# Oscillations of the Sending Window in Compound TCP

Alberto Blanc[1], Denis Collange[1], and Konstantin Avrachenkov[2]

[1] Orange Labs, 905 rue Albert Einstein, 06921 Sophia Antipolis, France
[2] I.N.R.I.A. 2004 route des lucioles, 06902 Sophia Antipolis, France

**Abstract.** One of the key ideas of Compound TCP is to quickly increase the sending window, until full link utilization is detected, and then to keep it constant for a certain period of time. The actual Compound TCP algorithm does not hold the window constant but, instead, it makes it oscillate around the desired value. Using an analytical model and ns-2 simulations we study these oscillations on a Linux implementation of Compound TCP, in the case of a single connection with no cross traffic. Even in this simple case we show how these oscillations can behave in different ways depending on the bandwidth delay product. We also show how it is important to take into account, in the analytical model, that some implementation subtleties may introduce non-negligible differences in the behavior of the protocol.

## 1 Introduction

Tan et al. have introduced Compound TCP [5] to improve the performance of TCP on networks with large bandwidth delay products. One of the main ideas of this new high speed TCP variant is to quickly increase the sending window as long as the network is underutilized and then stabilizing it when a certain number of packets is buffered in the network. To achieve this goal the sender monitors the round trip time: as long as the network is underutilized (i.e. no packets are queued) the round trip time will not change. This corresponds to the case where the window is smaller than the bandwidth delay product. As the window is increased the sending rate will eventually surpass the capacity of the bottleneck link and the round trip time will start to increase. In particular the sender estimates the number of packets currently backlogged in the network. When this estimate is greater than or equal to a threshold $\gamma$ the sender stops increasing its window. Figure 4 in [5], and some of the comments in that paper, give the impression that the window is then kept constant for a certain time.

While this might be a useful approximation in explaining and thinking about this new TCP version, it is easy to see, from equation (5) in [5], that the window will indeed oscillate during this phase. Figure 1 shows the evolution of the sending window from an ns-2 simulation. Clearly, at least in this case, the oscillations have a non-negligible amplitude. In the remainder of this paper we are going to analyze these oscillations in the case of a single Compound TCP connection with no competing traffic and no random losses (packets are dropped only when

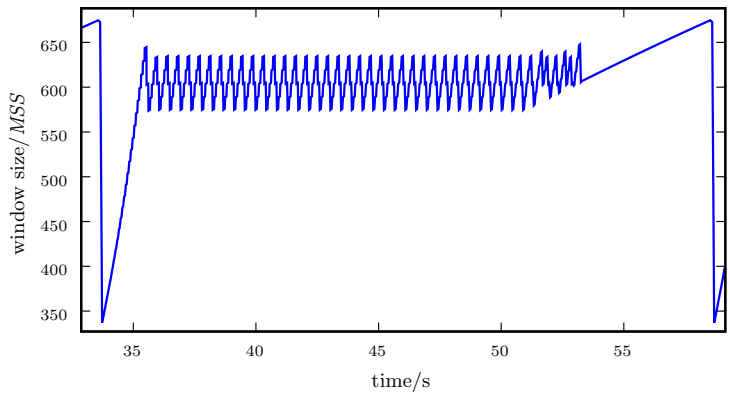


**Fig. 1.** The evolution of the sending window between two packet drops (ns-2 simulation, $\mu = 100\,\mathrm{Mb/s}$, $\tilde{\tau} = 69\,\mathrm{ms}$)

the buffer is full). While this is clearly the simplest possible scenario, even in this case, the oscillations have several possible patterns and, based on simulation results, their amplitude increases with the bandwidth delay product.

These oscillations may have many consequences on Compound TCP flows. Firstly, the packet loss probability in presence of other flows, using Compound TCP or not, will be increased during this phase. This may reduce the proportion of time during which the congestion window is larger than the bandwidth delay product, and then the efficiency of the protocol. Secondly, as these oscillations increase with the bandwidth delay product, the probability to saturate the bottleneck's buffer and will be higher and the efficiency will be lower in case of large round-trip times, as observed in [2]. Thirdly, these oscillations of the window, and, therefore of the buffer occupancy, may also degrade the performance of the network, increasing the delay, the jitter and the loss rate experienced by the other flows.

## 2 Evolution of the Compound TCP Window During the "Constant Window" Phase

During the "constant window" phase, every round trip time, the sender estimates the current backlog in the buffer of the bottleneck through the variable $\Delta$ (*diff* in [5]). An objective of Compound TCP is to keep $\Delta$ positive, to use the bottleneck at full speed, but small (close to $\gamma$), to minimize the buffer occupancy.

