Bandwidth Estimates in the TCP Congestion Control Scheme

Antonio Capone and Fabio Martignon

Politecnico di Milano, Italy

Abstract. Many bandwidth estimation techniques, somehow related to the TCP world, have been proposed in the literature and adopted to solve several problems. In this paper we discuss their impact on the congestion control of TCP and we propose an algorithm which performs an explicit and effective estimate of the used bandwidth. We show by simulation that it efficiently copes with the packet clustering and ACK compression effects without leading to the biased estimate problem of existing algorithms. We present numerical results proving that TCP sources implementing the proposed scheme with an unbiased used-bandwidth estimate fairly share the bottleneck bandwidth with classical TCP Reno sources. Finally, we point out the benefits of using the proposed scheme compared to TCP Reno in networks with wireless links.

1 Introduction

The Transmission Control Protocol (TCP) is based on the assumption that the network does not provide any explicit feedback to the sources. Therefore each source must form its own estimates of the network path properties, such as round-trip time (RTT) or usable bandwidth, in order to perform efficient end-to-end congestion control.

The TCP congestion control has actually the twofold aim to prevent congestion events and achieve a fair share of bandwidth among different connections. Therefore, according to the guidelines in [1] and [2], it's worth to define the *available bandwidth* as the maximum rate at which a TCP connection, exercising correct congestion control, should ideally transmit, and the *used bandwidth* as the rate at which the source is actually sending data.

The most widely deployed TCP implementations (TCP Reno and its extensions as SACKS [3] or New Reno [4]) do not explicitly estimate the available bandwidth. Instead, the end-systems maintain two state variables to regulate the transmission rate: the congestion window (*cwnd*), which usually determines the transmission window, and the slow start threshold (*ssthresh*) that marks the *cwnd* value which discriminates between the slow-start and the congestion avoidance phases. At the beginning of the connection, as the *ssthresh* is set to a big value, the source exponentially increases the number of packets in flight (slow start) until the network drops packets, thus signaling congestion. In response to congestion TCP Reno sets the *ssthresh* to one half of the bytes in

S. Palazzo (Ed.): IWDC 2001, LNCS 2170, pp. 614–626, 2001.

flight, and rapidly enters congestion avoidance phase during which the *cwnd* is linearly increased.

In [5] it has been shown that the general scheme of *additive increase and multiplicative decrease* (AIMD), on which the congestion control scheme of the TCP is based, leads to a fair share of the network bandwidth among different connections in an ideal scenario where all TCP connections take decisions in a synchronized fahion. So, ideally, the *ssthresh* gives an implicit estimate of the available bandwidth and the congestion avoidance is used to gently probe for extra bandwidth.

Unfortunately, it is well known that in real scenarios TCP Reno fails to achieve fair allocation of the bandwidth among connections sharing the same bottleneck when the connections experiment different conditions on the end-to-end path (as for example path delays). For these reasons the *ssthresh* can be considered as an implicit estimator only of the *used* rather than the *available* bandwidth.

Moreover, in TCP Reno, the implicit bandwidth estimate is strictly dependent on the congestion control events experienced by the connection. Therefore, as TCP Reno actually does an implicit estimate of the bandwidth it is using, we may ask whether it is worth performing an explicit run-time estimate of the used bandwidth and how this estimated value can be used by the congestion control scheme.

Various bandwidth estimation techniques, somehow related to the TCP world, have been proposed in the literature and adopted to solve different problems [2,6,7,8,9,10,11]. In this paper we first review these techniques pointing out their impact on the behavior of the TCP congestion control (Section 2). We then propose an algorithm which performs an explicit and effective estimate of the used bandwidth and show by simulation that it efficiently copes with the packet clustering and ACK compression effects without leading to the biased estimate problem of the algorithm proposed in [10,11] (Section 3). We show, however, that the best way to use the estimated value is to set the *ssthresh* to the byte-equivalent of the bandwidth/delay product only after congestion events as proposed in [10,11]. Moreover, we present numerical results proving that TCP sources implementing the proposed scheme with an unbiased used bandwidth estimate fairly share the bottleneck bandwidth with classical TCP Reno sources provided that the end-to-end path conditions are the same. Finally, we point out the benefits of using the explicit used bandwidth estimate compared to the implicit bandwidth estimate of TCP Reno when wireless links are on the path.

