Understanding TCP Incast Throughput Collapse in Datacenter Networks

WREN 2009

2009-08-21

Yanpei Chen
Rean Griffith
Junda Liu
Anthony D. Joseph
Randy H. Katz

This is the “Other Incast talk” or the “Response to the Incast talk” given on the last day of SIGCOMM 2009 by our colleagues at CMU.

It contains very recent findings beyond the writeup in the paper.

We strongly encourage questions, criticism, and feedback.

This is joint work with my co-authors at the Reliable, Adaptive, and Distributed Laboratory (RAD Lab) at UC Berkeley.

Outline – Incast is not solved

What is TCP incast throughput collapse and what has already been done?

What's a good methodology to study incast?

What are some instinctive first fixes and why are they not enough?

What are the root causes of incast?

How can we work towards a solution?

1

The bottomline of the story is that TCP incast is not solved. We will develop this story by answering several questions.

We begin by looking at what is incast, why do we care, and what has already been done.

We continue with a discussion on some subtle methodology issues.

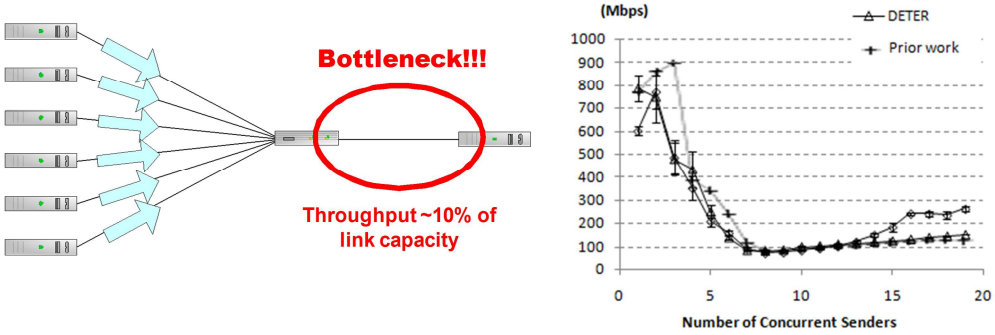
There are some instinctive first fixes. They work. But they are limited.

We will spend some time trying to discover the root causes of incast. It requires looking at the problem from several angles.

Lastly, we outline a promising path towards a solution.

Incast occurs in N-to-1 large data transfers

TCP pathology – application level "goodput" collapses far below link capacity



Affects key datacenter applications with synchronization boundaries

E.g. DFS, web search, MapReduce

2

Incast is a TCP pathology that occurs in N to 1 large data transfers.

It occurs most visibly on one-hop topologies with a single bottleneck.

The observed phenomenon is that the "goodput" seen by applications drops far below link capacity.

We care about this problem because it can affect key datacenter applications with synchronization boundaries, meaning that the application cannot proceed until it has received data from all senders.

Examples include distributed file systems, where block requests are satisfied only when all senders finished transmitted their own fragment of the block; MapReduce, where the shuffle step cannot complete until intermediate results from all nodes are fetched; or web search and similar query applications, where a query is not satisfied until responses from the distributed workers are assembled.

Incast is real, but can be masked by app inefficiencies

Observed somewhere, not observed everywhere

MapReduce: 3x performance improvement from better configuration

There is an experimental challenge to isolate incast

Better application makes the problem more visible

3

The problem is complex, because it can be masked by application level inefficiencies, leading to opinions that Incast is not “real”.

We encourage datacenter operators with different experiences to share their perspectives.

From our own experience in using MapReduce, we can get a three-fold performance improvement just by using better configuration parameters.

So the experimental challenge is to remove application level artifacts and isolate the observed bottleneck to the network.

We believe that as applications improve, incast would become visible in more and more applications.