Let $w(t)$ be the window size at time $t$ and let $t_i$ be the end of the $i$-th round trip, when the window is increased from $w_i$ to $w_{i+1}$. In order to make the $w(t)$ a left continuous function we set $w(t) = w_i$ if $t_{i-1} < t \leq t_i$. This way $w(t_i) = w_i$. Throughout this paper we will assume that $w$ is expressed in terms of packets, or Maximum Segment Size (MSS). Note that $t_{i-1} - t_i$ is one round trip time

so that the expression "every round trip time" refers to the time between two increments of the window. In the absence of loss, Compound TCP increments the window in the following way (see [5] for a complete description):

$$w_{i+1} = \begin{cases} w_i + \alpha w_i^k & , \text{if } \Delta_i < \gamma \\ w_i - \zeta \Delta_i + 1 & , \text{if } \Delta_i \geq \gamma \end{cases} \tag{1}$$

with

$$\Delta_i = w_i \left( 1 - \frac{\tilde{\tau}}{\tau_i} \right). \tag{2}$$

Where $\tilde{\tau}$ is the smallest round trip time observed so far, and $\tau_i$ is the latest estimate of the round trip time. For the remainder of the paper we assume that $\zeta = 1$, and for all numerical example we use $\alpha = 1/8$, $\gamma = 30$, and $k = 3/4$, as suggested by the authors of CTCP [5]. While selecting these values is an interesting problem in itself it is outside the scope of this work (see [5] for more details).

Clearly as $w$ increases so does $\Delta_i$ and, as the round-trip time is an increasing function of window, eventually $\Delta > \gamma$; and the window will be decreased, provided no packets are dropped in the meantime. Similarly, as smaller values of $w$ imply smaller values of $\Delta$, the window will be increased again. In other words one or more increasing phases are followed by one or more decreasing phases. We call a "cycle" the collection of increasing and decreasing phases starting with the first increasing phase and ending with the last decreasing phase. Figure 2 shows two possible cycles, the one on the left has 3 increasing phases and 1 decreasing phase, while the one on the right has 4 and 2, respectively. We will use two integers to classify cycles, with the first one representing the number of increments and the second one the number of decreasing phases. A 5:2 cycle, for example, has 5 increments and 2 reductions. In the figure dots and circles represents the value of the window *before* and *after* each increment, respectively.

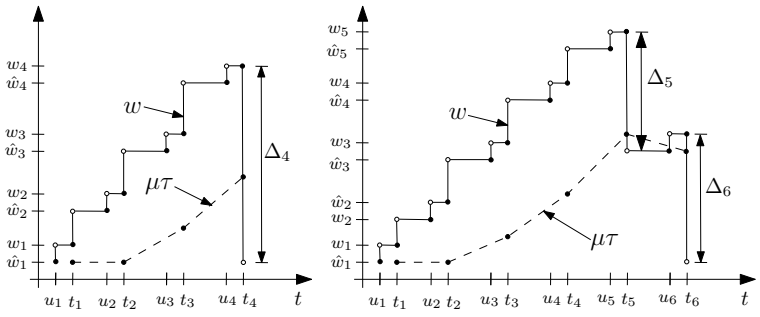


**Fig. 2.** The 3:1 and 4:2 cycles

The sending window of Compound TCP is the sum of two components that are incremented independently during each round trip time. The congestion component is incremented by one each round trip time, just like the window in TCP Reno. While the delay component ($w_{\mathrm{d}}$) is incremented, once per round trip time as well, as $w_{\mathrm{d}} = w_{\mathrm{d}} + \alpha w^k - 1$. The minus one compensates for the increase of the congestion component so that the total window grows as $w = w + \alpha w^k$. Similarly the plus one in (1) comes from the increment of the congestion window component, which happens also during the round trip time when the delay component is decreased. So that in a $m$:$n$ cycle there are $m$ increments of the delay component and $m + n$ increments of the congestion component. In Figure 2 the smaller increments, between $\hat{w}_i$ and $w_i$, represent the increments of the congestion component and they take place at time $u_i$ while the bigger increments are due to the delay component and take place at time $t_i$. Note that $w_1$ is the value *after* the initial increment of the congestion window, such increment represents the increment of the congestion window corresponding to the last reduction of the delay component at the end of the previous cycle.

