A Distributed Throttling Approach for Handling High Bandwidth Aggregates

Chee Wei Tan, *Student Member*, *IEEE*, Dah-Ming Chiu, *Senior Member*, *IEEE*, John C.S. Lui, *Senior Member*, *IEEE*, and David K.Y. Yau, *Member*, *IEEE*

Abstract—Public-access networks need to handle persistent congestion and overload caused by high bandwidth aggregates that may occur during times of flooding-based DDoS attacks or flash crowds. The often unpredictable nature of these two activities can severely degrade server performance. Legitimate user requests also suffer considerably when traffic from many different sources aggregates inside the network and causes congestion. This paper studies a family of algorithms that "proactively" protect a server from overload by installing *rate throttles* in a set of upstream routers. Based on an optimal control setting, we propose algorithms that achieve throttling in a distributed and fair manner by taking important performance metrics into consideration, such as minimizing overall load variations. Using ns-2 simulations, we show that our proposed algorithms 1) are highly adaptive by avoiding unnecessary parameter configuration, 2) provide max-min fairness for any number of throttling routers, 3) respond very quickly to network changes, 4) are extremely robust against extrinsic factors beyond the system control, and 5) are stable under given delay bounds.

Index Terms—Resource management, DDoS attacks, network security.

1 Introduction

This paper proposes a family of distributed throttling algorithms for handling high bandwidth aggregates in the Internet. High bandwidth aggregates, as well as persistent congestion, can occur in the current Internet even when links are appropriately provisioned, and regardless of whether flows use conformant end-to-end congestion control or not. Two prime examples of high bandwidth aggregates that have received a lot of recent attention are *distributed denial-of-service* (DDoS) attacks and *flash crowds* [7].

In a DDoS attack, attackers send a large volume of traffic from different end hosts and direct the traffic to overwhelm the resources at a particular link or server. This high volume of traffic will significantly degrade the performance of legitimate users who wish to gain access to the congested resource. Flash crowds, on the other hand, occur when a large number of users concurrently try to obtain information from the same server. In this case, the heavy traffic may overwhelm the server and overload nearby network links. An example of flash crowds occurred after the 9/11 incident [7], when many users tried to obtain the latest news from the CNN Web site.

- *C.W. Tan is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544. E-mail: cheetan@princeton.edu.*
- *D.-M. Chiu is with the Department of Information Engineering, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. E-mail: dmchiu@ie.cuhk.edu.hk.*
- *J.C.S. Lui is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. E-mail: cslui@cse.cuhk.edu.hk.*
- *D.K.Y. Yau is with the Department of Computer Science, Purdue University, 250 N. University St., West Lafayette, IN 47907-2066. E-mail: yau@cs.purdue.edu.*

Manuscript received 1 May 2006; revised 5 Aug. 2006; accepted 10 Aug. 2006; published online 8 Jan. 2007.
Recommended for acceptance by Y. Pan.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0106-0506.
Digital Object Identifier no. 10.1109/TPDS.2007.1034.

Note that network congestion caused by high bandwidth aggregates cannot be controlled by conventional flow-based protection mechanisms. This is because these traffic aggregates may originate from many different end hosts, each traffic source possibly of low bandwidth. One approach to handle high bandwidth aggregates is to treat it as a resource management problem and protect the resource *proactively* (i.e., ahead of the congestion points) from overload.

Consider a network server, say S, experiencing a DDoS attack. To protect itself from overload, S can install a *router throttle* at a set of upstream routers that are several hops away. The throttle specifies the maximum rate (in bits/s) at which packets destined for S can be forwarded by each router. An installed throttle will change the load experienced at S, which then adjusts the throttle rate in multiple rounds of feedback control until its load target is achieved. In [22], the authors proposed a throttle algorithm known as the additive-increase-multiplicative-decrease (AIMD) algorithm, and showed how such server-based adaptation can effectively handle DDoS attacks, while guaranteeing fairness between the deployment routers.

Although the AIMD throttle algorithm in [22] is shown to work under various attack scenarios, it also raises several important issues that need to be addressed, for example, the proper parameter setting so as to guarantee stability and fast convergence to the steady state. Addressing these issues, this paper leverages control theory to design a class of fair throttle algorithms. Our goal is to derive a throttle algorithm that is

1. highly *adaptive* (by avoiding unnecessary parameters that would require configuration),
2. able to *converge* quickly to the fair allocation,
3. highly *robust* against extrinsic factors beyond the system's control (e.g., dynamic input traffic and number/locations of current attackers), and
4. *stable* under given delay bounds.

While our discussion uses flooding DDoS attacks for context, our results are also applicable for congestion control involving a large number of distributed contributing sources. DDoS attacks complicate the network congestion control issue because additional detection mechanism such as statistical filtering [10], [14], [15], [20], [21] has to work jointly with overload control. In general, a robust defense mechanism has to minimize the disruption in terms of the amount of attackers' traffic getting through to the server and the time to bring down the congestion level without any knowledge of the number of compromised routers and the adopted attack distribution.

The balance of the paper is organized as follows: In Section 2, following the discussion in [22], we introduce the background and basic terminology of adaptive router throttling. We also comment on the limitations of the previous approach, and motivate the need for a formal approach in the design of a stable and adaptive system. In Section 3, we present a class of distributed throttle algorithms. In Section 4, we analyze the properties of our algorithms. Section 5 reports experimental results that illustrate various desirable properties of the algorithms. Related work is discussed in Section 6. Finally, the conclusions are drawn in Section 7.

2 Background of Router Throttling

In [22], the authors formulated the problem of defending against flooding-based DDoS attacks (i.e., one form of high bandwidth aggregates) as a resource management problem of protecting a server from having to deal with excessive requests arriving over a global network. The approach is *proactive*: Before aggressive traffic flows can converge to overwhelm a server, a subset of routers along the forwarding paths is activated to regulate the incoming traffic rates[1] to more moderate levels. The basic mechanism is for a server under attack, say S, to install a *router throttle* at a set of upstream routers several hops away. The throttle limits the rate at which packets destined for S can be forwarded by a router. Traffic that exceeds the throttle rate will be dropped at the router.

An important element in the proposed defense system in [22] is to determine appropriate throttling rates at the defense routers such that, globally, S exports its full service capacity U_S (in kb/s) to the network, but no more. These appropriate throttle rates should depend on the current demand distributions and, thus, must be negotiated dynamically between the server and the defense routers. The negotiation approach is *server-initiated*. A server operating below the designed load limit needs no protection, and so does not install any router throttles. As server load increases and crosses the designed load limit U_S, the server may start to protect itself by installing a rate throttle at the defense routers. If a current throttle rate fails to bring down the load at S to below U_S, then the throttle rate can be further reduced. On the other hand, if the server load falls below a low-water mark L_S (where $L_S < U_S$), then the throttle rate is increased to allow more packets to be forwarded to S. If an increase does not cause the load to significantly increase over some observation period, then the throttle is removed. The goal of our throttle algorithms is to keep the server load within $[L_S, U_S]$ whenever a throttle is in effect.

1. All traffic rate and server load quantities are in unit of Mbps, unless stated otherwise.

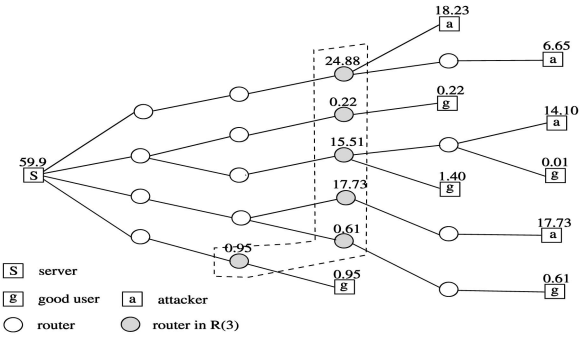


Fig. 1. Network topology illustrating $R(3)$ deployment points of router throttle. A square node represents a host and a round node represents a router. The host on the far left is the target server S. The deployment routers in $R(3)$ are shaded. The number above each host (or router) denotes the rate at which the host (or router) sends to S.

Router throttling does not have to be deployed at every router in the network. Rather, the deployment points are parameterized by a positive integer k, and are within the administrative domain of S. We denote this set of deployment points by $R(k)$. $R(k)$ contains all the routers that are k hops away from S and also routers that are less than k hops away from S, but are directly connected to an end host. To illustrate this concept, Fig. 1 shows an example network topology in which a square node represents a host and a round node represents a router. The host on the far left is the target server S. The deployment routers in $R(3)$ are shaded. Note that the bottom-most router in $R(3)$ is only two hops away from S, but is included because it is directly connected to a host. The number above each host (or router) denotes the rate at which the host (or router) sends to S. For this example, we use $U_S = 22$ and $L_S = 18$. Since the total offered rate of traffic to S is $\rho = 59.9$, S will be overloaded without protection.

In fairly allocating the server capacity among the routers in $R(k)$, a notion of *level-k max-min fairness* is introduced in [22]:

Definition 1 (level-k max-min fairness). *A resource control algorithm achieves level-k max-min fairness among the routers $R(k)$, if the allowed forwarding rate of traffic for S at each router is the router's max-min fair share of some rate r satisfying $L_S \leq r \leq U_S$.*

The AIMD algorithm in [22] (see Fig. 2) achieves level-k max-min fairness. It installs at each router in $R(k)$ a throttle rate r_s, which is the "*leaky bucket*" rate at which each router can forward traffic to S. Traffic that exceeds r_s will be dropped by each router. An installed throttle has very low runtime overhead as only a leaky-bucket throttle is required at each server that needs to handle high bandwidth aggregates.

