Understanding TCP Incast and Its Implications for Big Data Workloads

Yanpei Chen, Rean Griffith*, David Zats, Anthony D. Joseph, Randy Katz
University of California, Berkeley, *VMware
(ychen2, dzats, adj, randy)@eecs.berkeley.edu, *rean@vmware.com

1. Introduction

TCP incast is a recently identified network transport pathology that affects many-to-one communication patterns in datacenters. It is caused by a complex interplay between datacenter applications, the underlying switches, network topology, and TCP, which was originally designed for wide area networks. Incast increases the queuing delay of flows, and decreases application level throughput to far below the link bandwidth. The problem especially affects computing paradigms in which distributed processing cannot progress until all parallel threads in a stage complete. Examples of such paradigms include distributed file systems, web search, advertisement selection, and other applications with partition or aggregation semantics [5, 18, 25].

There have been many proposed solutions for incast. Representative approaches include modifying TCP parameters [18, 27] or its congestion control algorithm [28], optimizing application level data transfer patterns [21, 25], switch level modifications such as larger buffers [25] or explicit congestion notification (ECN) capabilities [5], and link layer mechanisms such as Ethernet congestion control [3, 6]. Application level solutions are the least intrusive to deploy, but require modifying each and every datacenter application. Switch and link level solutions require modifying the underlying datacenter infrastructure, and are likely to be logistically feasible only during hardware upgrades.

Unfortunately, despite these solutions, we still have no quantitatively accurate and empirically validated model to predict incast behavior. Similarly, despite many studies demonstrating incast for microbenchmarks, we still do not understand how incast impacts application level performance subject to real life complexities in configuration, scheduling, data size, and other environmental and workload properties. These concerns create justified skepticism on whether we truly understand incast at all, whether it is even an important problem for a wide class of workloads, and whether it is worth the effort to deploy various incast solutions in front-line, business-critical datacenters.

We seek to understand how incast impacts the emerging class of big data workloads. Canonical big data workloads help solve needle-in-a-haystack type problems and extract actionable insights from large scale, potentially complex and unformatted data. We do not propose in this article yet another solution for incast. Rather, we focus on developing a deep understanding of one existing solution: reducing the minimum length of TCP retransmission time out (RTO) from 200ms to 1ms [18, 27]. We believe TCP incast is fundamentally a transport layer problem, thus a solution at this level is best.

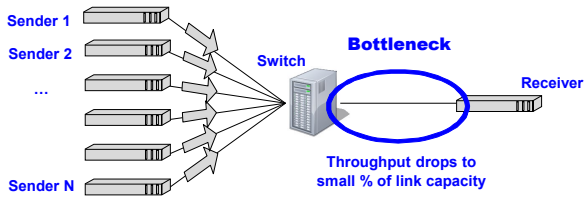

Figure 1. Simple setup to observe incast. The receiver requests k blocks of data from a set of N storage servers. Each block is striped across N storage servers. For each block request received, a server responds with a fixed amount of data. Clients do not request block $k+1$ until all the fragments of block k have been received.

The first half of this article develops and validates a quantitative model that accurately predicts the onset of incast and TCP behavior both before and after. The second half of this article investigates how incast affects the Apache Hadoop implementation of MapReduce, an important example of a big data application. We close the article by reflecting on some technology and data analysis trends surrounding big data, speculate on how these trends interact with incast, and make recommendations for datacenter operators.

2. Towards an Analytical Model

We use a simple network topology and workload to develop an analytical model for incast, shown in Figure 1. This is the same setup as that used in prior work [18, 25, 27]. We choose this topology and workload to make the analysis tractable.

The workload is as follows. The receiver requests k blocks of data from a set of N storage servers — in our experiments $k = 100$ and N varies from 1 to 48. Each block is striped across N storage servers. For each block request received, a server responds with a fixed amount of data.

Clients do not request block $k+1$ until all the fragments of block k have been received — this leads to a *synchronized read pattern* of data requests. We re-use the storage server and client code in [18, 25, 27]. The performance metric for these experiments is *application-level goodput*, i.e., the total bytes received from all senders divided by the finishing time of the *last* sender.