Prior and concurrent work found first fixes

Phanishayee et al. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. FAST 2008

- First description of the problem – coined the term “incast”
- Popular TCP variants (Reno, New Reno, SACK) all suffer
- Non-TCP work-around possible but problematic

Vasudevan et al. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. SIGCOMM 2009

- **Reduce TCP RTO MIN & use high resolution timers**

First fixes limited to some environments

4

There is significant prior and concurrent work on Incast by a group at CMU. We’re in regular contact and exchanging results and ideas.

Their paper in FAST 2008 gave a first description of the problem, and coined the term “incast”.

The key findings there are that all popular TCP variants suffer from this problem. There are non-TCP workarounds, but they are problematic in real-life deployment scenarios.

Their SIGCOMM 2009 paper, presented yesterday, suggested that reducing the TCP retransmission timeout minimum is a first step, and that high resolution timers for TCP also helps.

Our results are in agreement with theirs. However, we also have several points of departure convincing us that the first-fixes are limited.

Outline – Incast is not solved

What is TCP incast throughput collapse and what has already been done?

What's a good methodology to study incast?

What are some instinctive first fixes and why are they not enough?

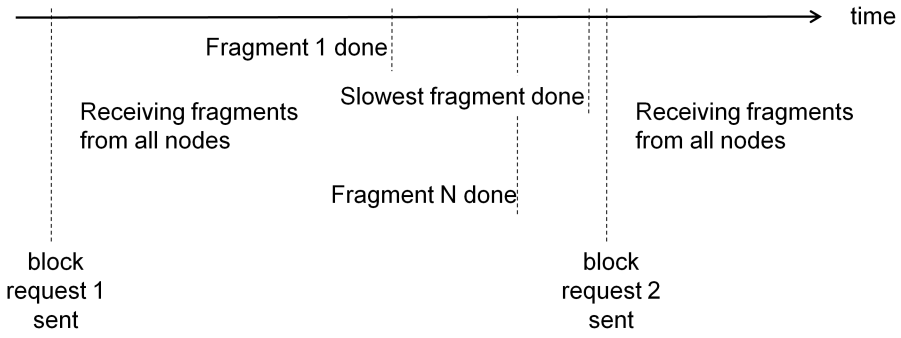
What are the root causes of incast?

How can we work towards a solution?

So, what is a good methodology to study incast?

Methodology – We need a simple workload

N-to-1 block transfer, each node sends a fragment of the block



5

We need a simple workload, to ensure that any performance degradation we see are not due to application inefficiencies.

We use a straight-forward N-to-1 block transfer workload. This is the same workload used in the CMU work, and it is motivated by data transfer behavior in file systems and MapReduce. What happens is this:

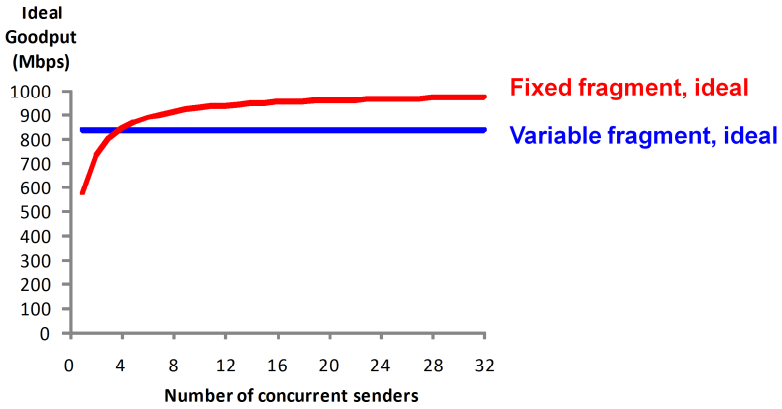
A block request is sent. Then the receiver starts receiving fragments from all senders. Some time later it finishes receiving different fragments. The synchronization boundary occurs because we send the second block request only after we received the last fragment.

There after the process repeats.