2 Estimation Techniques

In a classical IP architecture, to provide best effort service the network resources must be shared by all flows in an as fair as possible way. A centralized controller could in principle regulate the rate of all flows to ensure fairness based on the knowledge of the number of flows and the routing paths. However, a central

controller is unfeasible and too far from the IP philosophy, so the network must somehow estimate the bandwidth availability in a distributed way.

In Core Stateless Fair Queing (CSFQ) scheme [6], bandwidth estimate is performed at the IP router level. The router, knowing the bandwidth B_k of its k-*th* outgoing link and the number n_k of active flows by means of packet classification, estimates by B_k/n_k the bandwidth available to each flow. Through a run-time estimate of the bandwidth actually used by each flow, the router can decide to drop packets belonging to connections using bandwidth in excess, i.e. connections sending at a rate greater than the available bandwidth B_k/n_k. This approach has been shown to solve problems of unfairness among connections having different round trip times, and can be the basis for mechanisms designed to regulate *non TCP-friendly* or *unresponsive* flows [12].

CSFQ has the great advantage of forcing flows to fairly share bandwidth even when the congestion control mechanism of the transport protocol is not accurate. However, it requires relevant modifications in the IP routers and it cannot be easily deployed over the Internet.

If only the end-systems are in charge of the rate regulation without any explicit support from the network, some kind of bandwidth estimate must be performed at the TCP level. Explicit bandwidth estimation algorithms have been proposed to be used by the TCP sources at the beginning of the connection. Their main goal is to set the first value of the *ssthresh* in order to mitigate the effect of multiple losses due to the high default value commonly used [7]. Though the *ssthresh* should be set to the *available* bandwidth, most of the proposed schemes estimate the *bottleneck* bandwidth, a quantity which can be more easily tracked by analyzing the timing structure of received acknowledgments (ACKs). The "Packet Pair" algorithm [8] is based on the assumption that if two packets are sent with closely spaced timing, the interarrival time of the ACKs strictly reflects bottleneck bandwidth. However, as shown in [1], this technique performs poorly if implemented at the sender side, mainly due to the ACK compression [13] which alters the ACK spacing. Some variants of "Packet Pair" consist in tracking "Closely Spaced ACKs" (CSAs) [2,7].

A more sophisticated bandwidth estimation scheme which runs throughout the connection has been adopted in TCP Vegas [9]. While TCP Reno relies on packet losses in order to estimate the available bandwidth of the network, TCP Vegas estimates the available bandwidth of the network based on the difference between the expected and the actual flow rate. The expected and actual rates are given by $cwnd/baseRTT$ and $cwnd/RTT$, respectively, where $baseRTT$ is the minimum RTT ever recorded by the TCP source and RTT its last value. By this mechanism, when the network is not congested, the actual flow rate is close to the expected one, while, when network is congested, the actual rate is smaller than the expected flow rate.

TCP Vegas builds over this explicit and continuous bandwidth estimate a new congestion control scheme which leads to convergence of the congestion window to an equilibrium point. It has been shown, however, that even TCP Vegas fails to obtain a fair allocation of bandwidth especially in an heterogeneous

environment. Moreover, it is known that TCP Vegas is greatly penalized by the aggressive nature of TCP Reno, and so it receives very little bandwidth while Reno easily captures the rest [14]. Even the use of RED gateways [15], while bettering the situation, fails to fill the gap between Reno and Vegas. Finally, in [16] it has been pointed out that even in a homogeneous environment, TCP Vegas may fail to achieve fairness, fundamentally due to the convergence to fixed but different values of the *cwnd* parameters of competing connections.