In Fig. 2, r_s is the current throttle rate used by S. It is initialized to $(L_S + U_S)/f(k)$ where $f(k)$ is either some small constant, say 2, or an estimate of the number of throttle points needed in $R(k)$. The monitoring window W is used to measure the incoming aggregate rate. It is set to be

```
1. ρ_last := −∞;
2. r_s := (L_s + U_s)/f(k); /* initialize throttle rate */
3. While (1)
4.       sends current rate-r_s throttle to R(k);
5.       monitor traffic arrival rate ρ for time window W;
6.       If (ρ > U_S) /* throttle not strong enough */
                 /* further restrict throttle rate */
7.                r_s := r_s/2;
8.       else if (ρ < L_S) /* throttle too strong */
9.                If (ρ − ρ_last < ε)
10.                 remove rate throttle from R(k);
11.                 break;
12.               else
                  /* relax throttle by additive step */
13.                 ρ_last := ρ;
14.                 r_s := r_s + δ;
15.               end if;
16.      else
17.               break;
18.      end if;
19. end while;
```

Fig. 2. Pseudocode for the AIMD fair throttle algorithm.

somewhat larger than the maximum round trip time between S and a router in $R(k)$ so as to smooth out the variation of incoming aggregate rate. In Section 3.4, we examine the impact of the window size W in mitigating DDoS attack traffic. A constant additive step, δ, is used to ramp up r_s if a throttle is in effect and the current server load is below L_S. Each time S computes a new r_s that will be installed in $R(k)$, which causes the traffic load at S to change. The algorithm then iteratively adjusts r_s so that the server load converges to and stays within a point in $[L_S, U_S]$ whenever a throttle is in effect.

To illustrate the AIMD algorithm, consider the example shown in Fig. 1, with $U_S = 22$, $L_S = 18$, and a step size $\delta = 1$. We initialize r_s to $(L_S + U_S)/4 = 10$. Table 1a, reproduced from [22], shows the AIMD algorithm in action. When the algorithm is first invoked with throttle rate 10, the aggregate load at S drops to 31.78. Since the server load still exceeds U_S, the throttle rate is halved to 5, and the server load drops below L_S, to 16.78. As a result, r_s is increased to 6, and the server load becomes 19.78. Since 19.78 is within the target range [18, 22], the algorithm terminates. When that happens, the forwarding rates of traffic for S at the deployment routers (from top to bottom in the Fig. 1) are 6, 0.22, 6, 6, 0.61, and 0.95, respectively. This is the max-min fair allocation of a rate of 19.78 among the deployment routers, showing that level-k max-min fairness is achieved (in the sense of Definition 1).

Although the AIMD algorithm is shown to be effective under various attack scenarios, there remain some unresolved issues. In particular, the algorithm requires a *careful* choice of several control parameters, including the monitoring window W, step size δ, an initial estimate of the number of throttles $f(k)$, and the load limits U_S and L_S. The choice of these parameters will affect system stability and the speed of convergence to the stable load. For example, had we picked a smaller δ in the above example, it would have taken longer for the system to converge. This is shown in Table 1b. Furthermore, the parameters can be affected by a number of extrinsic factors that are beyond the algorithm's control, e.g.,

TABLE 1
Example of Using the AIMD Algorithm

Round	r_s	ρ
1	10	31.78
2	5	16.78
3	6	19.78

(a)

Round	r_s	ρ
1	10	31.78
2	5	16.78
3	5.15	17.23
4	5.30	17.68
5	5.45	18.13

(b)

(a) $\delta = 1$ *and (b)* $\delta = 0.15$.

the number of routers in $R(k)$, the delay of messages from S to the routers in $R(k)$, and changes in the offered traffic to S. As pointed out in [22], if the parameters are not set properly, the algorithm may not converge.

In the following section, we will address these robustness and stability issues by developing a *family* of distributed throttling algorithms that can effectively and automatically handle this problem.

3 A Family of Adaptive Fair Throttle Algorithms

In this section, we describe several algorithms with increasing sophistication. There are multiple objectives for these throttle algorithms:

1. **Fairness**: achieve level-k max-min fairness among the $R(k)$ routers.
2. **Adaptiveness**: use as few configuration parameters as possible.
3. **Fast convergence**: converge as fast as possible to the operating point for stationary or dynamic load.
4. **Stability**: algorithm converges regardless of the input rate distribution, and is robust against limited or unknown information of network parameters.

The throttle algorithm terminates if the aggregate load falls within a given region $[L_s, U_s]$, which contains C_0 where C_0 is a desired operating point. By default, we assume C_0 to be the midpoint between L_s and U_s. Each throttle algorithm is basically a probing algorithm that tries to determine a *common* r_s, to be sent by the server to all the $R(k)$ routers using negative feedback [4], [13]. Negative feedback is the process of feeding back the input a part of the system output. The principle of negative feedback thus implies that when the server asks the routers to decrease, we should ensure that the total load at the server will not increase, and when the server asks the routers to increase, the total load at the server will not decrease [4]. They are all designed to converge to the level-k max-min fairness criterion. All the algorithms differ in the way the server computes r_s, which will affect stability, i.e., whether the load converges to a point in $[L_s, U_s]$ and the convergence speed. We summarize our algorithms as follows:

1. Binary search algorithm: A simple algorithm that satisfies the first three objectives of a fair throttle algorithm for stationary load only.
2. Proportional aggregate control (PAC) with an estimate of the number of throttles: An algorithm

```
1. r_s := (U_s + L_s)/|R(k)|; /* initialize throttle rate */
2. ρ_last := −∞;
3. high := U_s;
4. low := 0;
5. While(1)
6.       sends current rate-r_s throttle to R(k);
7.       monitor traffic arrival rate ρ for time window W;
8.       If (ρ ≥ U_s)
9.            high := r_s;
10.           If (ρ_last − ρ < ε)
11.                low := 0; /* re-initialize BS */
12.           end if
13.      else if (ρ ≤ L_s)
14.           low := r_s;
15.           If (ρ − ρ_last < ε)
16.                high := U_s; /* re-initialize BS */
17.           end if;
18.      end if;
19.      ρ_last := ρ;
20.      r_s := (high + low)/2;
21. end while;
```

Fig. 3. Pseudocode for the binary search fair throttle algorithm.

that satisfies all the objectives of a fair throttle algorithm.

3. Proportional-aggregate-fluctuation-reduction (PAFR) algorithm: Improves the PAC algorithm by considering stability issues and tracking ability.

3.1 Binary Search (BS) Algorithm

The possible range for r_s is $[0, U_s]$, the two extremes corresponding to an infinite number of throttles and one throttle, respectively. A binary search algorithm can be applied: Reduce the search range by half at every iteration of the algorithm (see Fig. 3). Like the AIMD algorithm, this algorithm uses the aggregate rate ρ to determine the direction of adjustment. When the aggregate rate is more than U_s, r_s is reduced by half (same as before). However, when the aggregate rate is less than L_s, r_s is increased to the midpoint between the current r_s and U_s. This change removes the need to configure δ. The increase and decrease rules are symmetrical, each time reducing the search range by half. The pseudocode of the binary search algorithm is shown in Fig. 3. In the algorithm, we initialize the throttle rate r_s to $(U_s + L_s)/|R(k)|$, which corresponds to the guess that half of the $R(k)$ routers will be dropping traffic due to throttling.

The problem of optimal bandwidth probing assuming a known upper bound has been considered earlier by [9]. In their slightly different problem formulation of searching for available bandwidth by a single flow, they conclude that (a generalized version of) the binary search algorithm is near-optimal for certain reasonable cost functions of overshooting and undershooting.

The performance of binary search can be illustrated using the same example that we considered in Section 2. The results are shown in Table 2a. Note that the binary search algorithm converges quickly for a *stationary* load.

The basic binary search algorithm assumes that the traffic loading is stationary. As noted by Karp et al. [9], when the traffic conditions are dynamic, the situation is more complicated. They propose and analyze alternative algorithms for bounded variations in the traffic conditions. In our case, traffic can vary drastically since attackers may often vary their attack rates to reduce the chance of being revealed. The binary search algorithm must therefore detect the situation that the shrunken search range for r_s can no longer deal with the changed traffic conditions, and *re-initialize* the search range accordingly. Such a modification is included in the algorithm given in Fig. 3.

TABLE 2
Trace of r_s and Server Load Using the Binary Search Algorithm

Round	r_s	ρ
1	10	31.78
2	5	16.78
3	7.50	24.28
4	6.25	20.53

(a)

Round	r_s	ρ
1	5.63	24.11
⋮	⋮	⋮
∞	5.0	22.22

(b)

Round	r_s	ρ
1	5.63	24.11
⋮	⋮	⋮
7	5.01	22.25
8	2.51	12.77
9	3.76	18.50

(c)

(a) Binary search under stationary load. (b) Binary search becomes unstable without reinitialization. (c) The algorithm corrects itself using $\epsilon = 0.05$.