Methodology – Simple workload is complex …

Fixed fragment size as number of senders increase (CMU FAST 2008)

Variable fragment with fixed total data size across senders (CMU SIGCOMM 2009)



Differs only in ideal behavior – we used fixed fragment workload

6

There is some complexity even with this straight-forward workload. We can run the workload in two ways.

We can keep the fragment size fixed as the number of senders increase. This is how the workload was run in the FAST 2008 incast paper.

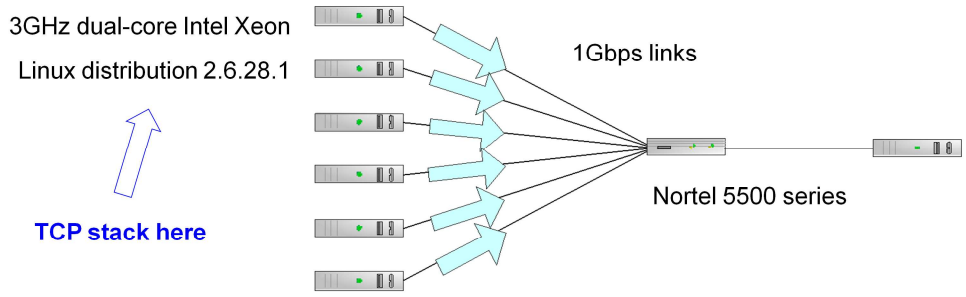
Alternatively, we can vary the fragment size such that the sum of fragments is fixed. This is how the workload was run in the SIGCOMM 2009 incast paper.

The two different workload flavors result in two different ideal behavior, but fortunately this is the only place that they differ.

We used the fixed fragment workload to ensure comparability with the results in the FAST 2008 incast paper.

Methodology – We need physical networks

We use real machines because default ns-2 models not detailed enough



Tools: TCP socket info, tcpdump + tcptrace

7

We also need to do measurements in physical networks.

Our instinct is to simulate the problem on event-based simulators like ns-2. It turns out that the default models in a simulator like ns-2 are not detailed enough for analyzing the dynamics of this problem. Later in the talk you will see several places where this inadequacy comes up. Thus we are compelled to do measurements physical networks.

We used Intel Xeons machines running a recent distribution of Linux, and we implement TCP modifications to the TCP stack in Linux.

The machines are connected by 1Gbps Ethernet through a Nortel 5500 series switch.

We did most of our analysis by looking at TCP socket state variables directly. We also used tcpdump and tcptrace to leverage some of their aggregation and graphing capabilities.

Outline – Incast is not solved

What is TCP incast throughput collapse and what has already been done?

What's a good methodology to study incast?

What are some instinctive first fixes and why are they not enough?

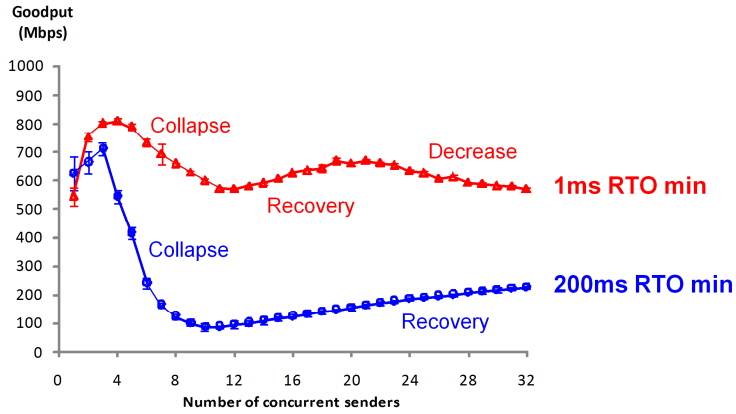
What are the root causes of incast?

How can we work towards a solution?

There are some first fixes to the problem. They work, but they are not enough.

Fixing RTT-RTO mismatch is a first step

Default TCP RTO min set to 200ms – optimized for WAN



Recovery and decrease not observed in concurrent work

8

Fixing the mismatch between RTO and the round-trip-time or RTT is indeed the first step, as demonstrated in the incast presentation yesterday.