We conduct our experiments on the DETER Lab testbed [12] where we have full control over the non-virtualized node OS, as well as the network topology and speed. We used 3GHz dual-core Intel Xeon machines with 1Gbps network links. The nodes run standard Linux 2.6.28.1. This was the most recent mainline Linux distribution in late 2009, when we obtained our prior results [18]. We present results using both a relatively shallow-buffered Nortel 5500 switch (4KB per port), and a more deeply buffered HP Procurve 5412 switch (64KB per port).

2.1 Flow rate models

The simplest model for incast is based on two competing behaviors as we increase N, the number of concurrent senders. The first behavior occurs before the onset of incast, and reflects the intuition that goodput is the block size divided by the transfer time. Ideal transfer time is just the sum of a round trip time (RTT) and the ideal send time. Equation 1 captures this idea.

$$\begin{aligned} Goodput_{beforeIncast} &= idealGooputPerSender \times N \\ &= \frac{blockSize}{idealTransferTime} \times N \\ &= \frac{blockSize}{RTT + \frac{blockSize}{perSenderBandwidth}} \times N \\ &= \frac{blockSize}{RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \end{aligned} \tag{1}$$

Incast occurs when there are some $N > 1$ concurrent senders, and the goodput drops significantly. After the onset of incast, TCP retransmission time out (RTO) represents the dominant effect. Transfer time becomes RTT + RTO + ideal send time, as captured in Equation 2. The goodput collapse represents a transition between the two behavior modes.

$$\begin{aligned} Goodput_{incast} &= goodputPerSender \times N \\ &= \frac{blockSize}{idealTransferTime + RTO} \times N \\ &= \frac{blockSize}{RTO + RTT + \frac{blockSize}{perSenderBandwidth}} \times N \\ &= \frac{blockSize}{RTO + RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \end{aligned} \tag{2}$$

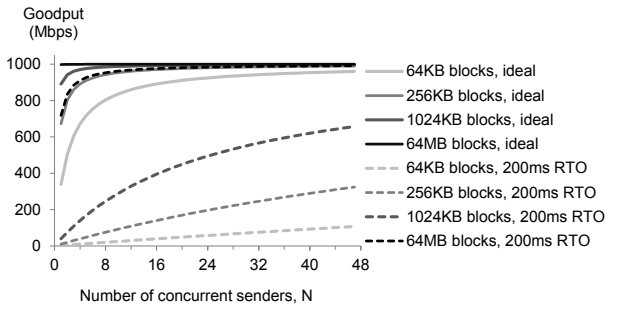

Figure 2. Flow rate model for incast. Showing ideal behavior (solid lines, Equation 1) and incast behavior caused by RTOs (dotted lines, Equation 2). We substitute $blockSize$ = 64KB, 256KB, 1024KB, and 64MB, as well as RTT = 1ms, and RTO = 200ms. The incast goodput collapse comes from the transition between the two TCP operating modes.

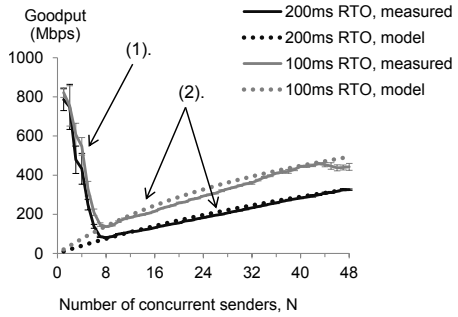

Figure 3. Empirical verification of flow rate incast model. Uses our previously presented data in [18]. The $blockSize$ is 256KB, RTO is set to 100ms and 200ms, and the model uses RTT = 1ms. Error bars represent 95% confidence interval around the average of 5 repeated measurements. The switch is a Nortel 5500 (4KB per port). Showing (1). Incast goodput collapse begins at N = 2 senders, and (2). Behavior after goodput collapse verifies Equation 2.

