2016 International Conference on Distributed Computing in Sensor Systems

Learning-based and Data-driven TCP Design for Memory-constrained IoT

Wei Li*, Fan Zhou†, Waleed Meleis‡ and Kaushik Chowdhury§
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, USA
Email: {*li.wei, †zhou.fan1}@husky.neu.edu, {‡meleis, §krc}@ece.neu.edu

***Abstract*—Advances in wireless technology have resulted in pervasive deployment of devices of a high variability in form factors, memory and computational ability. The need for maintaining continuous connections that deliver data with high reliability necessitate re-thinking of conventional design of the transport layer protocol. This paper investigates the use of Q-learning in TCP *cwnd* adaptation during the congestion avoidance state, wherein the classical alternation of the window is replaced, thereby allowing the protocol to immediately respond to previously seen network conditions. Furthermore, it demonstrates how memory plays a critical role in building the exploration space, and proposes ways to reduce this overhead through function approximation. The superior performance of the learning-based approach over TCP New Reno is demonstrated through a comprehensive simulation study, revealing 33.8% and 12.1% improvement in throughput and delay, respectively, for the evaluated topologies. We also show how function approximation can be used to dramatically reduce the memory requirements of a learning-based protocol while maintaining the same throughput and delay.**

***Keywords*—TCP; IoT; Q-learning; function approximation; Kanerva coding;**

I. Introduction

Current trends point to large scale deployments of wireless technology, such as in the case of Internet of Things (IoTs), where pervasively deployed sensors report critical data to a centralized server for analysis. Thus, not only must the network deliver this data with the highest possible throughput, but it must also minimize both the delay in transferring the data to the end server for immediate interpretation, and carry the returning acknowledgements (ACKs) to the source nodes. TCP is the de facto choice for the transport layer for reliable end to end delivery of packets, and thus, a clear starting point in the design of such protocols. In this paper, we propose a new technique for TCP *cwnd* adaptation based on Q-learning [1] that sets the optimal control parameters from historically observed network conditions. We then describe two variations on this approach based on CMAC [2] and fuzzy Kanerva-based function approximation [3]. We evaluate these three approaches and characterize their performance, quantified in terms of delay and throughput, and their memory requirements.

Classical TCP flavors, such as New Reno, rely on a probing operation (additive increase) of the *cwnd*, where it slowly increases the number of segments that can be transmitted by the source at a given time. When network conditions change suddenly, even if they follow a recurring pattern of bandwidth variations, New Reno must continue to explore the best setting for *cwnd* every time. This process involves progressively increasing the *cwnd* by 1 for each round trip time (RTT) in which the ACKs are received and sharply cutting the window by half or down to 1 (multiplicative decrease). Not only does this result in a lower network utilization, but it also impacts situations when the available bandwidth changes in short intervals, leaving insufficient reaction time. This motivates our approach of using techniques from machine learning where certain network conditions are observed and captured in terms of state-action pairs. When similar network situations are observed during operational time, a modified *cwnd* setting algorithm can immediately scale to meet the bandwidth availability. One of the key outcomes of this work is exploring the impact of system memory on TCP enhanced with learning. Towards this end, we study three different learning approaches — Q-learning, which incurs considerably higher memory consumption, CMAC which uses tiles to generalize and Q-learning coupled with fuzzy Kanerva-based function approximation. Both CMAC and fuzzy Kanerva-based Q-learning use function approximation technique to reduce the amount of memory needed to store the algorithm history. The latter approach is very useful for wireless nodes with very little memory, such as IoTs.

The overall operation of the protocol is explained as follows: We retain the slow start phase of New Reno, and enter into the learning-based adaptation only during the linear, or congestion avoidance stage. Based on the observed utility, measured as a weighted difference of the end to end throughput and delay, the *cwnd* is probabilistically changed by a scalar quantity. At each such instant, the network state and the expected utility is logged for later use, thereby creating a rapidly expanding table of network behavior, which is later leveraged in making the TCP window setting decisions.

The main contributions of our work are as follows:

- We propose a self-learning and unsupervised TCP *cwnd* adaptation technique based on Q-learning that can appropriately scale the *cwnd* as per previously seen network conditions.
- We analyze the impacts of function approximation on throughput and delay in CMAC and Fuzzy Kanerva-based TCP Q-learning. In both approaches, the memory required to store the algorithm history can be greatly reduced. With the reduction in memory, our proposed algorithm, Fuzzy Kanerva-based TCP Q-learning, achieves similar performance as pure Q-learning while only using 1.2% of memory compared to pure Q-learning. We then identify the optimal

2325-2944/16 $31.00 © 2016 IEEE
DOI 10.1109/DCOSS.2016.8

conditions where one or the other technique may be preferred.

- We demonstrate through packet level simulations in ns-3 significant improvement over TCP New Reno for the above approaches.

