Model-based Random Early Packet Dropping

Hyeong Soo Chang*, Robert Givan*, and Edwin K. P. Chong*

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
{hyeong,givan,echong}@ecn.purdue.edu

Abstract -- We study the use of a model of network traffic in making early packet-dropping decisions in the overload control of a finite buffer — our goal is to provide a controller that selects early dropping in order to minimize queueing delay given a fixed constraint on loss. We empirically compare several simulation-based approaches to the problem of incorporating a traffic model into selecting dropping, including two novel simulation-based policies called “policy switching” and “parallel roll-out” (the latter an extension to Bertsekas’ “roll-out” approach [7]). We show that multiple previously-published simulation approaches do not improve substantially on non-model-based controllers for this problem (including RED [16]), whereas our parallel roll-out approach reduces queueing delay by nearly a factor of two for generic simulated test traffic. In addition, we provide a new characterization of the optimal control sequence for this problem when the traffic is fully known in advance, along with an algorithm for offline selection of this optimal control — we use this new algorithm to give an upper bound on the achievable performance for a given traffic. Our early results show that the parallel rollout approach reduces queueing delay very nearly to this theoretical limit.

I. INTRODUCTION

This paper studies how the use of a model of network traffic can be effective in overload control of a finite buffer via random early packet dropping. We use the model to predict future packet arrivals by sampling and incorporate the sampled futures into the control of early packet dropping. We design and evaluate two novel decision-theoretic approaches involving the efficient use of online simulation — our “parallel roll-out” approach extends Bertsekas’ “roll-out” approach [7], and our “policy switching” approach uses sampling to evaluate policies as suggested by Kearns [23]. Both of these approaches assume that we start with a small set of simple heuristic policies that we wish to combine in an online fashion to generate a single controller. We compare several model-based and non-model-based approaches to the problem on simulated test traffic, and establish that buffer management can benefit significantly from a given or learned traffic model in reaching the goal of providing high throughput with low average delay. Our results show that previously-published simulation-based control approaches do not improve on simple non-model-based controllers (e.g. random early dropping), but that our parallel roll-out approach reduces queueing delay by nearly a factor of two.

We also study the problem of selecting the optimal dropping sequence offline, i.e., when the precise traffic is known ahead of time. This problem is closely related: it is a special case of having a more general traffic model. We provide a new characterization of the optimal control sequence in this case, along with an efficient algorithm for computing this sequence, with time complexity $O(H \log H)$ for traffic of length $H$. This algorithm can be used to find a theoretical upper-bound on the performance achievable on a given traffic sequence by an online controller (as was done for scheduling in [37]), and is also needed as the key step in implementing one of the competing approaches we test for using simulation in selecting a control action. Using this offline method to generate an upper-bound on possible performance for the traffic tested, we show empirically that our parallel roll-out method comes very close to achieving this bound.

We now discuss the problem we studied in more detail. We consider a controller managing the queue size in a finite buffer with a single server. A finite number of packets of a fixed length arrive into the buffer at each (discrete) time. The buffer can hold at most $N$ packets at once. At each time step, the server may drop some packets before it transmits one packet. We wish to determine how many packets need to be dropped at each time to provide a high throughput and a low queueing delay experienced by the packets transmitted, where the desired trade-off between these goals is set by a system parameter.

The advantage of controlling queue size via early dropping can be seen even in the simple situation where arrivals occur at a constant rate higher than the service rate (in our setting, this means more than one packet arrives per time step, constantly). Without early dropping the queue will fill up, and then every packet arrival but one will be dropped at each time step due to buffer overflow. This

---

* This research is supported by the Defense Advanced Research Projects Agency under contract F19628-98-C-0051. The equipment for this work was provided in part by a grant from Intel Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.
results in the maximum achievable throughput but also in unnecessary queueing delay for those packets that are transmitted because the queue is kept full all the time by the heavy arrivals. A queue management policy that recognizes this situation and responds by keeping only the packet to be served next in the queue achieves the same throughput but no queueing delay at all. Of course, if the constant traffic were to fall off suddenly, this queue management policy would suffer a loss in throughput because there would be no queued packets to serve — thus, any queue management strategy involving early dropping must base its dropping decisions on (implicit or explicit) expectations about the future traffic. In this work, we explore controlling dropping using an explicit stochastic model of the future traffic.

The early dropping problem has been studied as the buffer management problem in the networking community, or the customer rejection vs. acceptance problem in the operations research (OR) literature over the last two decades. The methods proposed to date are different from our approach primarily in four ways. First, we have found no previous work that attempts to select a dropping policy via simulation. Second, most previous dropping policies are based only on current buffer contents, not on the state of the current network environment (e.g. Floyd and Jacobson [16]; Romanov and Floyd [41]; Li [41]; Hashem [19]; Lam and Reiser [26]; Saad and Schwartz [43]; Elwalid and Mitra [13]) — in contrast, our approach maintains a model of the expected future traffic. Third, our system observes only packet arrivals and estimates the current environment of the network from the observed arrivals (i.e., we have partial observability) — the few systems that have used traffic models have assumed full observability (e.g. Kulkarni and Tedijanto [25]; Helm and Waldmann [20]). Finally, much related work (e.g. Lazar and others in [9],[22],[24], and [47]) has focussed on maximizing throughput subject to a delay constraint, or on maximizing weighted throughput in a multiclass setting with no regard for delay (e.g. [25]; Miller [32]); we focus here on the problem of minimizing delay subject to a throughput constraint.

