdot L_i$

    ignores losses below 5%, and hence is not fair to loss based protocols.
  - However other utility functions may be able to do this.
- Allegro causes high packet loss rate. If the available bandwidth decreases, since the source keeps transmitting at the higher rate, there is danger of overwhelming the network. In loss based protocols such as Reno, this issue is controlled by means of its window (also used in BBR).
- Intra-protocol RTT fairness
  - PCC Allegro shows a high degree of fairness for flows with differing RTTs.

Performance

Does not do very well with fluctuating bandwidth

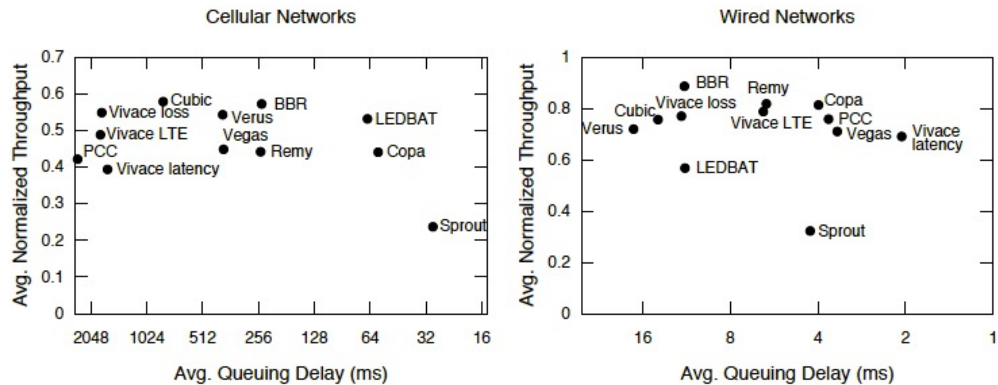


Figure 5: Real-world experiments on Pantheon paths: Average normalized throughput vs. queuing delay achieved by various congestion control algorithms under two different types of Internet connections. Each type is averaged over several runs over 6 Internet paths. Note the very different axis ranges in the two graphs. The *x*-axis is flipped and in log scale. Copa achieves consistently low queueing delay and high throughput in both types of networks. Note that schemes such as Sprout, Verus, and Vivace LTE are designed specifically for cellular networks. Other schemes that do well in one type of network don't do well on the other type. On wired Ethernet paths, Copa's delays are 10× lower than BBR and Cubic, with only a modest mean throughput reduction.

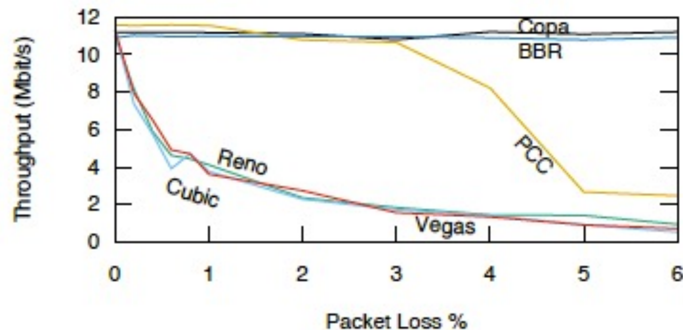
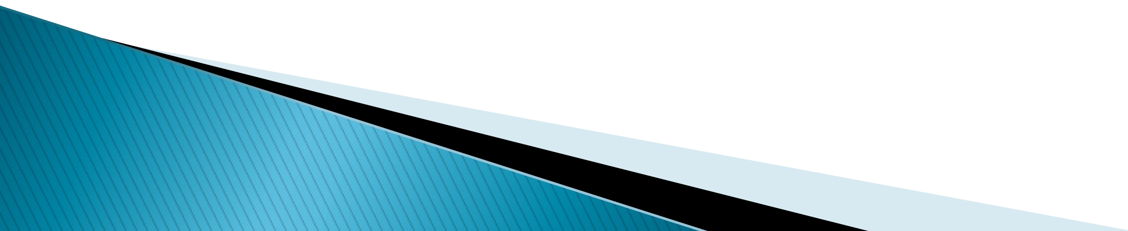


Figure 7: Performance of various schemes in the presence of stochastic packet loss over a 12 Mbit/s emulated link with a 50 ms RTT.