Figure 2 gives some intuition with regard to Equations 1 and 2. We substitute $blockSize$ = 64KB, 256KB, 1024KB, and 64MB, as well as RTT = 1ms, and RTO = 200ms. Before the onset of incast (Equation 1), the goodput increases as N increases, though with diminishing rate, asymptotically approaching the full link bandwidth. The curves move vertically upwards as block size increases. This reflects the fact that larger blocks result in a larger fraction of the ideal transfer time spent transmitting data, versus waiting for an RTT to acknowledge that the transmission completed. After incast occurs (Equation 2), RTO dominates the transfer time for small block sizes. Again, larger blocks lead to RTO forming a smaller ratio versus ideal transmission time. The curves move vertically upwards as block size increases.

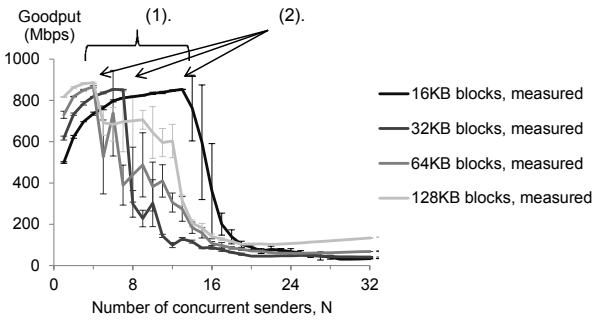

Figure 4. Empirical verification of flow rate TCP model before onset of incast. Measurements done on HP Procurve 5412 switches (64KB per port). RTO is 200ms. Error bars represent 95% confidence interval around the average of 5 repeated measurements. Showing (1). Behavior before goodput collapse verifies Equation 1, and (2). Onset of incast goodput collapse predicted by switch buffer overflow during slow start (Equation 3).

2.2 Empirical verification

This model matches well with our empirical measurements. Figure 3 superpositions the model on our previously presented data in [18]. There, we fix $blockSize$ at 256KB and set RTO to 100ms and 200ms. The switch is a Nortel 5500 (4KB per port). For simplicity, we use RTT = 1ms for the model. Goodput collapse begins at N = 2, and we observe behavior for Equation 2 only. The empirical measurements (solid lines) match the model (dotted-lines) almost exactly.

We use a more deeply buffered switch to verify Equation 1. As we discuss later, the switch buffer size determines the onset of incast. Figure 4 shows the behavior using the HP Procurve 5412 switch (64KB per port). Behavior before goodput collapse qualitatively verifies Equation 1 — the goodput increases as N increases, though with diminishing rate; the curves move vertically upwards as block size increases. We can see this graphically by comparing the curves in Figure 4 before the goodput collapse to the corresponding curves in Figure 2.

Takeaway: Flow rate model captures behavior before onset of incast. TCP RTO dominates behavior after onset of incast.

2.3 Predicting the onset of incast

Figure 4 also shows that goodput collapse occurs at different N for different block sizes. We can predict the location of the onset of goodput collapse by detailed modeling of TCP slow start and buffer occupancy. Table 1 shows the slow start congestion window sizes versus each packet round trip. For 16KB blocks, 12 concurrent senders of the largest congestion window of 5864 bytes would require 70368 bytes of buffer, larger than the available buffer of 64KB per port. Goodput collapse begins after N = 13 concurrent senders. The discrepancy of 1 comes from the fact that there is additional "buffer" on the network beyond the packet buffer on the switch, e.g., packets in flight, buffer at the sender machines, etc. According to this logic, goodput collapse should take place according to Equation 3. The equation accurately predicts that for Figure 4, the goodput collapse for 16KB, 32KB, and 64KB blocks begin at 23, 7, and 4 concurrent senders, and for Figure 3, the goodput collapse is well underway at 2 concurrent senders.

