Achieving Quality of Service with Adaptation-based Programming for Medium Access Protocols

Pingan Zhu, Jervis Pinto, Thinh Nguyen, Alan Fern
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, Oregon, USA 97331
Email: {zhup,pinto,thinh,afern}@eecs.oregonstate.edu

***Abstract*—Designing network protocols that work well under a variety of network conditions typically involves a large amount of manual tuning and guesswork, particularly when choosing dynamic update strategies for numeric parameters. The situation is made more complex by adding the Quality of Service (QoS) requirements to a network protocol. A fundamentally different approach for designing protocols is via Reinforcement Learning (RL) algorithms which allow protocols to be automatically optimized through network simulation. Unfortunately, getting RL to work well in practice requires considerable expertise and carries a significant implementation overhead. To help overcome this challenge, recent work has developed the programming paradigm of Adaptation-Based Programming (ABP), which allows programmers who are not RL-experts to write self-optimizing "adaptive programs". In this work, we study the potential of applying ABP to the problem of designing network protocols via simulation. We demonstrate the flexibility of our design method via a number of case studies, each of which investigates the performance of an adaptive program written for the backoff mechanism of the MAC layer in the 802.11 standard. Our results show that the learned protocols typically outperform 802.11 on a number of evaluation metrics and network conditions.**

I. Introduction

The design of network protocols is a complicated and often tedious endeavor. Consider, for example, the process of designing a MAC layer protocol for the popular 802.11 standard. Typically, a number of high-level design decisions are made (e.g. conditions for backoff steps) after which, a number of numeric parameters (e.g. backoff factors) are individually tuned for good (average) performance under a variety of network conditions. While the high-level designs are often theoretically well-motivated, the possible design space is vast and it is quite plausible that better protocols exist. Furthermore, the parameter tuning is extremely tedious which complicates an iterative design of the protocol since the smallest change to the protocol has to be followed by a search for the optimal parameter values.

Clearly, the designer is being forced to fully specify a solution to the complex problem of protocol design in the presence of considerable uncertainty. This method is one extreme of the programming paradigm spectrum. At the other extreme lie Reinforcement Learning (RL) techniques which ask very little of the programmer besides the problem specification and the objective function. Unfortunately, getting such RL programs to work well in practice is often very challenging, requiring considerable expertise and implementation overhead.

Adaptation-Based Programming (ABP) is a novel programming paradigm, which attempts to bridge these two extremes. ABP integrates RL into a modern programming language and is intended for users who are not experts in RL. It allows "adaptive" programs to be written in which certain decisions may be left "open" via `adaptive` objects. The programmer only needs to specify the information relevant to the decision and the set of candidate actions along with a performance objective given via `reward` statements which is used by the underlying learning system to learn a policy at each of the decision points. Importantly, the use of ABP adds practically no implementation overhead compared to writing a standard non-adaptive program.

The integration of a nearly unrestricted programming ability with modern RL algorithms makes for a promising new approach to the problem of network protocol design. The primary benefit is that the ABP system places little to no restrictions on the program unlike current RL libraries. In fact, an ABP program can be written in any situation where a Java program might be written. The programmer is free to specify as much of the program as she wants which allows her to focus on high-level decisions and leave the rest to the learning system. More generally, it allows the designer to only specify those parts of the protocol where the "best" action is intuitively clear and leave the rest unspecified.

To make the applicability of ABP more concrete, consider the problem of designing a "new" backoff mechanism in the MAC layer of 802.11. The current method uses a single bit of information as context: Whether the last attempted packet transmission at a node resulted in a collision or not. The set of actions are to double the current contention window after observing a collision or to reset to the minimum value of CWmin after a successful transmission. While this method is very simple and uses only a single bit for context, it is unclear if we might be able to do better by increasing the context or set of actions. As we will soon show, it is almost trivial to replace the above procedure with a single adaptive that uses the same context and action set. Modifying the context or the set of actions is also simple, making it very easy to hand-tune the protocol should the designer wish to do so. The other major advantage of using the ABP system is that it is very easy to specify any desired optimization objective (e.g. average bandwidth, cumulative profit). It is far easier to specify such an objective than it is to specify a protocol that will

effectively optimize the objective. ABP thus provides a way of automatically optimizing a protocol for specific objectives of interest with relative ease.