Many OS implementations contain an TCP RTO minimum constant that acts as the lower bound and the initial value for the TCP RTO timer.

The default value of this constant is hundreds of milliseconds, optimized for the wide area network, but far below RTT in datacenters.

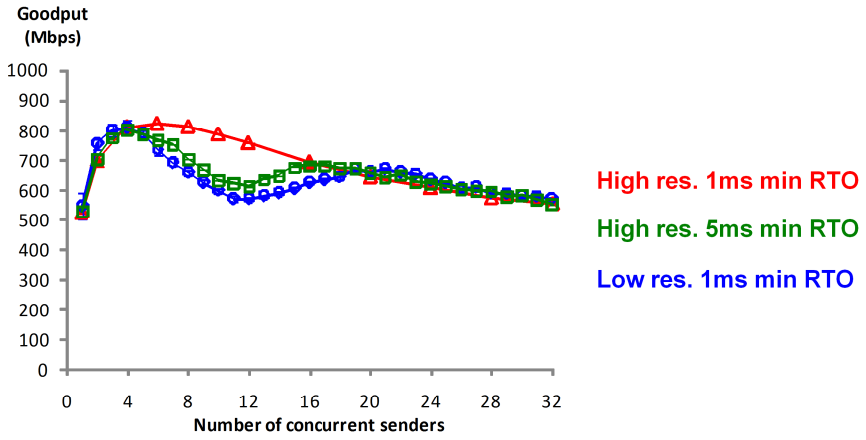
Reducing the default value gives us a huge improvement immediately.

More interesting, there are several regions in the graphs. As we increase the number of senders, we go through collapse, recovery, and sometimes encounter another decrease.

The recovery and decrease is not observed in concurrent work. Later you'll see why.

Smaller RTO needs high resolution timers

High res. timer 5ms RTO same as low res. timer 1ms RTO



Confirms the findings in concurrent work

9

When we have small RTO values, we need high resolution timers to make them effective.

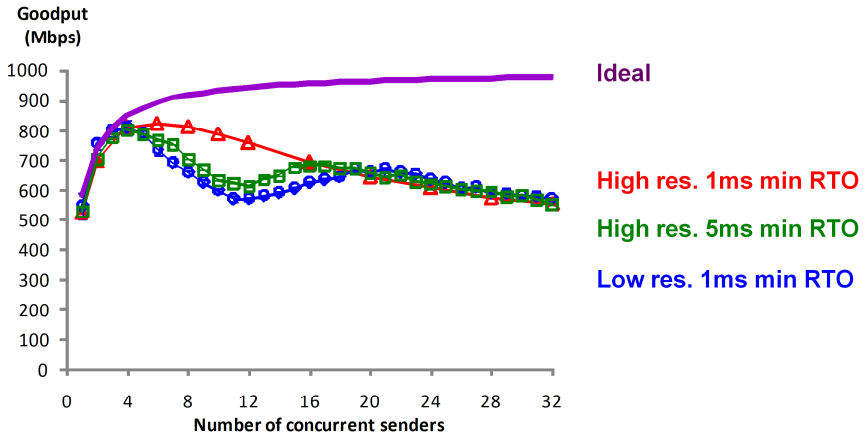
This graph shows that there is a difference between low and high resolution timers set to the same small value for the RTO min. If we increase the RTO min for the high resolution timer slightly, we get similar performance to the low resolution timer.

In other words, the default low resolution timer has granularity of approximately 5ms.

This confirms the findings in concurrent work, that high resolution timers are necessary.

The first fixes are not enough!

The best curve is still far from ideal



10

The reason these first fixes are not enough is because the line with the best goodput is still far from the ideal.

Also, beyond about 20 senders, the lines converge with a downward trend, meaning that the solution is not complete.