II. Motivation and Practical Relevance

An IoT is a network of interconnected objects, uniquely addressable through standard communication protocols. Such a network has capabilities of environmental sensing, local computation, wireless data transmission and may work cooperatively to satisfy a given goal, under the limitations posed by their processor and memory. We survey next a sample of IoT applications, and provide a high level description of these resources of the nodes in Table I, which may influence the choice of TCP protocol.

TABLE I: Examples of IoT applications

Application	Computing Power	Memory	Sample Device
Personalized healthcare	low	low	RFID
Environmental monitoring	medium	medium	TMote Sky platform
Smart city	high	high	Embedded PC

(1) Personalized healthcare: These applications involve medical sensors on the body, or implanted inside that captures the health condition of the subjects [4]. This kind of IoT application transmits sparing amount data, and does not require complex computation. Thus, the sensors are likely simple devices, and very limited processor capabilities and memory suffice for such applications.

(2) Environmental monitoring: Sensors may be deployed around a target area to automatically collect and report data on the various environmental effects (for e.g., seismic activity around volcanos [5], structural integrity, among others). The IoT devices in this case are embedded within the terrestrial environment for long durations, requiring more memory (compared to healthcare) to store the data. Also, they may need higher computing power to perform local aggregation on site.

(3) Smart city: Some of the current smart city deployments, such as CitySense project [6], rely on an embedded PC with a more powerful CPU and comparative RAM than the previous scenarios. Such sensors are not challenged in either computation or memory.

In the rest of this paper, we describe learning-based approaches that address these limitations. For example, while TCP with Q-learning has less computational needs, it does impose requirements on system memory. When Q-learning is coupled with function approximation within the TCP control mechanism, the memory usage can be flexibly set, but each iteration consumes large number of processor cycles. Thus the choice and the operating points of TCP can be hardware-dependent, and how these sensor-specific issues impact the end-to-end performance becomes an important subject of this paper.

III. Related Work

TCP is a well explored topic in both classical wired and wireless networking. For years many end-to-end congestion control mechanisms have been proposed. For example, Cubic uses a cubic function to tweak the *cwnd*, and is known for its ability to aggressively search for spare bandwidth. Other end-to-end congestion control approaches includes Vegas [7], Westwood [8] and Fast [9]. While these protocols all have their own unique properties, they share the similar idea of using some fixed functions or rules to change *cwnd* to adapt to new network conditions. One problem with this fixed-rule strategy is that they cannot adapt to the complexity and rapid evolution of modern data networks. They do not learn from experience or history and are not able to predict the consequences of each action taken. Even if an action reduces performance, the algorithm will mechanically select the same action repeatedly.

To solve this problem, some new techniques have been explored by the research community. Remy [10] comes close in principle to machine learning approach. It uses off-line training to find the optimal mapping from every possible network condition to the behavior of the sender (increase or decrease *cwnd*) with the goal of maximizing a utility function. The utility function captures the tradeoff between maximizing throughput and minimizing delay. Remy works well when prior assumptions about the network given at design time are consistent with the network situation in experiments. Performance may degrade when real networks violate the prior assumption [11]. The lookup table used in this approach stores mappings that are pre-calculated, which, as with other traditional TCP variants, cannot adapt to continuously varying network environment. In Remy's approach, the lookup table must be recomputed when new network conditions apply.

PCC [12] is a recently proposed protocol that can rapidly adapt to changing conditions in the network. It is an online learning algorithm that chooses actions based on recently observed results. However, the learning process does not exploit memory of previous experiences and their outcomes. PCC aggressively searches for better strategies that change the sending rate in real time. It employs a gradient-ascent learning method by which, in some cases, the exploration could be trapped at a local optimum, making it hard to achieve a globally optimal solution. Both Remy and PCC regard the network as a black box and focus on looking for the change in the sending rate that can lead to the best performance, without directly interpreting the environment or making use of previous experience.

There are few other works that apply machine learning to help improve the performance of TCP. For example, [13] builds a loss classifier using machine learning to differentiate link loss and congestion loss, and [14] and [15] use machine learning to better estimate RTT and throughput. None of these techniques tune *cwnd* directly.

IV. Q-Learning-based TCP

Our proposed algorithm, TCPLearning, is a protocol based on reinforcement learning. In reinforcement learning, the learning agent interacts with the environment with no prior knowledge, selects actions based on a learned policy, receives positive or negative rewards, and then observes the next state of the environment. The learning agent's goal is to develop a policy, a state space to action space mapping, that maximizes the long-term discounted reward. Henceforth, instead of using

probes to detect the effect of different actions on performance in PCC, TCPLearning uses the reinforcement algorithm Q-learning to learn an optimal policy to make action choices in each state directly by experience.