Distinguished among related work is the well-known Random Early Detection (RED) approach of Floyd and Jacobson [16]. RED uses an exponential weighted moving average (EWMA) to estimate average queue size and then drops packets according to a set of average queue-size thresholds with a drop probability determined by the estimated queue size; RED was specifically designed to control the buffer when facing responsive input traffic that uses Transmission Control Protocol (TCP). The performance of RED is very sensitive to the setting of the control parameters (e.g., the weight parameter of the moving average) and it is not clear how to set the parameters optimally. We compare our control policies to RED; our empirical work assumes that RED parameters are tuned for optimal performance on the given test traffic; however, RED can be expected to perform substantially worse in practice, magnifying our gains.

The closest previous work is also our own work — we have developed a “hindsight optimization” framework for designing network control using simulation. This framework is described further in Section III below, and is shown here to perform little better than the model-free methods such as RED (as just noted, however, our empirical work below assumes optimal tuning of the relevant parameters — such tuning is notoriously difficult for RED but rather straightforward for our approaches).

The new policy presented here, which we call Model-based Random Early Dropping (MRED), determines the number of packets to be dropped at each time step based on both the current instantaneous queue size and the (estimated) network environment. In this work, the network environment is represented by a traffic model capturing our expectations about future traffic — the current state of this model is hidden from the controller and must be estimated by observing the actual traffic.

The MRED policy is based on a control-theoretic sampling framework we call “parallel roll-out”, a novel extension of the “roll-out” framework presented by Bertsekas [7]. This framework provides a sampling-based heuristic policy for any problem formalized as a Partially-Observable Markov Decision Process (POMDP), given a small set of more primitive (or simple) heuristic policies to choose between. We discuss below how the buffer management problem can be viewed as a POMDP, and in turn then as an information-state MDP. The parallel roll-out approach provides a sampling technique for computing a lower-bound estimate of the value of each possible control action — a number of potential future traffic traces is generated, and then for each candidate action an evaluation against these traces is performed. The action with the best average performance in these offline evaluations is then taken. Each action evaluation involves simulating each of the primitive policies after taking the candidate action, and using the maximum of the values obtained as a lower-bound estimate of the value of the action. We can show that the resulting online policy performs better than any of the primitive policies used to generate it (assuming infinite sampling).

Parallel roll-out extends the original “roll-out” approach proposed by Bertsekas in [7]. This approach uses a similar technique to select an online policy that improves similarly on a single primitive policy (Bertsekas’ approach can be viewed as parallel roll-out where
A. Markov Decision Processes

We present here a brief summary of the essentials of the MDP formal model for a dynamic system. For a more substantial introduction, please see [5]. Consider a discrete-time dynamic system with $H$-horizon $x_{t+1} = f(x_t, u_t, w_t)$, for $t \in \{0, 1, ..., H-1\}$, where the function $f$ is given, and $x_t$ is the state in a set $X$, $u_t$ is the control to be chosen.
from a finite nonempty subset \( U(x_i) \) of \( C \), and \( w_t \) is a random disturbance, which determines the uncertainty of the system (for our problem, this corresponds to one-step random packet arrivals). The system operates over a long finite number of stages. We assume that \( w_t \) is selected randomly from the uniform distribution \([0,1]\) and independently from \( w_0, \ldots, w_{t-1} \). Note that a probability distribution \( p(x_{t+1} \mid x_t, u_t) \) that stochastically describes a possible next state of the system can be derived from \( f \).

Consider the control of the above system. The class of nonstationary policies consists of a sequence of functions \( \pi = \{ \mu_0, \mu_1, \ldots, \} \), where \( \mu_i : X \rightarrow C \) and such that \( \mu_i(x_i) \in U(x_i) \). Given an initial state \( x_0 \), the problem is to find an optimal policy \( \pi \) that maximizes the reward functional (total expected reward over \( H \))

\[
V_H^\pi(x_0) = E_{w_0, \ldots, w_{H-1}} \left\{ \sum_{t=0}^{H-1} r(x_t, \mu(x_t), w_t) \right\}
\]

subject to the system equation constraint \( x_{t+1} = f(x_t, u_t, w_t) \), \( t = 0, 1, \ldots, H-1 \), and the real-valued function \( r : X \times C \rightarrow [0,1] \rightarrow R \). The function \( f \), together with \( C, U, \) and \( r \) make up a Markov decision process (MDP). We now define

\[
V_{H-i}^*(x_i) = \max_\pi \ E_{w_i, \ldots, w_{H-1}} \left\{ \sum_{t=i}^{H-1} r(x_t, \mu(x_t), w_t) \right\},
\]

