Delay based additive increase for scalable congestion controls

Joakim Misund
University of Oslo, Norway
joakimmi@ifi.uio.no

ABSTRACT

As network capacity continue to increase additive increase's inability to claim capacity more quickly than one packet per round-trim time becomes more apparent. The constant increase of one packet cannot be changed without affecting important steady state properties such as convergence.

We propose that the fundamental problem is that we strive to have one algorithm to handle steady state and transient state. The transient state we focus on here is that in which the network is underutilized and one or more TCP connections are trying to bring it to full utilization. The two states are different enough to warrant different treatment.

The solution that we propose consist of two parts. The first is to be able to detect that a connection is in transient state that warrants more rapid increase. This detection is feasible to do for scalable congestion controls. So although this can be used for tradition TCP it is much more useful for scalable congestion controls that have little oscillation in steady state. The second part is a calculation of an appropriate increase factor that takes queue delay impact into consideration. When we use this alternative increase factor we say that we are in acceleration mode.

We show that the detection is working as intended, and that acceleration mode significantly improve acceleration without overshooting more that intended in various conditions. We implemented the algorithm in TCP Prague and compare it to BBRv2 and regular TCP Prague.

1 INTRODUCTION

Traditional TCP congestion control algorithms are slow to recover full utilization when there is a positive change in available capacity. The reason is that additive increase is used for increasing the congestion window in congestion avoidance. Depending on the round-trip time and number of flows it can take a couple of milliseconds or seconds to reach full utilizationn.

In this paper we focus on the situation where one of more flows in congestion avoidance loose a regular congestion signal. This situation will be called a open-loop phase. In steady-state the controller gets regular signals, but once the go way the controller becomes open-loop.

We introduce a faster acceleration mode that can be added to scalable congestion controls to improve acceleration. We found it necessary to introduce a new mode because any changes to the steady state behaviour seems to have negative consequences for oscillation and competition between flows. We transition from congestion avoidance to faster acceleration mode if we detect that congestion events do not happen when they ought to. This is feasible for scalable congestion controls because they have very little oscillation in steady state. This makes it timely to detect a change from the expected behaviour in steady state. In the acceleration mode the window is increased in such a way that the queue overshoot is bounded. For all the experiments in this paper the overshoot is set to 1 millisecond, but it can be configured to any value.

We combine the faster acceleration mode with a previously discussed mode called the overload mode. Overload mode is entered if there is a unusually large increase in queueing delay. In the overload mode the congestion window is reduced by either the proportion of CE marks or queueing delay depending on which gives the most conservative reduction. The decrease based on queueing delay is so that it tries to drain or remove the standing queue. When the algorithm has back-to-back reductions the effect of a previous change has not been measured when the next change occurs, therefore queueing delay is only reacted to every other round-trip time. However, ECN is reacted to every round-trip time as usual.

Together, these two modes make acceleration when available capacity increases and alleviation of congestion when available capacity reduces more timely. Consequentially, utilization and queueing delay is expected to improve. But as we will discuss further having higher utilization might actually hurt queueing delay as there is less headroom for competing traffic and to deal with sudden reductions in capacity.

2 PROBLEM AND MOTIVATION

Congestion control algorithms operate in two states; steady state and transient state. In steady state competing flows experience a reoccurring oscillation. Usually the term steady state implies no movement or change, but for most congestion controls steady state has a reoccurring oscillation. In control theoretical terms steady state often has a limit cycle. The oscillation is continually reoccurring and it is normally within certain limits which makes it somewhat predictable.

When a flow is in steady state it uses a algorithm called congestion avoidance. Each round-trip time the congestion window is increased by one segment, and at each congestion event (at most one per round-trip time) the window is reduced to a fraction of what it was before the reduction. Different congestion control algorithms use different fractions which means that they have different limit cycle lengths. TCP Reno uses a fraction of 0.5. TCP Cubic uses a fraction of roughly 0.7. DCTCP has its fraction proportional to the fraction of packets carrying a CE mark. Common to these variations is that they use additive increase between congestion events.

Additive increase makes the time to reach full utilization proportion to the difference between the bandwidth-delay product (BDP) and the current aggregate congestion window.

The number of round-trip times can be expressed as follows:

$$X = \frac{1}{N} * \left(\mathrm{BDP} - \sum_{i=0}^{N-1} W_i \right)$$

Where N is the number of flows and W is the congestion window of a flow. This can easily be converted to time by multiplying by the round-trip time. The worst case happens when $N = 1$.

