# WARD: a transmission control protocol-friendly stateless active queue management scheme

C.-Y. Ho, Y.-C. Chan and Y.-C. Chen

**Abstract:** In this article, the problem of providing a fair bandwidth allocation to the flows sharing a congested link in a router is investigated. Queue management, bandwidth share and congestion control are very important to both the robustness and fairness of the Internet. The buffer at the outgoing link is a simple FIFO, shared by packets belonging to the flows. A new transmission control protocol (TCP)-friendly router-based active queue management scheme, termed WARD, is proposed to approximate the fair queueing policy. WARD is a simple packet-dropping algorithm with a random mechanism which discriminates against flows that submit more packets per second than is allowed as their fair share. By doing this, it not only protects TCP connections from user datagram protocol flows, but also solves the problem of competing bandwidth among different TCP versions, such as TCP Vegas and TCP Reno. In addition, WARD works quite well for TCP flow isolation even with different round trip times. In other words, WARD improves the unfair bandwidth allocation properties. Furthermore, as it is stateless and easy to implement, WARD controls unresponsive or misbehaving flows with only a minimum overhead.

## 1 Introduction

The Internet provides a connectionless, best-effort, end-to-end packet service using the IP protocol. Its performance and stability depends on the congestion avoidance algorithm, introduced by Jacobson [1], being implemented in the transport layer protocols to provide good service under heavy load. In practice, TCP has been widely deployed to carry most of the traffic in the Internet. TCP helps a traffic source to determine how much bandwidth is available in the network and adjust its transmission rate accordingly. By using an additive increase/multiplicative decrease strategy, TCP keeps the network from being overloaded; it has become a crucial factor in the robustness and stability of the Internet.

However, a lot of TCP implementations do not include the congestion avoidance mechanism either deliberately or by accident [2]. Moreover, different types of applications, especially multimedia and audio/video streaming applications, are being increasingly deployed over the Internet. Such applications utilise the user datagram protocol (UDP) instead, which does not employ end-to-end flow and congestion control. In other words, the flows of these applications do not back off properly when they receive congestion indication. Rather, the sending rate is set by the application and normally little or no consideration of network congestion is taken into account during the transmission. As a result, UDP flows aggressively use up more bandwidth than other TCP compatible flows. This could eventually cause 'Internet Meltdown' [3] or result in two major problems already identified in the Internet: unfairness and congestion collapse [4]. Therefore it is necessary to have router mechanisms to shield responsive flows from unresponsive or aggressive flows, to effectively detect congestion in the network, to achieve fair share among flows and to provide a satisfactory quality of service to all users.

Besides, even when there is no UDP flow in the network, an unfairness problem may still occur when connections with different TCP versions such as TCP Vegas [5] and TCP Reno [6] coexist [7, 8], as their slow start, congestion avoidance and fast retransmit mechanisms are different. For example, TCP Vegas uses the difference between the expected and actual throughput, whereas TCP Reno detects the packet loss as an indicator to estimate the available bandwidth in the network, control the throughput and avoid congestion. TCP Vegas achieves much higher throughput, and has a fairer and stabler bandwidth share than TCP Reno [5]. The latter is an aggressive control scheme in which each connection captures more bandwidth until the transmitted packets are lost. TCP Vegas, however is a conservative scheme in which each connection obtains a proper bandwidth. Thus, the TCP Reno connections take bandwidth from the TCP Vegas connections when they coexist [9]. To solve this unfairness problem, several end-system-based solutions such as TCP NewVegas [10], TCP Vegas-A [11], TCP Vegas+ [12] and so on have been proposed. Nevertheless, these mechanisms merely solve the problem in some particular network environments. As a result, to regulate the flows causing the unfairness problem, router-based schemes are more appropriate.

This article takes a step in the direction of bridging fairness and simplicity. Specifically, we exhibit an active queue management (AQM) algorithm, called WARD, that is simple to implement and differentially penalises misbehaving flows by dropping more of their packets. By doing this, WARD aims to approximate max–min fairness for the flows that pass through a congested router.

© The Institution of Engineering and Technology 2007

doi:10.1049/iet-com:20060595

Paper first received 4th April 2006 and in revised form 11th June 2007

C.-Y. Ho and Y.-C. Chen are with the Department of Computer Science, National Chiao Tung University, No. 1001, Ta Hsueh Road, Hsinchu City 30050, Taiwan

Y.-C. Chan is with the Department of Computer Science and Information Engineering, National Changhua University of Education, No.1, Jin-De Road, Changhua City 50007, Taiwan

E-mail: cyho@csie.nctu.edu.tw

The basic idea behind WARD is that the contents of the FIFO buffer form a sufficient statistic about the incoming traffic and can be used in a simple fashion to penalise misbehaving flows. When a packet arrives at a router, WARD may draw two packets randomly from the FIFO buffer and compare them with the arriving packet. If two or all of them belong to the same flow, then they are dropped, else the randomly chosen packets are left intact and the arriving packet is admitted into the buffer. The reason for doing this is that the FIFO buffer is more likely to have packets belonging to a misbehaving flow and hence these packets are more likely to be chosen for comparison. Further, packets belonging to a misbehaving flow arrive more often in greater numbers and are more likely to trigger comparisons. The intersection of these two high-probability events is precisely the event in which packets belonging to misbehaving flows are dropped. Therefore, packets of misbehaving flows are dropped more often than packets of well-behaved flows.

The rest of this paper is organised as follows. Section 2 explains our motivation and goals for using the WARD mechanism and describes the WARD algorithm in detail. The simulation results are presented in Section 3. Finally, a summary of this work is provided in Section 4.