Round trip #	16KB blocks	32KB blocks	64KB blocks	128KB blocks
1	1,448	1,448	1,448	1,448
2	2,896	2,896	2,896	2,896
3	5,792	5,792	5,792	5,792
4	5,864	11,584	11,584	11,584
5		10,280	23,168	23,168
6			19,112	46,336
7				36,776

Table 1. TCP slow start congestion window size in bytes versus number of round trips. Showing the behavior for $blockSize$ = 16KB, 32KB, 64KB, 128KB. We verified using `sysctl` that Linux begins at 2× base MSS, which is 1448 bytes.

$$N_{initialGoodputCollapse} = \left\lceil \frac{perSenderBuffer}{largestSlowStartCwnd} \right\rceil + 1 \qquad (3)$$

Takeaway: For small flows, the switch buffer space determines the onset of incast.

2.4 Second order effects

Figure 4 also suggests the presence of second order effects not explained by Equations 1 to 3. Equation 3 predicts that goodput collapse for 128KB blocks should begin at N = 2 concurrent senders, while the empirically observed goodput collapse begins at N = 4 concurrent senders. It turns out that block sizes of 128KB represent a transition point from RTO-during-slow-start to more complex modes of behavior.

We repeat the experiment for $blockSize$ = 128KB, 256KB, 512KB, and 1024KB. Figure 5 shows the results, which includes several interesting effects.

First, for $blockSize$ = 512KB and 1024KB, the goodput immediately after the onset of incast is given by Equation 4. It differs from Equation 2 by the multiplier α for the RTO in the denominator. This α is an empirical constant, and represents a behavior that we call partial RTO. What happens is as follows. When RTO takes place, TCP SACK (turned on by default in Linux) allows transmission of further data, until the congestion window can no longer advance due to the lost packet. Hence, the link is idle for a duration of less than the full RTO value. Hence we call this effect partial RTO. For $blockSize$ = 1024KB, α is 0.6, and for $blockSize$ = 512KB, α is 0.8.

$$Goodput_{incast} = \frac{blockSize}{\alpha \times RTO + RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \qquad (4)$$

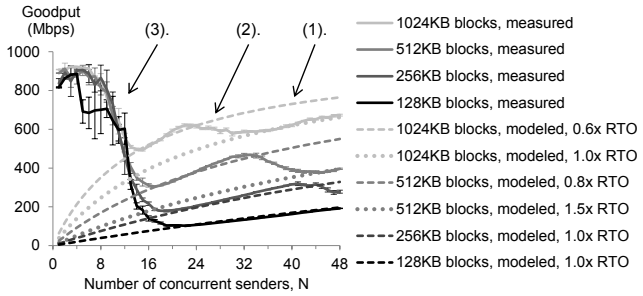

Figure 5. 2nd order effects other than RTO during slow start. Measurements done on HP Procurve 5412 switches (64KB per port). RTO is 200ms. Error bars represent 95% confidence interval around the average of 5 repeated measurements. Showing (1). Partial RTOs more accurately modeling incast behavior for large blocks, (2). Transition between single and multiple partial RTOs, and (3). Triple duplicate ACKs causing more gradual, $blockSize$ independent onset of incast.

Second, beyond a certain number of concurrent senders, α transitions to something that approximately doubles its initial value (0.6 to 1.0 for $blockSize$ = 1024KB, 0.8 to 1.5 for $blockSize$ = 512KB). This simply represents that two partial RTOs have occurred.

Third, the goodput collapse for $blockSize$ = 256KB, 512KB, and 1024KB is more gradual compared with the cliff-like behavior in Figure 4. Further, this gradual goodput collapse has the same slope across different $blockSize$. Two factors explain this behavior. First, flows with $blockSize \geq$ 128KB have a lot more data to send even after the buffer space is filled with packets sent during slow start (Equation 3 and Table 1). Second, even when the switch drops packets, TCP can sometimes recover. Empirical evidence of this fact exists in Figure 4. There, for $blockSize$ = 16KB and N = 13 to 16 concurrent senders, at least one of five repeated measurements manages to get goodput close to 90% of link capacity. Goodput collapse happens for other runs because the packets are dropped in a way that a connection with little additional data to send would observe only a single or double duplicate ACK, and go into RTO soon after. Larger blocks suffer less from this problem because the ongoing data transfers triggers triple duplicate ACK with higher probability. Thus, the connection retransmits, enters congestion avoidance, and avoids RTO. Hence the gradual goodput collapse.