Scaleable congestion controls can keep the average time between congestion events constant as capacity increases. They are able to do so because they react to the proportion of packets carrying CE marks. The steady state proportion for DCTCP and TCP Prague is 2 marks per round-trip time. As long as the network is able to produce 2 marks per round-trip time it does not matter what the capacity is. There will be no oscillation in theory.

Wireless technologies are common in today's internet and it is unlikely to change. These technologies often experience rate changes in either direction which makes it important for congestion control algorithms to be able to deal with the appropriately. It would be appropriate to be able to accelerate fast when available capacity increases, and react fast when available capacity decreases.

When there is an increase in available capacity it takes congestion avoidance with additive increase a long time to reach the new capacity. If the current window is W and the new steady state window changes to $2 * W$ it takes W round-trip times to reach full utilization. This is particularly slow if there are few long-running flows.

When available capacity decreases it takes scalable congestion controls that use smoothing on the CE-marks it receives usually a long time for a sufficient response. Changing the smoothing factor so that one responds more and faster usually has negative effect on steady state oscillation. This make detecting a abnormal state less timely. Additionally, delay is a more fine-grain signal compared to CE-marks when there is a overload scenario because marking probability tends to become saturated (100%). The information about how bad the congestion is not present. So making a change to how CE-marks are used seem like a difficult approach.

3 ACCELERATION MODE

This section is divided in two parts. The first part in about detecting a change to open-loop phase. The second part discussed different algorithms for increasing the congestion window faster than additive increase.

3.1 Determining the current state

To detect a change one has to know what to expect.

There are possible many ways to go about detecting a change to open-loop phase. One way is to actively test if there is more available capacity available periodically. This can be done using packet bursts or chirps for a certain duration. A probe can be analyzed and compared to what one would expect to see. While this might work fine when there is a single flow synchronization among multiple flows make this approach complicated.

A different way is to passively monitor how steady steady state behaves and look for unexpected progression. This is much simpler compared to doing a active measurement and detection.

In steady state a flow experience regular congestion events. The time between each congestion event can be used with the time since the last congestion event to determine whether or not a flow has entered a open-loop phase. In order for this to be feasible this detection has to be timely and reliable. If the condition for detection is too conservative the detection come too late to make a difference. However, if it is too sensitive it can lead to unwanted oscillation and flow rate inequality.

The simplest way to achieve a detection algorithm that uses the time between congestion events is to simply measure what the average time between congestion events is in steady state and compare it to the current time since the last congestion event. To account for variation, often caused by probabilistic congestion signals, the variation should be measured and used with the average. One way to measure variation is to maintain an average of the mean average deviation (MAD, note that M is typically median) over multiple samples.

Using the time between congestion events will mean that the timeliness of the detection algorithm will be strongly linked to the regular or expected time between congestion events. A congestion control that has a large sawtooth will naturally have longer time between congestion events. The longer the time between congestion events is the more time it takes to detect a change.The usefulness of the detection decreases as the expected time between events increase. This reduces the usefulness of a open-loop detection algorithm for TCP Reno and TCP Cubic. The algorithms that benefit the most are those that have little time between congestion events in steady state. One such group are scalable congestion controls which react to the proportion of congestion experienced. Because of this we focus on that class of algorithms in the remaining of the paper.

3.2 What should we do?

Assuming that we have decided that we should increase the congestion window faster than what additive increase would do we need to decide on how fast.

Go back to slow start. Slow start is used for the initial rate probing and can be used in open-loop phase too. There is a very significant difference between the start-up phase and open-loop phase after steady state. When a flow enters open-loop phase it has some useful information about what the available capacity was before it entered open-loop phase. This is not the case in the start-up phase.

It is likely that the new available capacity is not far from where the flow starts its open-loop phase. Slow start is likely to overshoot the new available capacity and cause queueing delay with a likely small improvement in time ti takes to reach full utilization.

Cubic increase function. TCP Cubic uses a cubic function to reduce time between congestion events and improve utilization for flows with large congestion windows. The congestion window increase becomes increasingly aggressive as the time since the next expected congestion event increases. The increase is capped at 50% increase over a round-trip time. Slow start for comparison has a 100% increase.