## 2 Motivation, goal and WARD

Our work is motivated by the need for a simple, TCP-friendly and stateless algorithm that can achieve fair bandwidth allocation and flow isolation, as there are some problems or drawbacks in the existing algorithms (Because the space is limited, the details and descriptions of each mechanism can be found in [13].). For example, how to set the value of minimum and maximum thresholds in the random early detection (RED) [14] and CHOose and Keep for responsive flow (CHOKe) [2] algorithms to suit a topology with variable connections is a tough problem because both RED and CHOKe are sensitive to parameter settings [15]. Moreover, improper parameter settings may lead to unsatisfactory TCP performance [16]. Similarly, how to decide the suitable size of the tables in a stochastic fair blue [7] algorithm is another question. We are seeking a solution to avoid these problems in the context of the Internet. In addition, we are motivated to find a solution that could penalise the 'unresponsive' or 'unfriendly' flows, such as UDP-based and poor implementations of TCP. Further, we hope that our solution will avoid global synchronisation, ensure low delays, keep buffer occupancies small and bias against bursty traffic. Thus, we propose a stateless queue management algorithm, WARD, which is discussed in the following paragraphs.

Next, we need a benchmark to compare the extent of fairness achieved by our solution. From [2], max–min fairness suggests itself as a natural candidate for two reasons: (a) It is well-defined and widely understood in the context of computer networks and (b) the fair queueing (FQ) algorithm is known to achieve it. However, it seems almost impossible for a scheme to achieve perfect max–min fairness without flow state information. Max–min fairness is not suitable in our context because we do not identify the flow(s) with the minimum resource allocation and maximise its (their) allocation, instead we identify and reduce the allocation of the flows that consume the most resources. In other words, we attempt to minimise the resource consumption of the maximum flow as a fairness index function (1), proposed in [18]. The resource freed up as a result of minimising the maximum flow consumption is distributed among the other flows. In the Internet context, the former flows are either unfriendly TCP or UDP and the latter flows are TCP.

$$F(x) = \frac{(\sum x_i)^2}{n(\sum x_i^2)} \quad (1)$$

### 2.1 Mechanism description

Suppose that a router maintains a single FIFO buffer for queueing the packets of all the flows that share an outgoing link. We describe an algorithm, WARD, that differentially penalises unresponsive and unfriendly flows. In addition, even though the idea of WARD is similar to CHOKe, their methods are different. (There are two major differences between WARD and CHOKe. One is that WARD is embedded in Drop Tail and CHOKe is embedded in RED. Therefore the two schemes deal with an incoming packet differently. The other is that in order to protect congestion-sensitive flows from congestion-insensitive or congestion-causing flows effectively, WARD selects more packets queued in the buffer to compare with an incoming packet than CHOKe does.) The state, taken to be the number of active flows and the flow ID of each parameter packet, is assumed to be unknown to the algorithm. The only observable for the algorithm is the total occupancy of the buffer.

Before describing the WARD algorithm, we give a weighted value to every position in the FIFO buffer. First, the index of every position is divided by the FIFO buffer size. Second, the weighted value of every position equals the first number beyond a decimal point of the above result. For example, assuming the FIFO buffer size is 100, then the weighted values of the 1st to 9th position are 0.0, the 10th to 19th 0.1, ..., the 90th to 99th 0.9 and the last position is 1.0. The packets entering the beginning or the head of a buffer means that there is no congestion yet; however, the congestion may occur when the buffer becomes full.

When a packet $k$, which may be queued into the position $P$, arrives at the buffer, we choose a uniformly distributed random decimal number $U$, which is no larger than 1. If $U$ is bigger than the weighted value of position $P$, this packet $k$ is queued into the FIFO buffer. Otherwise, the packet $k$ is compared with two packets $i$ and $j$ that are randomly selected from the FIFO buffer. First, if these three packets have the same flow ID, they will be all dropped. Second, the flow ID of either packet $i$ or $j$ is the same as that of the packet $k$; these two packets with same ID will both be dropped. Third, if packets $i$ and $j$ have the same flow ID, which is different from the flow ID of the packet $k$, both packets $i$ and $j$ will be dropped too. Otherwise, the randomly selected packets $i$ and $j$ are kept in the buffer (in the same position as before) and the arriving packet $k$ is queued in the position P. Besides, in the sensitive case, the buffer is full when the packet $k$ comes in. WARD will do all the above steps except queueing the packet $k$ in the last step. A flow chart of the WARD's algorithm is given in Fig. 1.

There are two reasons for choosing two packets randomly from the FIFO buffer. (1) The FIFO buffer is more likely to have packets belonging to a misbehaving flow and hence these packets are more likely to be chosen for comparison. (2) If we randomly choose more than two packets from the buffer, the comparison of those packets will be complicated. In addition, according to simulation results, the whole network performance and responsive flow throughput are decreased, although the dropping rate of the