Local Optimization
PCC Vivace



PCC Vivace

- Vivace also uses the PCC framework, but tries to solve the problems with PCC Allegro, with the following changes
  - Vivace has a utility function framework to replace the adhoc function in Allegro. The new utility function Incorporates latency awareness, thus mitigating the bufferbloat problem and the resulting packet loss and latency inflation.
  - Vivace incorporates a learning rate-control algorithm. This provides faster, more stable convergence, and reacts more quickly upon changes to network conditions.
  - The Vivace framework extends to heterogeneous senders with different utility functions, enabling flexible network-resource allocation.
  - Vivace induces more friendly behavior towards TCP, and thus is better suited for real-world deployment.

"PCC Vivace: Online-Learning Congestion Control", Dong et.al.

Vivace Utility Function

- As in Allegro, Vivace divides time into consecutive Monitor Intervals (MIs). At the end of each MI, sender i applies the following utility function to transform the performance statistics gathered at that MI to a numerical utility value

$$u = x_i^a - b x_i \frac{d(RTT_i)}{dt} - c x_i L_i$$

where $0 < a < 1$, $b \geq 0$, $c > 0$, are constants, x_i is sender i's sending rate and L_i is its observed loss rate.

- The term $\frac{d(RTT_i)}{dt}$ is the observed "RTT gradient" during this MI, i.e., the increase in latency experienced within this MI.
  This term is proportional to the rate at which the buffer size is increasing, and allows the sender to detect build-up of congestion before the absolute buffer size estimate.
  This is similar to the TIMELY protocol from Lecture 9 (and the P+I protocol in Lecture 3).

- Utility functions of the above form reward increase in throughput (via x_i^a), and penalize increase in both latency and loss.

Stability and Fairness

- When a ≤ 1, the family of utility functions falls into the category of "socially-concave" in game theory.

- A utility function within this category, when coupled with a theoretical model of Vivace's online learning rate-control scheme, guarantees high performance from the individual sender's perspective and ensures quick convergence to a global rate configuration.

- Let C denote the capacity of the bottleneck link. If

  $$b \geq an^{2-a}C^{a-1}$$

  where n is the number of flows sharing the bottleneck link, then the latency in equilibrium is the base RTT.

Vivace Rate Control

- Suppose the current sending rate is r. Then, in the next two MIs, the sender will test the rates $r(1+\epsilon)$ and $r(1-\varepsilon)$, compute the corresponding numerical utility values, u_1 and u_2, respectively, and estimate the gradient of the utility function to be

$$\gamma = \frac{u_2 - u_1}{2\epsilon r}$$

- Change in rate is given by

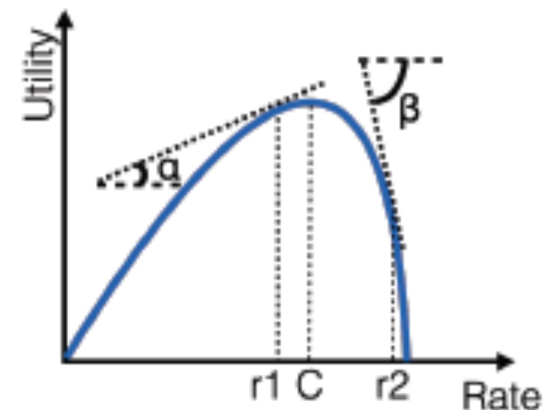
$$r \leftarrow r + \theta_o \gamma$$

where θ_o is the learning rate.

Optimization of u using Gradient Ascent

This allows Vivace to vary the step-size used while making rate changes. For example:

- At rate r_1 use a small step-size
- At rate r_2 use a large step size.



Vivace Rate Control: Confidence Multiplier

- Vivace varies the learning rate by using a "confidence multiplier" $m(\tau)$, so that

  $$r \leftarrow r + m(\tau)\theta_o\gamma$$

  After a sender makes τ consecutive decisions to change the rate in the same direction, θ is set to $m(\tau)\theta_o$.

- When the direction at which rate is adapted is reversed (increase to decrease or vice-versa), τ is set back to 0 (and the above process starts anew).

- The confidence amplifier is a monotonically nondecreasing function that assigns a real value $m(\tau)$ to any integer $\tau \geq 0$.