We'll explain next the difference in our results and the results you saw yesterday, and highlight the root causes of Incast.

Outline – Incast is not solved

What is TCP incast throughput collapse and what has already been done?

What's a good methodology to study incast?

What are some instinctive first fixes and why are they not enough?

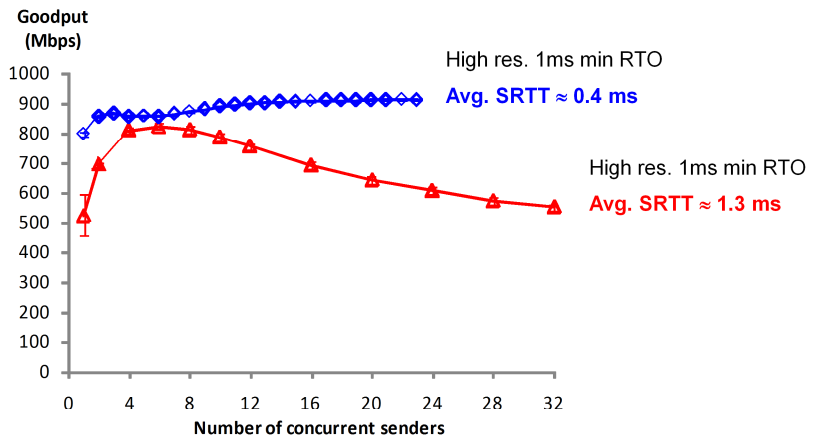
What are the root causes of incast?

How can we work towards a solution?

The root causes …

Different networks suffer to different degrees

Same protocol, bandwidth, nodes, switch vendor & model. Different results?!?



Turns out the switches have different hardware versions

11

First, an observation – different networks suffer the problem to different degrees.

We have two experimental test beds that are listed as identical – same OS, bandwidth, machines, switch vendor and even the same switch model. However, we get two very different results.

The red line is the result we saw before, on a network where TCP sees a relatively large SRTT. The results are different from what you saw yesterday. The blue line from a network where TCP sees a smaller SRTT. The results there are identical to the results from yesterday!!!

So different networks suffer to different degrees. The first fixes are sufficient only on some networks. Hence we believe the first fixes are limited.

It turns out that the switches have different hardware versions – they are sold as the same model, have the same firmware, but the different hardware versions resulted in the difference in round trip times.

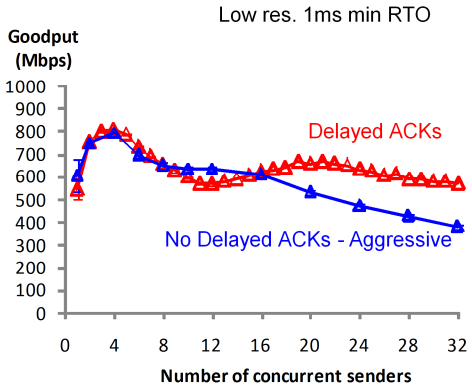
So the difference networks contribute to the difference between our results and the results from CMU.

ACK-clocked feedback has fundamental tradeoffs

Turning off delayed ACKs results in more aggressive behavior

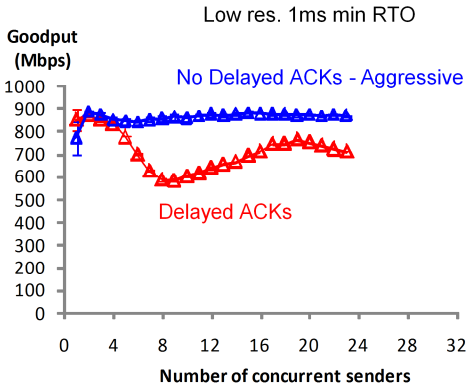
Large SRTT ≈ 1.3 ms

Aggressiveness harms



Small SRTT ≈ 0.4 ms

Aggressiveness helps



12

The different SRTT revealed a well-known fundamental tradeoff with ACK-clocked feedback mechanisms.