TCP Westwood, recently proposed in [10,11], performs an estimate of the available bandwidth by measuring the returning rate of acknowledgments, and uses this estimate to set the *ssthresh* and the *cwnd* after congestion events such as the receipt of three duplicate ACKs or coarse timeout expirations. TCP Westwood uses this faster recovery mechanism to avoid the blind halving of the sending rate as in TCP Reno after packet losses. Therefore, this explicit bandwidth estimation scheme has a deep impact over the performance of TCP Westwood sources, especially in presence of random, sporadic losses typical of wireless links or with paths with high bandwidth/delay product.

The bandwidth estimation algorithm performed by TCP Westwood as reported in [11] is described by the following pseudocode:

```
if (ACK is received)
   sample_BWE[k] = (acked * pkt_size * 8)/(now - lastacktime);
   BWE[k]= beta*BWE[k-1] + (1 - beta)*(sample_BWE[k] +
                                              sample_BWE[k-1])/2;
endif
```

Here, *acked* indicates the number of segments acked by the latest ACK, *pkt_size* indicates the segment size in bytes, *now* indicates the current time, *lastacktime* indicates the time the previous ACK was received, k and *k-1* indicate the current and previous value of the variables, BWE is the low-pass filtered measure of the available bandwidth, and *beta* is the pole used for the filtering (in [11] a value of $beta = 19/21$ is suggested).

The basic idea of the proposed scheme is to low-pass filter the bandwidth signal to obtain an accurate estimate of the bandwidth not affected by sporadic losses. Unfortunately, filtering directly the samples of BWE presents some drawbacks when the packet interarrivals are significantly different. The example depicted in Figure 1 shows a simple and typical situation where this scheme fails to correctly estimate the bandwidth:

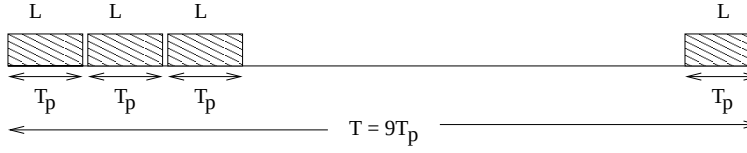


Fig. 1. Packet timing structure

Here L is the packet length expressed in bits, T_p the time interval between contiguous packets, T the total observed time. The bandwidth used by the connection is $\frac{4L}{T}$, and for simplicity $T = 9 * T_p$. The algorithm above, however, estimates approximately the value $\frac{7L}{T}$, as it averages the rates. By filtering we extract the average value of the rate, which is different from the used bandwidth. To be slightly more rigorous let the random variables Y and X represent the packet length and the interarrival time respectively. The average rate is given by:

$$E[\frac{Y}{X}] = E[Y] * E[\frac{1}{X}] \tag{1}$$

being X and Y independent. This value is in general different from the used bandwidth which is given by μ_y/μ_x, where $\mu_x = E[X]$ and $\mu_y = E[Y]$. If we expand the function $1/X$ around the value μ_x, up to the third term, we obtain:

$$E[\frac{Y}{X}] \approx \mu_y * [\frac{1}{\mu_x} + \frac{{\sigma_x}^2}{{\mu_x}^3}] \tag{2}$$

where ${\sigma_x}^2 = E[(X-\mu_x)^2]$. Even if the validity of the expression (2) is limited due to the approximations, it shows that the estimate is biased and the error depends on the variance of the interarrivals.