However, it takes time to realize the need for re-initializing the search, and it often takes more time than necessary for the new search to converge. For example, Table 2b shows that, if the last two sources in Fig. 1 with offered rates of 0.61 and 0.95 both change to 3.5 after round 4 in Table 2a, both sources are still within the rate throttle limit of r_s, but r_s cannot change to a value lower than $\text{low} = 5.0$ without reinitialization, and the system will be overloaded indefinitely. Table 2c shows that reinitialization corrects the algorithm by resetting the variable low. Although the binary search algorithm is *fair*, *fully adaptive*, and *fast*, it is not entirely satisfactory for dynamic traffic and packet delay.

3.2 Proportional Aggregate Control (PAC) with an Estimate on the Number of Throttles

The binary search algorithms directly adjust r_s, without using the magnitude of the overshoot (or undershoot) of the aggregate rate ρ beyond the target range. We now explore algorithms that do make use of this information, as well as an *estimate* of the number of routers in $R(k)$ that actually drop traffic. For clarity of exposition, we will call a deployment router whose offered rate exceeds r_s (and, thus, will drop traffic because of throttling) an *effective throttling router*. We call the throttle at such a router an *effective throttle*. If the number of effective throttles is n, then it is reasonable to set the change of r_s to

$$\Delta r_s = \frac{|\rho - C|}{n}, \tag{1}$$

where C represents the target rate (a value within $[L_S, U_S]$).

Since the traffic at the routers changes dynamically, it is not possible to know the exact number of effective throttling

```
1. ψ := 0.25;   /* low pass filter constant */
2. n_max := min(⌈ρ/r_s⌉, |R(k)|_max);
  /* estimate the number of effective throttles */
3. n := min(n_max, ⌈((1 − ψ) × n_last
        +ψ × (ρ − ρ_last)/(r_s − r_{s,last}))⌉);
4. n_last := n;
5. ρ_last := ρ;
6. r_{s,last} := r_s;
7. return n;
```

Fig. 4. Pseudocode for the algorithm to estimate the number of effective throttles n in $R(k)$ that assumes an upper bound on the size of $R(k)$, $|R(k)|_{\max}$.

routers. Furthermore, as r_s is adjusted, the number of these routers changes as well. By monitoring the past traffic and r_s, however, we can estimate n. The idea is to keep track of the last throttle rate r_{last} and last aggregate rate ρ_{last}. This allows us to estimate n as

$$n = \frac{|\rho - \rho_{last}|}{|r_s - r_{last}|}.$$

A pseudocode of the estimator is shown in Fig. 4. Exponential averaging is added to minimize inaccuracies due to fast changing components of the traffic load (see Line 3 of Fig. 4). Using this estimator, we can compute a suitable change to r_s using (1). This is the essence of the new algorithm, as shown in Fig. 5.

With the new algorithm, we also wish to address our fourth objective, namely, system *stability* in the face of delay between a throttle request sent by the server and the throttle taking effect at a deployment router. This objective can be achieved by correcting only a *fraction* K_P of the discrepancy between the aggregate rate and the rate target (see Line 19 of Fig. 5). The change Δr_s in (1) is thus *proportional* to the discrepancy; hence, we refer to such control as *Proportional Aggregate Control* (PAC). We will show in Section 4 how one can choose automatically (i.e., without the extra burden of

```
1. r_s := (U_s + L_s)/|R(k)|; /* initialize throttle rate */
2. ρ_last := −∞;
3. While(1)
4.          sends current rate-r_s throttle to R(k);
5.          monitor traffic arrival rate ρ for time window W;
6.          If (ρ ≥ U_s)
7.                  C := L_s;
8.          else if (ρ ≤ L_s)
9.                  If (ρ − ρ_last < ε)
10.                         remove rate throttle from R(k);
11.                         break;
12.                 else
13.                         C := U_s;
14.                         ρ_last := ρ;
15.                 end if;
16.         else
17.                  break;
18.         end if;
19.          φ := −K_P × (ρ − C);
20.          r_s := r_s + φ/est(ρ, r_s);
21. end while;
```

Fig. 5. Pseudocode for the proportional aggregate control fair throttle algorithm.

TABLE 3
Trace of the Throttle Rate and Server Load Using the PAC Algorithm

Round	r_s	ρ	$\triangle\rho/\triangle r_s$
1	10	31.78	–
2	8.28	26.62	3
3	6.84	22.30	3
4	6.12	20.14	3

Target capacity used is $L_s = 18$. Here, $K_P = 1/2$ so it reduces the search interval by half at every round.

tricky parameter configuration) an appropriate value for K_P based on the maximum delay between the server and a deployment router.

Another detail in the pseudocode is how to pick the value of C in (1). The value should be in the target range of $[L_S, U_S]$. Our current implementation sets the target capacity C to L_S when we have overload (see Line 7 of Fig. 5) and to U_S when we have underload (see Line 13 of Fig. 5). By advertising a larger aggregate rate discrepancy, we can maximize the likelihood that the resulting target rate will end up in the target range.

The PAC algorithm is applied to the same example used before. The results are shown in Table 3. An estimate of the number of effective throttles is given by the change in server load over the change in throttle rate, i.e., $\triangle\rho/\triangle r_s$. For stationary traffic demand, the rate of convergence is monotonic and relatively fast. The speed of convergence depends on the setting of K_P. However, the setting of K_P can affect system stability. This is discussed next.

3.3 Proportional-Aggregate-Fluctuation-Reduction (PAFR) Algorithm

We now introduce one more optimization to obtain an algorithm that is better than the PAC algorithm presented in the last section. Whenever there is a change in the offered load at the routers in $R(k)$, a higher proportional gain can adapt faster to the change. However, such "strong" control may result in large fluctuations in r_s, which may be annoying to the end users and also cause instability. On the other hand, a "weak" control causes prolonged congestion at the server. The goal is thus to reduce fluctuations while still achieving fast convergence. The stability problem that arises when a high proportional gain is used can be mitigated by considering a derivative control parameter, K_D. As we will show in Section 4, it turns out that K_D can be optimized to a constant value, independent of the operating conditions, for a given design *cost function*. Hence, K_D does not introduce any extra burden of parameter configuration. This *Proportional-aggregate-fluctuation-reduction* (PAFR) fair throttle algorithm is illustrated in Fig. 6.

As shown at Line 18 in Fig. 6, whenever we have underload or overload, the total expected change in the total load in $R(k)$ at the next interval is given by

$$\triangle\rho'_{i+1} = -K_P(\rho_i - C) - K_D\triangle\rho_i; \quad \triangle\rho'_1 = 0, \qquad (2)$$

where $\triangle\rho_i$ is the actual change and $\triangle\rho'_i$ is the expected change in the interval i.

The first term in (2) is necessary for the mismatch regulation, and the second term tracks the rate of change in

```
1. r_s := (U_s + L_s)/|R(k)|; /* initialize throttle rate */
2. ρ_last := −∞;
3. While(1)
4.          sends current rate-r_s throttle to R(k);
5.          monitor traffic arrival rate ρ for time window W;
6.          If (ρ ≥ U_s)
7.                  C := L_s;
8.          else if (ρ ≤ L_s)
9.                  If (ρ − ρ_last < ε)
10.                         remove rate throttle from R(k);
11.                         break;
12.                 else
13.                     C := U_s;
14.                 end if;
15.         else
16.                  break;
17.         end if;
18.         φ := −K_P × (ρ − C) − K_D × (ρ − ρ_last);
19.         r_s := r_s + φ/est(ρ, r_s);
20.         ρ_last := ρ;
21. end while;
```

Fig. 6. Pseudocode for the proportional-aggregate-fluctuation-reduction fair throttle algorithm.

the feedback. The first term has the largest impact on the aggressiveness of the algorithm. A smaller K_P will have slower convergence rate. On the other hand, a larger K_P may result in larger overshoots, sometimes even causing the system to become unstable, i.e., the load oscillates indefinitely around the band $[L_s, U_s]$ without convergence. The second term is more responsive to changes in the mismatch than the first term, and also has a damping effect on the fluctuation of the throttle rate. A larger K_D implies more damping, which improves the stability of the system. As shown in the simulation results in Section 5, the PAFR algorithm can reduce the amplitude of overshoot significantly during transient, while maintaining fast convergence.

Table 4 shows the PAFR algorithm for $K_P = 0.73$ and $K_D = 0.48$ using the previous example. An estimate of the number of effective throttles is given by the change in server load over the change in throttle rate, i.e., $\triangle\rho/\triangle r_s$. Although this simple example does not show any improvement in convergence time, the simulation results in Section 5 will clearly illustrate such improvements. In addition, the PAFR algorithm tracks the target rate range better. The tracking performance metric will be made precise in the next section.

3.4 Setting the Window Size W

The criterion for the right window size is twofold: obtain a more accurate estimate of the effective number of throttled routers n, which improves the adaptive throttle algorithms, and enforce rate limiting. For example, the filter at each throttle may employ some form of deterministic or probabilistic policy that drops packet according to specific features in the packet. Furthermore, it is recommended in [15] that a filter policy cannot be completely memoryless while enforcing rate limiting, i.e., in order for such kind of a filter to operate effectively based on its past measurement of the number of packets in a window length W, r_s cannot change too drastically in consecutive windows. Similarly, to obtain an accurate estimate of n, W cannot be too small. On the other hand, a large W affects the convergence time of the overload control system.