In this paper, we illustrate the utility of ABP through several case studies. Our focus is on the uncertainty regarding the best design for the backoff mechanism in standard 802.11 and enhanced 802.11e for QoS provisioning. Our results show that the learned policies always do at least as well as the 802.11x and often outperform them. The rest of the paper is organized as follows: In Section II, we review RL literature and its relations to ABP. In Section III, we show examples of adaptive program to optimize the backoff mechanism of the standard 802.11. Section IV describes the network simulations and the experimental setup followed by two case studies which demonstrate the potential of combining learning with existing knowledge and hand-tuning. We discuss the related work in Section V and conclude with a discussion and future work.

II. Reinforcement Learning and Adaptation-Based Programming

Reinforcement Learning (RL). The problem of reinforcement learning is typically formalized in the sequential decision-making framework of Markov Decision Processes (MDPs). An MDP consists of a state space, a set of actions, a stochastic transition function, and a reward function. The transition function specifies the probability of transitioning to a particular next state after taking a particular action in the current state. The reward function specifies a real-valued reward for each state of the system, indicating the desirability of being in the state. A typical objective is to find an action selection policy, mapping states to actions, that maximizes the average total reward acquired while executing the policy in the MDP. RL is the problem of optimizing such a policy without being given knowledge of the MDP transition or reward function. Rather an RL algorithm must interact with the MDP in order to find an optimal policy via intelligent trial and error. There is a large body of work on RL [1], [2] and it has also been successfully applied to a variety of networking problems, e.g. [3], [4], [5], [6], [7], [8].

While many RL algorithms come with theoretical guarantees (e.g. convergence in the limit, or finite-time approximation guarantees), the successful application of RL to real-world problems is still somewhat of an art that requires significant experience and effort in selecting a learning algorithm, formulating the problem as RL, implementing the algorithm and connecting it to the application environment. While there are a number of RL libraries [9], [10] that simplify experimenting with RL, the libraries are primarily designed for use by RL researchers to ease the evaluation and comparison of algorithms.

Adaptation-Based Programming (ABP). ABP is a novel programming paradigm [11], [12] that aims to lower the bar for successfully applying RL algorithms to real-world problems. ABP incorporates RL into a modern programming language, Java in this paper, and is intended for programmers with little to no RL experience. ABP allows the user to write "adaptive programs", which are simply standard programs that are allowed to contain open decision points, which reflect the programmer's uncertainty about what the best action is at that point. As illustrated in the next section, open decision points are included in an adaptive Java program via the `adaptive` construct (Java object) which can be used at such points to "suggest" an action according to some underlying policy which must be learned through experience. Thus, instead of forcing the programmer to guess the correct program action to be taken at such a point, ABP allows the programmer to be agnostic, with the idea that the program will automatically optimize those choices.

To facilitate such optimization, the programmer must specify the objective that should be optimized by the program, e.g. attained bandwidth. This is done in ABP via the use of "reward" statements which may be placed anywhere in the program. During the program execution, each reward statement provides the ABP system with a positive or negative numeric reward, indicating the "goodness" of the program execution. For example, a positive reward might be provided for each successfully transmitted packet. The learning objective of the ABP system is for the adaptive objects to learn to make decisions so that the expected total reward over program executions is maximized. The learning is done via RL algorithms that run behind the scenes during program executions. For example, the ABP system used in this work is based on policy-gradient RL algorithms (see [12]), the details of which are not important to this paper. Importantly, the programmer need not understand the learning mechanism, but is only responsible for specifying appropriate decision points and rewards. See [11] for a detailed description of the constructs and semantics of the Java ABP system used here while [12] describes the learning algorithm employed. We now give examples of adaptive programs for network protocol design, which will be used later in our experimental investigation.

III. Example Adaptive Programs

The adaptive programs used in this work are based on the backoff mechanism in standard 802.11 protocol shown in Figure 1 as a non-adaptive program. This program is executed after every node attempts to send a packet. The only decision point is the action of adjusting window size after a transmission. If the previous transmission is a collision, it will double the contention window(CW) size; otherwise resetting CW to the minimum value.

```
if(collision)
      cw = cw * 2;
else
      cw = CW_MIN;
```

Fig. 1. Code snippet for standard 802.11 backoff mechanism

We start with the original 802.11 backoff mechanism shown in Figure 1 as a standard non-adaptive program. This mechanism is executed at any node that has just attempted to send

a packet. The only decision point is how to adjust the current size of the contention window (cw) under various network conditions. While there are many possibilities for such a design space, 802.11 uses a simple approach of doubling the window when there is a collision, otherwise resetting it.