TCPLearning sender uses the normal slow start phase of New Reno protocol. If slow start ends when *cwnd* exceeds threshold, the congestion control process enters the congestion avoidance phase and our learning algorithm takes over controlling *cwnd*. If slow start ends because congestion is observed, the New Reno protocol continues and the learning algorithm is not used. If packet loss is detected while in the congestion avoidance phase, the learning algorithm halts and the New Reno protocol is applied to implement fast retransmission and fast recovery.

As in New Reno, the most important task of TCPLearning is adjusting the size of the *cwnd*. During each time period, usually one RTT, our algorithm collects throughput and RTT values by processing the ACK information, and then combines them into a single utility function U. The utility function increases as throughput increases and delay decreases. The algorithm's goal is to learn how changes in the size of the *cwnd* can increase the value of the utility function.

The learning algorithm uses Q-learning to learn a policy to choose actions and achieve its goal. Q-learning uses a simple value iteration update process. At time t, for each state s_t and each action a_t, the algorithm calculates an update to its expected discounted reward, or action-value function $Q(s_t, a_t)$ as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t(s_t, a_t)[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (1)$$

where r_{t+1} is an immediate reward at time $t+1$, $\alpha_t(s_t, a_t)$ is the learning rate such that $0 \leq \alpha_t(s_t, a_t) \leq 1$, and γ is the discount factor such that $0 \leq \gamma < 1$. Q-learning stores $Q(s_t, a_t)$ values in a table, called the Q-table. The time complexity to update the value of $Q(s_t, a_t)$ is $O(|A|)$ where $|A|$ is the number of actions.

The state of the system is represented by four state variables and the values of state variables are partitioned by discretization:

- A moving average of the inter-arrival time between newly received ACKs, discretized into 10 intervals.
- A moving average of the inter-arrival time between packets sent by the sender, discretized into 10 intervals.
- The ratio between current RTT and the best RTT found so far, discretized into 10 intervals.
- The slow start threshold, discretized into 10 intervals.

TABLE II: *cwnd* modification options

Change in *cwnd*	Extent of change (bytes)
Reduce	-1
No change	0
Increase	+5
Increase	+10
Increase	+20

The available actions for changing the *cwnd* are summarized in Table II.

The reward function is based on the changes in the value of the utility function: $U = \log_e(\text{throughput}) - \delta \times \log_e(\text{delay})$ where δ expresses the relative importance of delay over throughput. In our experiments, δ is set to 1.

The reward function is equal to:

- +2, if the utility increases after time period t
- -2, if if the utility decreases after time period t

where t is set to 0.1s (one RTT in our experiments).

V. Fuzzy Kanerva-based TCP Q-learning

The requirement that an estimated value be stored for every state-action pair limits the size and complexity of the learning problems that can be solved. The Q-table is typically large because of the high dimensionality of the state-action space, or because the state or action space is continuous. While discretization can be used to map a continuous state or action space onto a discrete set of values [16], it may still need a relative large discretized state-action space to achieve good learning performance. Therefore, function approximation may be needed to store an approximation of the entire discretized table to further reduce the memory cost while maintaining the same good learning performance [17] .

Many function approximation techniques exist, including coarse coding [18], tile coding (also known as CMAC) [2] and radial basis function networks (RBFNs) [19].

The Cerebellar Model Articulation Controller (CMAC), or Tile Coding, is a computationally-efficient function approximation technique [18]. In CMAC, k *tilings* are selected, each of which partitions the state-action space into tiles. The receptive field of each feature corresponds to a tile, and a θ-value is maintained for each tile. A state-action pair p is *adjacent* to a tile if the receptive field of the tile includes p. The Q-value of a state-action pair is equal to the sum of the θ-values of all adjacent tiles. In binary Tile Coding, which is used when the state-action space consists of discrete values, each tiling corresponds to a subset of the bit positions in the state-action space and each tile corresponds to an assignment of binary values to the selected bit positions.

Using small tiles can improve the precision of a learned policy since the finer grained state space makes it easier to select unambiguous actions. However, small tiles increases the learning time by making it harder to generalize between similar but different states. By using multiple overlapping tilings, CMAC maintains the ability to generalize while achieving high resolution. Using relatively large tiles in each tiling to provide coarse grain generalization and introducing a number of tilings to provide fine grain generalization, CMAC can consume less memory while maintaining comparable performance than TCPLearning algorithm mentioned in previous section.