Capacity estimation and acceleration algorithm. One can use a capacity estimation algorithm that rely on packet bursts or chirps to estimate the new available capacity. However, when there are multiple competing flows there is a significant chance they miss each other and cause a queue overshoot.

Capacity estimation algorithms struggle to estimate capacity over discontinuous links with aggregation such as WiFi. This seriously reduce the usability in todays Internet.

Delay preserving scalable additive increase. A difference between a flow in the start-up phase and a flow entering open-loop phase after steady state is that the latter has a lower limit on the networks capacity. It knows that the current sending rate is lower than the available rate at the bottleneck. This can be used to calculate how fast the congestion window can be increased without causing a unacceptable amount of delay.

3.3 Algorithms put together

We have the following variables:

Variable	Name/Value	Description
C	cycle_counter	Number of rtts since last reaction
I	cycle_increase	Number of increases since last reaction
I_a	cycle_avg	Average of I
I_m	cycle_mad	Average Mean absolute deviation of I
a	scale	Scale to apply to AI
Q	1000	Acceptable queue overshoot in us
R	srtt_us	srtt
W	snd_cwnd	Congestion window
g	1/16	EWMA factor for I_a, I_m

The initialization of the variables follows:

$C \leftarrow 0$
$I \leftarrow 0$
$I_a \leftarrow 0$
$I_m \leftarrow 0$

At the completion of a whole round-trip time the values are updated as follows:

$C \leftarrow C + 1$
$I \leftarrow I + a$
$a \leftarrow 1$

If the state is deemed to be open-loop the scale is updated as follows. The condition for being in open-loop state is that $C > I_a + 2 * I_m$.

if $(C > I_a + 2 * I_m)$ **then**
  $a \leftarrow \frac{(W-a)*Q}{R}$
end if

The scale is subtracted from the current congestion window to make it so that the window used has been tested. Using the current window would include a round of scaled additive increase without feedback confirming it.

Every time there is a congestion reaction the averages are updated and counters reset:

$I_m \leftarrow I_m - I_m * g + |I_a - I| * g$
$I_a \leftarrow I_a - I_a * g + I * g$
$C \leftarrow 0$
$I \leftarrow 0$

3.3.1 Determining the current state. The state is deemed to be transient if $C > I_a + 2 * I_m$. In other words if the number of round-trip times is greater than the average number of increases done in-between reductions pluss two times the variation of increases. The use of increases instead of number of round-trip times avoids issues where a higher scale becomes a new steady state. It makes it more difficult for a flow that is in transient state and has a high scale to stay there once a reaction occurs. It improves sensitivity.

The choice of multiplying I_m by 2 is to approximate 95% confidence value for normally distributed random variable. However, it is really arbitrary.

3.3.2 Dynamic change to additive increase. To further describe this part of the algorithm we define the following variables.

δ_S Inter sending time of packets at current sending rate.
δ_C Serialization delay at the bottleneck.

When we are in transient state we adjust the scale according to the current sending rate and the queue delay target. We assume that $\delta_C < \delta_S$ (higher capacity than sending rate).

First, sending inter-packet gap is computed.

$$\delta_S = \frac{R}{W - a} \tag{1}$$

Then the scale that satisfies the max queue delay target is computed. Again, this can be a fraction of the RTT instead of a constant delay.

$$a = \frac{Q}{\delta_S} \tag{2}$$

The scale is then applied to the additive increase. If Q is set to a fraction of the RTT the result is multiplicative increase as the RTT cancels and the fraction is multiplied by W.

Are multiple flows more aggressive? No, they ensure the same queueing delay target (at least in theory).

Assume that we have N competing flows over a network with a capacity of C capacity and R RTT. We have that each flow should have a window and sending rate as follows.

$$\delta_S = \delta_C * N \tag{3}$$

$$W - a = \frac{RTT}{\delta_S} \tag{4}$$

Then one of the flows terminate, and the remaining flows start to work their way to the new equal share. At the point that the flow terminates there are N-1 flows still sending with the same inter-packet gap, $\delta_C * N$. Assume for now that the