Figure 2 shows the bandwidth estimated by one of 20 TCP Westwood connections performing the rate estimation algorithm and sharing the same 10 Mb/s bottleneck. Similar curves are observed for the other connections. The bottleneck queue was designed to hold a number of packets equal to the bandwidth/delay product. The one-way-RTT was 50 ms, the test lasted 600 simulated seconds to simulate an FTP session, and *beta* was set to 0.995. These results as all the others presented in this paper were obtained using the Network Simulator, 'ns' ver.2 [17]. To provide a comparison, the dotted line represents the bandwidth estimated by a TCP Reno source running the DFT algorithm we propose and describe in detail in the next section. Since the fair-share value is 500 kb/sec, while TCP Westwood algorithm estimates more than 8 Mb/s we conclude that variance of packet interarrivals is quite large.

The interarrivals would be almost regular if packets belonging to different connections could alternate on the channel. On the contrary, it has been shown that TCP transmissions tend to be clustered so that on a channel we usually observe many consecutive packets of the same connection [13]. Note that the bias on the bandwidth estimate does not depend on the value *beta* chosen for the pole of the IIR filter. It's easy to understand that any fixed value of the pole leads to the same problem.

Filtering directly the rate measured considering the ACK arrival times also exposes the algorithm to the phenomenon of the ACK compression. This happens when the time spacing between returning ACKs is altered due to congestion of the routers on the return path. As one or more ACKs spend some time in the queue of the congested router in the reverse path, subsequent ACKs may reach each other and their original spacing is lost. It has also been shown that ACK compression is quite relevant for real networks operation [18], and therefore it

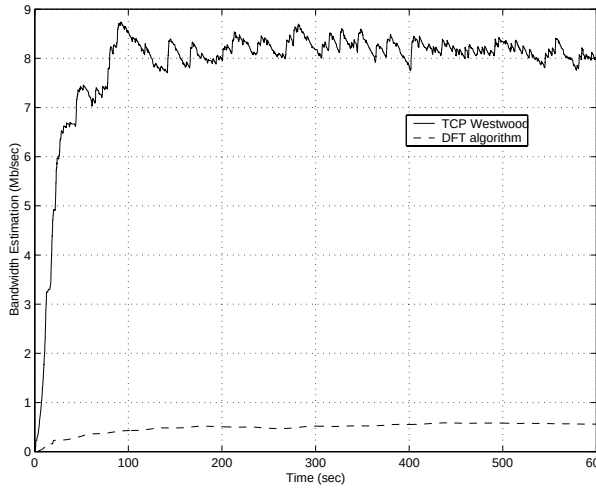


Fig. 2. Bandwidth estimated by TCP Westwood

cannot be neglected. To evaluate the impact of the ACK compression on the algorithm we have considered a scenario with two connections sharing the same 10 Mb/s bottleneck link, but transmitting data in opposite directions, as described in [13]. The end-to-end propagation delay was 100 ms, and the bottleneck queue could contain a number of packets equal to the bandwidth/delay product. The two routers at each end of the bottleneck are therefore charged with both packets from one connection and ACKs of the other, thus leading to the situation of ACK compression described before. The results show that the impact on the estimate of TCP Westwood is dramatic since the estimate is about 25 times higher than that shown in Figure 2.

Finally, we point out that an algorithm implemented in the TCP source according to the TCP Westwood approach can estimate the *used bandwidth* and not the *available bandwidth*. Therefore, the benefits of the new scheme are not due to the estimate of the available bandwidth which cannot be estimated end-to-end, but on the possibility to explicitly estimate the used bandwidth taking into account the short-medium term history of packet arrivals. Moreover, the bandwidth which the algorithm tries to explicitly estimate is the same implicitly considered by TCP Reno and reflected by the *ssthresh* in steady-state conditions. So, if the estimate is accurate enough we expect that the bandwidth used by TCP Reno and by a TCP exploiting a bandwidth explicit estimate are almost the same. This can provide a fair behavior in homogeneous (all sources using the algorithm) and heterogeneous scenarios (with also classical TCP Reno sources).