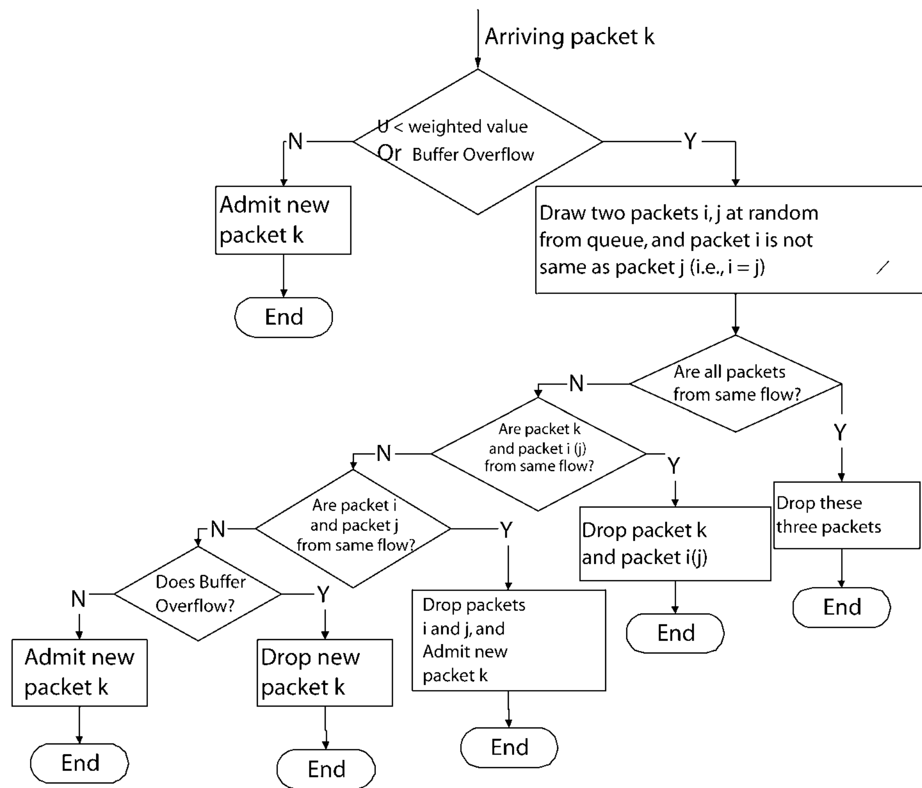


**Fig. 1** *WARD algorithm*

unresponsive flows for choosing more packets is (just a little) higher than that for choosing two packets.

WARD is a truly stateless algorithm. It does not require any special data structure. Compared to a pure FIFO queue, WARD just needs to perform a few simple extra operations: giving weighted values, choosing a random number, drawing two packets randomly from the queue, comparing flow IDs and possibly dropping the incoming and the candidate packets. We may say that WARD is embedded in Drop Tail, which is a commonly used scheme, so there is no big problem in using WARD. In other words, as a stateless algorithm, it is nearly as simple to implement as RED. To see this, let us consider the details of implementation. We use two numerals $P$ and $F$ to represent the current buffer length and the fixed physical buffer size, respectively. Thus, a weighted value of each position can be counted by $P/F$. Drawing two packets randomly can be implemented by generating a random address from which a packet flow ID is read out. Comparison of flow ID can be done easily in hardware. It is arguably more difficult to drop one or two randomly chosen packets because this means removing them from a linked list. Instead of doing this, we add one extra bit to the packet header, as is done in [2]. If the potential candidate is to be dropped, the bit is set to one and the value of $P$ is decreased at the same time. When a packet advances to the head of the FIFO buffer, the status of this bit determines whether it is to be immediately discarded or transmitted to the outgoing link.

In the following section, we demonstrate that the proposed scheme improves the fairness of bandwidth allocation on the basis of numerical results. Indeed, a deterministic fluid model that explicitly models the feedback equilibrium of TCP/WARD system and the UDP throughput behaviour with WARD can be found in [9]. (When the number of TCP flows is large, the UDP bandwidth share peaks at $(2e)^{-1} = 0.184$ as well as drops to zero as the UDP input rate tends to infinity.)

## 3 Simulation results

This section presents simulation results of WARD's performance in penalising misbehaving flows and thus approximating fair bandwidth allocation. We use the RED, CHOKe, and Drop Tail schemes for comparison. The mechanisms that require full per-flow state information are not included here because of the practical limitations of scalability, especially in high-speed routers that usually handle thousands of flows. The simulations range over a spectrum of network configurations and traffic mixes. The results are presented in five parts: single unresponsive flow, multiple unresponsive flows, TCP sources with different versions, TCP sources with different round trip times, and web-mixed experiments. (In fact, we observed that when the number of independent flows increases to ten or hundreds of thousand flows, two chosen packets and the incoming packet may come from different flows. In the worst case, the behaviour of WARD is the same as that of Drop Tail. However, only in core routers, a circumstance of many tens or hundreds of thousand flows are likely. In this section, we assumed WARD is used in an edge router and no large flows connect to an edge router. In our next article, we will present a modified mechanism of WARD to handle many thousands flows in a core router.)

### 3.1 Simulation setup

We use the network simulator *ns2* [20] and the dumb-bell topology shown in Fig. 2 to assess the performance of WARD, which will be compared with Drop Tail, RED and CHOKe. The congested link in this network is between the routers *R1* and *R2*. The link, has a capacity

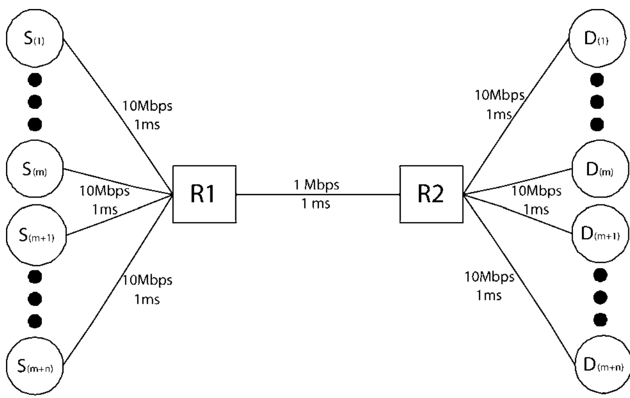


**Fig. 2** *Simulation topology*

of 1 Mbps, is shared by $m$ TCP (with one version) and $n$ UDP or $m$ TCP Vegas and $n$ TCP Reno flows. An end host is connected to the routers using a 10 Mbps link, which is ten times the bottleneck link bandwidth. All links have a small propagation delay of 1 ms except the last scenario, so that the delay experienced by a packet is mainly caused by the buffer delay rather than the transmission delay. The maximum window size of TCP is set to 500 segments such that it does not become a limiting factor of a flow's throughput. The TCP flows are derived from FTP sessions, which transmit large sized files. The UDP hosts send packets at a constant bit rate (CBR) of $\gamma$ Kbps, where $\gamma$ is a variable. The size of all packets are set to 1 KB.