which is the maximally achievable expected reward over the remaining \( H-i \) horizon given state \( x_i \) at time \( i \), and is called optimal value of \( x_i \) for the \( H-i \) horizon. Then from the standard MDP theory, we can write recursive optimality equations given by

\[
V_{H-i}^*(x_i) = \max_{u \in U(x_i)} \left\{ r(x_i, u, w_i) + V_{H-i-1}^*(x_{i+1}) \right\},
\]

where \( x_i \in \{0, \ldots, H-1\} \), \( x_{i+1} = f(x_i, u, w_i) \), and we assume \( V_0^*(x_H) = 0 \). We write

\[
Q_{H-i}(x_i, u) = E_{w_i} \left\{ r(x_i, u, w_i) + V_{H-i-1}^*(x_{i+1}) \right\}
\]

for the \( Q \)-value of control \( u \) at \( x_i \). In particular, we will write

\[
Q_{H-i}^V(x_i, u) = E_{w_i} \left\{ r(x_i, u, w_i) + V(x_{i+1}) \right\}
\]

for the \( Q \)-value of \( u \) when \( V^* \) is taken to be \( V \). Then, the policy defined by

\[
\mu_i^*(x_i) = \arg\max_{u \in U(x_i)} Q_{H-i}(x_i, u)
\]

is an optimal policy achieving \( V_H^*(x_0) \). In particular, for a fixed horizon \( H \), the control \( u^* = \mu_0^*(x_0) \) is an optimal “current” action.

---

1. Dividing the total expected reward over a finite horizon by the horizon gives expected average reward per time step over the horizon.

---

**B. Traffic Modeling**

The MRED policy selects its control based on a model of the expected future traffic. There are multiple possible rationales for assuming that such a model may be available. First, if model-based control methods are found to be advantageous (as the are here) in a wide array of contexts, network flows could be expected to provide a stochastic model matching their usage patterns — such a model could be selected from a library of typical usage patterns (e.g. MPEG video, email, etc.). Alternatively, a model inference algorithm (e.g., Expectation Maximization (EM)) operating at a coarser time scale can be used to learn a model that approximates the observed traffic pattern \([4][46]\). In future work we expect to demonstrate that a model learned by EM is adequate to enable MRED to achieve substantial advantage over techniques lacking access to any model at all.

We use HMMs as traffic models as a straightforward starting point for our work. It is well-known that HMM can capture a wide variety of interesting traffic and approximate self-similar traffic.

We assume that time \( t \) is discrete, i.e., \( t \in \{0,1,2,\ldots\} \), and that each packet takes exactly one time step to be served. The random disturbance \( w_t \) describes the random packet arrivals in the time interval \((t-1, t]\). Suppose we have \( M \) traffic sources where each source generates at most \( K \) packets per time step according an HMM model.\(^2\)

The HMM model for source \( i \) has a finite set of states \( \Delta_i \), and a state \( s \in \Delta_i \) is associated with a packet generation probability distribution \( G_i^s \) and a next state transition probability distribution \( F_i^s \) — a state \( s \in \Delta_i \) generates \( k \) packets according to the probability of \( G_i^s(k) \) and then make a stochastic transition to \( s' \) with probability \( F_i^s(s') \).

**C. The Buffer Management Problem as an MDP**

In this section we formulate the buffer management problem as an MDP.

1) The State Space X

The key difficulty is in formulating the state space \( X \). A natural state space to consider is the space of possible queue states (there is one such state for each possible queue length) crossed with the space of possible traffic source states (one for each way of selecting an HMM state \( x_i \) for each traffic source \( i \) from the corresponding HMM state space \( \Delta_i \)).

However, for our problem, the current state from this

---

\(^2\) There are alternative fitting algorithms based on moment matchings (e.g.[18]), but the results cannot be used for sampling methods.

\(^3\) The superposition of multiple HMMs is also an HMM. Therefore, this formulation allows for aggregate traffic even though we use one HMM source.
state space would not be fully observable to our controller — because the state of each traffic source is hidden from the controller, which can only observe the traffic generated. As a result, the dynamic system for the buffer management problem is partially observable. One standard way (e.g. see [29]) to handle partial observability is to use a state space consisting of the information state of the controller. We observe that the controller can maintain a probability distribution, called a belief state, over the possible hidden state assignments describing the traffic sources. This distribution can be updated using Bayes’ rule each time new arrivals are generated from the sources (we assume that each arrival is labelled as to which source generated it). A belief state represents the controller’s estimation of the current network environment, and is a summary of the history of the system.

Our state space can now consist of the possible queue states crossed with the possible belief states — so that a system state will be a pair of a queue state and a belief state. We note that this state space is continuous and infinite. Transition probabilities between states can be calculated (as just mentioned) using Bayes’ rule based on the known arrival models $G_i$ and transition models $F_i$ for the individual sources, as shown below. We now formally define the state space for the MDP form of the buffer management problem.