### 2.1 Linux Implementation

In order to study the behavior of Compound TCP we ran simulations using ns-2 version 2.33 [4] with a Compound TCP implementation for Linux [1]. This implementation uses a slightly different formula to compute $\Delta_i$: instead of $w_i$ it uses $w_{i-1}$ so that (2) is replaced by:

$$\Delta_i = w_{i-1}(1 - \tilde{\tau}/\tau_i). \tag{3}$$

Recall that $w_i$ is the value of the window *before* the increment at time $t_i$ so that $w_{i-1}$ is the value of the window during $(t_{i-2}, t_{i-1}]$. While [5] is somewhat vague about the details of these computations, it is appropriate to use $w_{i-1}$ instead of $w_i$. One way of thinking about $\Delta_i$ is that it tries to estimate the number of backlogged packets by comparing the current round trip time with the smallest round trip time observed so far. Given that the sender uses acknowledgments to measure the round trip time, any such sample corresponds to the round trip time experienced by the packet last acknowledged and such a packet was sent when $w = w_{i-1}$. In other words all the round trip samples are "one round trip time old."

Another aspect to take into account is that, while it is possible to find several equivalent expression for $\Delta_i$, the Linux kernel does not use floating point operations. So that all the operations have to be approximated with integer ones. This introduces an error that can be minimized but that can lead to non-negligible differences between the implemented protocol and a theoretical model. In order to minimize the approximation error $\Delta_i$ is computed as:

$$2\Delta_i = 2w_{i-1} - \left\lfloor \frac{2w_{i-1}\tilde{\tau}}{\tau} \right\rfloor \tag{4}$$

and $\gamma$ is multiplied by 2 whenever it is compared with $\Delta_i$. (We use $\lfloor x \rfloor$ to represent the integer part of $x$.) Computing the window increment $\alpha w_n^k$ presents

a similar problem. As $\alpha = 1/8$ and $k = 3/4$ (as suggested in [5]) the following formula is used:

$$\alpha w_n^k = \left\lfloor \frac{1}{2^3} \left\lfloor \frac{2^8 w_n}{\sqrt{2^{16}\sqrt{w_n}}} \right\rfloor \right\rfloor \tag{5}$$

where all the multiplications (and divisions) by a power of 2 are implemented as shift operations and the square root is implemented using the int_sqrt() function of the Linux kernel.

### 2.2 Modeling the Linux Implementation

For the case of a single connection with no other traffic $\tilde{\tau}$ is equal to the propagation delay, which we assume to be known. In this case it is also possible to express $\tau_i$ (the latest estimate of the round trip time) as a function of the window and the bottleneck capacity $\mu$. In the Linux kernel the round trip times are measured in microseconds so that, even if integers are used, the precision is extremely high.

Assuming that the backlog is non-zero, we can compute the round trip time dividing the window by the bottleneck capacity so that:

$$\tau_i = \min \left[ \left\lfloor \frac{w_{i-2} + 1}{\mu} \right\rfloor, \tilde{\tau} \right]. \tag{6}$$

This is is because the implementation in question uses the smallest round trip time sample among all the samples collected during the last round trip time, between $t_{i-1}$ and $t_i$. (One comment in the source code explains that the choice of using the smallest round trip sample is to minimize the effect of delayed acknowledgments.) As the window, and therefore the round trip time, is an increasing function of time, the smallest value corresponds to the smallest time value; that is the beginning of the round trip. For example in Figure 2, at time $t_4$ the sender considers all the samples relative to the packets sent between times $t_2$ (excluded) and $t_3$ (included), whose acknowledgments were received between $t_3$ (excluded) and $t_4$ (included). At time $t_2$ the window was increased from $w_2$ to $w_3$, but, while this increment is instantaneous, the round trip time grows by smaller increments (more precisely by $1/\mu$) so that the smallest round trip sample *observed* between $t_3$ and $t_4$ corresponds to the packet that was sent when the window was $w_2+1$. Clearly $\tau_i$ cannot be smaller than the propagation delay, hence the minimum with $\tilde{\tau}$. This can indeed happen as $w_i$ can be smaller than $\mu\tilde{\tau}$, either during the initial growing phase during the oscillation phase in the case of multiple reductions.