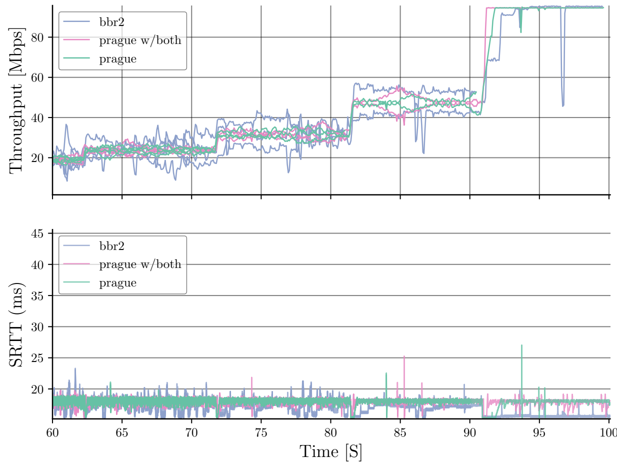

Figure 1: 5 flows exiting staggered. RTT and capacity is 15ms and 100mbps. Each algorithm run separately. AQM is step at 3ms. Acceleration gives large improvement when given time and condition to kick in.

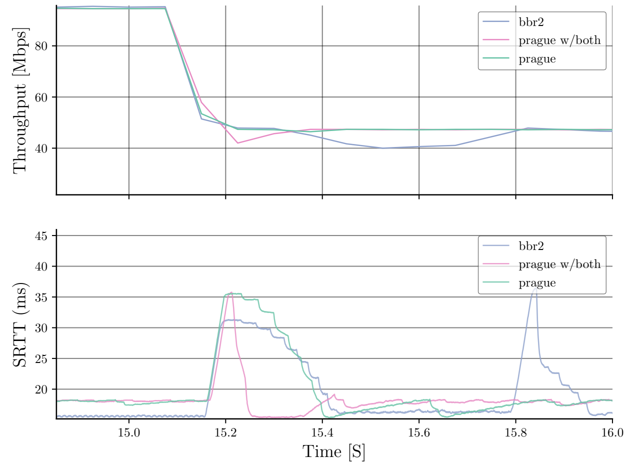

Figure 2: Single flow, each algorithm run separately. 15ms RTT and 100mbps capacity. Capacity halved to 50mbps. AQM is step at 3ms. Overload mode's rapid decrease sometimes lead to under-utilization that can be recovered by acceleration mode.

flows switch to transient state immediately. Each flow will adjust their additive increase to $(W - a)/(\delta_C * N)$ and the aggregate increase becomes

$$\frac{N-1}{N} * \frac{Q}{\delta_C}$$

So depending on what N is the aggregate increase is in the range of $[0.5, 1) * \frac{Q}{\delta_C}$. However, it can never be greater than $\frac{Q}{\delta_C}$, which ensures that the latency target is kept.

Delayed feedback. We have assumed that the delay in feedback is non-existent, which is untrue. The consequence is that the maximum delay target will be violated by a small amount. The delay cannot be removed, so compensating for it by setting the target a bit lower might be necessary.

Increase favors flows with larger window. Because the scale depends on the current congestion window there can be unequal acceleration for competing flows. Flows with lower round-trip time will more quickly detect a open-loop transition and claim spare capacity faster than a competing flow with a longer round-trip time. This is not an issue as the flows will converge to steady state once full utilization has been reached.

4 EVALUATION

In this section we evaluate how the algorithm performs in a range of scenarios. If nothing else is specified we use a delay target, Q, of 1 millisecond. In the experiments dctcp-pattern will be regular DCTCP, and dctcp-fast-c will be DCTCP with the algorithm described above.

4.1 Staggered flows

Figure 1 shows 5 flows exiting one after another, leaving spare capacity for the other to claim. When the capacity difference is great enough for acceleration mode to kick in it reaches full utilization faster.

4.2 Rate decrease

This experiment is simply to cut the available capacity to a fraction of the current capacity. Capacity is reduced by changing the link rate.

Figure 2 shows how throughput and queue length changes when available capacity decreases from 100mbps to 50mbps. BBRv2's acceleration is overly aggressive.

Figure 3 shows how throughput and queue length changes when available capacity decreases from 100mbps to 80mbps. The overload mode reduces excess queueing delay rapidly, but it comes at the cost of under-utilization. However, acceleration mode kicks in and reclaim the available capacity.

4.3 Rate oscillation

Figure 4 shows how different algorithm handle an available capacity change. Acceleration mode significantly improves acceleration when more capacity becomes available. Even when system-relates noise causes a congestion-unrelated spike and premature overload reduction around 43 seconds acceleration mode recovers quickly.