Let $\Pi_{i,t+1}$ be a probability distribution over the states in $\Delta_i$ at time $t+1$, giving the probability that each state is in fact the traffic source $i$’s state at time $t+1$. Given $k_i$ packet arrivals from the source $i$ at time $t$, we can update $\Pi_{i,t+1}$ by applying a form of Bayes rule given by

$$\Pi_{i,t+1}(s) = \alpha G_i^t(k_i) \sum_{s' \in \Delta_i} F_i^t(s') \Pi_{i,t}(s'),$$

where $\alpha$ is a normalizing factor selected so that the $\Pi_{i,t}(s)$ values sum to one over all $s$ in $\Delta_i$.

Let $b_i$ be the number of packets in the system immediately prior to the $t$'th arrival instant and immediately after one packet is served at time $t$-1, if any. The number of packet arrivals during $(t-1,t)$ is denoted by $a_i$. The state $x_t$ of our problem then has two components: the set of probability vectors $\Pi_i$ for $i \in \{1, ..., M\}$ and $l_{i,t} = a_i + b_i$, where we can view $\Pi_i$ as the state estimate of the traffic source $i$, and $l_{i,t}$ as the number of packets, or the current queue size before service. The precise relationship between $b_i$ and $l_{i,t}$ is given by

$$b_i = \max(0, l_{i,t-1} - u_{i,t-1} - 1, u_{i,t-1} \in U(x_{t-1}))$$

$$l_{i,t} = a_i + b_i$$

We note that the state space just described has the Markov property — given the state at time $t$, the future trajectory of states is independent of any previous state.

2) The Control Sets $U(x_t)$

The admissible control set $U(x_t)$ for $x_t$ is the number of packets that can be dropped before the server transmits one packet. We are forced to drop packets if there is buffer overflow, and we can drop any additional packet currently pending in the buffer, leaving one to be served. That is, $U(x_t) = \{i_t - N, ..., l_t - 1\}$ if $l_t > N$, $\{0, ..., l_t - 1\}$ if $0 < l_t < N$, and $\{0\}$ if $l_t = 0$ (i.e., the buffer is empty).

3) The Reward Function $r(x_t, u_t)$

Maximizing throughput and minimizing average delay are conflicting objectives. In a networking context, reasonable throughput can naturally be taken to be a higher priority objective; in this view the goal is to minimize queuing delay subject to a given minimum bound $\alpha$ on throughput. Our empirical results in Section V show the queuing delay achieved by early dropping strategies that meet various constraints on $\alpha$ — looser constraints on $\alpha$ allow more aggressive strategies to reduce queuing delay. We design our system to minimize delay while attempting to meet a given throughput constraint heuristically.

Our selection of a reward function $r$ must is constrained by two issues: first, the MDP formalism we use requires the reward associated with a given time step to be determined independent of the system state at other times (we say the objective must be time-decomposable); second, the reward function must be inexpensive to compute in order to be used in our online simulations at low cost.

One option available is to study the constrained MDP optimization problem of finding a minimum delay policy under a given throughput constraint. There has been previous work addressing constrained MDP optimization [1] that could be applied here. However, our heuristic framework does not at this point apply to constrained MDPs, so we instead formulate a reward function that rewards throughput and penalizes delay according to a trade-off parameter $\lambda > 0$ that corresponds roughly to the constraint $\alpha$ above. Specifically, we set the goal of maximizing $T_{\text{avg}} - \lambda L_{\text{avg}}$, where $T_{\text{avg}}$ is the average throughput over a finite horizon $H$, and $L_{\text{avg}}$ is the average queue length over $H$. To maximize $T_{\text{avg}} - \lambda L_{\text{avg}}$, we define the one-step reward $r(x_t, u_t)$ of taking control $u_t$ at state $x_t$ to be $1 - \lambda(l_{t} - u_t)$ if $l_t - u_t > 0$ and 0 otherwise. This reward function has the following property: the total reward at horizon $H$ is equal to the difference of the total throughput and the total queue length weighted by $\lambda$ (here, the total queue length is the sum of the queue lengths before service at each time step, and is an easier-to-compute stand-in for the total delay experienced). Note that using queue length as a measure of delay corresponds to the assumption that all packet dropping is done at admission.
to the queue — however, in our setting we allow packets to be dropped at later time steps as well. Nevertheless, we have found that this reward function accurately guides the controller to achieve low queuing delay given throughput constraints, as shown in our empirical results.

III. FINDING A MODEL-BASED EARLY-DROPPING POLICY

In this section, we briefly discuss alternative approaches to MDP control, and present the policy switching and parallel rollout techniques for heuristically controlling an MDP system with online simulation.

A. Previous Approaches to Solving MDPs

1) Dynamic Programming Approaches

Several approaches have been developed for solving or approximately solving MDP problems like those just described in time polynomial in the size of the state space $X$ (see Puterman [40] for a discussion of stochastic dynamic programming, value iteration, policy iteration, and linear programming approaches to such problems). Unfortunately, the state space for most natural problems is at least exponential in size relative to the natural problem description (as discussed above, the state space size for our problem is actually infinite). Large state spaces generally make dynamic programming solutions infeasible for practical MDP problems, including ours.