Traditional Kanerva coding [18] can also be used to reduce the memory needed to store the state-action value table. The state here is described by a sequence of state-variables of the domain and each state-variable can take on a range of values. A state-action pair is represented as a combination of one

state and one action. In Kanerva coding, a collection of k *prototype state-action pairs*, (prototypes) is selected. Given a state-action pair s and a prototype p_i, $||s - p_i||$ represents the number of state-variables whose values differ between them, plus 1 if the action values differ. A state-action pair s and a prototype p_i are said to be *adjacent* if $||s - p_i|| \leq 1$. We define the *membership grade* $\mu_i(s)$ of state-action pair s with respect to prototype p_i to be equal 1 if s is adjacent to p_i, and 0 otherwise. A state-action pair's *membership vector* consists of its membership grades with respect to all prototypes. A value $\theta(i)$ is maintained for the ith prototype, and $\hat{Q}(s)$, an approximation of the value of a state-action pair s, is then the sum of the θ values of the adjacent prototypes. That is $\hat{Q}(s) = \sum_i \theta(i)\mu_i(s)$.

If two state-action pairs have the same membership vector, that is, the same *membership grades* over all prototypes, a prototype *collision* is said to have taken place. Kanerva coding works best when each state-action pair has a unique membership vector. If prototypes are not well distributed across the state-action space, many state-action pairs will either not be adjacent to any prototype, or adjacent to identical sets of prototypes that may cause prototype collisions. Such prototype collisions reduce the quality of the results because the solver cannot distinguish distinct state-action pairs, and the estimates of their Q-values will be equal.

An advantage of this approach is that each prototype contains information about all dimensions of the state-action space. The performance of a reinforcement learner with Kanerva coding depends largely on the number of prototype state-action pairs and the size of the target state-action space [19]. Dynamically selecting prototypes can improve the performance of function approximation [19], [20].

A more flexible and powerful approach is to allow a state-action pair to update θ-values of all prototypes, not only neighboring prototypes. Instead of being binary values, membership grades vary continuously between 0 and 1. Such fuzzy membership grades are larger for closer prototypes and smaller for more distant ones. Since prototype collisions occur only when two state-action pairs have same real values in all elements of their membership vectors, collisions are less likely.

In fuzzy Kanerva coding [3], the membership grade is defined as follows. Given a state-action pair s, the ith prototype p_i, and a constant variance σ^2, the membership grade of s with respect to p_i is $\mu_i = e^{-\frac{||s-p_i||^2}{2\sigma^2}}$. A value $\theta(i)$ is maintained for the ith prototype and an approximation $\hat{Q}(s)$ of the value of a state-action pair is $\hat{Q}(s) = \sum_i \theta(i)\mu_i(s)$. The effect of an update $\Delta\theta$ to a prototype's θ-value is now a continuous function of the place difference $||s - p_i||$.

Algorithm 1 describes our fuzzy Kanerva-based TCP Q-learning algorithm (Fuzzy TCPLearning). A vector $\vec{p}$ of prototypes is selected randomly, and a corresponding vector of θ values $\vec{\theta}$ is initialized to zero.

For each prototype p_i, the algorithm calculates an update to its expected discounted reward, $\theta(p_i)$ for all prototypes as follows:

$$\theta(p_i) \leftarrow \theta(p_i) + \mu_i(s,a)\alpha_t[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad (2)$$

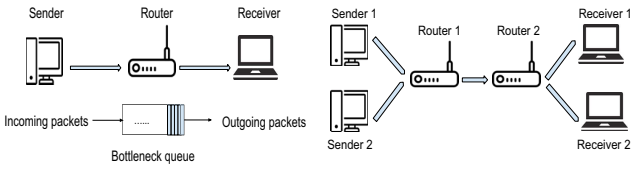
Algorithm 1: Fuzzy Kanerva-based TCP Q-learning

Input: $\vec{p}$: a set of prototypes, $\vec{\theta}$: a set of θ-values, each associates with one prototype p_i
Output: *cwnd*: changed by the selected action

1 **Procedure** `Main()`
2 Choose a set of prototypes $\vec{p}$ and initialize corresponding θ-values to 0
3 **for** *each new ACK received in congestion avoidance mode in TCP New Reno* **do**
4 **if** *observe updated utility* **then**
5 `Fuzzy-Kanerva(pre_s, pre_a, p, θ)`
6 **else**
7 Take action *pre_a* and apply to *cwnd*
8 **Procedure** `Fuzzy-Kanerva(pre_s, pre_a, p, θ)`
9 Observe reward *r*, get current state *s*
10 $\vec{\mu}(pre\_s, pre\_a) = e^{(-\frac{\|(pre\_s, pre\_a)-\vec{p}\|^2}{2\sigma^2})}$
11 $Q(pre\_s, pre\_a) = \sum_i \mu_i(pre\_s, pre\_a) * \theta(i)$
12 **for** *each action a' under current state* s **do**
13 $\vec{\mu}(s,a') = e^{(-\frac{\|(s,a')-\vec{p}\|^2}{2\sigma^2})}$
14 $Q(s,a') = \sum_i \mu_i(s,a') * \theta(i)$
15 $\delta = r + \gamma * max_{a'} Q(s,a') - Q(pre\_s, pre\_a)$
16 $\Delta\vec{\theta} = \vec{\mu}(pre\_s, pre\_a) * \alpha_t * \delta$
17 $\vec{\theta} = \vec{\theta} + \Delta\vec{\theta}$
18 **if** *randomly generated probability* > ε **then**
19 $a = argmax_{a'} Q(s,a')$ where $Q(s,a') = \sum_i \mu_i(s,a') * \theta(i)$
20 **else**
21 $a =$ random action
22 Take action *a* and apply to *cwnd*
23 *pre_s* = *s*
24 *pre_a* = *a*
25 **return** *cwnd*