Delayed ACKs is a default feature in TCP where the receiver acknowledges every other packet instead of every packet. The original motivation was to reduce ACK traffic for applications like Telnet. Turning it off results in more aggressive behavior because we can act on feedback from every packet instead of every other packet.

In the graphs, the blue line represents aggressive behavior. It turns out that in a large SRTT network, aggressiveness harms, but in a small SRTT network, aggressiveness helps.

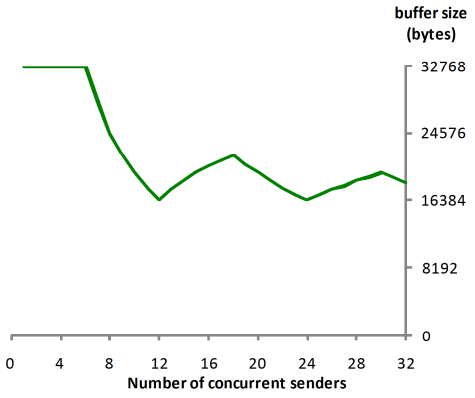
This insight is similar to the findings from XCP, TCP Fast, and similar work – the larger the bandwidth-delay product, the less helpful aggressive behavior would be.

Switch Buffer Management Affects Goodput

Complex buffer management strategy:

786432B total buffer shared between 48 ports

Each port allowed to use 1/6 of the buffer space for each group of 12 ports



13

Switch buffer management is another factor that affects goodput.

The switch we used has a pretty complex buffer management strategy.

One interpretation gives per-port average buffer size like so.

As we increase the number of concurrent senders, there is a harmonic behavior in the average buffer size.

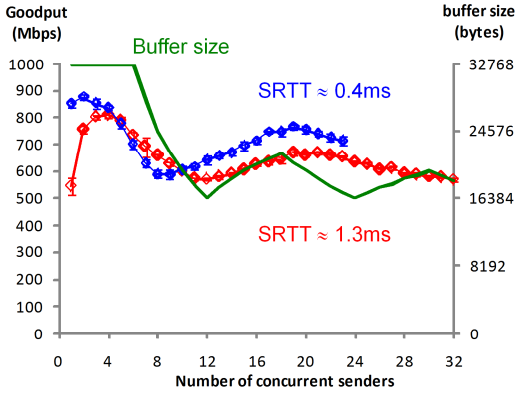
We know from the FAST 2008 incast paper that larger buffer sizes mitigate the problem.

Switch Buffer Management Affects Goodput

Complex buffer management strategy:

786432B total buffer shared between 48 ports

Each port allowed to use 1/6 of the buffer space for each group of 12 ports



Low res. 1ms min RTO

ns-2 simulations fail to capture this behavior

13

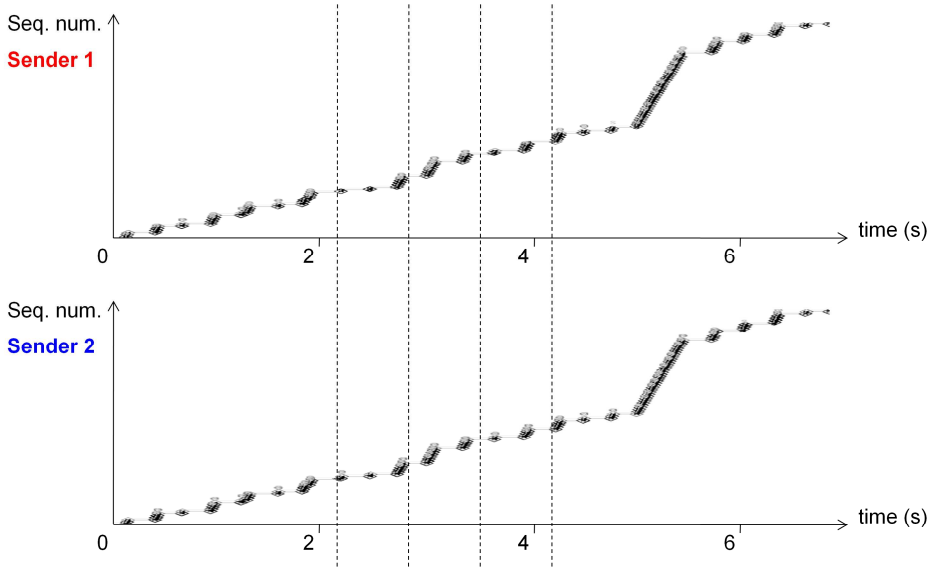
If we superimpose the goodput results on the same graph, we do indeed see some overlap. The overlap is not exact because there are many simultaneous effects, and their superposition is not linear.