2) Sampling to Compute Approximately Optimal Control

Sampling possible trajectories is an alternative approach to exact solution via dynamic programming. However, computing a true approximation via sampling is also intractable — mainly due to the need to approximate the value of a nested sequence of alternating expectations over the entire state space and maximizations over the action choices that represents the “expected trajectory” of the system under optimal control. See Kearns et al. [23] for a description of a method for computing such an approximation using a number of samples independent of the state space size. Kearns derives bounds on the number of samples needed to give a near-optimal controller; these bounds grow exponentially with the accuracy desired. Our empirical work (not shown here) has indicated that obtaining reasonable approximations of the true value of a control action via this method is impractical (see [10] for more details).

3) Sampling to Improve a Given Policy by “Roll-out”

Sampling has also been used by Bertsekas [6] [7] to design a heuristic method of policy improvement called roll-out. This method uses sampling to improve a given heuristic policy $\pi$ in an online fashion: at each time step, the given policy is simulated using sampling, and the results of the simulation are used to select the (apparently) best current action (which may differ from that prescribed by the given policy). The action selected is the action with the highest $Q_{\pi}(s)$-value at the current state, as estimated by sampling. It is possible to show that the resulting online policy outperforms the given heuristic policy. Roll-out is a viable practical alternative and our empirical results below include results for buffer-management policies selected by roll-out.

4) Hindsight Optimization Sampling

The method of the hindsight optimization ([10] and [11]) computes an upper bound on the true $Q$-value of each action by simply interchanging the order of expectations and maximizations in the recursive definition of $Q$-value, applying Jensen’s inequality. More precisely, we upper bound the true $Q$-value for a current control choice $u_0$ at $x_0$ with the hindsight-optimal $Q$-value $Q_H$.

$$\hat{Q}_H(x_0, u_0) = \mathbb{E}_{w_0, \ldots, w_{H-1}} \left\{ r_0 + \max_{u_1, \ldots, u_{H-1}} \sum_{t=1}^{H-1} r_t \right\},$$

where $r_t=r(x_t, u_t, w_t)$ and $H_s \leq H$ is the sampling horizon. Because the hindsight optimal $Q$-value is simply a mean of a function over $[0,1]^H$, Monte Carlo simulation can estimate this value quickly. In theory, the number of the sampled random disturbance sequences required to estimate the mean does not depend on $C$ or $H$ [45]. Various techniques (e.g., importance sampling [42]) can reduce the variance of the estimate, if desired. We note that using the same sampled random disturbance sequences across controls for calculating the hindsight-optimal $Q$-values reduces the error variance among the hindsight optimal $Q$-values. This has the same flavor as common random number simulation in discrete event systems [21] and also resembles differential training for the roll-out algorithm mentioned above [6].

Based on this method for estimating $\hat{Q}_H(x_0, u_0)$ values, a hindsight-optimal controller generates a set of sampled traces of future packet arrivals. For each possible current control, the controller then solves the (deterministic) hindsight optimization problem to get the best subsequent control sequence for each sampled trace, and calculates the reward sum achieved by following that control sequence. Averaging these reward sums over the set of sampled traces will give the (approximated) hindsight-optimal $Q$-value for the selected control. The control with the highest hindsight optimal $Q$-value is then taken. Please see [11] for more details.

B. New Approaches to MDP Control

The new approaches we describe here assume that we have a small set $\Pi$ of simple “primitive” heuristic poli-
cies \( \{\pi_1, \ldots, \pi_m\} \) that we wish to combine into an online controller. For the buffer management problem, we assume that we have such a policy “buffer-k” for each queue size \( k \) between 1 and the buffer size \( N \). The policy buffer-k selects the minimum dropping so that the queue never exceeds length \( k \) (i.e. buffer-k behaves like a drop-tail policy with buffer size \( k \)). We now present two methods for combining online a finite set of policies.

Both approaches rely on the fact that we can approximate the value function \( V^*_\pi \) for each \( \pi \) in \( \Pi \) by sampling/simulation (see Kearns [23] for example).

1) Parallel Rollout

The parallel rollout approach selects the action with the highest \( Q^V \) value at the current state, for \( V = \max_{\pi \in \Pi} V_\pi \). The resulting policy (assuming infinite sampling to properly estimate \( V_\pi \)) can be proven to compute a policy that returns value no less than the value function \( V \) (i.e. better than any policy in \( \Pi \)). For our buffer management problem, parallel rollout can thus do no worse than the best of the fixed buffer size buffer-k policies.

2) Policy Switching

This policy simply selects the action given by the policy \( \pi \) in \( \Pi \) that have the highest \( V_\pi \) estimate at the current state. This policy resembles the parallel rollout policy in that it must compute the value function \( V_\pi \) for each \( \pi \) in \( \Pi \), but it does not do the one-step look ahead involved in computing the \( Q^V \) function for each action.

We have now reviewed a variety of MDP control approaches, most of which will be compared in our empirical results on the buffer management problem, as well as briefly presented two novel approaches. Before presenting our empirical results, we first describe new methods for offline solution of the same problem — these methods allow us to find a theoretical limit on performance for use in evaluating our control policies.