In [5] the authors say that the window should be updated "once per round trip time." The implementation we used accomplishes this as follows: whenever the window is updated the sequence number of the next segment to be transmitted (say $n_i$) is recorded. Once the corresponding acknowledgment arrives the window is updated another time. As a consequence, whenever the window is reduced because $\Delta \geq \gamma$ the next segment cannot be sent immediately (the window is smaller than the number of unacknowledged packets). The sender resumes

transmission only after receiving acknowledgments for $\Delta$ packets. Due to this pause in the transmission the backlog experienced by packet $n_i$ is smaller. When the corresponding acknowledgment arrives the round trip time (for this packet) is:

$$\tau_i = \min\left[\left\lfloor \frac{w_{i-1} - \Delta_{i-1}}{\mu} \right\rfloor, \tilde{\tau}\right]. \tag{7}$$

Where $w_{i-1} - \Delta_{i-1}$ is the window size after the reduction. Therefore, if at time $t_{i-1}$ the window was reduced, $t_i$ (i.e. the beginning of the new round trip time) corresponds with the arrival of the acknowledgment for packet $n_i$. As we have already mentioned, the implementation we have used takes the smallest of all the round trip samples collected between the last window update, at time $t_{i-1}$ (excluded), and the current time $t_i$ (included). This implies that, each time the window was reduced at $t_{i-1}$, the smallest round trip time sample is due to the last acknowledgment received. And the corresponding packet is the first one sent after the transmission pause.

Note that, provided the queue at the bottleneck link does not empty during the pause, when the sender resumes sending packets the backlog size will not change until the window is updated at time $t_i$. This implies that the value given by (7) is actually the "true" value of the round trip time between $t_{i-1} + \Delta_{i-1}/\mu$ and $t_i$ , where $\Delta_{i-1}/\mu$ is the duration of the transmission pause (it is the time needed to receive enough acknowledgments to compensate for the reduction of the window).

Given that we have defined a cycle as a series of increasing phases followed by one or more decreasing phase, the first phase of a cycle will always follow a window reduction so that, at $t_1$ (7) is used. In this case we also have that $w_m - \Delta_m = w_1 - 1$, where $m$ is the last phase of the previous cycle. (Recall that we have defined $w_1$ as the value after the first increment of the congestion component.) And we can write $\tau_1 = (w_1 - 1)/\mu$. At time $t_2$ the smallest round trip samples received between $t_1$ and $t_2$ is again $\tau_1$ because the packets whose acknowledgments arrive during $(t_1, t_2]$ were sent between during $(t_m + \Delta_m/\mu, t_1]$. As previously observed, all the packets sent during this time experience the same round trip time: $(w_m - \Delta_m)/\mu$.

In general, $\tau_i$ is given by (6) unless the window was reduced at time $t_{i-1}$, in which case (7) should be used, or if the window was reduced at time $t_{i-2}$ and incremented at time $t_{i-1}$, in which case $\tau_i = \tau_{i-1}$. Using these formulas, together with those for $\Delta_i$ (3) and for the window increment (5) it is possible to model the evolution of the window. At the same time it is important to use the same integer approximations used in the Linux kernel: for example if we compute $w_4(w_1)$ (that is the value of the window after three increments with a starting value of $w_1$) using the same formulas used in the Linux Kernel or using floating point operations the difference between the two quantities is between 0 and 2. Where we use the formula $\alpha w_n^k$ to compute the window increment and compute $\Delta_i$ as $\Delta_i = w_i(1 - \tilde{\tau}/\tau)$ when we use floating point operations. Finally, Figure 3 compares $\Delta_5$ using integer and floating point operations.

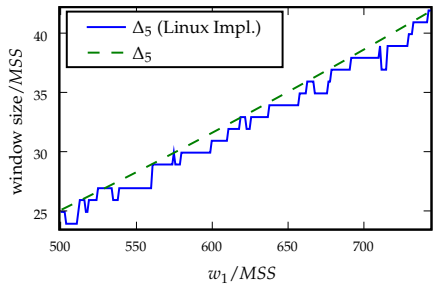


**Fig. 3.** $\Delta_5$ using floating point and integer operations

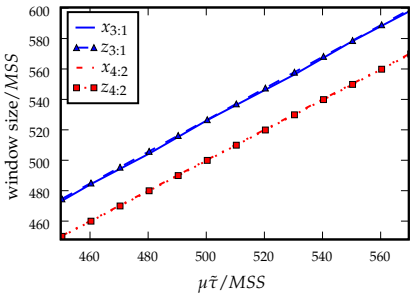


**Fig. 4.** Solutions of the fixed point equation for the 3:1 and 4:2 cycles using floating point ($z$) and integer ($x$) operations