where r is an immediate reward, s' is the next state, α_t is the learning rate such that $0 \leq \alpha_t \leq 1$, and γ is the discount factor such that $0 \leq \gamma < 1$. An ϵ-greedy approach is used to select the next action to take. For some value of ϵ, $0 \leq \epsilon \leq 1$, a random action is selected with probability ϵ, and an action that gives the largest Q-value for state s is selected with probability $1 - \epsilon$ (lines 18-21). $\mu_i(s,a)$ has the effect that the value $\theta(p_i)$ is updated most if the state-action pair is identical to the prototype p_i and is changed less if the difference between the state-action pair and prototype p_i become larger.

The algorithm begins by randomly generating a set of prototypes and initializing the θ-value of each prototype. Whenever receiving a new ACK in congestion avoidance mode, the learning procedure changes the *cwnd* if the value of the utility function is updated in each time period t (one RTT in our experiments). Otherwise, the previously chosen action takes effect and changes the *cwnd* (lines 3-7). When encountering a state-action pair and its accompanying reward, the learning procedure updates each prototype's θ-value based on the received reward and differences between the prototype and the state-action pair using Equation 2 (lines 9-17). All updated prototypes' θ values would contribute to varying degrees to the Q-value of each encountered state-action pair. The time complexity to update θ values of all prototypes is $O(|A||S||P|)$ where $|A|$ is the number of all possible actions, $|S|$ is the number of state variables and $|P|$ is the number of prototypes. The space complexity is $O(|P|)$. All prototypes



(a) (b)

Fig. 1: (a) Single sender/receiver topology and (b) Dumbbell network topology.

and θ value of each prototype should be stored in memory.

VI. Performance Evaluation

We use ns-3 based packet level simulations to evaluate the performance of TCPLearning, CMAC and Fuzzy TCPLearning in varying bandwidth conditions by comparing with TCP New Reno. We begin with a single-bottleneck network shown in Fig. 1(a) and later extend the evaluation to a more complex multi-flow network shown in Fig. 1(b) for fairness-related studies. We use these topologies to demonstrate the characteristic features of learning in controlled environments and show the impact on throughput and delay. The bottleneck bandwidth (on the router-receiver link) switches alternately between 7.5Mbps and 2.5Mbps every 800s. The network RTT is set to 100ms and the buffer size is set to BDP, which is 50 packets in our simulation. We conduct 8 experiments using each algorithm and report the average throughput and delay. The standard deviation of values is shown using error bars.

A. TCPLearning without Function Approximation

In this scenario, we disable function approximation and set the exploration rate ϵ for TCPLearning to 0.1. The initial learning rate α is set to 0.3, and it is decreased by a factor of 0.995 after each 10s. The total simulation time is set to 6400s.

1) Average Throughput and Delay: Fig. 2(a) compares the average throughput achieved by TCP New Reno and TCPLearning as the bottleneck bandwidth switches between 7.5Mbps and 2.5Mbps every 800s. The results show that TCPLearning significantly outperforms TCP New Reno as the bottleneck bandwidth fluctuates. We observe that at bottleneck bandwidth of 7.5Mbps, the average throughput of TCPLearning is 6.72Mbps while the average throughput of TCP New Reno is 4.46Mbps. At bottleneck bandwidth of 2.5Mbps, the average throughput of TCPLearning is 2.27Mbps while the average throughput of TCP New Reno is 2.26Mbps. We note that since the default buffer size is optimal under a network RTT of 100ms and bottleneck bandwidth of 2.5Mbps, TCP New Reno fully uses the buffer and TCPLearning achieves equally good performance.

Fig. 2(b) compares the average RTT achieved by TCP New Reno and TCPLearning under the same network settings. The results show that at bottleneck bandwidth of 7.5Mbps, the average RTT of TCPLearning is 111ms, while the average RTT of TCP New Reno is 109ms. At bottleneck bandwidth of 2.5Mbps, the average RTT of TCPLearning is 114ms while the average RTT of TCP New Reno is 154ms.