Vivace Rate Control: Dynamic Change Boundary

- Sampled utility-gradient can be excessively high due to unreliable measurements or large changes to network conditions between MIs.

- To prevent a huge sudden change in rates, the following mechanism is used:
  - Whenever the computed rate change Δ_r exceeds ωr, the effective rate change is capped at ωr.
  - The value of ω is itself varied according to the rule. ω is updated according to the rule

$$\omega = \omega_0 + k \cdot \delta$$

    following k consecutive rate adjustments in which the gradient-based rate-change Δ_r exceeded the dynamic change boundary, for a predetermined constant $\delta > 0$.
  - Whenever $\Delta_r \leq r\omega$, Vivace recalibrates the value of k in the formula $\omega = \omega_0 + k\delta$ to be the smallest value for which $\Delta_r \leq r\omega$.
  - k is reset to 0 when the direction of rate adjustment changes (e.g., from increase to decrease).

RTT Gradient Estimation

- The RTT gradient $\frac{d(RTT_i)}{dT}$ in MI could be estimated by quantifying the RTT experienced by the first packet and the last packet sent in that MI. To estimate the RTT gradient more accurately, Vivace utilizes linear regression.
- It assembles the 2-dimensional data set of (sampled packet RTT, time of sampling) for the packets in a MI, and uses the linear-regression-generated slope as the RTT gradient.

Reacting to Random Loss

- We say a rate-control protocol is p-loss-resilient if that protocol does not decrease its sending rate under random loss rate of at most p.
- For Vivace to be p-loss-resilient, we need to set c in the its utility function framework to be

$$c = \frac{aC^{a-1}}{p}$$

- There is an interesting trade-off involved in having a larger value for p.
  In a system with n sources that are experiencing a congestion related loss L, the value of L increases with p.

$$u = x_i^a - bx_i \frac{d(RTT_i)}{dt} - cx_i L_i$$

TCP Interaction

$$u = x_i^a - bx_i \frac{d(RTT_i)}{dt} - cx_iL_i$$

- How can Vivace be both latency sensitive and also avoid being over-whelmed by loss based algorithms such as CUBIC

- Consider the scenario that a Vivace sender is the only sender on a certain link. It tries out two rates that exceed the link's bandwidth, and the buffer for that link is not yet full.
  - Vivace's utility function will assign a higher value to the lower of these rates, since the achieved goodput and loss rate are identical to those attained when sending at the higher rate, but the latency gradient is lower.
  - Thus, in this context, the Vivace sender behaves in a latency-sensitive manner and reduces its transmission rate.

- Now, consider the scenario that the Vivace sender is sharing a link that is already heavily utilized by many loss-based protocols like TCP CUBIC and the buffer is, consequently, almost always full.
  - When testing different rates, the Vivace sender will constantly perceive the latency gradient as roughly 0, and thus disregard latency and compete against the TCP senders over the link capacity, effectively transforming into a loss-based protocol.