### 3.2 Single unresponsive flow

To study how much bandwidth a single nonadaptive UDP source can obtain when the routers use different queue management schemes, we set up a simulation with 32 TCP sources (*Flow*1 to *Flow*32) and 1 UDP source (*Flow*33) in the network. The UDP source sends packets at a rate of 2 Mbps, twice the bandwidth of the bottleneck link, such that the link *R1–R2* becomes congested.

To observe how WARD achieves fair bandwidth allocation, the individual throughput of each of the 33 connections with buffer size 132 (4 packets per flow), along with their ideal fair shares, are plotted in Fig. 3. Although the throughput of the UDP flow (*Flow*33) is still higher than the rest of the TCP flows, it can be seen that each TCP is allocated a bandwidth relatively close to its fair share. Furthermore, the dropping probability of UDP flow is about 96%. A packet may be dropped because of a match or buffer overflow in WARD. A misbehaving flow, which

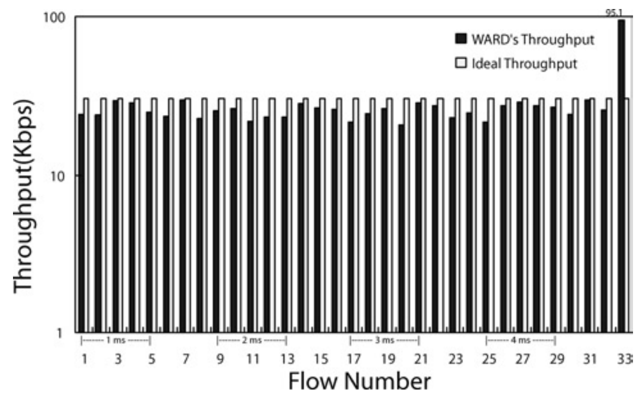


**Fig. 4** *Throughput per flow with different RTTs*

has a high arrival rate and a high buffer occupancy, incurs packet dropping mostly because they are matched. On the other hand, the packets of a responsive flow are unlikely to be matched, so they will be dropped mainly because of buffer overflow. Moreover, we change the link delays for the 32 TCP flows to 1, 2, 3, and 4 ms, respectively. In other words, the 32 TCP flows are separated into four groups. The simulation result is shown in Fig. 4. Although the throughput of the UDP flow (*Flow*33) is still higher than the rest of the TCP flows, it can be seen that each TCP is allocated a bandwidth relatively close to its fair share no matter what the round trip propagation delay is due to.

The throughput of the UDP flow under different queue management algorithms, Drop Tail, RED, CHOKe and WARD, is plotted in Fig. 5. The minimum threshold in the RED and CHOKe is set to 100, allowing on average around three packets per flow in the buffer before a router starts dropping packets. Following [14], we set the maximum threshold to be twice the minimum threshold. In addition, with no partiality, the buffer size of Drop Tail and WARD is fixed at 200 packets due to the maximum threshold mentioned above. From Fig. 5, we clearly see that the Drop Tail and RED gateways do not discriminate against unresponsive flows. The UDP flow takes away more than 85% of the bottleneck link capacity and the TCP connections only obtain the remaining 150 Kbps. Although CHOKe improves the throughput of the TCP flows dramatically by limiting the UDP throughput to 250 Kbps, the UDP throughput is still much higher than each of TCP throughput. WARD boosts the total TCP flows' throughput from 150 Kbps (in Drop Tail gateway) to at least 850 Kbps and limits UDP throughput to at most 150 Kbps, which is only around 15% of the link capacity.