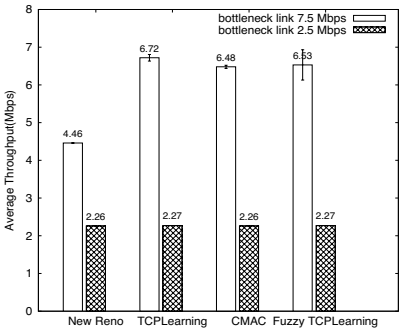
TCPLearning, under either bottleneck bandwidth, outperforms TCP New Reno in term of average throughput. Fig. 2(a) demonstrates that TCPLearning increases the average throughput by 33.8% in this high bandwidth fluctuating network. When considering the delay shown in Fig. 2(b), although TCPLearning performs slightly worse, 1.8% degradation in this case, at bottleneck bandwidth of 2.5Mbps, it outperforms TCP New Reno 26% at bottleneck bandwidth of 7.5Mbps. On average, TCPLearning achieves 12.1% reduction in delay.

We observe that the average throughput of TCP New Reno is 4.46Mbps, which is much less than the bottleneck bandwidth 7.5Mbps. This is because TCP New Reno's predefined congestion avoidance algorithm increases *cwnd* beyond what the connection can support, and finally congests the network, which ultimately results in a significant drop in *cwnd* and throughput. Even worse, since the TCP New Reno algorithm has no memory of past actions and the effect of those actions on performance, it repeats the same behavior. Fig. 3 shows the size of the *cwnd* as a function of time during a simulation of TCP New Reno. The plot shows that the algorithm repeatedly makes the same incorrect decision which reduces performance.

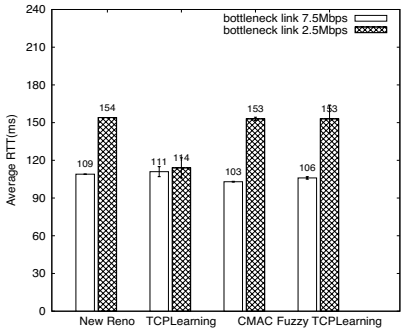
In addition, TCP New Reno takes a significant amount of time to recover after each significant drop in *cwnd* since it has to increase the *cwnd* linearly during the congestion avoidance phase. However, TCPLearning overcomes this defect by learning from experience. Fig. 3 also shows the size of the *cwnd* as a function of time during a simulation of TCPLearning. The plot shows that as the learning process proceeds, TCPLearning experiments with different actions that modify *cwnd* until 110s. After 110s, the learned action-value function $Q(s, a)$ converges to the optimal action-value function $Q^*(s, a)$. At this time, TCPLearning finds an optimal action which fully uses the buffer and does not trigger any packet loss. This learned action makes the *cwnd* large enough to achieve good performance, but slightly smaller than the upper bound beyond which packet loss occurs. The resulting high throughput gained by this optimal action remains stable until 800s after which the bottleneck bandwidth switches.

2) Real-time Throughput: Fig. 4 shows the real-time throughput of TCP New Reno and TCPLearning with the bottleneck bandwidth switching between 7.5Mbps and 2.5Mbps every 800s. The plot shows that when the bottleneck bandwidth is 7.5Mbps (in the first 800s), TCP New Reno experiences repeated packet loss that consequently results in low and unstable average throughput. When the bottleneck bandwidth switches to a much smaller value, 2.5Mbps after 800s, TCP New Reno fully uses the buffer and gets high and stable throughput. We observe that those scenarios with high fluctuating bottleneck bandwidths effectively and severely reduce throughput when using TCP New Reno. However, fluctuated bottleneck bandwidths have little effect on the throughput achieved by TCPLearning. As shown in Fig. 4, TCPLearning takes 110s to learn the optimal policy at 7.5Mbps bottleneck bandwidth, and remains high and stable throughput until 800s. When the bottleneck bandwidth switches to 2.5Mbps after 800s, TCPLearning converges very quickly and still achieves stable throughput until the bottleneck bandwidth switches again.

3) Real-time RTT: Fig. 5 shows the real-time RTT of TCP New Reno and TCPLeaning under the same bandwidth switch-



(a) Throughput comparisons



(b) RTT comparisons

Fig. 2: Average throughput and RTT over TCP New Reno, TCPLearning, CMAC and Fuzzy TCPLearning algorithms under fluctuating bottleneck bandwidths