Notice that the 802.11 backoff program can be viewed as looking at a context, here whether there was a collision or not, and then selecting between two actions depending on the value of the context. This is exactly the behavior of adaptive objects in ABP. Each adaptive object is provided with a context when making a decision and returns one of a specified set of actions. Figure 2 shows our simplest adaptive program, which corresponds as closely as possible to standard 802.11. The first line provides the program with a reward that is equal to the difference between the newly successfully sent packets (at most one) and the number of packets that were dropped from the packet queue since the node's last attempted transmission. Note that this code and hence reward statement is executed after each attempted transmission. Thus, the cumulative reward is more positive if there are many successes and few drops. The next two lines set the context to be used by an adaptive for decision making and the actions available to the adaptive. In this case, the context is simply equal to $h0$, a bit indicating whether there was a collision or not, which is the same context used by 802.11. The actions are also the same as for 802.11. Next is the call to the adaptive's `suggest` which may return one of the two actions, `RESET` or `MUL_BY_TWO`. This choice will depend on the context and the mapping from context to actions is what should be learned by the adaptive program.

```
reward(send - drop)                                    (1)
ctx = h0                                               (2)
actSet = { RESET, MUL_BY_TWO }                         (3)
action = adaptive.suggest(ctx,actSet)                  (4)
if(action == MUL_BY_TWO)                               (5)
     cw = cw * 2                                       (6)
else if(action == RESET)                               (7)
     cw = CW_MIN                                       (8)
```

Fig. 2. Basic adaptive program with the same policy space as 802.11.

How might 802.11 perform if we allowed it to consider more information when making a decision i.e. consider a larger context? What if we consider selecting from a larger set of actions rather than just doubling or resetting? While in concept it is clear that these are likely to lead to an improved protocol, it is very unclear how to define the function that selects among actions based on the complex context. This is the type of decision that we would like to leave open to be automatically optimized. Figure 3 shows an adaptive program which has more context and a larger set of candidate actions to choose from. The context contains information that intuitively, could be relevant to the adaptive's choice. For instance, using more collision history bits and the arrival rate of the node seems relevant to making a good decision. Note that it is not necessarily the case that all of the context is relevant for making the decision, but the programmer need not worry about that. It is the job of the ABP learning system to uncover the relevance of parts of the context for decision making. Similarly, it is unclear whether the two new actions under consideration (halve the window, leave unchanged) will be useful, but it is conceivable and left for the ABP learning system to decide. By making these small changes to the adaptive program, a much wider space of possible protocols will be considered by the learning system. As our experiments will show, this can lead to considerably improved performance with no further effort from the programmer.

```
reward(send - drop)
ctx = h0,h1..h4, b/w, arr. rate, drop bit
actSet = { RESET, MUL_BY_TWO, DIV_BY_TWO, REMAIN }
action = adaptive.suggest(ctx,actSet)
if(action == MUL_BY_TWO)
     cw = cw * 2
else if(action == RESET)
     cw = CW_MIN
else if(action == DIV_BY_TWO)
     cw = cw / 2
else if(action == REMAIN)
     // do nothing
```

Fig. 3. Extended adaptive program with a larger policy space.

IV. Simulation Study

A. Environmental Setup

To gain fundamental insights into the behaviors of the network protocols produced by adaptive programs, we choose to restrict our experiments to a few simple but important simulation settings. Specifically, we study the performance of ABP-based MAC protocols in a single-hop 802.11x network consisting of N wireless terminals or users. All users are in proximity and a transmission from one user will interfere with all others. In addition, we consider scenarios in which upstream traffic are dominant, i.e., users send data more than they receive. This scenario is important for the evaluation of MAC protocols since typical wireless MAC protocols depend critically on the channel contention mechanism when a user tries to access the wireless medium to send its data. Moreover, we focus on a single point of uncertainty in the MAC layer and restrict our simulation to that of the MAC layer alone. All simulations are performed using a time-slot accurate simulator similar to the wireless module in NS-2 [13].