To the best of our knowledge, ns-2 default FIFO queuing models don't capture this kind of behavior. We believe this is one reason that default ns-2 simulations don't reflect the experimental results.

So, different switch buffer management also contributes to the difference between our results and the results from CMU.

Senders are synchronized

TCP sequence number dynamics – low res. 200ms RTO:



14

Another observation is that the senders are synchronized.

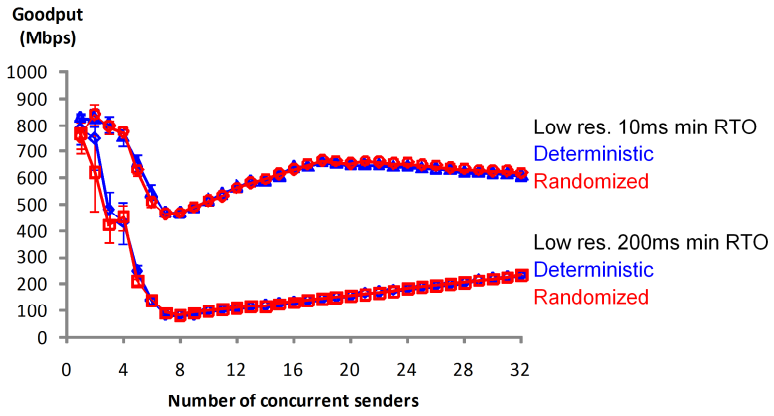
You see here the TCP sequence number trace for two different senders plotted on the same time axes.

The vertical dotted guide lines show that the key features line up almost exactly.

We got this result from using the default TCP. So for default TCP, the senders are synchronized.

Randomize RTO doesn't help

Intuition – randomize RTO using bit mask with TCP sequence number



This idea works in ns-2 but not on a physical network!

15

The reflex response is to add some randomization.

We bit-masked the initial RTO with the TCP sequence number. This operation gives us a random RTO because most TCP stacks implement random sequence numbers to prevent sequence number guessing attacks.

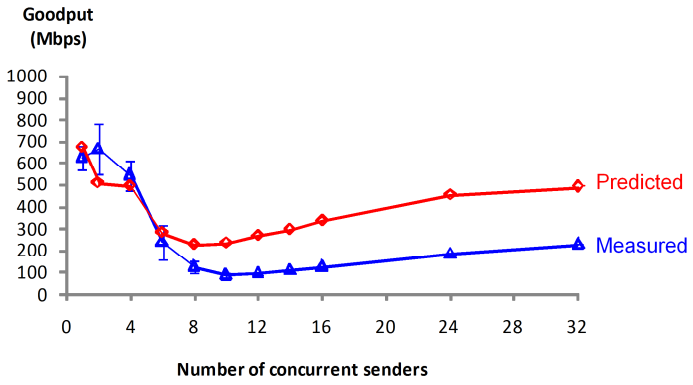
It turns out that the randomized RTO and deterministic RTO give us identical performance.

This is yet another idea that works fine in ns-2, but not on a physical network. We believe the reason here is that the TCP stack in ns-2 is the "ideal", and departs from the actual implementation in the OS kernel.