We should point out that SACK semantics are independent of duplicate ACKs, since SACK is layered on top of existing cumulative ACK semantics [23].

Takeaway: Second order effects include partial RTO due to SACK, multiple partial RTOs, and triple duplicate ACKs causing more gradual onset of incast.

2.5 Good enough model

Unfortunately, some parts of the model remain qualitative. We admit that the full interaction between triple duplicate ACKs, slow start, and available buffer space requires elaborate treatment far beyond the flow rate and buffer occupancy analysis presented here.

That said, the models here represent the first time we quantitatively explain major features of the incast goodput collapse. Comparable results in related work [25, 28] can be explained by our models also. The analysis allows us to reason about the significance of incast for future big data workloads later in the article.

3. Incast in Hadoop MapReduce

Hadoop represents an interesting case study of how incast affects application level behavior. Hadoop is an open source implementation of MapReduce, a distributed computation paradigm that played a key part in popularizing the phrase "big data". Network traffic in Hadoop consists of small flows carrying control packets for various cluster coordination protocols, and larger flows carrying the actual data being processed. Incast potentially affects Hadoop in complex ways. Further, Hadoop may well mask incast behavior, because network forms only a part of the overall computation and data flow. Our goal for this section is to answer whether incast affects Hadoop, by how much, and under what circumstances.

We perform two sets of experiments. First, we run stand-alone, artificial Hadoop jobs to find out how much incast impacts each component of the MapReduce data flow. Second, we replay a scaled-down, real life production workload using previously published tools [17] and cluster traces from Facebook, a leading Hadoop user to understand the extent to which incast affects whole workloads. These experiments take place on the same DETER machines as those in the previous section. We use only the large buffer Procurve switch for these experiments.

3.1 Stand-alone jobs

Table 2 lists the Hadoop cluster settings we considered. The actual stand-alone Hadoop jobs are `hdfsWrite`, `hdfsRead`, `shuffle`, and `sort`. The first three jobs stress one part of the Hadoop IO pipeline at a time. Sort represents a job with 1-1-1 ratio between read, shuffled, and written data. We implement these jobs by modifying the `randomwriter` and `randomtextwriter` examples that are pre-packaged with recent Hadoop distributions. We set the jobs to write, read, shuffle, or sort 20GB of terasort format data on 20 machines.

3.1.1 Experiment setup

The TCP versions are the same as before – standard Linux 2.6.28.1, and modified Linux 2.6.28.1 with `tcp_rto_min` set to 1ms. We consider Hadoop versions 0.18.2 and 0.20.2. Hadoop 0.18.2 is considered a legacy, basic, but still rel-

Parameter	Values
Hadoop jobs	hdfsWrite, hdfsRead, shuffle, sort
TCP version	Linux-2.6.28.1, 1ms-min-RTO
Hadoop version	0.18.2, 0.20.2
Switch model	HP Procurve 5412
Number of machines	20 workers and 1 master
fs.inmemory.size.mb	75, 200
io.file.buffer.size	4096, 131072
io.sort.mb	100, 200
io.sort.factor	10, 100
dfs.block.size	67108864, 536870912
dfs.replication	3, 1
mapred.reduce.parallel.copies	5, 20
mapred.child.java.opts	-Xmx200m, -Xmx512M

Table 2. Hadoop parameter values for experiments with stand-alone jobs.

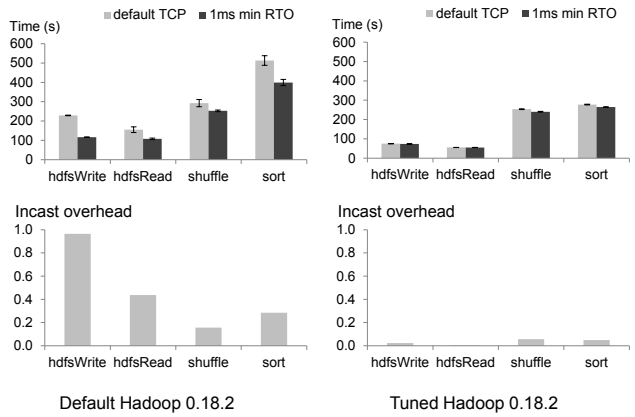

Figure 6. Hadoop stand alone job completion times. HP Procurve 5412 switches. Showing job completion times (top) and overhead introduced by incast (bottom) for default Hadoop-0.18.2 (left), and tuned Hadoop-0.18.2 (right). The error bars show 95% confidence intervals from 20 repeated measurements. The tuned Hadoop-0.18.2 leads to considerably lower job completion times. The confidence intervals are not overlapping for both settings. However, the default Hadoop has higher incast overhead.