TABLE 4
Trace of the Throttle Rate and Server Load Using the PAFR Algorithm

Round	r_s	ρ	computed $\triangle\rho_i'$	$\triangle\rho/\triangle r_s$
1	10	31.78	0	–
2	7.48	24.22	-10.06	3
3	7.18	23.32	-0.91	3
4	6.05	19.93	-3.38	3

Target capacity used is $L_s = 18$.

To cope with this problem, we use a moving window with a parameter, which is a function of the maximum round trip time in the system to obtain a smooth measurement of the incoming aggregate rate ρ. Specifically, we define $\lambda = 2D/W$, where D is the average round trip delay from each router in $R(k)$ to S. Averaging between consecutive windows, we have $\overline{\rho}_i = \lambda\rho_{i-1} + (1-\lambda)\rho_i$, where the subscript denotes the ith window. We verify by simulations in Section 5 that the above approximation works very well in measuring the aggregate traffic rate, which is used as an input in our throttle algorithms.

4 An Analysis of the Fair Throttle Algorithms

In this section, we use a control-theoretic approach to study router throttling. Particularly, a fluid flow model is used to analyze the stability of the PAC and PAFR fair throttle algorithms. We model S as a single bottleneck node, and denote the total traffic load from $R(k)$ to S by $y(t)$, which serves the same role as ρ_i in our algorithms. In both the PAC and PAFR algorithms, the expected change in aggregate traffic load is computed using (2) where $K_D = 0$ in the PAC algorithm.

Since the offered traffic at each router changes dynamically, $y(t)$ will be larger than the expected change in (2) if 1) the load at some previously unthrottled routers increases, but is still less than the throttle limit, or 2) additional routers are added to $R(k)$ while the throttle rates are changing. On the other hand, $y(t)$ is less than the expected change if 1) the offered load at some previously throttled routers fall significantly below the throttle limit, or 2) some routers are removed from $R(k)$ while the throttle rates are changing. These can be viewed as uncertainties in the control system. Here, stability is defined in the *bounded-input-bounded-output* sense (see [13], p. 319). Besides, the estimation of parameters such as the number of throttles introduces additional uncertainty to the system. The use of feedback is thus to combat uncertainty, and to determine how to adjust the throttle rate while maintaining a stable system. A properly designed negative feedback controller is our focus in the following.

First, we introduce a fluid model that governs the system we are analyzing. In our fluid model, we treat the target capacity to be a constant value, C, which can be the midpoint between L_S and U_S. The rate of change in $y(t)$ is given as

$$dy(t)/dt = -K_P(y(t-D) - C) - K_D(dy(t-D)/dt), \quad (3)$$

where D is the delay, which is upper bounded by a constant D_0, and K_P and K_D are the proportional and differential gains, respectively. Selecting appropriate K_P and K_D is part

of the design of the throttle algorithms. We first consider the PAC algorithm where the change in the search range converges to zero if the traffic is stationary. Our first result shows that the amount of feedback is inversely proportional to the amount of delay that the system can tolerate.

Lemma 1. *The system in (3) is asymptotically stable if and only if* $0 < K_P < \min(\pi/(2D_0), 1)$, *where* D_0 *is the largest delay in the system.*

Proof. Please refer to the Appendix. □

Remark 1. When $K_P = 1$, the exact mismatch between $y(t)$ and C is used as a feedback. Intuitively, the system is most responsive. However, the system is also the least tolerant to time delay. Hence, there is a tradeoff between the speed of response and robustness to delay (see proof of Theorem 1).

Remark 2. Recall that overestimating the number of throttles in $R(k)$ has the effect of reducing the proportional gain. For example, if the estimator always assumes that the number of effective throttles is the maximum number of routers in $R(k)$, we have a form of doubly conservative PAC algorithm.

Next, we introduce a *cost function*, which is a performance metric that measures the effectiveness of a fair throttle algorithm, and is given as

$$J = \sum_{i=1}^{T_s} (x_i)^2, \tag{4}$$

where x_i is the corresponding mismatch between C and ρ at interval i, and T_s is a given terminating interval of the algorithm. Note that x_i is positive if the system is overloaded, or negative if the system is underloaded, hence the use of the squared function. Note that (4) is similar to the proposed cost function (gentle cost function) in [9] where both overloading and underloading are penalized. In this paper, we treat the cost of overloading and underloading to be the same.

In (4), T_s plays the same role as *settling time* in control theory (see [13], p. 272), which is defined as the time required for the system response to decrease and stay within a specified percentage of its final value (e.g., a frequently used value is 5 percent). The criterion in (4) is tolerant to *small* but punitive to *large* errors at every interval. In other words, a smaller J is preferred. For a given $[L_s, U_s]$ that is relatively narrow, a sufficiently long T_s and a given traffic demand, J is useful for differentiating the performance of various algorithms. An overconservative algorithm tends to have a larger J. Besides, an algorithm that causes large variation in r_s at most of the intervals often results in a larger J.

The design of the PAC and PAFR algorithms is based on an optimization approach where K_P and K_D are selected by minimizing J in (4) subject to the constraint that the integral is finite for a given input sequence of x_i. To this end, we consider infinitesimally small intervals, and let $T_s \to \infty$, thus we obtain the following integral of squared error:

$$J = \int_0^\infty x^2(t)\, dt, \tag{5}$$

which belongs to a family of integral cost functionals in control theory for determining fixed order control parameters for a given input [13]. For the PAC fair throttle algorithm, we have the following main result.

Theorem 1. *The PAC algorithm that minimizes (5) has* $K_P = (\sqrt{3} - 1)/D_0$ *for a delay* D_0. *Furthermore, it stabilizes the system for any positive delay* $D \le D_0$.

Proof. Please refer to the Appendix. □

For the PAFR fair throttle algorithm, we have the following main result.

Theorem 2. *The PAFR algorithm that minimizes (5) has* $K_P = 0.80/D_0$ *and* $K_D = 0.48$ *for a delay* D_0. *Furthermore, it stabilizes the system for any positive delay* $D \le D_0$.

Proof. Please refer to the Appendix. □

Remark 3. In both PAC and PAFR algorithms, only a single optimized parameter K_P needs to be set with respect to the maximum delay in the system.

Remark 4. Theorems 1 and 2 are valid for any number of routers in $R(k)$. If K_P stabilizes the system for a given D_0, the $R(k)$ router deployment using the same K_P is feasible anywhere in the network as long as the delay is less than D_0.

Corollary 1. *The PAFR algorithm that minimizes (5) always has a smaller* J *than the PAC algorithm for all* $D > 0$.

Proof. Please refer to the Appendix. □

Corollary 1 implies that the PAFR algorithm outperforms the PAC algorithm in minimizing (4) given that the algorithms run for a sufficiently long time, and is validated by the simulation results in the next section.

5 Simulation Results

In this section, we study the performance of the various fair throttle algorithms in terms of performance metrics such as convergence time, throttle rate fluctuation, and adaptation to different network parameters using ns-2. We denote the instantaneous offered traffic load at router i as $T_i(t), \forall i$. For all the experiments, we set $L_s = 100\text{Mbps}$ and $U_s = 115\text{Mbps}$. The initial throttle rate r_s is set as 200 with $W = 3D$, and the low pass filter time constant of $\text{est}(\rho, r_s)$ is set as 0.25, unless noted otherwise.

5.1 Experiment A (Heterogeneous Network Sources with Constant Input)

First, we consider 50 heterogeneous sources. Initially, each source has a constant offered traffic load $T_i(t) = 30, \forall\, i$. Next, 10 sources change their offered load to 1Mbps and 4Mbps at $t = 50$ and $t = 100$, respectively. The network delay D between S and each source is 100ms. The cost metric J and settling time T_s for each algorithm are reported under the figures for the input disturbance at $t = 50$ only if the algorithm converges. Fig. 7 shows the aggregate server load when AIMD ($\delta = 0.02\text{Mbps}$, $\delta = 0.2\text{Mbps}$, and $\delta = 0.4\text{Mbps}$), binary search ($\epsilon = 1\text{Mbps}$), PAC ($K_P = 0.5$) and PAFR ($K_P = 0.5, K_D = 0.48$) algorithms are used, respectively. From Fig. 7,

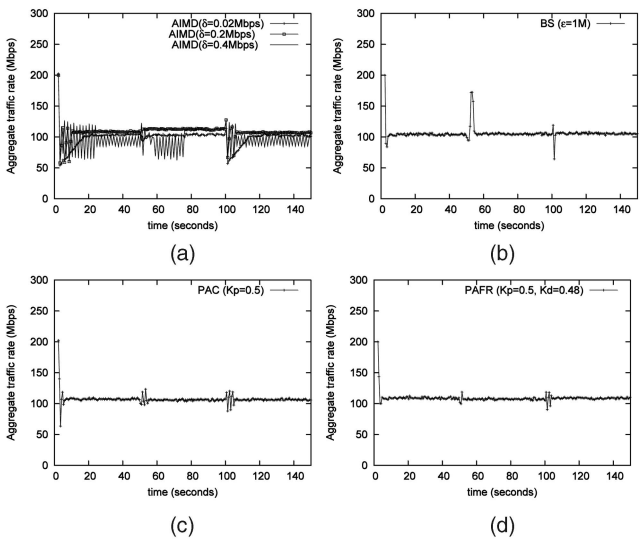


Fig. 7. (a) AIMD algorithm with step size of $\delta = 0.02M$ ($T_s = 4.0s$, $J = 74.1Mb$)$\delta = 0.2M$ ($T_s = 2.5s, J = 50Mb$), and $\delta = 0.4M$ ($T_s = 42.7$, $664.3Mb$). (b) Binary search algorithm. $\epsilon = 1M$ ($T_s = 6.6s, J = 183.9Mb$). (c) PAC ($K_P = 1/2$) algorithm ($T_s = 4.8s, J = 38.7Mb$). (d) PAFR algorithm with $K_P = 0.5$ and $K_D = 0.48$ ($T_s = 1.5s, J = 18.6Mb$).

