Performance Analysis of TCP-Reno and TCP-Sack in the Case of a Single Source

WANG Xin-miao, YAN Pu-liu, HUANG Tian-xi
College of Electronic Information, Wuhan University, Wuhan 430072, China

Abstract: This work explores the behavior of both TCP-Reno and TCP-Sack under a simple scenario, where a single TCP source transmits the packets continuously over a single bottleneck node characterized by its queue size, bandwidth and propagation delay. The analysis allows to derive the performance of TCP, the utilization tends to 75% of the bottleneck throughput when the bandwidth × propagation delay pipe becomes very large, while it tends to 100% when the queuing delays are predominant because the queue is never empty. In the transient analysis we show how the initial phase of the session can degrade the performances. These results are proved through simulation.

Key words: TCP flow control, TCP-Reno, TCP-Sack

CLC number: TN 915

1 Algorithms Presentation

TCP-Reno is basically built on five flow control algorithms that adapt the throughput of TCP by modifying its sending window. First the sending window is set to 1 and the source starts in the so-called Slow-Start algorithm in which the window is increased by one each time an acknowledgment is received until a threshold window is reached. When this threshold is reached TCP enters the Congestion Control algorithm in which the window is increased by one once all packet acknowledgments have been received. It is important to note that every packet is acknowledged by the receiver and a source emits a packet only if an acknowledgment has been received; this is the Self-Clocking algorithm of TCP. This event triggers the Slow-Start mode by setting the window to 1. In the Congestion Avoidance mode the window is increased by one once all packet acknowledgments have been received. It is important to note that every packet is acknowledged by the receiver and a source emits a packet only if an acknowledgment has been received; this is the Self-Clocking algorithm of TCP. This algorithm makes that acknowledgments transmitted at the rate packets arriving at the receiver so that the rate at which the sender transmits is approximately the rate of the bottleneck node.
When the window is too large, losses occur in
the congested node. Losses are detected through
the Fast Retransmit algorithm when the sender re-
ceives a small number of duplicated acknowledg-
ments (in general three). After a loss, Reno does
not resume a Slow-Start as in Tahoe. Rather,
Reno wants to keep advantage of the self-clocked
acknowledgments in the pipe. However as a loss
has occurred it is necessary to reduce the window
size $W$, before loss was detected. For that purpose
the source resumes a new window size of $W/2$ after
acknowledgments in excess have been received and
discarded (this is the Fast Recovery algorithm).
The problem with this algorithm is that, if multi-
ple losses occur during the same window, it is very
hard to know when to start the $W/2$ train of self-
clocked acknowledgments. This is because the
number of remaining acknowledgments is unknown
and the number of losses is not the assumed one
packet loss. This leads to desynchronization, time-
out and poor performance. The Selective Acknowl-
edgment option\textsuperscript{[3]} has been designed to overcome
this problem of Reno. With Sack, when the receiv-
er detects holes in the sequence of packets, it ad-
vises the sender by selectively acknowledging
blocks of packets that have been received correct-
ly. With this procedure the sender knows exactly
the number of acknowledgments in the pipe and
thus can synchronize with the $W/2$ train of self-
clocked acknowledgments. In fact, adding Sack to TCP does not change the basic
underlying congestion control algorithms. Sack
provides a more robust response to a regime of
high successive losses rather than a modification of
the basic performance characteristics of Reno. As a
consequence the subsequent analysis does not focus
on the Sack mechanism and assumes that the syn-
chronization after losses detection is perfect i.e.,
the sender is always able to recognize the number
of holes in the train of acknowledgments. This
assumption is true for Reno as far as the number of
loss is at most one for a given window, and it is al-
ways true for TCP-Sack as far as acknowledgments
are not lost.

2 System Model

The system, including the source, router,
links and receiver, is analyzed through the follow-
ing model shown in Fig. 1.

![System model](image)

The chain of routers and links from the source
to the destination is modeled through its bottleneck
node (i.e., the node connected to the lower link).
The throughput of the node is $\mu$ and its queue ca-
pacity is $\gamma$ (including the packet being trans-
mitted). The scheduling policy is FIFO and the
dropping policy is drop-tail. The delays other
than bottleneck transmission time and queuing are
modeled through a single deterministic parameter
$\tau$. These parameters have no specific units, if $\mu$
is expressed in packets/s then $\tau$ is in seconds and in
packets.

We assume that the source always has packets
to transmit and the window advertised by the re-
ceiver is always greater than the congestion win-
dow. In other words, the performance bottleneck
is the network rather than the receiver host. In ad-
dition, taking into account that all parameters are
known and the source transmits packets continu-
ously, so we can study this system through a de-
terministic analysis.

3 Throughput Analysis

3.1 System behavior

The source starts with a window of one in the
Slow-Start mode. After the threshold window is
reached the Congestion Avoidance algorithm ap-
pies. The window is regularly increased and fills
the bandwidth propagation delay pipe ($\mu\tau$ in our
model). When the $\mu\tau$ window is reached packets
are accumulated in the queue. When the window
reached $q = \mu\tau + \gamma$, the next increase provokes con-
gestion and a packet loss. These two phases are il-
ustrated in Fig 2.

In our model there is only one loss because af-
after the packet in excess has been dropped, follow-
ing packets arrive at the bottleneck rate of $1/\mu$
(this is the self clocking effect of TCP) and over-