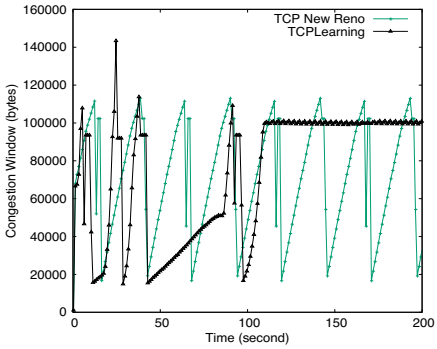
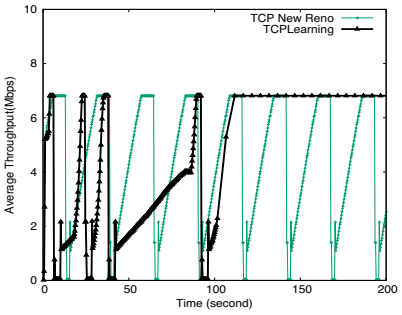
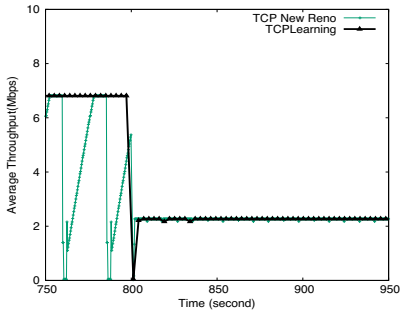


Fig. 3: Real-time *cwnd* of TCP New Reno and TCPLearning during simulation from 0s to 200s



(a) For duration from 0s to 200s



(b) For duration from 750s to 950s

Fig. 4: Real-time throughput of TCP New Reno and TCPLearning under fluctuating bottleneck bandwidths (7.5Mbps and 2.5Mbps), and throughputs for both figures remain same between 200s to 800s

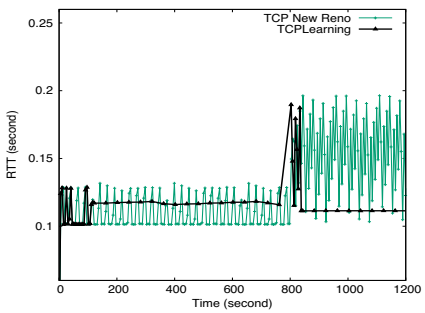


Fig. 5: Real-time RTT of TCP New Reno and TCPLearning under fluctuating bottleneck bandwidths (7.5Mbps and 2.5Mbps)

ing scenario described above. We observe that TCPLearning achieves more stable and lower RTT than TCP New Reno under fluctuating bottleneck bandwidths.

B. TCPLearning with Function Approximation

We evaluate the performance of the CMAC algorithm and the Fuzzy TCPLearning algorithm by applying both to the same network topology shown in Fig. 1(a). The CMAC algorithm partitions the state-action space into a set of different tiles and creates a number of tilings to provide both coarse and fine grain generalization in learning. In our experiment, we use 5 tilings and each tiling has 3,125 tiles since we have 5 possible actions and 4 state variables, each of which is evenly partitioned into 5 intervals. To learn the action values, we need to store 15,625 θ-values whose amount equals to 3,125 tiles per tiling multiplied by 5 tilings. Since each tiling has large tiles, relatively little memory is needed to store all the θ-values. The Fuzzy TCPLearning algorithm uses function approximation combined with continuous membership grades to control and significantly reduce the amount of memory needed to store the learned values, Q-table in the case of TCPLearning, while maintaining performance.

To run the experiments, we first randomly generate a set of 100 prototypes and initialize the corresponding θ-values. Then, the θ_i-value of each prototype is updated by the Q-learning process using Equation 2.

1) Average Throughput and Delay: Fig. 2 also compares the average throughput and delay achieved by CMAC and Fuzzy TCPLearning at two alternating bottleneck bandwidths. We observe that both CMAC and Fuzzy TCPLearning outperform TCP New Reno in terms of throughput and delay at two different bottleneck bandwidths. We note that both CMAC and Fuzzy TCPLearning have slight degradations in throughput compared to TCPLearning when the bottleneck bandwidth is 7.5Mbps. When the bottleneck bandwidth is 2.5Mbps, nearly identical throughputs are observed. In addition, both CMAC and Fuzzy TCPLearning achieve better performance than New Reno, and worse performance than TCPLearning, in term of delay at both bottleneck bandwidths. We conclude that, on average, TCPLearning performs best in terms of both throughput and delay. However, taking advantage of function approximation techniques, CMAC and Fuzzy TCPLearning significantly reduce memory usage while achieving comparable performance.

2) Impact of Reduced Memory Use: The TCPLearning algorithm allocates memory to store each of the 50,000 state-action pairs that may be encountered. Since 4 bytes are used to store the Q-value corresponding to one state-action pair, 200KB of memory storage is used by TCPLearning. In contrast, the CMAC algorithm only needs to store the θ-values which can be much less than the number of state-action pairs. The total number of θ-values used in our experiments is 15,625 and the final memory usage is 62.5KB which is less than 1/3 of memory used by TCPLearning. Fuzzy TCPLearning algorithm allocates storage for 100 state-action pairs. Since 20 bytes are needed to store one state-action pair, and another 400 bytes are used to store the θ-values of 100 prototypes, it only uses 2.4KB of memory, making it ideal for IoT applications.