Performance

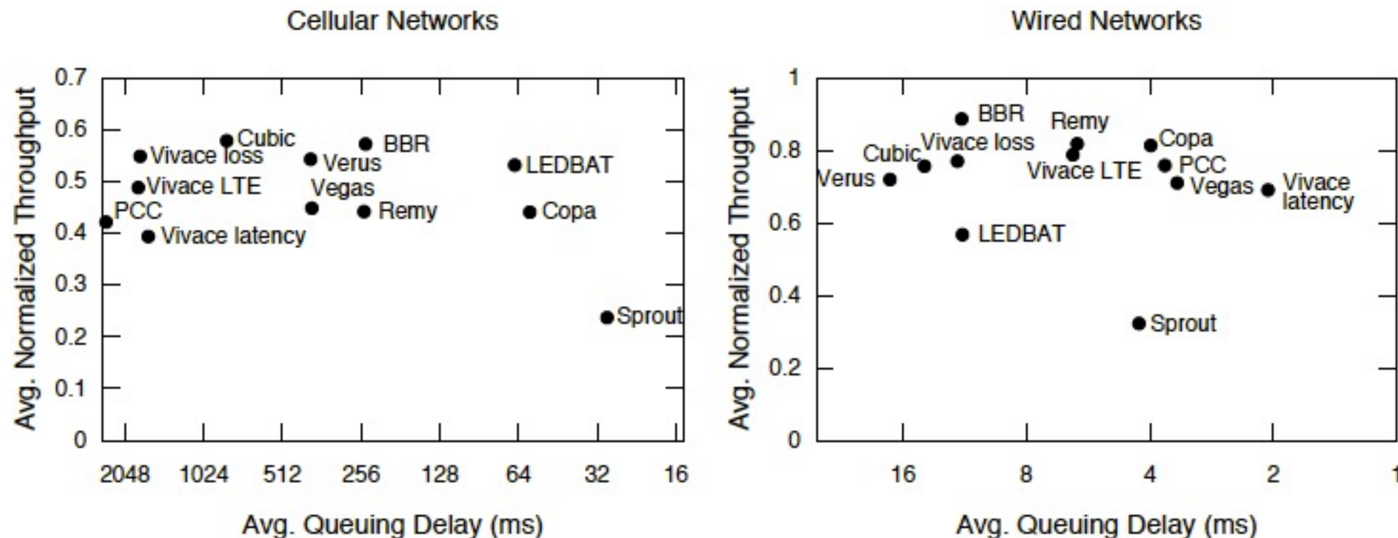
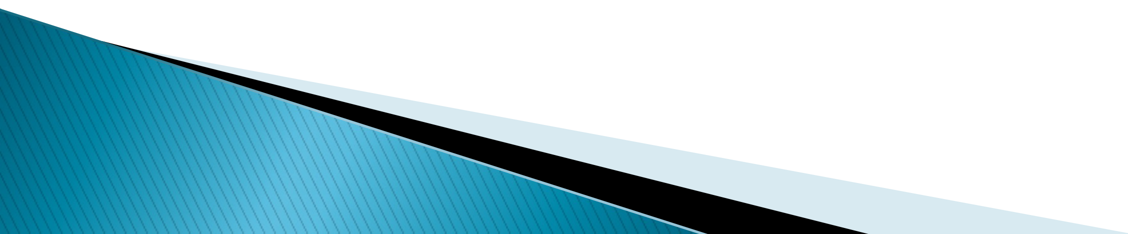


Figure 5: Real-world experiments on Pantheon paths: Average normalized throughput vs. queuing delay achieved by various congestion control algorithms under two different types of Internet connections. Each type is averaged over several runs over 6 Internet paths. Note the very different axis ranges in the two graphs. The *x*-axis is flipped and in log scale. Copa achieves consistently low queueing delay and high throughput in both types of networks. Note that schemes such as Sprout, Verus, and Vivace LTE are designed specifically for cellular networks. Other schemes that do well in one type of network don't do well on the other type. On wired Ethernet paths, Copa's delays are 10× lower than BBR and Cubic, with only a modest mean throughput reduction.

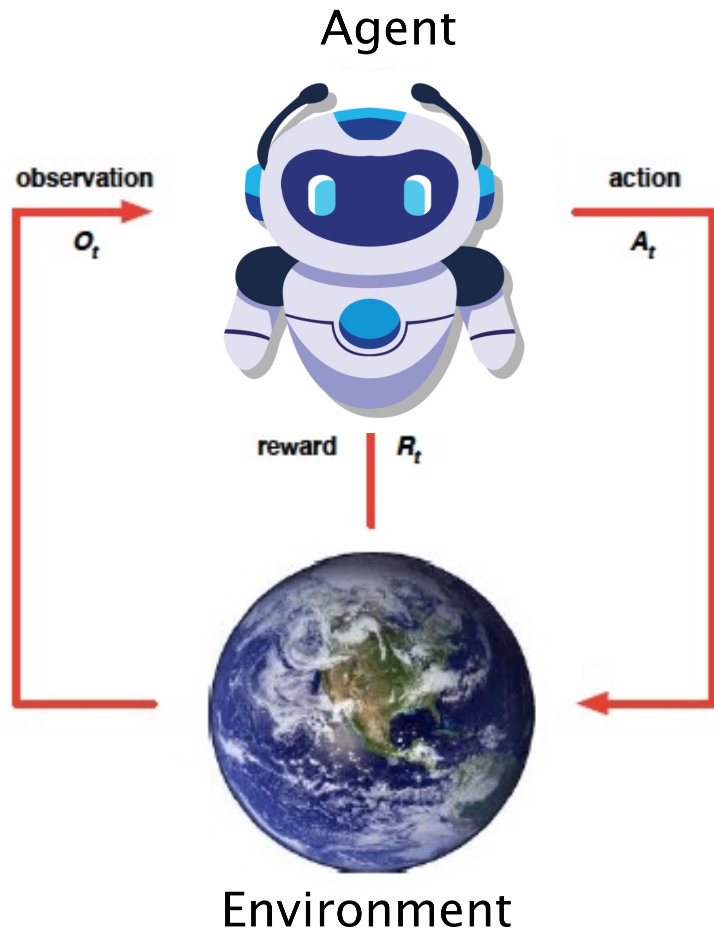
Deep Reinforcement Learning based Congestion Control