Training adaptive programs: Each adaptive program is trained separately on both of the above scenarios as follows: The adaptive program is allowed to learn on a newly initialized simulation for a duration of 60 seconds, after which we reset the simulator and restart. In order to evaluate the learning mechanism, after every 100 episodes we turn off the learning algorithm and evaluate the protocol for certain performance metrics. After learning for 50,000 episodes, we pick up the best performed policy in terms of accumulated reward to proceed evaluation. Since the ABP system uses randomization

Packet Size	12000 bits
SLOT-TIME	$20\mu s$
DIFS	50 μs
CWmin	31
CWmax	1023
Channel Bit Rate	11Mbps
Maximum Queue Length	100

TABLE I
SIMULATOR PARAMETERS ADAPTED FROM 802.11.

```
if(mostRecentPacketIsDropped) {
  if(t-0 send attempt is successful)
    return RESET
  else {
    if(t-4 send attempt is successful)
      return RESET
  }
}
else {
  if(all five previous send attempts are successful and
                    estimated node bandwidth is moderate)
    return RESET
}
return MUL_BY_TWO
```

Fig. 5. Decision tree representation of learned policy in the B-H scenario of Case 1. Here `t-0` refers to the most recent send attempt while `t-4` refers to the fifth previous send attempt.

and the learning algorithm might get stuck in local minima, we repeat the above procedure five times and report the best learned protocol among all the runs.

Evaluating protocols: We evaluate a given protocol on a given network scenario by simulating the network for 600 seconds. During the evaluation, the learning mechanism is switched off. Therefore the adaptive deterministically picks the action it thinks is best for a given context. To remove the effects of randomization, we perform ten such evaluations and report the mean and standard deviation of each metric.

B. Case Study 1: Standard 802.11

We simulate 20 users, each with its own packet queue with packets arriving at a constant rate. Packets are dropped if a maximum queue length is exceeded. This restriction on the number of packets in a queue produces packet drops as a feedback signal to the MAC protocol indicating that the rate of packets generated by an application or by another upstream node in an ad-hoc network exceeds the current sending rate. A well-designed MAC protocol ought to be able to use this information to adapt its sending rate effectively.

In order to simulate different traffic densities, we use three different arrival rates. Two arrival rates are chosen to simulate the extreme traffic density scenarios where all the queues are mostly full (H) or mostly empty (L). The third arrival rate is a moderate scenario (M) between these extremes. The most basic version of the simulator assumes one of these three arrival rates for every one of the 20 nodes. We call this network model "balanced" (B) and combined with the arrival rates, gives us three network variants {B-H, B-M, B-L}. To better simulate scenarios where different types of traffic are on the same network, we "unbalance" the network by assigning arrival rates according to one of three schemes, each containing a different mixture of arrival rates which leads to the next three variants {U-H, U-M, U-L}. We use the system parameters shown in Table II.

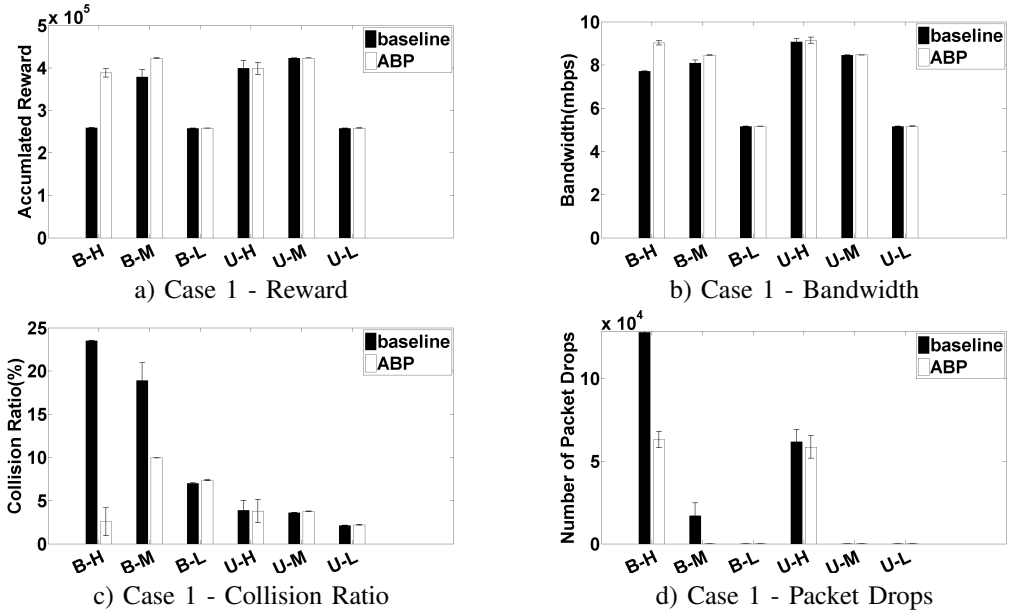
Actions, Context, and Reward. Since we eventually want to use the protocol in a distributed manner, each node must be able to compute its context locally (i.e. without any communication between nodes or a centralized controller). Specifically, we use the context composed as follows: a) *Five history bits* which represent the status of the last five transmissions with 1 for a collision and 0 for a successful transmission. These features generalize the single history bit used by standard 802.11. b) *Estimated available bandwidth* which we coarsely quantize into three bins. c) *Estimated arrival rate*, also coarsely quantized into three bins. d) *Packet drop bit*, whether the last arriving packet is dropped or not.