## 3 Fixed Points

Given that, as previously discussed, a series of increasing phases is always followed by one or more decreasing phases it is natural to ask if such oscillations follow a specific pattern and, above all, if they reach a steady state. It is sufficient to look at a few simulations to guess that oscillations do reach a steady state very quickly (after one or two cycles). Depending on the system parameters, we have observed 5 types of cycles: 3:1, 2:1, 5:2, 4:2 and 3:2. For each cycle type it is possible to find the steady state solution by numerically solving a fixed point equation. For the 3:1 cycle, for example, if $w_1$ is the value of the window at the beginning of each phase the value at the end of the cycle is $f_{3:1}(w_1) \triangleq w_4(w_1) - \Delta_4(w_1) + 1$ where $w_{n+1} = w_n + \alpha w_n^k$ and $\Delta_4$ is given by (3). Note that the only independent variable is $w_1$ so that solving the fixed point equation $f_{3:1}(w_1) = w_1$ it is possible to find the steady state solution. The plus one takes into account the fact that, during each cycle, there are three increments of both window components and one increment (by one) of the congestion component combined with one reduction of the delay component. To be precise, here $w_1$ corresponds to the initial value of the total window after an increment by one of the congestion component (see Figure 2).

For each cycle $m$:$n$ it is possible to define the corresponding function $f_{m:1}(w_1) = w_{m+1}(w_1) - \Delta_{m+1}(w_1) + 1$ if $n = 1$ and $f_{m:n} = w_{m+1}(w_1) - \Delta_{m+1}(w_1) - \Delta_{m+2}(w_1) + 2$ if $n = 2$. For the latter case the plus two compensates for the two increments of the congestion component corresponding to the two reductions of the delay component. Figure 4 shows the solution of $f_{3:1}$ and $f_{4:2}$ as a function of the bandwidth-delay product ($\mu\tilde{\tau}$). In this case the difference between using floating point and integer operations is not very significant, especially for the 4:2 case where the error is negligible. While Figure 4 shows only two cases, they are the most representative ones. The solutions for the 2:1 case are close to those of the 3:1 case. And the solutions for all the $m$:2 cases are almost identical. It is interesting to note how the solutions for the $m$:2 cases are very close (and in some cases equal) to $\mu\tilde{\tau}$. As $w_1 - 1$ is the minimum value of the window is

$w_1 = \mu\tilde{\tau}$ then $w_1 - 1 < \mu\tilde{\tau}$ which implies that the buffer will be empty for the first part of each oscillation. The simulations do confirm this, showing that the buffer will be empty for half a round trip time during each oscillation. This is caused by the increment of the congestion component, which usually takes place after half a round trip time after the last reduction. While half round trip time over a cycle of a few round trip times is not a big portion it does nonetheless lead to an under utilization of the bottleneck link during this "constant window" phase negating one of the main design ideas of Compound TCP.

## 4 Conclusions and Future Work

Contrary to what suggested by one of the figures in [5], we have shown how the Compound TCP window does oscillate during the "constant window" phase. These oscillations converge quickly to a cyclic behavior whose mode (number of increases:number of decreases) depends on the bandwidth delay product. Some implementation details on Linux have also an influence on the mode, and on the amplitude of these cycles, especially the discretization of the state variables, increments of the window and backlog estimates.

These oscillations may explain some of the inefficiencies observed for Compound TCP on some tests, especially when the round-trip times are large or the buffers small [3, 2]. This phenomenon may also degrade the performance of the other simultaneous flows. We plan on further analyzing the influence of these oscillations on the performance of long lived Compound TCP connections and on other simultaneous flows. We would also like to understand the relationship between the parameters of the system and the type of cycles followed by the oscillations. While the simulations seem to indicate that this depends on the bandwidth delay product we do not yet know the exact relationship.

## References


1. L. Andrew. Compound TCP Linux module. available at http://netlab.caltech.edu/lachlan/ctcp/, April 2008.
2. Andrea Baiocchi, Angelo Castellani, and Francesco Vacirca. YeAH-TCP: Yet Another Highspeed TCP. In *Proc. 5th Int. Workshop on Protocols for FAST Long-Distance Networks*, March 2007.
3. Y-T. Li. Evaluation of TCP congestion control algorithms on the Windows Vista platform. Technical Report SLAC-TN-06-005, Stanford Linear Accelerator Center, June 2005.
4. S. McCanne, S. Floyd, et al. ns network simulator. available at http://www.isi.edu/nsnam/ns/".
5. K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *INFOCOM 2006. Proc. 25th IEEE Int. Conf. on Computer Communications.*, 2006.