DRL based Congestion Control

- Objective: To provide a fully-automated mechanism to train a DRL agent by interacting with a real-world network environment, and to avoid hand-tuned heuristics as much as possible.
- A DRL agent uses both the states observed from the environment and a scalar reward to train its deep neural network model, which is the agent's strategy that it uses to produce an action, called its policy.
- In the context of congestion control, the action to be taken by the DRL agent may be an increase in the sending rate or the size of the congestion window.
- The objective of the DRL agent is to train a policy that maximizes the expected cumulative reward.
- Can a well-designed reward function and a curated set of states be used to train the DRL agent effectively by learning from the actual network environment, more so than handtuned heuristics?

RL Framework

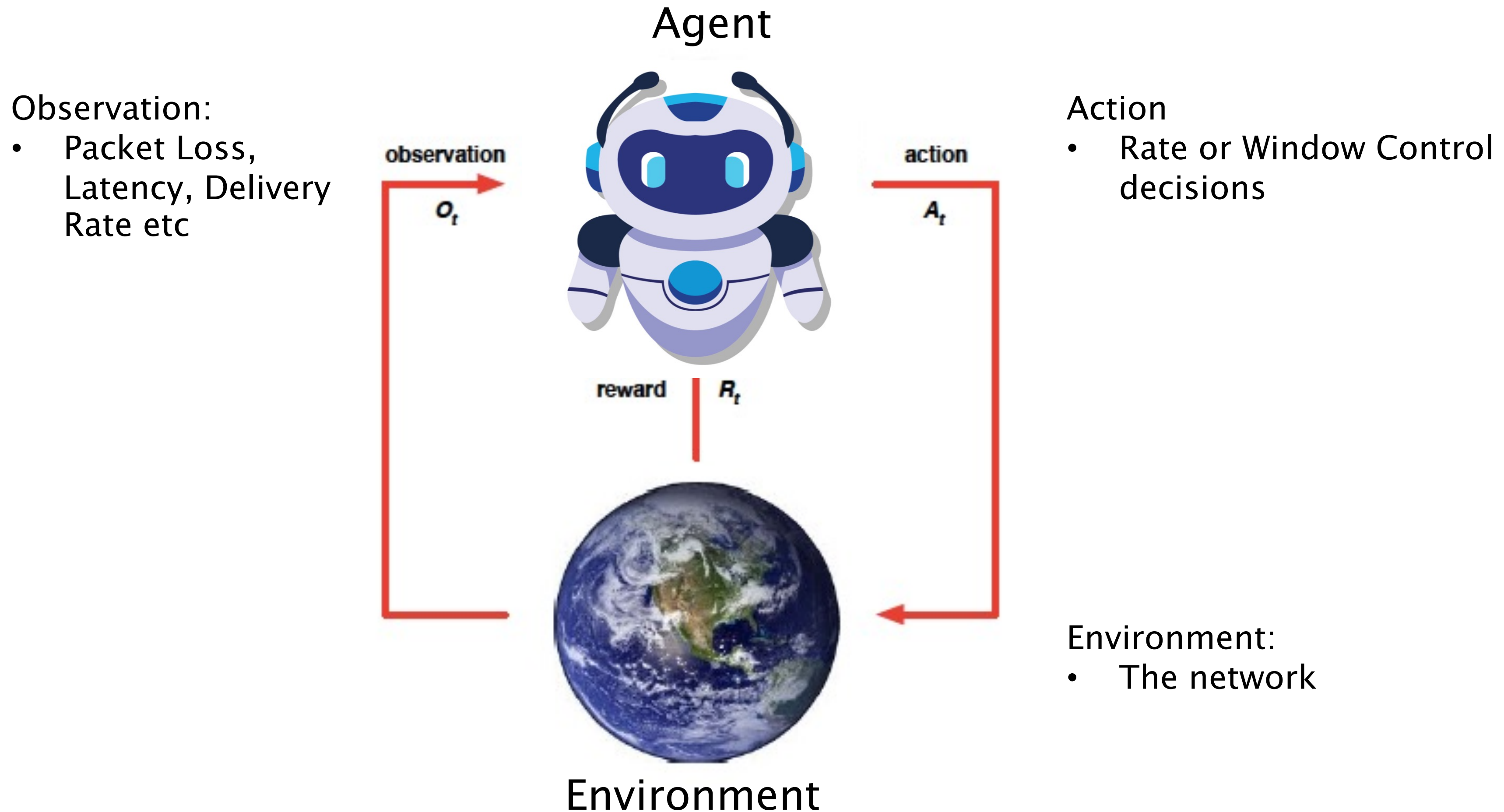


State: Summary of the past observations that the Agent uses to choose the next Action

- At each step t the agent:
  - Executes action A_t
  - Receives observation O_t
  - Receives scalar reward R_t
- The environment:
  - Receives action A_t
  - Emits observation O_{t+1}
  - Emits scalar reward R_{t+1}
- t increments at env. step

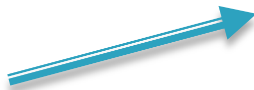
Agent has no control over the Environment's response

RL Framework for Congestion Control



RL Objective: Maximize Rewards

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$



Update made at end of an Episode

Model Free
Sample Sweep
Sample Backup

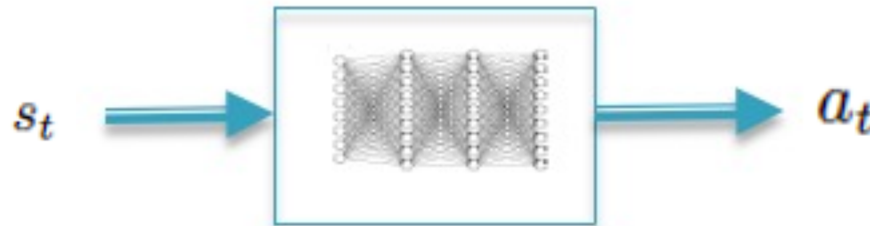
S_t S1 4 S2 3 S1 2

Instead of a Model, we now have sample episodes from the MDP

Aurora: Actions