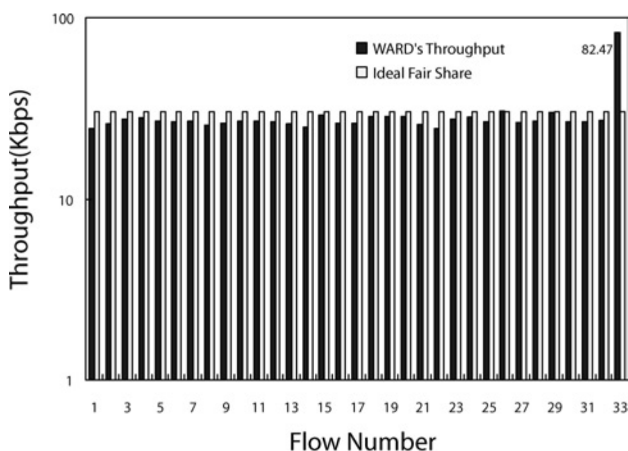


**Fig. 3** *Throughput per flow with same RTT*

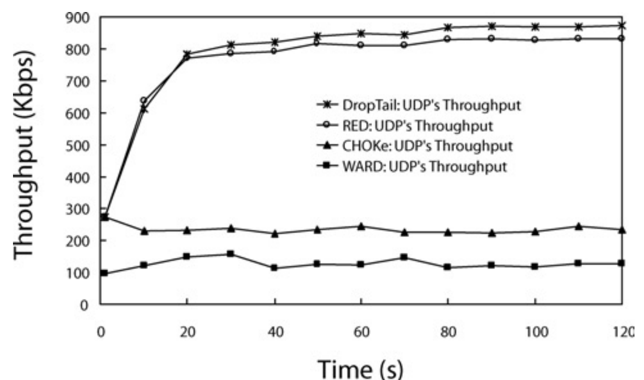


**Fig. 5** *UDP throughput comparison*

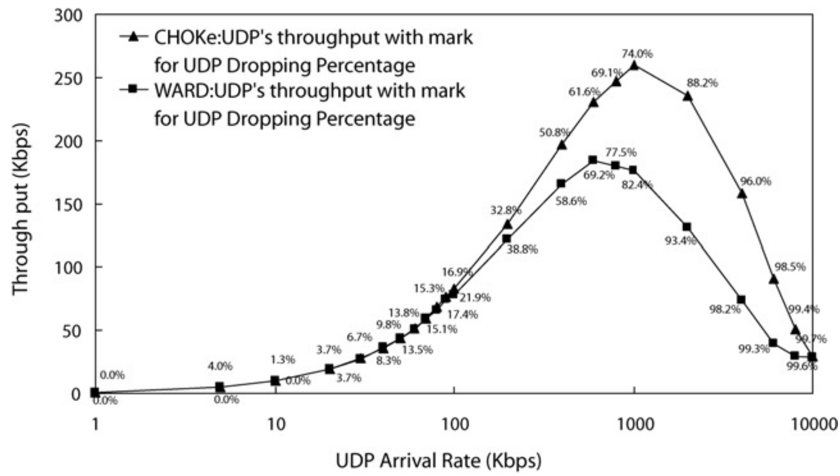


**Fig. 6** *Performance under different traffic load*

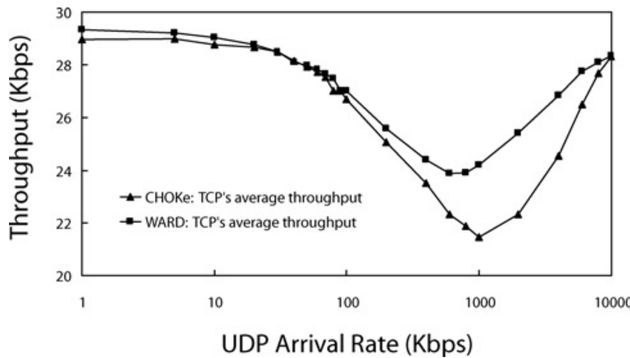


**Fig. 7** *Average TCP throughput under different traffic load*

With the fixed buffer size (200), we vary the UDP arrival rate $\gamma$ to investigate WARD's performance under different traffic load conditions. The simulation results are summarised in Figs. 6 and 7, where the UDP and average TCP throughput are plotted against the UDP flow arrival rate. The drop percentage of the UDP flow is also shown in Fig. 6. From Fig. 6, we see that the maximum UDP bandwidth share is 0.1845 (=184.5 Kbps/1 Mbps) when the UDP's sending rate is set to 600 Kbps. From the plots, we can observe some characteristics of WARD as compared with CHOKe. (1) When the UDP arrival rate is lower than the fair share bandwidth, WARD protects the throughput of the UDP flow as best as it can. For example, no packets are dropped from UDP flow when its arrival rate is 10 Kbps. As the UDP arrival rate increases, the drop percentage goes up as well. For instance, WARD drops 21.9% of the UDP packets when its rate achieves 100 Kbps. Moreover, WARD drops almost all packets (99.7%) when the arrival rate reaches 10 Mbps since the probability of obtaining a matched UDP packet increases with the increasing arrival rate of UDP flow. In other words, the packets of UDP flow have higher probability to be matched. (2) The average throughput of TCP flows with the WARD algorithm is higher than that with CHOKe. The drop percentage of UDP flow when using WARD is higher than when using CHOKe when its rate is higher than the fair share bandwidth. In addition, we do not compare WARD with Drop Tail and RED here because the comparisons of CHOKe, RED and Drop Tail have already been made in [2].

Now, a UDP source uses variable bit rate (VBR) with Pareto model and with shape 1.5 to generate the packets. In addition, UDP source is an ON–OFF source. During ON periods, UDP sends data at 2 Mbps. The average throughput of the UDP flow is from 0 Kbps to 1 Mbps, which is the link capacity. The simulation results are shown in Table 1, where the traffic load is defined as the average throughput of UDP flow divided by 1 Mbps, $TCP_{avg}$ is the average TCP throughput (Kbps), $UDP_{thr}$ is the UDP throughput (Kbps) and $P_{drop}$ is the drop percentage of the UDP flow, respectively. No matter what the traffic load is, WARD protects TCP flows as best it can.

**Table 1: Comparisons of WARD with different traffic load**

| Traffic load | 0 | 0.1 | 0.2 | 0.3 |
|---|---|---|---|---|
| $TCP_{avg}$ | 29.30 | 27.35 | 26.51 | 25.63 |
| $UDP_{thr}$ | – | 64.60 | 95.26 | 129.06 |
| $P_{drop}$ | – | 32.7% | 41.0% | 51.4% |
| Traffic load | 0.4 | 0.5 | 0.6 | 0.7 |
| $TCP_{avg}$ | 24.71 | 24.50 | 24.55 | 23.90 |
| $UDP_{thr}$ | 155.26 | 160.93 | 163.33 | 182.00 |
| $P_{drop}$ | 63.1% | 68.2% | 72.3% | 71.9% |
| Traffic load | 0.8 | 0.9 | 1.0 | – |
| $TCP_{avg}$ | 24.03 | 24.3 | 24.10 | – |
| $UDP_{thr}$ | 179.93 | 170.00 | 179.26 | – |
| $P_{drop}$ | 78.1% | 80.0% | 81.3% | – |

With the UDP sending packets at CBR of 2 Mbps, we vary the buffer size from 66 (2 times 33 flows) to 330 (10 times 33 flows) to study WARD's performance. The details of different conditions are shown in Table 2. Although the average TCP flow throughput is still close to the fair share throughput, the UDP throughput increases

**Table 2: Performance of WARD with different buffer sizes**

| Buffer size | 66 | 132 | 200 |
|---|---|---|---|
| $TCP_{avg}$ | 28.87 | 28.61 | 27.02 |
| $UDP_{thr}$ | 65.26 | 82.46 | 131.33 |
| $P_{drop}$ | 96.7% | 95.9% | 93.4% |
| Buffer size | 264 | 330 | – |
| $TCP_{avg}$ | 25.15 | 24.2 | – |
| $UDP_{thr}$ | 193.8 | 225.53 | – |
| $P_{drop}$ | 90.3% | 88.7% | – |

**Table 3: Average throughput of TCP and throughput of each UDP**

| 2UDP | TCP | UDP1 | UDP2 | UDP3 | UDP4 |
|---|---|---|---|---|---|
| Thr | 26.3 | 65.1 | 41.2 | – | – |
| SR | – | 2000 | 1000 | – | – |
| $P_{drop}$ | – | 96.75% | 95.88% | – | – |
| 4UDP | TCP | UDP1 | UDP2 | UDP3 | UDP4 |
| Thr | 24.8 | 64.4 | 62.7 | 57.6 | 25.5 |
| SR | – | 2000 | 1000 | 100 | 30 |
| $P_{drop}$ | – | 96.78% | 93.73% | 42.40% | 15.11% |

relatively. The reason is that, when the buffer size becomes larger, more UDP packets will be queued in the buffer even the dropping probability of UDP packets increases.