3 Double Filtering Technique

In this section we present a new technique, the Double Filtering Technique (DFT), which using the basic idea of TCP Westwood succeedes to obtain correct estimates of the bandwidth used by the TCP source.

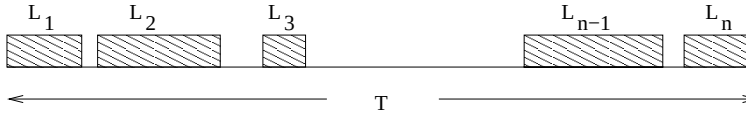


Fig. 3. Packet timing structure

To explain the rationale of DFT let us refer to the example in Figure 3 where transmissions occurring in a period T are considered. Let n be the number of packets belonging to a connection and $L_1, L_2 ... L_n$ the lengths, in bits, of these packets. The average bandwidth used by the connection is simply given by $\frac{1}{T}\sum_{i=1}^{n} L_i$. If we define $\overline{L} = \frac{1}{n}\sum_{i=1}^{n} L_i$, we can express the bandwidth (Bw) occupied by the connection as:

$$Bw = \frac{n\overline{L}}{T} = \frac{\overline{L}}{\frac{T}{n}} \tag{3}$$

The basic idea is to perform a run-time sender-side estimate of the average packet length, $\overline{L}$, and the average interarrival, $\frac{T}{n}$, separately. Following the TCP Westwood approach this can be done by measuring and low-pass filtering the length of acked packets and the intervals between ACKs' arrivals. However, since we want to estimate the used bandwidth we can also low-pass filter directly the packets' length and the intervals between sending times.

Note that sending time intervals can be very very short when groups of packets are generated by TCP sources. However, this is not a problem for DFT since the estimate is performed directly on the interarrival samples. Different would be the case for algorithms that filter the bandwidth samples, such as TCP Westwood, since these samples are close to infinity.

The pseudocodes of the two bandwidth estimation schemes are the following:

1)Processing the stream of *sent packets*:

```
if (Packet is sent)
  sample_length[k] = (packet_size * 8);
  sample_interval[k] = now - last_sending_time;
  Average_packet_length[k] = alpha * Average_packet_length[k-1] +
                                  (1-alpha)*sample_length[k];
  Average_interval[k] = alpha * Average_interval[k-1] +
                                  (1-alpha )* sample_interval[k];
  Bwe[k] = Average_packet_length[k] / Average_interval[k]
endif
```

where *packet_size* indicates the segment size in bytes, *now* indicates the current time, *last_sending_time* the time the previous packet was sent, *k* and *k-1* indicate the current and previous values of the variables. *Average_packet_length* and *average_interval* are the low-pass filtered measures of the packet length and the interval between sending times. *Alpha* is the pole of the two low-pass filters. *Bwe* is the measure of the available bandwidth.

2)Processing the stream of *received ACKs*:

```
if (Packet is received)
  sample_length[k] = (acked * packet_size * 8);
  sample_interval[k] = now - last_ack_time;
  Average_packet_length[k] = alpha * Average_packet_length[k-1] +
                                  (1-alpha)*sample_length[k];
  Average_interval[k] = alpha * Average_interval[k-1] +
                                  (1-alpha )* sample_interval[k];
  Bwe[k] = Average_packet_length[k] / Average_interval[k]
endif
```

where the quantities are the same as before. Here, *acked* indicates the number of segments acked by the latest ACK. In order to compute this value, the algorithm shown in [11] must be used.