### IV. Offline Buffer Management

The hindsight-optimization framework discussed above requires that we provide a method for offline solution of the MDP problem once the random disturbances \( w_t \) are known. Moreover, offline solution of this problem provides a theoretical upper-bound on performance attainable for any given traffic trace. For buffer management, the offline problem corresponds to finding the optimal dropping plan for any given traffic arrival sequence. In this section we give a novel characterization of the optimal dropping plan, provide a new algorithm for computing this optimal plan, and prove the correctness and claimed time complexity for that algorithm.

### A. Defining and Characterizing the Optimal Plan

Throughout this section, we focus on a fixed arrival sequence \( a_t \in \{0, \ldots, H_s-1\} \), where \( H_s \) is our sampling horizon, and discuss how to find an optimal dropping plan for that sequence. All the definitions below are relative to this fixed arrival sequence.

We begin by formally defining the problem. A dropping plan is a sequence of nonnegative integers \( d = [d_0, d_1, \ldots, d_{H_s-1}] \) giving the number of packets \( d_i \) to drop at each time \( t \). Given a dropping plan \( d \), the unbounded queue-length trajectory \( l_t(d) \) under that plan is given by the following evolutionary equations:

\[
\begin{align*}
l_0(d) &= a_0 - d_0 \\
l_{i+1}(d) &= \min(0, l_i(d) - 1) + a_{i+1} - d_{i+1}
\end{align*}
\]

Note that there is no limit on buffer size incorporated here. The plan \( d \) is said to meet buffer constraint \( k \) if \( l_t(d) \) is not exceed \( k \) for any \( t \). The cumulative reward associated with a dropping plan is given by

\[
R(d) = \sum_{t=0}^{H_s-1} \text{sgn}(l_t(d)) - \lambda(l_t(d)),
\]

and we say that a dropping plan is optimal if it meets buffer constraint \( N \) and achieves cumulative reward on the given arrival sequence \( a_t \) no less than any other plan meeting constraint \( N \).

We now define an optimal plan \( \delta \) and prove its optimality. The intuition behind this plan is that our reward function \( R(d) \) is such that we prefer to drop any packet that must wait \( 1/\lambda \) time steps for service. It follows that we must drop just enough the number of packets must wait \( 1/\lambda \) time steps — moreover, we must drop any packet that will be dropped (either for the reason just given or for buffer overflow) the moment it arrives, to minimize queue length. We now formalize these ideas.

Given a dropping plan \( d \), the next empty time at time \( t \), denoted \( n(d, t) \), is the least time \( t' \geq t \) such that \( l_{t'} \leq 1 \). This is the first time not before \( t \) at which the queue will be empty after service if we follow plan \( d \), ignoring the limit on buffer size. We say that a dropping plan \( d \) is acceptable at time \( t \) if the next empty time at time \( t \), \( n(d, t) \), is less than \( \min(t + (1/\lambda)H_s^{-1}) \), and the queue trajectory \( l_{t'}(d) \) is less than or equal to the buffer size \( N \) for all times \( t' \) in the interval \([t, n(d, t)]\). We note that acceptability is monotone relative to increased dropping: if \( d \) is acceptable at time \( t \), than any plan \( d' \) that drops no less than \( d \) at each time step is also acceptable at \( t \). Also, this definition ensures that an acceptable trajectory leaves the queue empty at the horizon, after the last service interval.

In the definition below, we treat a partial dropping plan \( [d_0, \ldots, d_t] \) for some \( t \leq H_s \) as a full dropping plan by assuming that \( d_{t'} = 0 \) for all \( t' \) in the interval \((t, H_s] \).
Note that under this interpretation, [] is the empty dropping plan (drops no packets).

**Definition 1:** The dropping plan \( \delta = [\delta_0, ..., \delta_{H_{s-1}}] \) is defined recursively as follows: for all \( i \geq 0 \), \( \delta_i \) is the smallest \( x \) such that the time \( i \) is acceptable relative to the dropping plan \( [\delta_0, ..., \delta_{i-1}, x] \).

**Theorem 1:** \( \delta \) is an optimal plan for arrival sequence \( a_r \).

**B. A Method for Computing the Optimal plan**

In Fig 1 we give pseudocode for an \( O(H_s \log H_s) \) algorithm for computing the plan \( \delta \) described in the previous subsection (\( \delta \) is called plan[ ] in the pseudocode), and we explain the design of this code here.

The algorithm assumes we have already computed the unbounded queue-length trajectory \( I_t([l]) \) for no dropping and stored the trajectory in traj[t], for each \( t \) — this is easy to do in \( O(H_s) \) time. The main loop of the algorithm scans this trajectory using a time-indexed variable looktime. A second time index variable droptime also moves through the time interval \( [0, H_s) \) in a monotonically increasing fashion, marking the location in plan[ ] that is currently being updated. As droptime is increased, the variable offset is maintained to summarize the relevant effect of the dropping actions selected for time indices 0 through droptime – 1 — specifically, the difference between the no-dropping trajectory \( I_t([l]) \) and the trajectory under the dropping policy given by the current plan[ ], where the difference computed applies at any time \( t \) between droptime and the next empty time.