### *3.3 Multiple unresponsive flows*

We follow the traffic model mentioned above (i.e. Fig. 2). Recall that the first model includes 32 TCP flows (*Flow*1 to *Flow*32) and 1 UDP flow (*Flow*33); we do not change any variables here except the number of sources with TCP or UDP flows. The second traffic model includes 31 TCP flows (*Flow*1 to *Flow*31) and 2 UDP flows (the sending rate of *Flow*32) is 2 Mbps and that of *Flow*33 is 1 Mbps). There are 29 TCP flows (*Flow*1 to *Flow*29) and 4 UDP flows (*Flow*30 to *Flow*33) in the third traffic model. The rates of the UDP flows are 2 Mbps, 1 Mbps, 100 Kbps and 30 Kbps, which is smaller than the ideal fair share throughput. The results of these two traffic models are shown in Table 3, where Thr is throughput (Kbps) of a flow, SR is sending rate (Kbps) of a UDP flow and $P_{drop}$ is dropping probability, respectively. In addition, the ideal fair share throughput is 30.3 Kbps. From Table 3, we see that the UDP flow with a low rate is also treated fairly. In other words, the dropping probability is greater when the sending rate becomes higher. When the number of TCP and UDP flow changes, the WARD algorithm tries to achieve FQ. If we consider the input rates of UDP flows, the performance is really satisfactory.

### *3.4. TCP sources with different versions*

When a TCP Vegas user competes with other TCP Reno users, it does not receive a fair share of bandwidth due to the conservative congestion avoidance mechanism used by TCP Vegas. The reason is that TCP Reno continues to increase the window size until a packet is lost. This would occur mainly due to buffer overflow (if the queue management algorithm is Drop Tail). This bandwidth estimation mechanism results in a periodic oscillation of window size and buffer-filling behaviour of TCP Reno. Thus, while TCP Vegas tries to maintain a smaller queue size, TCP Reno keeps inserting many more packets into the buffer and stealing more bandwidth [7–11]. How to drop some packets sent by TCP Reno before buffer overflow is an interesting issue.

In this subsection, we use 20 TCP sources in the dumbbell topology, as shown in Fig. 2. Moreover, the ideal fair share throughput of each flow is 50 Kbps. Among these 20 TCP sources, the source traffic from TCP Vegas decreases from 20 to 0. We show the simulation results of using a WARD algorithm with different buffer sizes in Tables 4, 5, and Fig. 8, where *B* is the buffer size, *V* the TCP Vegas and *R* the TCP Reno, respectively. In Table 4,

**Table 4: Fairness index of different TCP versions and buffer sizes**

| Fairness index | | | | | |
|---|---|---|---|---|---|
| *B* | Vegas–Reno flows | | | | |
| | 20–0% | 15–5% | 10–10% | 5–15% | 0–20% |
| 60 | 99.7 | 99.6 | 99.4 | 99.5 | 99.5 |
| 70 | 99.8 | 99.6 | 99.7 | 99.4 | 99.4 |
| 80 | 99.7 | 99.7 | 99.5 | 99.6 | 99.6 |
| 90 | 99.7 | 99.8 | 99.6 | 99.5 | 99.3 |
| 100 | 99.1 | 99.6 | 99.7 | 99.6 | 99.5 |
| 110 | 99.9 | 98.7 | 99.2 | 99.5 | 99.8 |
| 120 | 99.5 | 99.5 | 99.2 | 99.6 | 99.6 |

**Table 5: Average throughput of different TCP versions and buffer sizes**

| Average throughput, Kbps | | | | | | |
|---|---|---|---|---|---|---|
| *B* | Vegas–Reno flows | | | | | Ver. |
| | 20–0 | 15–5 | 10–10 | 5–15 | 0–20 | |
| 60 | 49.9 | 50.2 | 51.1 | 50.6 | – | V |
| | – | 47.0 | 46.0 | 46.8 | 47.4 | R |
| 70 | 50.0 | 50.0 | 50.1 | 49.3 | – | V |
| | – | 47.0 | 47.3 | 47.7 | 47.3 | R |
| 80 | 50.0 | 50.0 | 48.8 | 48.7 | – | V |
| | – | 47.0 | 48.3 | 47.6 | 47.4 | R |
| 90 | 50.1 | 50.2 | 48.0 | 45.9 | – | V |
| | – | 46.8 | 49.2 | 48.5 | 47.2 | R |
| 100 | 49.8 | 49.8 | 47.3 | 46.1 | – | V |
| | – | 47.8 | 49.9 | 48.6 | 47.5 | R |
| 110 | 50.0 | 47.9 | 45.0 | 44.0 | – | V |
| | – | 53.0 | 52.1 | 49.2 | 47.3 | R |
| 120 | 50.0 | 47.9 | 44.7 | 43.9 | – | V |
| | – | 53.3 | 52.4 | 49.2 | 47.3 | R |

we see a fairness index of each buffer size of over 99%, regardless of the number of sources using either TCP Vegas or TCP Reno in the network. Furthermore, the detailed average throughput of TCP Vegas and TCP Reno are shown in Table 5 because in Table 4, the fairness index demonstrates the whole situation. Figure 8 uses a bar chart to present the details of average throughput