C. Fairness Observations

We evaluate the fairness of TCPLearning algorithm by evaluating its performance in the dumbbell network topology shown in Fig. 1(b). The topology includes 2 senders and 2 receivers sharing the bottleneck bandwidth of 2.5Mbps with 100ms RTT. The bottleneck router buffer size is set to 100 packets. The transmission of data in the two flows starts simultaneously. Table III shows the average throughput of the two competing flows for TCP New Reno and TCPLearning. We observe that the average throughput of both flows is nearly identical using both TCP New Reno and TCPLearning, and therefore both score equally in the Jain's fairness index.

TABLE III: Fairness behavior of competing flows

Algorithm	TCP New Reno	TCPLearning
Average throughput of flow 1	1.14 Mbps	1.15 Mbps
Average throughput of flow 2	1.13 Mbps	1.12 Mbps
Average throughput of each flow	1.135 Mbps	1.135 Mbps
Jain's Fairness Index	0.99	0.99

VII. Conclusion

Learning methods can be used to reconstruct and optimize traditional TCP to better utilize available bandwidths in network traffic. We described a novel Q-learning based TCP that developed a policy to selects actions that adjust the *cwnd* size. The policy converges quickly to a stable set of actions. Our TCPLearning algorithm achieved better throughput and delay performance than TCP New Reno in the presence of high bandwidth fluctuations. To reduce the memory usage needed in the learning process while still maintaining comparable performance, we introduce function approximation techniques to our learning algorithms. We describe the CMAC algorithm that uses less than 1/3 of the memory required by TCPLearning. Furthermore, Fuzzy TCPLearning algorithm significantly reduces the memory usage, using 1.2% of the memory needed by TCPLearning. Fuzzy TCPLearning still outperforms the TCP New Reno with respect to throughput and delay in a network with large bandwidth fluctuations. This approach can be sued to implement learning-based TCP on memory-constrained wireless nodes.

Acknowledgment

This work was supported in part by the Office of Naval Research under grant number N000141410192.

References

[1] C. Watkins, "Learning from delayed rewards," *Ph.D thesis, Cambridge Univeristy, Cambridge, England*, 1989.

[2] W. T. Miller III, F. H. Glanz, and L. G. Kraft III, "Cmas: An associative neural network alternative to backpropagation," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1561–1567, 1990.

[3] C. Wu and W. Meleis, "Fuzzy kanerva-based function approximation for reinforcement learning," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1257–1258.

[4] S. Amendola, R. Lodato, S. Manzari, C. Occhiuzzi, and G. Marrocco, "Rfid technology for iot-based personal healthcare in smart spaces," *Internet of Things Journal, IEEE*, vol. 1, no. 2, pp. 144–152, 2014.

[5] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 381–396.

[6] R. N. Murty, G. Mainland, I. Rose, A. R. Chowdhury, A. Gosain, J. Bers, and M. Welsh, "Citysense: An urban-scale wireless sensor network and testbed," in *Technologies for Homeland Security, 2008 IEEE Conference on*. IEEE, 2008, pp. 583–588.

[7] L. S. Brakmo and L. L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *Selected Areas in Communications, IEEE Journal on*, vol. 13, no. 8, pp. 1465–1480, 1995.

[8] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "Tcp westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 287–297.

[9] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "Fast tcp: motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking (ToN)*, vol. 14, no. 6, pp. 1246–1259, 2006.

[10] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 123–134.

[11] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 479–490.

[12] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "Pcc: Re-architecting congestion control for consistent high performance," *NSDI'15 Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[13] A. Jayaraj, T. Venkatesh, and C. Murthy, "Loss classification in optical burst switching networks using machine learning techniques: improving the performance of tcp," *Selected Areas in Communications, IEEE Journal on*, vol. 26, no. 6, pp. 45–54, 2008.

[14] B. A. Nunes, K. Veenstra, W. Ballenthin, S. Lukin, and K. Obraczka, "A machine learning approach to end-to-end rtt estimation and its application to tcp," in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE, 2011, pp. 1–6.

[15] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to tcp throughput prediction," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1. ACM, 2007, pp. 97–108.

[16] A. Lampton, J. Valasek, and M. Kumar, "Multi-resolution state-space discretization for q-learning with pseudo-randomized discretization," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE, 2010, pp. 1–8.

[17] L. Baird, "Residual algorithms: Reinforcement learning with function approximation," in *Proc. of the 12th Intl. Conf. on Machine Learning*. Morgan Kaufmann, 1995.

[18] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Bradford Books, 1998.

[19] B. Ratitch and D. Precup, "Sparse distributed memories for on-line value-based reinforcement learning," in *Proc. of the European Conf. on Machine Learning*, 2004.

[20] C. Wu and W. Meleis, "Adaptive kanerva-based function approximation for multi-agent systems." in *Proc. of 7th Intl. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 2008.