Figure 5

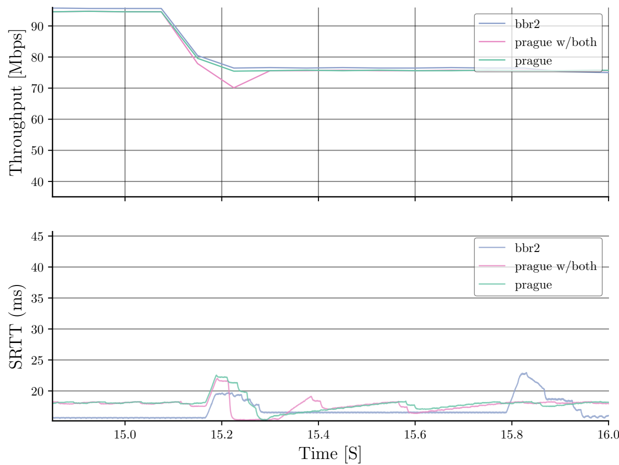

Figure 3: Single flow, each algorithm run separately. 15ms RTT and 100mbps capacity. Capacity reduced to 80mbps. AQM is step at 3ms. Overload mode's rapid decrease sometimes lead to under-utilization that can be recovered by acceleration mode.

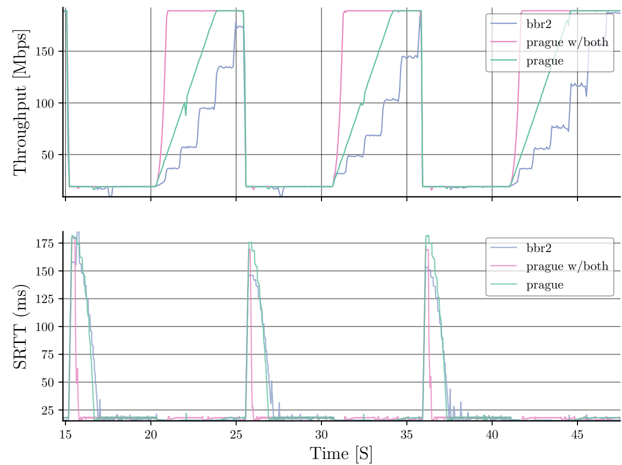

Figure 5: Single flow, each algorithm run separately. 15ms RTT and 200mbps capacity. Capacity periodically changed to 20mbps. AQM is step at 3ms. Acceleration mode recovers capacity rapidly without overshooting excessively.

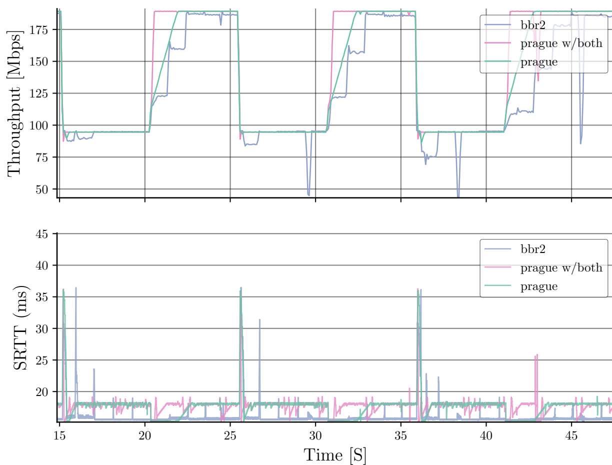

Figure 4: Single flow, each algorithm run separately. 15ms RTT and 200mbps capacity. Capacity periodically changed to 100mbps. AQM is step at 3ms. Acceleration mode recovers capacity rapidly without overshooting excessively.

5 FUTURE WORK

6 DISCUSSION

6.1 A constant delay or a proportion of RTT?

A flow has a base round-trip time that it cannot reduced. The utility that a flow has for a user running a time-sensitive application is usually s-shaped as a function of round-trip time. Using this one can say that as long as the change in queueing delay resulting from a more aggressive increase does not decrease the utility significant the increase is acceptable. There are multiple problems with such an argument. Looking at the utility from a single flow is insufficient because there might be multiple flows from the same user for different applications. The utility-function depends on the application using the TCP transfer. There is currently no API that allows a application to tell the TCP congestion controller how important latency is to that application or user. Such an API should also include a users willingness to sacrifice capacity seeking to give possible competing time-sensitive flows low latency.