Note that all these contexts can be measured or estimated at a node. For example, the estimated available bandwidth can be estimated by the number of idle slots that a node observes which ensures that a node's policy can be implemented using only local information. Now, depending on the context, one of the following actions are taken by the protocol: RESET, REMAIN, MUL-BY-TWO and DIVIDE-BY-TWO. For the reward, we use the difference between number of successful sent packets and collision packets.

The results of comparing the learned protocols with 802.11 are reported in Figure 4(a-d). In all cases, the adaptive program learns a protocol that either outperforms the baseline (on B-H, B-M) or does as well as the baseline for all the metrics. In particular, on the B-H and B-M cases, substantial improvements in bandwidth, dropped packets and the collision ratio are observed. An inspection of the learned policies shows that they are optimized for the particular network scenario they were trained on, as expected. For instance, in the B-H scenario, most of the queues are typically close to full. In this scenario, immediately resetting the contention window back to the small value of CW_MIN after a single successful transmission is intuitively bad since it increases the probability of a collision at the next send attempt. We can summarize the behavior of the learned protocol via a decision tree shown in Figure 5. If we prune the decision tree down to a single bit of information, the most informative bit appears to be whether the last packet is dropped or not; with the protocol choosing to reset upon observing a dropped packet, otherwise doubling the contention window.

For the B-M scenario, the decision tree representation of the policy is quite complex and cannot be represented in the simple form above. However, pruning the tree reveals the most informative features to be the five send attempt history bits and the estimated node bandwidth. Unsurprisingly, the drop feature is no longer relevant since the node queues are rarely full. On this scenario, as Figure 4(d) shows, standard 802.11 drops four orders of magnitude more packets than the best learned protocol, which drops practically none.

As the traffic densities decrease, a large number of node



a) Case 1 - Reward b) Case 1 - Bandwidth

c) Case 1 - Collision Ratio d) Case 1 - Packet Drops

Fig. 4. (a-d) shows the comparison of performance metrics of the best learned protocol vs. standard 802.11.

queues are empty and the learned protocols keep the node contention windows very small which explains the close similarity in performance between the learned protocols and the baseline. Initially, we expected to see large improvements in the U-H and U-M scenarios. A closer inspection revealed that only a small number of nodes have packets arriving too quickly, making packet drops unavoidable. In these scenarios, the learned protocol simply sets each node's contention window to the smallest value possible which seems to be the right thing to do as demonstrated by the near-zero collision ratio. The lower number of collisions and the fewer number of sending nodes allows the learned protocol (and 802.11) to transmit very efficiently.

An obvious shortcoming of our training mechanism is that the learned policies might not generalize well to previously unseen or rapidly changing network conditions. However, our objective here is to investigate if we can use ABP to discover better protocols for specific network conditions and our training mechanism is validated by the observed improvements in performance and efficiency. Furthermore, this approach is far from unreasonable. For instance, one might envision a two-step learning procedure where in step one, we separately learn a suite of protocols on a number of network "classes" as done here and in step two, we train another adaptive to learn when to switch between the protocols in the suite. In summary, the simple adaptive program used here is an incremental change to the existing protocol and is still able to achieve significant improvements in performance (i.e. higher bandwidth, fewer dropped packets) and efficiency (i.e. lower collision ratio).