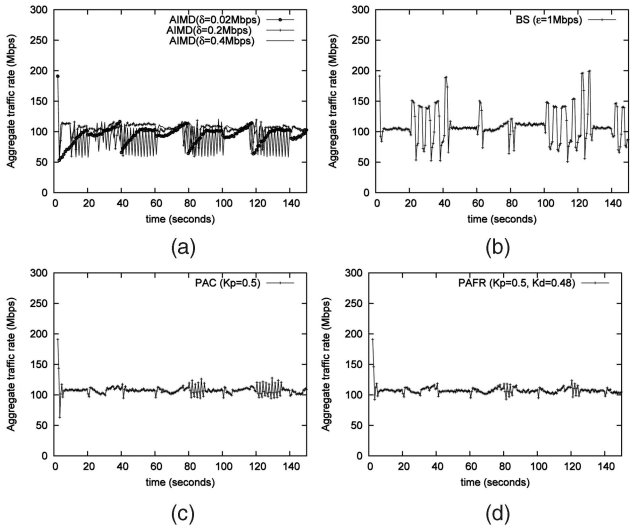


Fig. 8. (a) AIMD algorithm with step size of $\delta = 0.02M$ ($T_s = 9.3s$, $J = 114Mb$), $\delta = 0.2M$ ($T_s = 2.0s, J = 28.1Mb$), and $\delta = 0.4M$ ($T_s = 16.1s$, $J = 236.4Mb$). (b) Binary search algorithm. $\epsilon = 1M$ ($T_s = 9.8s$, $J = 229.2Mb$). (c) PAC ($K_P = 1/2$) algorithm ($T_s = 2.4s, J = 12.9Mb$). (d) PAFR algorithm with $K_P = 0.5$ and $K_D = 0.48$ ($T_s = 1.3s, J = 7.1Mb$).

we observe that the PAFR algorithm is stable, converges, and has a much less over/undershoot.

5.2 Experiment B (Heterogeneous Network Sources with Varying Input)

Next, we repeat the above experiment, but the first 30 sources are configured to be constant sources with $T_i(t) = 4\text{Mbps}$ for $i = 1, \ldots, 30$. The next 10 sources are sinusoidal sources where $T_i(t) = 1.5\sin(0.16t) + 2.5\text{Mbps}$ for $i = 31, \ldots, 40$. The network delay for each of these sinusoidal sources is 50ms. The last 10 sources are square pulse sources with $T_i(t) = 4\text{Mbps}$ for the time interval $20k \le t \le 10(2k+1)$ and $T_i(t) = 1\text{Mbps}$ for the time interval $10(2k+1) \le t \le 20(k+1)$, $i = 41, \ldots, 50$, and $k \in \{0, 1, 2, \ldots\}$. The network delay for each of these square pulse sources is 50ms. Fig. 8 shows the aggregate server load for the second experiment.

Based on the above experiments, we can make the following observations.

Remark 5. The stability and performance of the AIMD algorithm is dependent on the step size used. The system is stable and slow for small δ, but is unstable when δ is large. Furthermore, there is a significant overshoot in the initial transient response for all the three different step sizes used.

Remark 6. When the binary search algorithm is used, the transient response becomes unsatisfactorily if the number of sources gets large as in the experiments. Large overshoots are observed in the transient before the algorithm converges, and are due to the large variation in r_s as shown in Fig. 9. In the dynamic traffic case, slow response to reinitialization is the main cause for the large overshoots between $t = 50$ and $t = 100$.

Remark 7. The PAC ($K_P = 0.5$) algorithm is comparatively faster and stable, but there are large overshoots when there is a disturbance to the system at $t = 50$ and $t = 100$. As a result, the convergence time might be longer as shown in Fig. 8.

Remark 8. As compared to the other algorithms, the PAFR algorithm is more robust to external disturbances with much smaller overshoots, and has faster convergence time. Fig. 9 compares the throttle rate variation between the binary search and the PAFR algorithms. We observe that damping reduces the throttle rate variation considerably. From Fig. 10, we note that the estimator algorithm is extremely effective in estimating accurately the number of throttles in $R(k)$ at $t = 50$ and $t = 100$.

5.3 Experiment C (Effect of System Parameters on the Binary Search Algorithm)

We next evaluate an improved version of the binary search algorithm using the estimator algorithm. Reinitializing the parameters in the binary search algorithm that leads to a *gradual* change in r_s can be done by taking into account the load mismatch and the total number of effective throttles. The goal of a gradual change is to reduce the large throttle rate variation that we observe in the previous experiments. Particularly, if we have overload, we replace Line 11 in Fig. 3 by the following:

$$\text{low} = \min(\max(r_s + (C - \rho)/est(\rho, r_s), 0), C). \qquad (6)$$

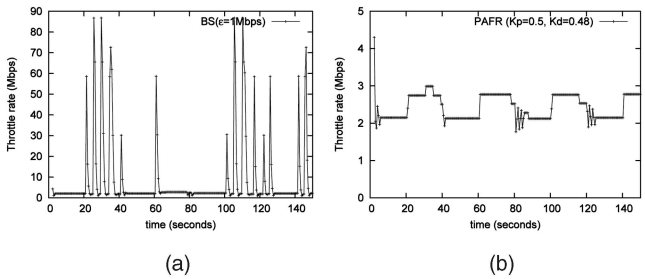


Fig. 9. (a) Binary search algorithm exhibits large throttle rate variation. (b) PAFR algorithm with $K_P = 0.5$ and $K_D = 0.48$. Damping provides better transient response while maintaining excellent response time.

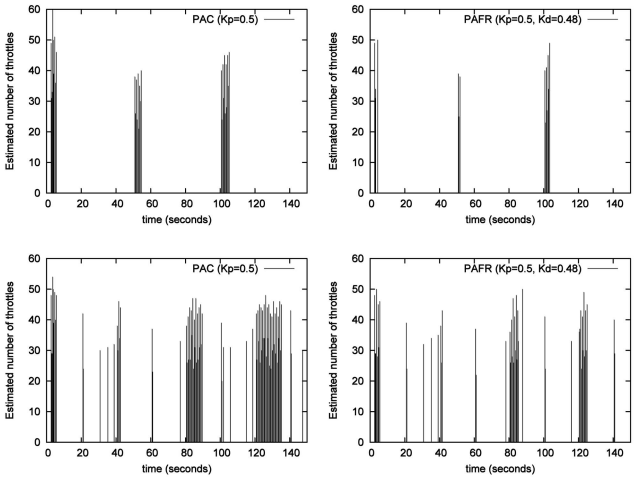


Fig. 10. Estimating the number of throttles in $R(k)$ using the $\mathrm{est}(\rho, r_s)$ algorithm.

Likewise, if we have underload, we execute (6) with the variable "low" replaced by the variable "high" at Line 16 in Fig. 3. We term this the improved binary search algorithm.

We repeat the second experiment using the improved binary search, and observe the system response when the step disturbance occurs at $t = 50$. Fig. 11 shows the effect of different ϵ on T_s and J of the improved binary search algorithm. We observe that if ϵ is too small, the convergence time is much longer. A longer time is also needed to detect that reinitialization is required, which results in a transient response characterized by large overshoots. If ϵ is too large, the algorithm becomes more sensitive to small perturbation in the offered load. As compared to the previous experiment, we observe that using (6) tends to stabilize the binary search algorithm. In fact, by using information from the load mismatch, the gradual reinitialization mechanism in (6) brings the improved binary search a step closer and comparable to a PAC algorithm. However, the system response is still sensitive to the parameter setting of ϵ, which is hard to tune. Hence, the performance of the improved binary search algorithm is unsatisfactorily as compared to the PAC or PAFR algorithm.

5.4 Experiment D (Effect of Parameters U_s and L_s)

We repeat the second experiment, but increase the range $[L_s, U_s]$ to [100Mbps, 125Mbps]. Fig. 12 shows the aggregate server load for binary search, PAC $(K_P = 0.5)$ and PAFR algorithms, respectively. From Experiments B and D, we note that a narrow $[L_s, U_s]$ makes the binary search algorithm unstable, again due to the need for repeated reinitializations and delays. The PAC algorithm is stable, but overshoots remain in the transient response, while the PAFR algorithm maintains its excellent performance with fast convergence time. In general, a narrow $[L_s, U_s]$ is more demanding for the throttle algorithms.

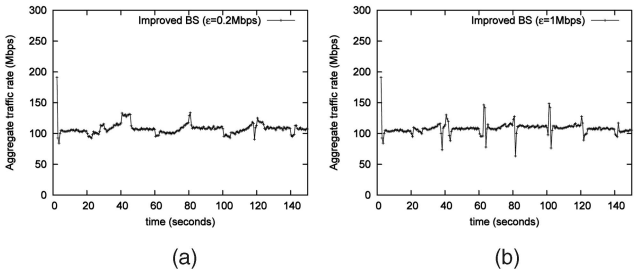


Fig. 11. Improved binary search algorithm with (a) $\epsilon = 0.2M$ $(T_s = 3.7s, J = 27.0Mb)$ and (b) $\epsilon = 1M$ $(T_s = 1.7s, J = 13.2Mb)$.