If we consider the minimum RTT measured by the TCP source (RTT_{min}) as a good estimator of the end-to-end propagation delay, then we can set:

$$Ssthresh = Bwe * RTT_{min} \tag{4}$$

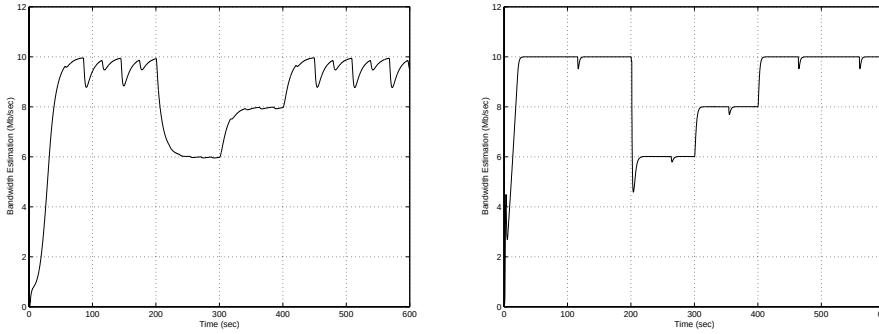
The *ssthresh* is set to the value of equation (4) only after three duplicate ACK's, or after a coarse-grained timeout expiration, following the guidelines of [11].

Simulation results show that DFT is not biased, and obtains bandwidth estimates which oscillate around the fair-share value when all TCP sources experience almost the same path conditions. In order to smooth these oscillations and ensure an estimate closer to the right value, we propose to further filter the value of Bwe as follows [6]:

$$Bwe[k] = (1 - e^{\frac{-T[k]}{T_0}}) * \frac{Average\_packet\_length[k]}{Average\_interval[k]} + e^{\frac{-T[k]}{T_0}} * Bwe[k-1] \tag{5}$$

where $T[k]$ is the instantaneous time interval between two estimates and T_0 is a time constant we set equal to 1 second in our simulations. By binding the value of the pole to $T[k]$, we perform an adaptive filtering which exploits the oscillations of the signal Bwe in order to quickly follow variations in the available bandwidth.

Figures 4a and 4b show the behavior of DFT without and with the filtering performed by Equation (5), respectively. For both the figures the scenario consists of a single TCP connection running over a 10 Mb/s link. In the interval between 200 and 300 seconds, an UDP flow, having the same priority as TCP, transmits at a rate of 4 Mb/s. Then, in the interval between 300 and 400 seconds,



(a) (b)

Fig. 4. DFT bandwidth estimate (a) without adaptive filtering (b) with adaptive filtering

another UDP flow starts transmitting at 2 Mb/s. The bottleneck queue, managed with a drop-tail policy, was designed to contain a number of packets equal to the bandwidth/delay product, and the simulation lasted 600 simulated seconds. In Figure 4a, the oscillations are evident, even if the estimate is unbiased. Moreover, the algorithm adapts slowly to changes in the bandwidth available to the connections. In Figure 4b, instead, the oscillations have been smoothed, and the estimate follows more quickly bandwidth variations in the underlying network.

Following the approach in [11] we set the *ssthresh* to the estimated value only after congestion events. This choice is supported by the worst performance observed with more frequent updating of the *ssthresh* value as shown in Figure 5. The scenario and the parameters considered are the same as in Figure 4, but the *ssthresh* is continuously updated. We observe that the estimate is not accurate mainly because the continuous updating of the *ssthresh* to the estimated used bandwidth value forces the source in the congestion avoidance phase and prevents to follow available bandwidth variations. Similar results have been obtained with a periodic updating with period equal to 0.5 s.

Figure 6 compares the performance of DFT algorithm (the version filtering the stream of *sent* packets), and the TCP Reno, referring to a simulation scenario that considers 10 connections sharing a single bottleneck link of 10 Mb/s with an end-to-end delay of 100 ms. The buffer contains a number of packets equal to the bandwidth/delay product, and FIFO queueing management is adopted, to test DFT even in absence of a somewhat fair queueing. Several simulations have been run and the results have been averaged in order to eliminate phase effects [19]. We have numbered the connections from 1 to 10: the first 5 used DFT algorithm, the other 5 TCP Reno. We observe that an almost fair division of the link has been obtained and both algorithms use almost the same bandwidth.