C. Case Study 2: 802.11e Adaptive Program with QoS Model

In this case study, we examine the utility of using ABP to design network protocols with QoS requirements. The main idea is to incorporate the QoS requirements into the reward function and adaptive contexts. To simplify the setup, in this simulation, we consider a three-node wireless network. Each node is assumed to generate a different type of traffic in order to model three representative classes of applications: voice-IP applications with stringent delay but small throughput requirement, video streaming applications with moderate delay and good throughput requirement, and web-browsing/FTP with less stringent requirements on both delay and bandwidth. Specifically, we use the MPEG-2 trace in [14] to model the traffic generated by the video streaming node. This video trace has an average bandwidth of 3 Mpbs and the TXOP is set to 3 ms. For the voice-IP node, we use a 20 kbps Poisson traffic with TXOP set to 1.5.ms. For the FTP node, we use a Poisson model with a fixed rate within an experiment but vary them from 3 Mpbs to 8 Mbps for different experiments. We use FIFO queuing policy with drop tail policy for all the queues at different nodes, i.e., a newly arrival packet is dropped if the queue is full. The list of simulation parameters for this case study are shown in Table II.

SLOT-TIME	$20\mu s$
DIFS	$50\mu s$
Channel Bit Rate	11Mbps
Maximum Queue Length	100
CWmin Voice	7
CWmax Voice	15
CWmin Video	15
CWmax Video	31
CWmin Data	31
CWmax Data	1023
MAX TXOP Voice	1.5ms
MAX TXOP Video	3ms

TABLE II
SIMULATOR PARAMETERS ADAPTED FROM THE 802.11E STANDARD.

We note that standard 802.11e gives voice and video traffic

a relatively high priority compared to regular traffic by setting CWmax and CWmin to small values. Specifically, CWmax for voice and video traffic are 15 and 31, respectively, while CWmax for regular data traffic can be up to 1023. Also, the bandwidth is set at 11 Mbps which is substantially lower than the current technology. However, this is only a scaling factor, and should not significantly affect the relative performance between ABP and 802.11e.

Actions, Context, and Reward. The action set is the same as Case 1. We use the following context which is again restricted to be locally computable. a) *History bits*, representing the status of the last five transmissions with 1 for a collision and 0 for a successful transmission. b) *Estimated available bandwidth*, coarsely quantized into three bins. c) *Estimated arrival rate*, again quantized into three bins. d) *Packet drop bit* corresponding to whether the last arriving packet is dropped or not. e) *bits indicating type of traffic and throughput requirement*: IP-telephone, video streaming, or data with corresponding throughput requirements for voice (20 kbps), video (3 Mbps), and data (none).

We implement the following ABP reward. Every node estimates its goodput at every 0.1 seconds. For voice and video traffic, it is important that their packets arrive on time and the overall throughput satisfy the requirements. As such for voice and video traffic, if the estimated throughput of video or voice traffic in a period of 0.1 second lies in the 5% range of the requirements, no penalty will be given. Otherwise, we give -1 for that period of time. Also, for the data traffic, if the estimated throughput in a certain period is better than the previous period, we give +1 to as a reward.

There are many metrics for comparing standard 802.11e with ABP. For example, one can compare the average video throughput, or voice throughput, or average throughput of all traffic produced by 802.11e and ABP. However, this is a vector comparison and there might not be an absolute vector that is better than all others. On the other hand, it is reasonable to say one protocol is better than other if, for both protocols, voice and video traffic satisfy the specified throughput requirements while the data throughput of the better protocol is larger than the other. In fact, the protocol learned by ABP is better than 802.11e in this sense. Fig. 6(a) shows that the average throughput for data traffic using ABP is consistently larger than that using the 802.11e protocol for different data traffic densities ranging from 5 to 8 Mbps. Fig. 6(b) shows the collision ratio for data traffic as a function of data traffic density. As seen, there are more collisions for ABP when the data traffic density increases. This is an acceptable trade-off to increase the overall throughput. Note that most of the collisions happen with data traffic, and rarely interfere with the voice and video traffic. On the other hand, Fig. 6(c) shows the number of packet drops due to overbuffering at the data queue is much more for 802.11e than that of ABP. This occurs as ABP tries to send out packets faster at the expense of increased collision but reducing the number of packet drops. Fig. 6(d) show all video, voice and data throughput for ABP. An important observation here is that the voice and video traffic are satisfied the specified throughput requirements as they vary very little around 20 kbps and 3 Mbps.