The variable droptime does not have to take on every value in \( [0, H_s-1] \) due to the following property. Given dropping action \( \delta_j \) from the plan \( \delta \), the definition of acceptability implies that the actions \( \delta_j \) for all \( j \) in the interval \( (i, n) \) for \( n = n([\delta_0, ..., \delta_i]) \) are all zero. This fact justifies updating droptime to the next empty time plus one after each dropping action is selected.

During the looktime scan (i.e. in the body of the main for loop), the following computations are performed to select dropping actions. Whenever looktime encounters an overflow, or reaches a point at least 1/\( \lambda \) steps ahead of droptime without finding the next empty time, it is necessary to add to the amount of dropping done at droptime in order to make plan[ ] acceptable at droptime. The amount of dropping added is selected (in the lines using “argmin” and “min” to set dropamt) to ensure that either the buffer overflow is prevented, or that a queue-empty time occurs between droptime and looktime (the latter is required if looktime is at \( 1/\lambda \) steps ahead of droptime).

If the amount of dropping added results in an empty queue between droptime and looktime, then droptime is advanced to one more than this empty time. Likewise, if the looktime scan encounters a point at which the trajectory is empty under plan[ ], then droptime is advanced beyond this point.

The operation “argmin” is used in the pseudocode shown in two places, in each case to select the minimum amount of dropping to add to the plan at time droptime in order to cause the next empty time under the resulting plan to be earlier than looktime. This use of “argmin”...
ensures that we drop only the minimum number of packets at `droptime` to either avoid buffer overflow at `looktime` or ensure a next empty time before `looktime`. Specifically, “argmin” is used to compute the lowest point in the unbounded dropping-free trajectory between `droptime` and `looktime` (breaking ties in favor of the latest time) — it follows that dropping (at `droptime`) all but one packet in the current trajectory that at time is the least additional drop at `droptime` that results in an empty time in the interval `[droptime, looktime]`. We then drop that amount if an empty time is sought (i.e., after the first use of “argmin” in the pseudocode) or possibly less than that amount if we are only seeking to avoid buffer overflow (i.e., after the second use of “argmin”).

C. Correctness and Time Complexity

Theorem 2: The code in Fig. 1 computes the plan \( \delta \).

Proof: The following invariants can be proven together by induction to hold of the code in Fig. 1.

- The variables `offset` and `droptime` are maintained so that the variable `offset` takes the value \( l_t(\text{plan}) - l_t(\text{plan}) \) on each entry to the `for` loop body, for any \( t \) in the interval `[droptime, looktime]` (the difference does not depend on \( t \) in this interval).
- The variable `droptime` is always less than or equal to `looktime` on entry to the `for` loop body.
- `plan[t]` = \( \delta_t \) for all \( t < \text{droptime} \).
- `plan[t]` = 0 for all \( t > \text{droptime} \).
- `plan[t]` ≤ \( \delta_t \) for \( t = \text{droptime} \).
- `traj[droptime]-offset > 1` on entry to the main `if`.
- On entry to the `for` loop, the unbounded queue-length trajectory for the dropping specified in `plan[]` shows neither buffer overload nor a queue of size 0 or 1 in the time interval `[droptime, looktime]`.
- The variable `looktime` is never greater than `droptime+1/\lambda`, when the `for` loop is entered.

These invariants, together with the easily proven fact that `droptime = H_s` on termination establish that `plan[t] = \delta_t` for all \( t \) when the algorithm terminates. Q.E.D.

Theorem 3: The code in Fig. 1 runs in \( O(H_s \log H_s) \) time.

Proof: We claim that the two argmin operations can be supported by a priority queue implemented as a heap in \( O(\log H_s) \) time per invocation. The full paper will show and discuss pseudocode detailing this implementation.

The number of iterations of the main loop is clearly \( H_s \). Each operation in the body of this loop, excluding the `while` loop, takes at most \( \log(H_s) \) time, so the amount of time spent in the main `for` loop altogether (excluding the inner `while` loop) is \( O(H_s \log H_s) \).

To conclude, we must argue that the total time spent in the `while` loop is at most \( O(H_s \log H_s) \). We first observe that the `while` loop body takes at most \( \log H_s \) time for a single iteration, and then argue that over the entire algorithm at most \( H_s \) iterations occur. We divide such iterations into two groups: terminal iterations and continuing iterations, where a “terminal iteration” is one after which the `while` loop terminates, and a “continuing iteration” is followed by yet another iteration of the loop. We note that there can be at most \( O(H_s) \) terminal iterations of (*) because such iterations are followed inevitably by the end of an iteration of the `for` loop. But it is easy to show that continuing iterations occur only when `droptime = traj[mintime]` − `offset` − 1 (i.e. `droptime` is less than `excess` due to the min{...} application), and that `droptime` is increased at every such iteration. It is also possible to show that `droptime` starts at 0, never decreases, and is never greater than \( H_s \). Therefore there can be at most \( O(H_s) \) increases in `droptime` and as a result at most \( O(H_s) \) continuing iterations. We conclude that the `while` loop iterates at most \( O(H_s) \) times. Q.E.D.