atively stable and mature distribution. Hadoop 0.20.2 is a more fully featured distribution that introduces some performance overhead for small jobs [17]. Subsequent Hadoop improvements have appeared on several disjoint branches that are currently being merged, and 0.20.2 represents the last time there was a single mainline Hadoop distribution [30].

The rest of the parameters are detailed Hadoop configuration settings. Tuning these parameters can considerably improve performance, but requires specialist knowledge about the interaction between Hadoop and the cluster environment. The first value for each configuration parameter in Table 2 represents the default setting. The remaining values are tuned values, drawn from a combination of Hadoop sort benchmarking [1], suggestions from enterprise Hadoop vendors [4], and our own experiences. One configuration worth further explaining is `dfs.replication`. It controls the degree of data replication in HDFS. The default setting is three-fold data replication to achieve fault tolerance. For use cases constrained by storage capacity, the preferred method is to use HDFS RAID [14], which achieves fault tolerance with 1.4× overhead, much closer to the ideal one-fold replication.

3.1.2 Results

Figure 6 shows the results for Hadoop 0.18.2. We consider two performance metrics — job completion time, and incast overhead. We define incast overhead according to Equation 5, i.e., the difference between job completion time under default and `1ms-min-RTO` TCP, normalized by the job completion time for `1ms-min-RTO` TCP. The default Hadoop has very high incast overhead, while for tuned Hadoop, the incast overhead is barely visible. However, the tuned Hadoop-0.18.2 setting leads to considerably lower job completion times.

$$\begin{aligned} t &= jobCompletionTime \\ IncastOverhead &= \frac{t_{defaultTCP} - t_{1ms-min-RTO}}{t_{1ms-min-RTO}} \end{aligned} \quad (5)$$

The results illustrate a subtle form of Amdalh's Law, which explains overall improvement to a system when only a part of the system is being improved. Here, the amount of incast overhead depends on how much network data transfers contribute to the overall job completion time. The default Hadoop configurations result in network transfers contributing to a large fraction of the overall job completion time. Thus, incast overhead is clearly visible. Conversely, for tuned Hadoop, overall job completion time is already low. Incast overhead is barely visible because the network transfer time is low.

We repeat these measurements on Hadoop 0.20.2. Compared with Hadoop 0.18.2, the more recent version of Hadoop sees a performance improvement for the default configuration. For the optimized configuration, Hadoop 0.20.2 sees performance overhead of around 10 seconds for all four job types. This result is in line with our prior comparisons between Hadoop versions 0.18.2 and 0.20.2 [17]. Unfortunately, 10 seconds is also the performance improvement for using TCP with 1ms-min-RTO. Hence, the performance overhead in Hadoop 0.20.2 masks the benefits of addressing incast.

Takeaway: Incast does affect Hadoop. The performance impact depends on cluster configurations, as well as data and compute patterns in the workload.

3.2 Real life production workloads

The results in the above subsection indicate that to find out how much incast *really* affects Hadoop, we must compare the default and 1ms-min-RTO TCP while replaying real life production workloads.

Previously, such evaluation capabilities are exclusive to enterprises that run large scale production clusters. Recent years have witnessed a slow but steady growth of public

Figure 7. Per job input, shuffle, and output size for each workload. `FB-*` workloads come from a six-months cluster trace in 2009 and a 45-days trace in 2010. `CC-*` workloads come from traces of up to 2 months long at various customers of Cloudera, which is a vendor of enterprise Hadoop.

knowledge about front line production workloads [9, 10, 15, 17, 29], as well as emerging tools to replay such workloads in the absence of production data, code, and hardware [16, 17].