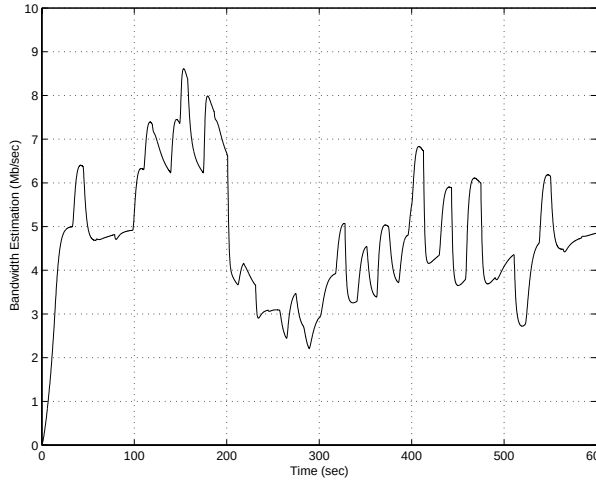


Fig. 5. Bandwidth estimate with continuous *ssthresh* updates

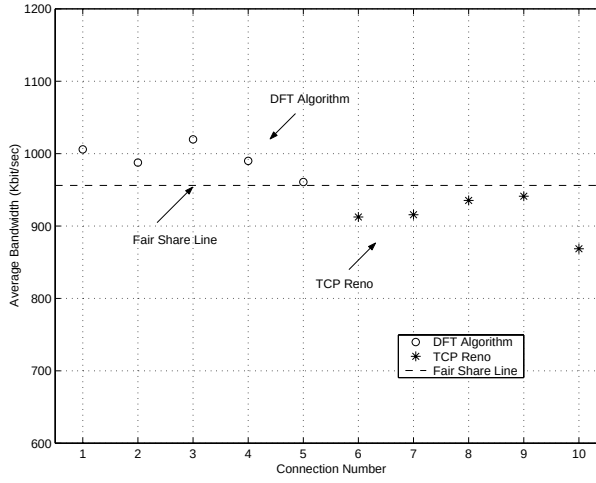


Fig. 6. DFT fairness towards TCP Reno in a 10 Mb/s bottleneck ($alpha = 0.99$)

We have run also simulations over different scenarios covering link bandwidths ranging from few kb/s to 150 Mb/s, varying the number of competing connections and using also a more complex topology with multiple congested gateways. The conclusions obtained are the same: DFT obtains a no worse level of fairness than TCP Reno. Simulation results also show that the strength of DFT lies in its scalability: as more connections share the bottleneck link, as the estimate variance reduces. The presence of constant rate flows, such as UDP flows for IP telephony or video conference, makes DFT perform better as it reduces the dimension of packet clusters.

So far we have proved the accuracy of the DFT algorithm and shown that TCP sources using this algorithm are fair to other sources. To complete the performance evaluation we need to verify the ability to achieve high throughput in presence of links affected by sporadic losses as achieved by TCP Westwood.

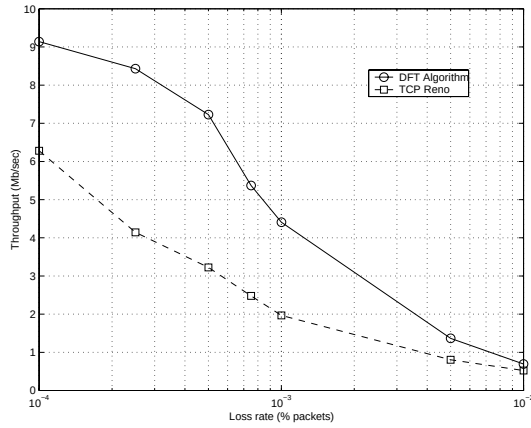


Fig. 7. DFT and RENO throughput vs link error rate

To this purpose, in Figure 7 we compare the throughput achieved by a connection running the DFT algorithm to that of a TCP Reno connection over a link with random errors. The link has a capacity of 10Mb/s, and the FIFO queue can contain a number of packets equal to the bandwidth/delay product. The one-way RTT is 50 ms, and the link drops packets according to a Poisson process with average ranging from 0.01% to 1%. We observe that DFT can sustain higher throughput than TCP Reno at all drop rates considered. This is due to the filtering process which keeps in account also the past history of the bandwidth estimates avoiding to confuse network congestion signals due to queue drops with losses due to link errors.