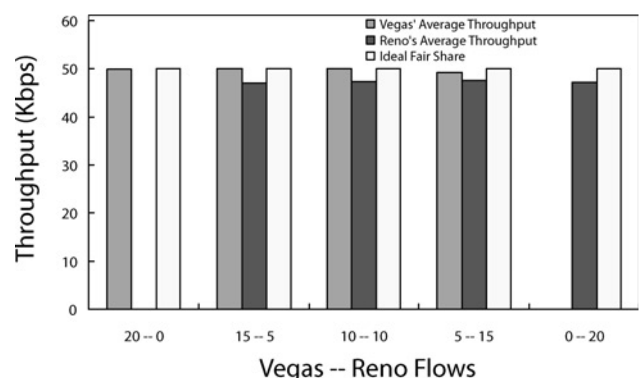


**Fig. 8** *Average throughput of different TCP versions as buffer size is 70*

**Table 6: Fairness index of different queue management algorithms**

| Fairness index algorithm | Vegas–Reno flows | | | | |
|---|---|---|---|---|---|
| | 20–0% | 15–5% | 10–10% | 5–15% | 0–20% |
| Drop Tail | 96.3 | 60.7 | 76.9 | 95.8 | 97.7 |
| RED | 96.3 | 74.2 | 83.4 | 91.3 | 98.9 |
| CHOKe | 96.3 | 84.3 | 92.9 | 97.1 | 99.3 |
| FRED | 96.3 | 62.2 | 78.4 | 91.5 | 99.7 |
| WARD | 99.4 | 99.5 | 99.2 | 99.6 | 99.6 |

when the buffer size is 70. When the buffer size is smaller than 90, the throughput of TCP Vegas is better than that of TCP Reno. On the other hand, if the buffer size is greater than 90, the performance of TCP Vegas is not much worse than that of TCP Reno, since the dropping probability of TCP Reno packets is greater than that of TCP Vegas. Thus, we think that TCP Vegas could compete for network resources with TCP Reno when using WARD algorithm in a router. Furthermore, TCP Vegas and TCP Reno share the network resources fairly when the buffer size is four times the number of total flows.

We compare the fairness index of WARD with other algorithms, such as Drop Tail, RED, CHOKe and fair random early detection (FRED) [21]. In addition, the physical buffer size is 120 for each algorithm and minimum and maximum thresholds are set to 60 and 120 for RED, CHOKe and FRED algorithms. The result is shown in Table 6. We when see that the performance of WARD is much better than others, even the algorithm requires full per-flow state information.

### 3.5 TCP sources with different RTTs

In the early TCP versions such as TCP Reno, when TCP connections feature different RTTs, the sources with shorter RTT get more network resources [22]. Different from TCP Reno, TCP Vegas is not biased against the connections with longer RTT [23, 24]. Thus, we show the results of TCP Reno sources with different RTTs in Fig. 9. The network topology is a dumb-bell, as shown in Fig. 2. There are three cases in our test. In case 1, there are 20 sources competing for 1 Mbps bottleneck bandwidth, so the ideal fair share throughput is 50 Kbps. The RTT of ten sources are 20 ms and 40 ms for the others. From the simulation result, the average throughput of the shorter RTT is 47.31 Kpbs and that of the longer, 48.04 Kbps. We add another ten sources with 60 ms RTTs to the network topology in Case 2. As a result, the ideal fair share throughput is 33.3 Kbps. The average throughput of sources with 20, 40 and 60 ms RTTs are 32.05, 31.65 and 31.42 Kbps, respectively. Similarly, ten sources with 80 ms RTTs are added in Case 3 and the ideal fair share throughput is 25 Kbps; the result is demonstrated in Fig. 9. Although these 40 sources come with different RTTs, the average throughput for each RTT is about 24 Kbps. In addition, the buffer size in the router is 5 times that in sources. For example, the buffer size is 150 (=5 × 30) in Case 2.

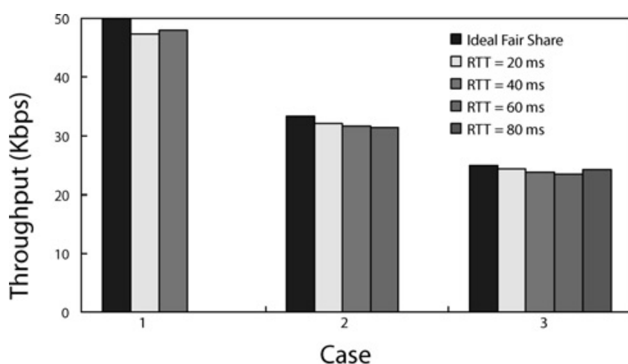


**Fig. 9** *TCP sources with different round trip times*

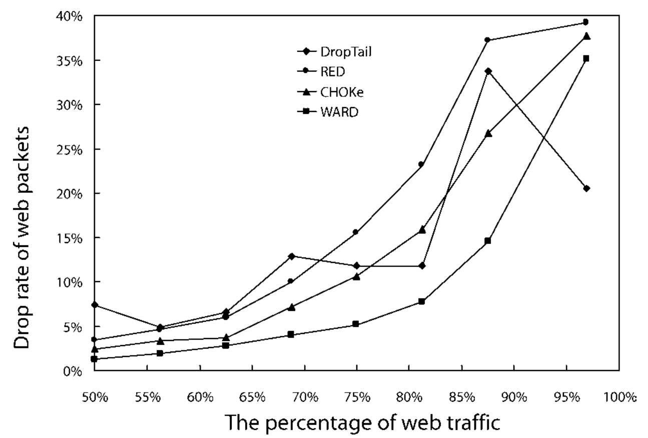


**Fig. 10** *Dropping rate of web packets under different web traffic load*

### 3.6 Web-mixed experiments

Fig. 10 shows the dropping rate of web packets of Drop Tail, RED, CHOKe and WARD for the web-mixed experiments with different web traffic load and FTP flows. In addition, the dropping rate of web packets is counted from the number of dropping web packets divided by the number of dropping packets. From this figure, we see that when the web traffic load is lower than 93%, WARD drops less web packets than the other three mechanisms. In other words, short-lived TCP connections may have better protection from long-lived TCP flows with WARD when the web traffic load is lower than 93%. On the other hand, when the web traffic load is higher than 93%, only the performance of Drop Tail is better than that of WARD. Maybe this is because the probability of choosing packets from the same web source is growing. In order to further improve the performance of WARD when there is a high web traffic load, the scheme will need to be fine-tuned.