V. EMPIRICAL RESULTS

We implemented the MRED scheme in the Network Simulator (NS) [35], and this section provides the resulting simulation data. Our focus in the simulation study is to show to what extent the throughput provided by MRED is close to the optimal throughput provided by Droptail, while at the same time MRED gives a lower average queueing delay comparing with the previously proposed sampling approaches, RED, and any buffer-k policy. For our simulation, the buffer size \( N \) is 25 and each time maximally 7 packets can arrive.

A. Simulation Test Traffic Model

We have conducted experiments by selecting an HMM model for the arrival distributions. To give a fairly complex bursty traffic behavior, we randomly create an HMM subject to the following constraints: the HMM must have four 10-state statespace regions, where each region has a different average traffic load and a random pattern of traffic given that load. There is one highly loaded region (load=2), one medium (0.9), one low (0.3), and one very low (0.1) loaded region. From any state, there is a small probability (< 0.02) of leaving the current region and entering one of the other regions at its entry point. Selecting this probability to be small ensures that our controller has some ability to infer the system state.

Each region is a circularly structured Markov chain, where (state) transition probabilities and packet generation probabilities were selected randomly and then normalized to the desired region load. The resulting set of regions is finally tuned to set the overall traffic load close
to 0.75 to ensure a reasonable overall load.

We believe that the traffic model selected by this way is a good reflection of real network traffics by two natural assumptions. We can assume that the current network router experiences the traffic bursts which all of the packets in the bursts cannot be served or can be served but overall load is almost saturated. We can also assume that the network traffic has a multi-time scale behavior. As a future work, we are currently investigating the usage of EM to fit a real network traffic data into this model and extend our simulation studies.

B. Random Early Detection

We briefly review the RED buffer management scheme for reference. At each packet arrival instant, RED calculates a new average queue size estimate \( q_t \) using \( q_t \leftarrow (1-w_q)q_{t-1} + w_q l_t \) where \( w_q \) is the EWMA parameter and \( l_t \) is the instantaneous queue size at time \( t \). RED determines the dropping probability at time \( t \) from a function \( D: q_t \to [0, 1] \). The function \( D \) is defined by three parameters \( \text{minth} \), \( \text{maxth} \) and \( \text{maxh} \) as follows: if \( q < \text{minth} \), \( D(q)=0 \), and if \( q \geq \text{maxth} \), \( D(q)=1 \), and \( (\text{maxh}-\text{minth})^{-1}(q-\text{minth})\text{maxh} \) otherwise, as shown in Fig. 3. Please see [16] for a detailed discussion.

Note that RED parameter setting includes a familiar fixed- \( k \)-threshold dropping policy which we call the buffer- \( k \) policy by setting \( \text{minth} = 1 \), \( \text{maxth} = k \) and \( w_q = 1 \).

C. Simulation Results

We simulated each candidate approach over 62,500 time steps where one time step corresponds to 1.6 ms. In Fig. 3, we summarize our experiments by plotting each approach’s performance for a randomly selected traffic from the test model. These plots are unusual and simulation-intensive in the following manner: each plot corresponds to many runs of the corresponding algorithm with different parameter settings — the plot shows the maximum performance achieved at each value of \( \alpha \). This approach corresponds to the assumption that the system designer can select the approach parameters appropriately for the desired \( \alpha \) (our other experiments substantiate this assumption in general).

The horizontal axis of the figure signifies of the percentage loss tolerated from the optimal throughput (given a maximum buffer size of 25). In particular, for RED, we tested a wide variety of the RED parameters and plotted the maximum performance over many parameter settings, including the setting approach suggested by Floyd in [16]. Included in the RED parameter choices is the buffer- \( k \) for each possible \( k \). For each sampling-based approach, we used the same traffic, selecting the \( \lambda \)-values parameterizing the controller from [0.001,0.07], again plotting best performance. The (standard) rollout algorithm’s plot in the graph corresponds to the maximum over 25 plots: one for rollout with each possible buffer- \( k \) policy as the base policy. To give an idea of the importance of having a correct model, we also tested the parallel-rollout approach with the same traffic but using a simple 0.75-load Bernoulli model during control. As we can see from the graph, we establish that buffer management can benefit significantly from a given traffic model in reaching the goal of providing high throughput with low average delay. Making the use of a wrong model yields a poor performance. As we can expect, the rollout algorithm improves RED but the performance is the almost same as RED. Low (high) loss tolerance performance can be obtained by setting the \( \lambda \)-values to be small (large).

Our results show that previously-published simulation-based control approaches (as well as policy switching) do not improve on simple non-model-based controllers (e.g. RED), but that our parallel rollout approach reduces queueing delay by nearly a factor of two. Furthermore, we can see that the parallel rollout approach reduces queueing delay very nearly to the theoretical limit calculated as described in Section IV.
REFERENCES