3.2.1 Workload analysis

We obtained seven production Hadoop workload traces from five companies in social networking, e-commerce, telecommunications, and retail. Among these companies, only Facebook has so far allowed us to release their name and synthetic versions of their workload. We do have permission to share some summary statistics. The full analysis is under publication review.

Several observations are especially relevant to incast. Consider Figure 7, which shows the distribution of per job input, shuffle, and output data for all workloads. First, all workloads are dominated by jobs that involve data sizes of less than 1GB. For jobs so small, scheduling and coordination overhead dominate job completion time. Therefore, incast will make a difference only if the workload intensity is high enough that Hadoop control packets alone would overwhelm the network. Second, all workloads do contain jobs at the 10s TB or even 100s TB scale. This compels the operators to use Hadoop 0.20.2. This version of Hadoop is the first to incorporate the Hadoop fair scheduler [29]. Without it the small jobs arriving behind very large jobs would see FIFO head of queue blocking, and suffer wait times of hours or even days. This feature is so critical that cluster operators use it despite the performance overhead for small jobs. Hence, it is likely that in Hadoop 0.20.2, incast will be masked by the performance overhead.

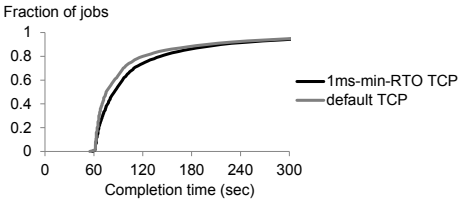

Figure 8. Distribution of job completion times for the `FB-2009` workload. The distribution for 1ms-min-RTO is 10-20 seconds right shifted compared with the distribution for default TCP.

3.2.2 Workload replay

We replay a day-long Facebook 2009 workload on the default and 1ms-min-RTO versions of TCP. We synthesize this workload using the method in [17]. It captures in a relatively short synthetic workload the representative job submission and computation patterns for the entire six-month trace.

Our measurements confirm the hypothesis earlier. Figure 8 shows the distribution of job completion times. We see that the distribution for 1ms-min-RTO is 10-20 seconds right shifted compared with the distribution for default TCP. This is in line with the 10-20 seconds overhead we saw in the workload-level measurements in [17], as well as the standalone job measurements earlier in the article. The benefits of addressing incast are completely masked by overhead from other parts of the system.

Figure 9 offers another perspective on workload level behavior. The graphs show two sequences of 100 jobs, ordered by submission time, i.e., we take snapshots of two continuous sequences of 100 jobs out of the total 6000+ jobs in a day. These graphs indicate the behavior complexity once we look at the entire workload of thousands of jobs and diverse interactions between concurrently running jobs. The 10-20 seconds performance difference on small jobs becomes insignificant noise in the baseline. The few large jobs take significantly longer than the small jobs, and stand out visibly from the baseline. For these jobs, there are no clear patterns to the performance of 1ms-min-RTO versus standard TCP.

The Hadoop community is aware of the performance overheads in Hadoop 0.20.2 for small jobs. Subsequent versions partially address these concerns [22]. It would be worthwhile to repeat these experiments once the various active Hadoop code branches merge back into the next mainline Hadoop [30].

Takeaway: Small jobs dominate several production Hadoop workloads. Non-network overhead in present Hadoop versions mask incast behavior for these jobs.

4. Incast for Future Big Data Workloads

Hadoop is an example of the rising class of big data computing paradigms, which almost always involve some amount of network communications. To understand how incast affects future big data workloads, one needs to appreciate the tech-

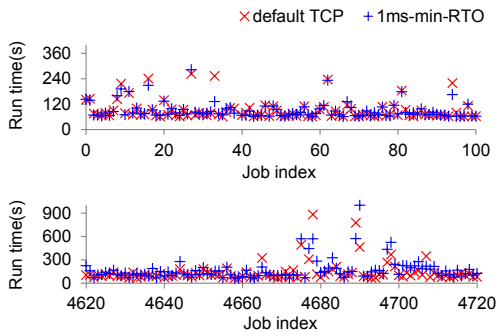

Figure 9. Sequences of job completion times. Showing two continuous job sequences of 100 jobs. The few large jobs have long completion times, and stand out from the baseline of continuous stream of small jobs.

nology trends that drive the rising prominence of big data, the computational demands that result, the countless design and mis-design opportunities, as well as the root causes of incast.