V. Related Work

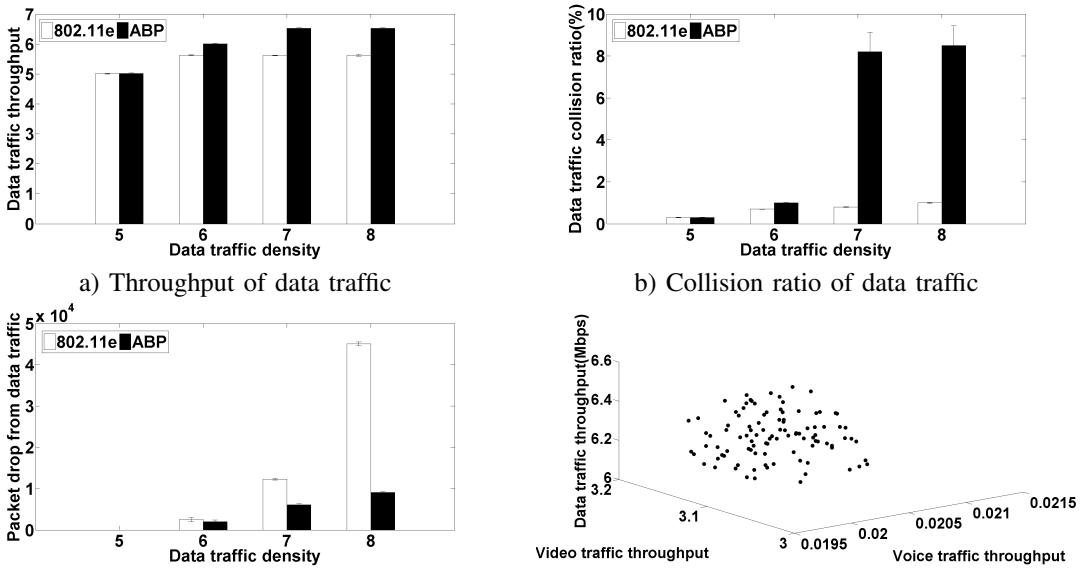
While our ABP framework for designing network protocols is novel, tools and methodologies for designing network protocols have been well explored. Under certain restricted assumptions about the network conditions, it is often that theoretical analysis alone is sufficient to yield well-designed protocols in practice [15]. On the other hand, it is often difficult to develop precise theoretical analysis for many real-world scenarios in which many simplistic assumptions about the networks no longer hold. In this case, network protocol designers often rely on simulations to validate the performance of their theoretically or intuitively motivated protocols. As such, many network simulators have been developed to aid the designers. The most predominant simulator is NS-2 [13], which allows the designers to write their protocols in high-level Tcl/Tk scripts. However, unlike our ABP framework, NS-2 is incapable of automatically suggesting good policies using the designer's hints. JavaSim [16] is also another popular network simulators capable of performing both coarse and fine-grain simulations to allow for trade-off in accuracy and simulation time. Like NS-2, Javasim is another "unintelligent" network simulator. Interested readers may refer to [17] for a list of network simulation tools.

There have been many previous attempts at ABP [18], [19], [20]. Of these the ALISP [19], [20] is the most similar to the Java ABP system used in this paper. ALISP integrates choice points (similar to adaptives) into the LISP language. However, the system is intended for RL experts and constraints the programmer who must write the program in a certain way. ALISP relies on the notion of an external world (MDP) that provides rewards and the accompanying MDP theory is hard for typical programmers to understand. The ABP system used in this paper has been designed explicitly for such programmers and is applicable wherever a standard Java program can be written.

ABP systems are often confused for a RL library [9], [10]. However, such libraries are designed to simplify the experimentation with and comparison between RL algorithms and demand considerable RL knowledge which limits their usability. To the best of our knowledge, the ABP system used here is the only system of its kind which allows non-RL experts to easily apply modern RL algorithms to their programs.