- Framework: Aurora adopts the notion of monitor intervals (MIs) from PCC. Time is divided into consecutive intervals. In the beginning of each MI t, the sender can adjust its sending rate x_t, which then remains fixed throughout the MI.
- Actions
  - Actions are expressed as changes to the current rate

$$x_t = \begin{cases} x_{t-1} * (1 + \alpha a_t) & a_t \geq 0 \\ x_{t-1} / (1 - \alpha a_t) & a_t < 0 \end{cases}$$



Dense Feedforward Neural Network
with 2 Hidden Layers (32 and 16 neurons respectively)

Model trained using the PPO Algorithm

Model trained using a network simulator. Testing done on an actual network

Aurora: State and Reward Function

- State: Aurora collects the following Statistics Vectors:
  - Latency Gradient (as in PCC Vivace), the derivative of latency with respect to time; latency ratio (as in Remy),
  - The ratio of the current MI's mean latency to minimum observed mean latency of any MI in the connection's history; and
  - Sending ratio, the ratio of packets sent to packets acknowledged by the receiver.
  - The State is a function of a fixed-length history of the above statistics vectors collected from packet acknowledgements sent by the receiver. It is defined as

$$s_t = (v_{t-(k+d)}, \ldots, v_{t-d}),$$

    for a predetermined constant $k > 0$ and a small number d representing the delay between choosing a sending rate and gathering results.
- Rewards: Chosen as

$$10 * throughput - 1000 * latency - 2000 * loss$$

  where throughput is measured in packets per second, latency in seconds, and loss is the proportion of all packets sent but not acknowledged. The scale of each factor was chosen to force models to balance throughput and latency for the chosen training parameters