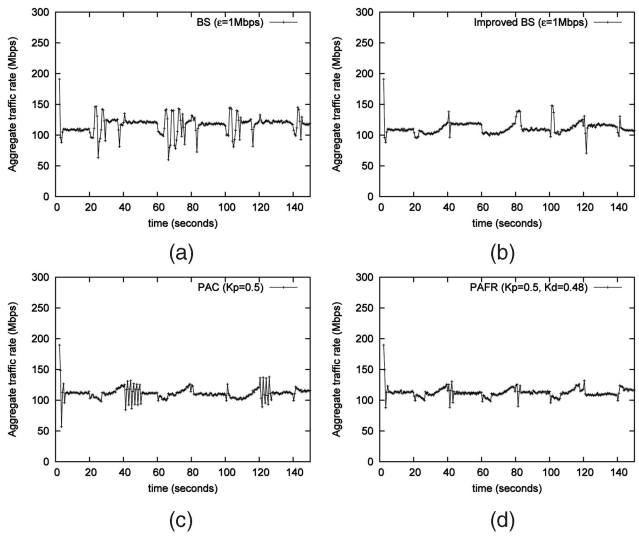


Fig. 12. (a) Binary search with $\epsilon = 1M$ $(T_s = 3.9, J = 47.0Mb)$. (b) Improved binary search with $\epsilon = 1M$ $(T_s = 1.5s, J = 12.9Mb)$. (c) PAC algorithm with $K_p = 1/2$ $(T_s = 3.2, J = 19.4Mb)$ (d) PAFR algorithm with $K_p = 0.5, K_D = 0.48$ $(T_s = 1.0s, J = 7.2Mb)$. As compared to Experiment B, a larger $[L_s, U_s]$ reduces the cost J.

5.5 Experiment E (Heterogeneous Network Sources with Randomly Varying Input)

In this experiment, we introduce stochastic effects in the load level of the $R(k)$ routers. The presence of stochastic elements represents traffic with background flows in a network, which may lead to a larger variation in the input with a lower stability margin. The delay in each link is uniformly distributed from $1s$ to $3.5s$, and $W = 2s$. There are altogether 50 routers in $R(k)$. Fifteen routers have sinusoidal varying sources—five with a rate of $0.5\sin(0.16t) + 3\mathrm{Mbps}$, five with a rate of $0.5\sin(0.16t + 0.5\pi) + 1.5\mathrm{Mbps}$, and five with $\sin(0.16t + 1.5\pi) + 3.5\mathrm{Mbps}$. Ten routers have square pulses that vary between 1.5Mbps and 2.5Mbps with a period of $80s$. Twenty five routers have Gaussian distributed varying source rates —10 with mean 3Mbps and variance 1Mbps, 10 with mean 3.5Mbps and variance 0.5Mbps, and five with mean 4Mbps and variance 1Mbps.

We perform the experiment for two scenarios: In the first scenario, S assumes $|R(k)|_{\max}$ as 50. As shown in Figs. 13a, 13b, and 13c, the response of the control system is relatively smooth for small K_p in the presence of large delays and large variation in the sources. A large K_p makes the system sensitive to noise in the load level. Fig. 13d shows that $\mathrm{est}(\rho, r_s)$ predicts that there are close to 50 effective throttles in the system. In the second scenario, we assume a smaller $|R(k)|_{\max} = 10$. As shown in Figs. 13a, 13b, and 13c and Figs. 14a, 14b, and 14c, the responses for $K_P = 0.05$ and $K_P = 0.1$ are relatively smoother than that for $K_P = 0.2$. However, as shown in Fig. 14d, even though the number of routers can be grossly underestimated, the PAFR algorithm can still function properly and meet the target load. We repeat the experiment for the case of $|R(k)|_{\max} = 50$, and increase the low

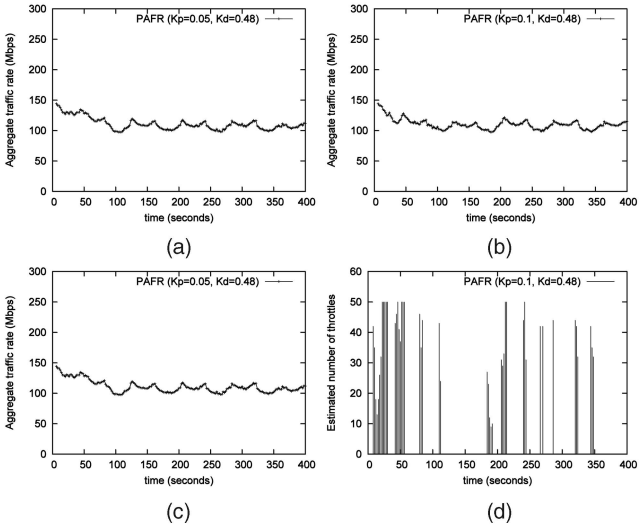


Fig. 13. Performance of the PAFR throttle algorithm with an estimate that $|R(k)|_{\max} = 50$, $K_D = 0.48$ as K_P takes the value of (a) $K_P = 0.05$ ($T_s = 13.4s, J = 92.0Mb$), (b) $K_P = 0.1$ ($T_s = 8.6s$, $J = 50.6Mb$), and (c) $K_P = 0.2$ ($T_s = 13.5s, J = 71.0Mb$). (d) Estimated number of effective throttles using the $\mathrm{est}(\rho, r_s)$ algorithm for the case $K_P = 0.1$.

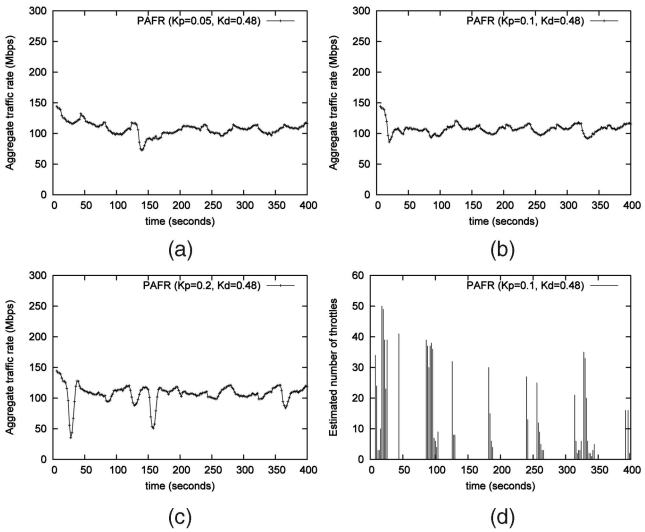


Fig. 15. Performance of the PAFR throttle algorithm using low pass filter time constant $\psi = 0.95$ for $\mathrm{est}(\rho, r_s)$ with an estimate of upper bound $|R(k)|_{\max} = 50$ for (a) $K_P = 0.05$ ($T_s = 16.1s, J = 84.6Mb$), (b) $K_P = 0.1$ ($T_s = 12.9s, J = 73.7Mb$), and (c) $K_P = 0.2$ ($T_s = 18.6s, J = 180.0Mb$). (d) Estimated number of effective router for the case of $K_P = 0.1$.

pass filter time constant to $\psi = 0.95$. As shown in Fig. 15d, a larger ψ tracks a larger variation in the number of effective throttle routers, but the equilibrium points are not affected much as shown in Figs. 15a, 15b, and 15c.

Remark 9. The PAFR algorithm is more effective in reducing the fluctuation when there are many unknown parameters, e.g., attack distribution and the number of effective throttles, and is also more robust than the other algorithms in minimizing load variation regardless of the window size used, thus allowing statistical filtering to be done with a higher degree of confidence.

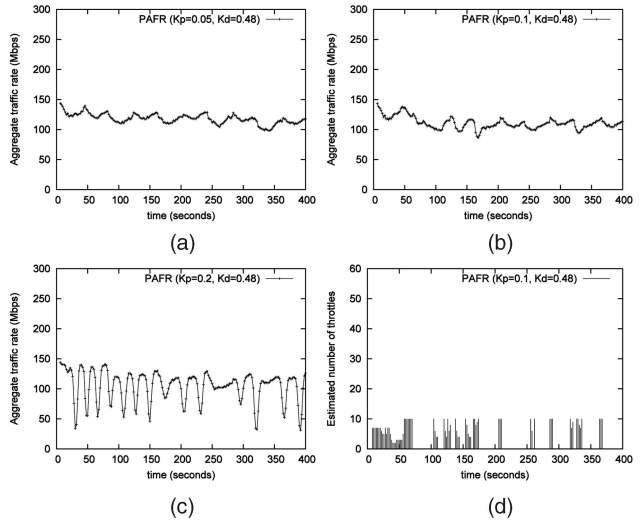


Fig. 14. Performance of the PAFR throttle algorithm with complete information that $|R(k)|_{\max} = 10$, $K_D = 0.48$ as K_P takes the value of (a) $K_P = 0.05$ ($T_s = 35.4s, J = 286.8Mb$), (b) $K_P = 0.1$ ($T_s = 19.5s$, $J = 145.0Mb$), and (c) $K_P = 0.2$ ($T_s = 60.0s, J = 790.1$). (d) Estimated number of effective throttles using the $\mathrm{est}(\rho, r_s)$ algorithm for the case $K_P = 0.1$.