It is more important for flows with low base round-trip time to preserve low latency than it is for flows with high low base round-trip time. Therefor it makes sense to make Q a proportion of the base round-trip time. However, we have to keep in mind that the increase in round-trip time is a increase in queuing delay. A increase in queuing delay that is added to the different base round-trip times of all the flows traversing that bottleneck. A single user might have a mix of latency-sensitive and greedy connections with different round-trip times. We wouldn't want a 100ms base round-trip time greedy connection impose a 25ms increase in queueing delay if we have a time-sensitive flow with a base-round trip time of 5ms.

6.2 Overload and acceleration mode interaction

The small overshoot caused by the acceleration mode can cause overload mode to be entered right away. This is not necessarily such a bad thing as long as the overload mode does not increase its reduction over subsequent overshoots by the acceleration mode.

There are two key design points of the algorithms that prevent oscillatory behaviour. The first is that the acceleration mode measure the number of increases, not the number of round-trip times, between congestion events for finding an appropriate threshold for mode change. The second is that the overload mode moves q_{max} towards the queueing delayed cause by the acceleration mode, making the threshold greater for each cycle of interaction.

6.3 Flow-rate fairness

If two flows enter acceleration mode with different sending rates the one with the highest sending rate will accelerate fastest in acceleration mode. This is not an issue because (1) there is no feasible way for each flow to know of the other flow, and (2) a flow with higher rate has more or better information. Competing flows will eventually converge in congestion avoidance, which will give flow rate fairness.

A similar argument can be made for different reduction rates for competing flows with different round-trip times. A flow with a small round-trip time should react as fast as it can because (1) it cannot know whether it is competing with other flows, and (2) has more or better information about congestion.

7 CONCLUSION

We have demonstrated that TCP Prague's and other scalable congestion control's acceleration to fully utilize a underutilized network can be greatly improved with little risk.

REFERENCES

A MACHINE CONFIGURATION

The testbed consist of four machines and we will call them clients, server and aqm.

Both the clients and the server use a custom 5.10.31 Linux kernel. The kernel is 5.10.31-3cc3851880a1-prague-37 and can be found at the following page https://github.com/L4STeam/linux/releases/tag/testing-build. The clients and the server use 18.04.04. The aqm runs Ubuntu 16.04.7 with a custom 5.5.0 Linux kernel which can be found at https://github.com/JoakimMisund/net-next/tree/series_l4s_20210421-1025-pc. Each machine has a 1 Gigabit Intel I350 Gigabit network card. The clients and the server has a 2-port version of the card, while the aqm has a 4-port version of the card. The driver is igb and the firmware version is 1.63.

The machines are connected in serial and through a switch. The switch makes it possible to have management traffic between the machines without interfering with the experimental traffic.

Delay and rate limiting is added by tc-netem and tc-htb respectively on the aqm machine. To avoid timer related issues that we have previously experienced using tc-htb and tc-netem we set the cpu scaling governor to performance and the highest C-state to 0 on every cpu on the aqm. We sat the burst and cburst of tc-htb to 3080 because we experienced persistent underutilization during some runs for different configurations when it was set to 1. Delay is applied on the forwarding path and the reverse path, i.e. a round-trip time of 10ms implies 5ms in forwarding path and 5ms is reverse path.

Table 1 and table 2 shows sysctl and ethtool configuration, respectively.

Variable name	Value
tcp_no_metrics_save	1
tcp_low_latency	1
tcp_autocorking	0
tcp_fastopen	0
tcp_fin_timeout	5
tcp_tw_reuse	1
tcp_ecn	1
tcp_ecn_fallback	0
tcp_max_syn_backlog	2048
somaxconn	2048
netdev_max_backlog	2000
rmem_max	8388608
rmem_default	8388608
wmem_max	8388608
wmem_default	8388608
ip_local_port_range	"20000 61000"
tcp_rmem	"8388608 8388608 8388608"
tcp_wmem	"8388608 8388608 8388608"
tcp_mem	"8388608 8388608 8388608"

Table 1: Sysctl variables and their values. Others are set to default values.

Configuration name	Value
rx-usecs	0
tx-usecs	0
gso	off
tso	off
gro	off
tx-gso-partial	off
sg	off
autoneg/tx/rx	off

Table 2: Interface configuration on all machines, set through ethtool.