4 Conclusions

In this paper we proposed the DFT algorithm which performs an explicit and effective run-time estimate of the used bandwidth of a TCP source. It is based on separate filtering of both the intervals between sending times of TCP packets and the packets' lengths.

Following the approach of TCP Westwood, we used the estimate to set the *ssthresh* after congestion events. We proved by simulation that the accuracy of the estimation algorithm allows the proposed scheme to be fair with TCP Reno connections sharing the same bottleneck channel. As a result it is suitable to be gracefully adopted in the IP world with no coexistence problems. In addition

the new scheme, differently from TCP Reno, is effective to cope with channel random errors as it occurs in wireless environments.

Acknowledgements

The authors wish to thank Professor Luigi Fratta for his help and suggestions.

References

1. V. Paxson. End-to-End Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):272–292, 1997.
2. M.Allman and V.Paxson. On Estimating End-to-End Network Path Properties. In *Proceedings of ACM SIGCOMM'99*, 1999.
3. S.Floyd, M.Mathis, J.Mahdavi, and A.Romanow. TCP Selective Acknowledgement Option. *RFC 2018*, April 1996.
4. S.Floyd and T.R.Henderson. The NewReno Modifications to TCP's Fast Recovery Algorithm. *IETF RFC 2582*, 26(4), April 1999.
5. D.Chiu and R.Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance In Computer Networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.
6. I.Stoica, S.Shenker, and H.Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *Proceedings of ACM SIGCOMM'98*, Vancouver, Canada, September 1998.
7. J.C.Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. *ACM SIGCOMM Computer Communications Review*, 26(4):270–280, October 1996.
8. S.Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM'91*, pages 3–15, September 1991.
9. L.S. Brakmo and L.L. Peterson. TCP Vegas: End-to-End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.
10. C. Casetti, M. Gerla, S.S. Lee, S. Mascolo, and M. Sanadidi. TCP with Faster Recovery. In *Proceedings of Milcom 2000*.
11. S. Mascolo, C. Casetti, M. Gerla, S.S. Lee, and M. Sanadidi. TCP Westwood : congestion control with faster recovery. Technical report, UCLA CS Technical Report #200017, 2000.
12. Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, Aug.1999.
13. L.Zhang, S.Shenker, and D.Clark. Observations on the Dynamics of a Congestion Control Algorithm : The Effects of Two-Way Traffic. In *Proceedings of SIGCOMM'91 Symposium on Communications Architectures and Protocols*, pages pages 133–147, Zurich,September,1991.
14. J.Mo, V.Anantharam, R.J.La, and J.Walrand. Analysis and Comparison of TCP Reno and Vegas. In *Proceedings of ACM GLOBECOMM'99*, 1999.
15. S.Floyd and V.Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.
16. Go Hasegawa, M.Murata, and H.Miyahara. Fairness and Stability of Congestion Control Mechanisms of TCP. In *Proceedings of INFOCOM'99*, 1999.

17. ns-2 network simulator (ver.2).LBL. URL: http://www.isi.edu/nsnam.
18. J.C.Mogul. Observing TCP Dynamics in Real Networks. In *Proceedings of ACM SIGCOMM'92 Symposium on Communications Architectures and Protocols*, pages 305–317.
19. S.Floyd and V.Jacobson. On Traffic Phase Effects in Packet-Switched Gateways. *Internetworking:Research and Experience*, 3(3):115–156, September 1992.