Analytical model is difficult – overlapping effects

When one effect dominates, an analytical model is effective

E.g. For large RTO values, the effect of RTOs dominate



Model fails when many effects have non-linear superposition

16

Having explored the problem space, we attempted to put together all the overlapping effects into a quantitative model. After much effort, we realized that doing so in a general fashion is very hard.

When one effect dominates, such as when the effect of RTOs dominate, we can get an effective model.

A model based solely on the effect of RTOs gives us the same shape of the curve as the measured data. We have a systematic over-prediction because we don't account for secondary effects such as inter-packet wait time.

The model fails when no single effect dominates, because different effects don't have a linear overlap. We know this because we constructed models based on linear superposition, and the predicted and measured behavior depart.

The difficulty in our modeling efforts highlighted to us the difficulty in implementing a general solution.

Outline – Incast is not solved

What is TCP incast throughput collapse and what has already been done?

What's a good methodology to study incast?

What are some instinctive first fixes and why are they not enough?

What are the root causes of incast?

How can we work towards a solution?

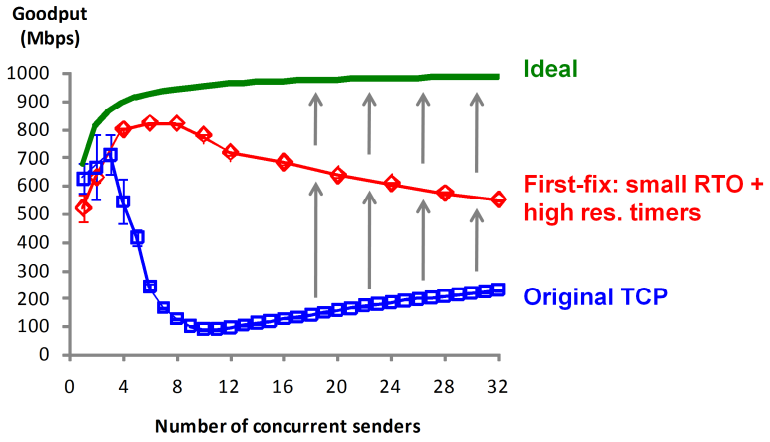
Towards a solution …

A solution is hard but possible

Better TCP needs to deal with different network, switch, background traffic

We can limit parameter space by looking at correlations in TCP state variables

Working on adaptive CWND management – promising preliminary results



17

The solution is hard, but we believe it is possible. We believe we already have a good understanding of the problem space.

A general solution to incast needs to survive different network environments, switch buffer management strategies, and background traffic conditions, which we have not even investigated yet.

We can narrow down the parameter space by looking at correlations in TCP state variables. Out of the many TCP state variables, only a small subset has strong correlations to each other.

We're working on an adaptive congestion window management scheme, with some promising preliminary results. We can talk more about it if you'd like.

The first fixes are already a big improvement on the default TCP. We believe that we can push it further towards the ideal.

Take-Away Message – Incast is not solved

TCP incast is a real problem that affects key datacenter applications

Contributing factors include

- Protocol misconfiguration – fixed
- Feedback response & network characteristics
- Switch buffer management

General and comprehensive solution coming soon!!!

18

The takeaway message is that Incast is not solved.

It is a real problem affecting key applications.

We've already fixed protocol misconfiguration, so now we need to fix everything else.

A general and comprehensive solution to incast remains elusive, but we're confident that we'll have it soon.

Take-Away Message – Incast is not solved

TCP incast is a real problem that affects key datacenter applications

Contributing factors include

- Protocol misconfiguration – fixed
- Feedback response & network characteristics
- Switch buffer management

General and comprehensive solution coming soon!!!

Thank You!

18

Thank you!

Emails

{ychen2, rean, liujd, randy, adj} at eecs dot berkeley dot edu

Feedback welcome.