## 4 Conclusions

In this article, we propose a packet-dropping scheme called WARD. It is not only a stateless queue management algorithm but also a TCP-friendly router-based mechanism. The goal of WARD is to approximate FQ at a minimal implementation cost. Simulations demonstrate that it works well in protecting congestion-sensitive flows from congestion-insensitive or congestion-causing flows. Also, it solves the problem of competing bandwidth among different TCP versions, such as TCP Vegas and TCP Reno. Furthermore, WARD's performs still better in web-mixed experiments. Further work involves studying the performance and spatial characteristic analysis of this algorithm under a wider range of parameters, network topologies and real traffic traces, obtaining more accurate theoretical

models and insights and considering hardware implementation issues. Moreover, the analysis and simulations of WARD with short-lived TCP flows such as web traffic and the amount of TCP traffic sources hunting for bandwidth will be also discussed in our future work.

## 5 References


1. Jacobson, V.: 'Congestion Avoidance and Control'. Proc. ACM SIGCOMM'88, August 1988, pp. 314–329
2. Pan, R., Prabhakar, B., and Psounis, K.: 'CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation'. IEEE INFOCOM'2000, March 2000, vol. 2, pp. 942–951
3. Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and Zhang, L.: 'Recommendations on queue management and congestion avoidance in the internet'. IETF RFC (Informational) 2309, April 1998
4. Floyd, S., and Fall, K.: 'Promoting the use of end-to-end congestion control in the internet', *IEEE/ACM Trans. Netw.*, 1999, **7**, (4), pp. 458–472
5. Brakmo, L.S., and Peterson, L.L.: 'TCP Vegas: end to end congestion avoidance on a global internet', *IEEE J. Sel. Areas Commun.*, 1995, **13**, pp. 1465–1480
6. Jacobson, V.: 'Modified TCP congestion avoidance algorithm', Mailing list, end-to-end-interest, April 1990
7. Ait-Hellal, O., and Altman, E.: 'Analysis of TCP Vegas and TCP Reno,' IEEE ICC'97, June 1997, vol. 1, pp. 495–499
8. Mo, J., La, R.J., Anantharam, V., and Walrand, J.: 'Analysis and comparison of TCP Reno and Vegas'. IEEE INFOCOM'99, March 1999, vol. 3, pp. 1556–1563
9. Lai, Y.C., and Yao, C.L.: 'Performance comparison between TCP Reno and TCP Vegas'. IEEE ICPADS'2000, July 2000, pp. 61–66
10. De Vendictis, A., Baiocchi, A., and Bonacci, M.: 'Analysis and enhancement of TCP Vegas congestion control in a mixed TCP Vegas and TCP Reno network scenario', *Perform. Eval.*, 2003, **53**, pp. 225–253
11. Srijith, K.N., Jacob, L., and Ananda, A.L.: 'TCP Vegas-A: solving the fairness and rerouting issues of TCP Vegas'. IEEE IPCCC'2003, April 2003, pp. 309–316
12. Hasegawa, G., Kurata, K., and Murata, M.: 'Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the internet', IEEE ICNP'2000, November 2000, pp 177–186
13. Chatranon, G., Labrador, M.A., and Banerjee, S.: 'A survey of TCP-friendly router-based AQM schemes', *Comput. Commun.*, 2004, **27**, (15), pp. 1424–1440
14. Floyd, S., and Jacobson, V.: 'Random early detection gateways for congestion Avoidance', *IEEE/ACM Trans. Netw.*, 1993, **1**, (4), pp. 397–413
15. Bonald, T., May, M., and Bolot, J.C.: 'Analytic evaluation of RED performance'. IEEE INFOCOM'2000, March 2000, vol. 3, pp. 1415–1424
16. May, M., Bolot, J., Diot, C., and Lyles, B.: 'Reasons not to deploy RED'. Proc. Int. Workshop Quality-of-Service (IWQoS), 1999, pp. 260–262
17. Feng, W., Kandlur, D.D., Saha, D., and Shin, K.G.: 'Stochastic fair blue: a queue management algorithm for enforcing fairness'. IEEE INFOCOM'2001, 2001, pp. 1520–1529
18. Jain, R., Chiu, D., and Hawe, W.: 'A quantitative measure of fairness and discrimination for resource allocation in shared computer systems', DEC Research Report TR-301, 1984
19. Ho, C.-Y., Chan, Y.-C., and Chen, Y.-C.: 'WARD study: a deterministic fluid model', *IET Commun.*, 2007, **1**, (4), pp. 711–717
20. http://www.isi.edu/nsnam/ns
21. Lin, D., and Morris, R.: 'Dynamics of random early detection'. ACM SIGCOMM'97, September 1997, pp. 127–137
22. Floyd, S., and Jacobson, V.: 'Connection with multiple congested gateways in packet-switched networks, part1: one-way traffic', *ACM Comput. Commun. Rev.*, 1991, **21**, (5), pp. 30–47
23. Mo, J., La, R.J., Anantharam, V., and Walrand, J.: 'Analysis and comparison of TCP Reno and Vegas'. IEEE INFOCOM99, March 1999, vol. 3, pp. 1556–1563
24. Hasegawa, G., Murata, M., and Miyahara, H.: 'Fairness and stability of congestion control mechanism of TCP', *Telecommun. Syst. J.*, 2000, pp. 167–184