6 Related Work

Handling high bandwidth aggregates using router throttling can be viewed as a congestion control problem. A large body of work on congestion control assumes that the feedback is in the form of a binary signal. In our context, we have the luxury of sending an explicit throttling rate to the traffic regulators (routers in this case). So, it is simpler for us to achieve max-min fairness without using AIMD-style distributed algorithms. In this respect, our approach is similar to XCP [8], in being able to separate the problem of fair allocation with the problem of controlling the total aggregate traffic.

In searching for the simplest control (for the total aggregate traffic), we consider binary search, which is inspired by the work in [9]. But, this approach quickly becomes more complex and less effective when considering dynamic traffic and delay.

Our work is similar to those efforts in analyzing Internet and ATM congestion control, for example, the ATM Available Bit Rate (ABR) model in [3], [11] and the TCP model in [6], [12]. The departure is in the detailed model where we use explicit rate feedback, and introduce a hysteresis for the desired capacity. In addition, unlike congestion control in the Internet, an effective estimator that estimates the number of effective throttles and the role of the window size is crucial for distinguishing legitimate users and attackers.

Defending against DDoS flooding-based attack or flash crowds using different rate throttling mechanism at upstream routers can be found in [16]. In [16], a *pushback* max-min fairness mechanism used for controlling large bandwidth aggregates is proposed, which is initiated at the congested source and applied recursively to the upstream routers. However, this approach may cause upstream routers with low traffic demand to be excessively penalized.

Other router-assisted network congestion control schemes that handle high bandwidth aggregates in a fair manner include detection and dropping using Active

Queue Management (AQM) mechanism in congested routers such as balanced RED [1], RED with Preferential Dropping [17], Stochastic Fair Blue [5], and CHOKe [19]. The Network Border Patrol (NBP) with enhanced core-stateless fair queueing [2] is a framework used to control flow aggregates. However, some of these schemes require either full or partial state information and some schemes do not guarantee max-min fairness in the shortest possible time. Besides, these schemes are more complex than rate throttling, which has low runtime overhead.

7 Conclusion

We analyze how a class of distributed server-centric router throttling algorithms can be used to achieve max-min fairness in a number of routers that are k hops away when the server is overloaded by large bandwidth aggregates, which may occur under DDoS attack or flash crowd scenarios. Our proposed throttle algorithms, which are based on a control-theoretic approach, are shown to be highly adaptive to dynamic traffic situations. The stability and convergence issues of these algorithms are also studied. In particular, we show that the PAFR fair throttle algorithm is robust against a wide range of settings—not only does it guarantee convergence, but, in addition, it has the least load fluctuation. Level-k max-min fairness is guaranteed for all the throttling routers in $R(k)$.

Designing throttle algorithms to counter DDoS attacks with limited information is challenging, thus it is not surprising that many issues remain open in this topic. First, it will be desirable to have more in-depth analysis of the proposed throttle algorithms in terms of convergence rate and transient performance. Second, we can investigate other cost functions for optimal setting, which may lead to other optimal fair throttle algorithms. The effectiveness of various throttle algorithms can thus be examined under a variety of cost functions that may suit different network scenarios under DDoS attacks. Third, we have taken delay into account in analyzing our throttle algorithms. The next step would be to test our algorithms in network scenarios under which the deployment depth varies. This has implication on the practical deployment of throttle algorithms in a real network as well as providing different lines of defense against DDoS attacks. Related issues on the impact of deployment depth on throttle algorithms can also be found in [22].

Future directions of our work include testing our algorithms in networks with diverse delays. We also plan to implement our proposed throttle algorithms in OPERA, which is an open-source programmable router platform that provides flexibility for researchers to experiment with security services [18]. Particularly, OPERA allows dynamically loadable software modules to be installed on the fly without shutting down the routing functionalities in the router, which facilitates testing different throttle and statistical filtering algorithms on a security testbed.

Appendix A

Proof of Lemma 1

We first let $K_D = 0$ in (3) to obtain $dy(t)/dt = K_P(C - y(t - D))$, which is rewritten as $dx(t)/dt = -K_P x(t - D)$, where $x(t) = y(t) - C$. Next, a necessary and sufficient proof that follows the frequency domain method in [23] is given as follows: For stability, the roots of any characteristic equation must lie on the left half complex plane (see [13], p. 321). We wish to determine K_P such that the characteristic equation $s + K_P e^{-sD} = 0$ has stable poles with delay from 0 up to D_0, and this set of K_P is denoted as S_R (following the convention in [23]). First, by the Routh-Hurwitz criterion (see [13], p. 322), the set of K_Ps that stabilize the delay-free plant (given as S_0 in [23]) is $S_0 = \{K_P > 0\}$. Second, there exists K_P such that the characteristic equation is stable if $D > 0$. Third, we compute the set $S_L = \{K_P | K_P \notin S_N \text{ and } \exists\ D \in [0, D_0], w \in R,\ s.t.\ jw + K_P e^{-jwD} = 0\}$, which is given by $S_L = \{K_P \geq \pi/(2D_0)\}$. Hence, $S_R = S_0 \setminus S_L$ and, thus, $S_R = \{0 < K_P < \pi/(2D_0)\}$. But, K_P must necessarily be not more than 1 for convergence as the system dynamics is a search for a value between the current server load and C. Hence, $0 < K_P < \min(\pi/2D_0, 1)$.

Next, we denote the open loop transfer function as $L(s, K_P, D) = K_P e^{-sD}/s$. For a fixed w and for any positive $D < D_0$, $\angle L(jw, K_P, D) > \angle L(jw, K_P, D_0)$, i.e., we always have positive phase margins. Hence, the PAC algorithm can stabilize the system for $D \leq D_0$. To prove convergence of $x(t)$ to zero, note that $|K_P e^{-jsD}/s| \to \infty$ as $s \to 0$. Hence, zero steady state error is ensured in the face of a step disturbance. The tradeoff between system response and delay tolerance can be verified by noting that the closed loop system bandwidth w_B satisfies $K_P < w_B < 2K_P$, and w_B is roughly inversely proportional to the response time of the system in response to a step disturbance.

Appendix B

Proof of Theorem 1

For J in (5) to exist, $x(t)$ must converge to zero in the limit as $t \to \infty$. From Lemma 1, this is true if and only if $0 < K_P < \min(\pi/2D_0, 1)$. Hence, any optimal K_P, if it exists, must satisfy Lemma 1. Next, we use Parseval's theorem to compute J, which is rewritten as $J = (1/j2\pi)\int_{-j\infty}^{j\infty} X(s)\ X(-s)\ ds$ where $X(s)$ denotes the Laplace transform of $x(t)$ (cf. [13], p. 583). Ignoring initial conditions, we let $X(s) = 1/(s + K_P e^{-sD})$ for the PAC algorithm. Next, we obtain

$$\begin{aligned} J &= (\tan K_P D + \sec K_P D)/(2K_P) \\ &= (\tan(K_P D/2 + \pi/4))/(2K_P). \end{aligned}$$

To compute the optimal K_P, we let $dJ/dK_P = 0$, which results in $\cos K_P D = K_P D$. Using Taylor's series approximation, this results in $1 - (K_P D)^2/2 \approx K_P D$, which gives $K_{P,opt} = (\sqrt{3} - 1)/D$. Since

$$d^2 J/dK_P^2|_{K_P = K_{P,opt}} = \left(1 + \sqrt{1 - (K_{P,opt} D)^2}\right)/(2K_{P,opt}^2 D) > 0,$$

J is indeed the minimum. Moreover, for all D, $K_{P,opt}$ satisfies Lemma 1, thus $K_{P,opt}$ indeed stabilizes the system for all delay $D < D_0$.

Appendix C

Proof of Theorem 2

Continuing along the same line of proof as Theorem 1, we let $X(s) = 1/(s + (K_P + K_D s)e^{-sD})$. Now, J in (5) exists

since $|(K_P + K_D s)e^{-jsD}/s| \to \infty$ as $s \to 0$. The parameters K_P and K_D that satisfy the sufficient and necessary condition for asymptotic stability can again be obtained by the approach in [23] and the Nyquist criterion to yield $S_{PD} = \{0 < K_P < 1, \text{and } K_P D/\sqrt{1-K_D^2} < \arccos(-K_D)\}$. Hence, the optimal set of control parameters $K_{P,opt}$ and $K_{D,opt}$ must lie in the set, S_{PD}. Next, we have

$$J = 1/(2K_P\sqrt{1-K_D^2})(K_P \cos\kappa D + K_D \kappa \sin\kappa D)/(\kappa - K_P \sin\kappa D + \kappa K_D \cos\kappa D),$$

where $\kappa = K_P/\sqrt{1-K_D^2}$, which can be simplified as $J = D\tan((\upsilon+\Psi)/2)/(2\Psi\sin^2\upsilon)$, where $\Psi = K_P D/\sqrt{1-K_D^2}$ and $\upsilon = \arccos K_D$. To yield the optimal K_P and K_D, we have $\partial J/\partial\upsilon = 0$ and $\partial J/\partial\Psi = 0$ for $0 < \upsilon < \pi$ and $0 < \Psi < \pi - \upsilon$. Finally, we have $\Psi = \sin(\upsilon + \Psi)$ and $\Psi = 0.5\tan\upsilon$. Solving these two equations numerically, we obtain the optimal Ψ_{opt} as $\Psi = 0.92$. Since $\Psi_{opt} = 0.5\tan\upsilon$, we have $K_P = 2\Psi_{opt}^2/(D\sqrt{1+4\Psi_{opt}^2})$ and $K_D = 1/\sqrt{1+4\Psi_{opt}^2}$, which we obtain $K_P \approx 0.8/D$ and $K_D \approx 0.48$. We invoke the same steps as before by fixing w and, for all $D \le D_0$, the loop transfer function always has a positive phase margin.