VI. Discussion and Future Work

This paper has introduced the framework of adaptation-based programming for network protocol design. The framework is appropriate whenever a protocol designer can specify the objective to be optimized but has uncertainty about how to build a completely specified protocol that optimizes it. The ABP framework allows the designer to completely specify parts of the protocol code that he is confident about, while leaving other uncertain choices open for the ABP system



a) Throughput of data traffic

b) Collision ratio of data traffic

c) Packet drops from data traffic

d) Throughput of different traffic types

Fig. 6. (a-d) shows the comparison of performance metrics of the best learned protocol vs. standard 802.11e.

to automatically optimize. Our case studies investigated the application of ABP to extending the 802.11 backoff mechanism in various ways. The results show that the learned protocols are able to outperform the standard 802.11 and 802.11e mechanisms on a variety of performance metrics.

In future work, we plan to apply the ABP approach to more complex protocols, where multiple choice points can be optimized. We also plan to fully integrate the system with standard network simulators. In addition, we plan to consider the problem of online optimization of adaptive programs in the context of network. In this work, the protocol was designed by running the adaptive program in a simulator during the learning phase. The resulting learned protocol would then be executed on a real network without further adaptation. There is clear utility in allowing adaptation during real execution in response to changing network conditions, that may not have been reflected in simulation. However, such adaptation must be carefully controlled to ensure stability and fast adaptation to changes. Understanding sound methods for doing this is an important open problem.

REFERENCES

[1] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[2] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2000.

[3] M. Littman and J. Boyan, "A distributed reinforcement learning scheme for network routing," in *tech. report CMU-CS-93-165, Robotics Institute, Carnegie Mellon University*, 1993.

[4] N. Tao, J. Baxter, and L. Weaver, "A multi-agent, policy-gradient approach to network routing," in *ICML*, 2001.

[5] B. Awerbuch, D. Holmer, and H. Rubens, "Provably secure competitive routing against proactive byzantine adversaries via reinforcement learning," in *http://www.cnds.jhu.edu /research /networks /archipelago /publications /LearningByzantineRouting-TechnicalReport.pdf*, 2003.

[6] Y. Chang, T. Ho, and L. Kaelbling, "All learning is local: Multi-agent learning in global reward games," in *NIPS*, 2004.

[7] ——, "Mobilized ad-hoc networks: A reinforcement learning approach," in *International Conference on Autonomic Computing*, 2004.

[8] S. Dejmal, A. Fern, and T. Nguyen, "Reinforcement learning for vulnerability assessment in peer-to-peer networks," in *IAAI*, 2008.

[9] B. Tanner and A. White, "RL-Glue : Language-independent software for reinforcement-learning experiments," *JMLR*, vol. 10, pp. 2133–2136, September 2009.

[10] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "PyBrain," *JMLR*, vol. 2010, p. 4, 2010.

[11] T. Bauer, M. Erwig, A. Fern, and J. Pinto, "Adaptation-based programming in java," in *PEPM*, ser. PEPM '11. ACM, 2011, pp. 81–90.

[12] J. Pinto, A. Fern, T. Bauer, and M. Erwig, "Improving policy gradient estimates with influence information," *ACML*, vol. 20, pp. 1–18, 2011.

[13] http://nsnam.isi.edu /nsnam /index.php /UserInformation. The official ns-2 webpage.

[14] http://trace.eas.asu.edu/mpeg4/index.html. Trace files and statistics: Mpeg-4 part 2 video trace library.

[15] F. Kelly and T. Voice, "Stability of end-to-end algorithms for joint routing and rate control," *SIGCOMM Comput. Commun. Rev.*, vol. 35, pp. 5–12, April 2005.

[16] http://omni.bus.ed.ac.uk /javasim /usermanual /usermanual.html. The official javasim webpage.

[17] http://www1.cse.wustl.edu / jain /cse567-08 /ftp/simtools /index.html. A survey of network simulation tools: Current status and future developments.

[18] C. Simpkins, S. Bhat, C. L. I. Jr., and M. Mateas, "Towards adaptive programming: integrating reinforcement learning into a programming language," in *OOPSLA*, 2008, pp. 603–614.

[19] D. Andre and S. J. Russell, "Programmable reinforcement learning agents," in *NIPS*, 2000, pp. 1019–1025.

[20] ——, "State abstraction for programmable reinforcement learning agents," in *AAAI*, 2002, pp. 119–125.