APPENDIX D

Proof of Corollary 1

For a fixed D, from Theorem 1, the cost function for the PAC algorithm is $J_P = (1+\sqrt{1-(K_{P,opt}D)^2})/2K_{P,opt}^2 D$, which is approximately $1.6D$, and, for the PAFR algorithm, from Theorem 2, $\kappa = K_{P,opt}/\sqrt{1-K_{D,opt}^2} \approx 0.91/D$, thus we have $J_{PD} \approx 1.1D$. For all $D > 0$, $J_P > J_{PD}$.

ACKNOWLEDGMENTS

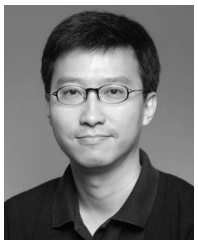
The authors thank the editor for coordinating the review process and the anonymous reviewers for their insightful comments and constructive suggestions. They also thank Wang Yue for his help in the ns-2 simulation. The research of D.M. Chiu was supported in part by the RGC Earmark Grant 4232/04Grant. The research of John C.S. Lui was supported in part by the RGC Grant 4218/04.

REFERENCES

[1] F.M. Anjum and L. Tassiulas, "Fair Bandwidth Sharing among Adaptive and Non-Adaptive Flows in the Internet," *Proc. IEEE INFOCOM '99,* Mar. 1999.
[2] C. Albuquerque, B.J. Vickers, and T. Suda, "Network Border Patrol: Preventing Congestion Collapse and Promoting Fairness in the Internet," *IEEE/ACM Trans. Networking,* vol. 12, no. 1, pp. 173-186, Feb. 2004.
[3] L. Benmohamed and S.M. Meerkov, "Feedback Control of Congestion in Packet Switching Networks: The Case of a Single Congested Node," *IEEE/ACM Trans. Networking,* vol. 1, no. 6, pp. 693-708, Dec. 1993.
[4] D.M. Chiu and R. Jain, "Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks," *J. Computer Networks and ISDN,* vol. 17, no. 1, pp. 1-14, June 1989.
[5] W. Feng, K. Shin, D. Kandlur, and D. Saha, "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness," *Proc. IEEE INFOCOM '01,* Apr. 2001.
[6] C.V. Hollot, V. Misra, D. Towsley, and W. Gong, "Analysis and Design of Controllers for AQM Routers Supporting TCP Flows," *IEEE Trans. Automatic Control,* vol. 47, no. 6, pp. 945-959, June 2002.
[7] K.Y.K. Hui, J.C.S. Lui, and D.K.Y. Yau, "Small-World Overlay P2P Networks," *Computer Networks,* vol. 50, no. 15, Oct. 2006.
[8] D. Katabi, M. Handley, and C. Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks," *Proc. ACM SIGCOMM,* pp. 89-102, Aug. 2002.
[9] R. Karp, E. Koutsoupias, C. Papadimitriou, and S. Shenker, "Optimization Problems in Congestion Control," *Proc. 41st Ann. IEEE CS Conf. Foundations of Computer Science,* Nov. 2000.
[10] Y. Kim, W.C. Lau, M.C. Chuah, and J.H. Chao, "PacketScore: Statistics-Based Overload Control against Distributed Denial-of-Service Attacks," *Proc. IEEE INFOCOM '04,* Mar. 2004.
[11] A. Kolarov and G. Ramamurthy, "A Control-Theoretic Approach to the Design of an Explicit Rate Controller for ABR Service," *IEEE/ACM Trans. Networking,* vol. 7, no. 5, pp. 741-753, Oct. 1999.
[12] S. Kunniyur and R. Srikant, "Analysis and Design of an Adaptive Virtual Queue Algorithm for Active Queue Management," *Proc. ACM SIGCOMM,* pp. 123-134, Aug. 2001.
[13] B.C. Kuo, *Automatic Control Systems.* Prentice Hall, 1975.
[14] K.T. Law, J.C.S. Lui, and D.K.Y. Yau, "You Can Run, But You Can't Hide: An Effective Statistical Methodology to Trace Back DDoS Attackers," *IEEE Trans. Parallel and Distributed Systems,* vol. 16, no. 9, Sept. 2005.
[15] Q. Li, E. Chang, and M.C. Chan, "On the Effectiveness of DDoS Attacks on Statistical Filtering," *Proc. IEEE INFOCOM '05,* Mar. 2005.
[16] R. Mahajan, S.M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Schenker, "Controlling High Bandwidth Aggregates in the Network," *ACM SIGCOMM Computer Comm. Rev.,* vol. 32, no. 3, pp. 62-73, July 2003.
[17] R. Mahajan, S. Floyd, and D. Wetherall, "Controlling High-Bandwidth Flows at the Congested Router," *Proc. Ninth IEEE Int'l Conf. Network Protocols,* Nov. 2001.
[18] B.C.B. Chan, J.C.F. Lau, and J.C.S. Lui, "OPERA: An Open-Source Extensible Router Architecture for Adding New Network Services and Protocols," *J. Systems and Software,* vol. 78, no. 1, pp. 24-36, Oct. 2005.
[19] R. Pan, B. Prabhakar, and K. Psounis, "CHOKe: A Stateless AQM Scheme for Approximating Fair Bandwidth Allocation," *Proc. IEEE INFOCOM '00,* Mar. 2000.
[20] H. Wang, D. Zhang, and K.G. Shin, "Detecting SYN Flooding Attacks," *Proc. IEEE INFOCOM '02,* June 2002.
[21] T.Y. Wong, J.C.S. Lui, and M.H. Wong, "An Efficient Distributed Algorithm to Identify and Traceback DDoS Traffic," *Computer J.,* vol. 49, no. 4, 2006.
[22] D.K.Y. Yau, J.C.S. Lui, F. Liang, and Y. Yeung, "Defending against Distributed Denial-of-Service Attacks with Max-Min Fair Server-Centric Router Throttles," *IEEE/ACM Trans. Networking,* vol. 13, no. 1, Feb. 2005.
[23] H. Xu, A. Datta, and S.P. Bhattacharyya, "PID Stabilization of LTI Plants with Time-Delay," *Proc. 42nd IEEE Conf. Decision and Control,* Dec. 2003.

Chee Wei Tan received the MA degree in electrical engineering from Princeton University, Princeton, New Jersey, in 2006. He is currently working toward the PhD degree in the Department of Electrical Engineering at Princeton University. He was a research associate with the Advanced Networking and System Research Group at the Chinese University of Hong Kong in 2004. He spent the summer of 2005 at Fraser Research Lab, Princeton, New Jersey. His current research interests include queueing theory, nonlinear optimization, communication networks, and wireless and broadband communications. He is a student member of the IEEE.

Dah-Ming Chiu (SM'03) received the BSc degree from Imperial College London in 1975, and the PhD degree from Harvard University in 1980. He worked for Bell Labs, Digital Equipment Corporation, and Sun Microsystems Laboratories. Currently, he is a professor in the Department of Information Engineering at the Chinese University of Hong Kong. He is on the editorial board of the *International Journal of Communication Systems*. He is a senior member of the IEEE.



John C.S. Lui (SM'02) received the PhD degree in computer science from the University of California at Los Angeles (UCLA). He worked at the IBM T.J. Watson Research Laboratory and the IBM Almaden Research Laboratory/San Jose Laboratory after his graduation. He participated in various research and development projects on file systems and parallel I/O architectures. He later joined the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests are in systems and in theory/mathematics. His current research interests are in theoretic/applied topics in data networks, distributed multimedia systems, network security, OS design issues, and mathematical optimization and performance evaluation theory. Dr. Lui has received various departmental teaching awards and the CUHK Vice-Chancellor's Exemplary Teaching Award in 2001. He is an associate editor of the *Performance Evaluation Journal*, a member of the ACM, and an elected member of the IFIPWG7.3. He was the TPC cochair of ACM Sigmetrics 2005, and is currently on the Board of Directors for ACM SIGMETRICS. He is a senior member of the IEEE.

David K.Y. Yau (M'97) received the BSc (first class honors) degree from the Chinese University of Hong Kong, and the MS and PhD degrees from the University of Texas at Austin, all in computer sciences. From 1989 to 1990, he was with the Systems and Technology group of Citibank, NA. He is currently an associate professor of computer sciences at Purdue University, West Lafayette, Indiana. Dr. Yau was the recipient of an IBM graduate fellowship. He received a US National Science Foundation (NSF) CAREER award in 1999, for research on network and operating system architectures and algorithms for quality of service provisioning. His other research interests are in network security, value-added services routers, and mobile wireless networking. He is a member of the IEEE and the ACM, and serves on the editorial board of the *IEEE/ACM Transactions on Networking*.