We believe that the top technology trends driving the prominence of big data include (1). Increasingly easy and economical access to large scale storage and computation infrastructure [7, 11], (2). Ubiquitous ability to generate, collect, and archive data about both technology systems and the physical world [19], and (3). Growing desire and statistical literacy across many industries to understand and derive value from large datasets [2, 13, 20, 24].

Several data analysis trends emerge, confirmed by the cluster operators who provided the traces in Figure 7. (1). There is increasing desire to do interactive data analysis, as well as streaming analysis. The goal is to have humans with non-specialist skills explore diverse and evolving data sources, and once they discover a way to extract actionable insights, such insights should be updated based on incoming data in a timely and continuous fashion. (2). Bringing such data analytic capability to non-specialists requires high-level computation frameworks built on top of common platforms such as MapReduce. Examples of such frameworks in the Hadoop MapReduce ecosystem include HBase, Hive, Pig, Sqoop, Oozie, and others. (3). Data sizes grow faster than the size per unit cost of storage and computation infrastructure. Hence, efficiently using storage and computational capacity are major concerns.

Incast plays into these trends as follows. The desire for interactive and streaming analysis requires highly responsive systems. The data size required for these computations are small compared with those required for computations on historical data. We know that when incast occurs, the RTO penalty is especially severe for small flows. Applications would be potentially forced to either delay the analysis response, or give answers based on partial data. Thus, incast could emerge as a barrier for high quality interactive and streaming analysis.

The desire to have non-specialists use big data systems suggests that functionality and usability should be the top design priorities. Incast affects performance, which can be interpreted as a kind of usability. It becomes a priority only after we have a functional system. Also, as our Hadoop experiments demonstrate, performance tuning for multi-layered software stacks would need to confront multiple layers of complexity and overhead.

The need for storage capacity efficiency entails storing compressed data, performing data deduplication, or using RAID instead of data replication to achieve fault tolerance. In such environments, memory locality becomes the top concern, and disk or network locality becomes secondary [8]. If the workload characteristics permits a high level of memory or disk locality, network traffic gets decreased, the application performance increases, and incast becomes less of a concern.

The need for computational capacity efficiency implies that computing infrastructure needs to be more highly utilized. Network demands will thus increase. Consolidating diverse applications and workloads multiplexes many network traffic patterns. Incast will likely occur with greater frequency. Further, additional TCP pathologies may be revealed, such as the similarly phrased TCP outcast problem, which affects link share fairness for large flows [26].

5. Recommendations

Set TCP minimum RTO to 1ms.

Future big data workloads likely reveal TCP pathologies other than incast. Incast and similar behavior are fundamentally transport-level problems. It is not resource effective to overhaul the entire TCP protocol, redesign switches, or replace the datacenter network to address a single problem. Setting `tcp_rto_min` is a configuration parameter change – low overhead, immediately deployable, and as we hope our experiments show, it does no harm inside the datacenter.

Deploy better tracing infrastructure.

It is not yet clear how much incast impacts future big data workloads. The article discusses several contributing factors. We need further information to determine which factors dominate under what circumstances. Better tracing helps remove the uncertainty. Where possible, such insights should be shared with the general community. We hope the workload comparisons in this article encourage similar, cross-organizational efforts elsewhere.

Apply a scientific design process.

We believe future big data systems demand a departure from some design approaches that emphasize implementation over measurement and validation. The complexity, diversity, scale, and rapid evolution of such systems imply that mis-design opportunities proliferate, redesign costs increase,

experiences rapidly become obsolete, and intuitions become hard to develop. Our approach in this article involves performing simplified experiments, developing models based on first principles, empirically validating these models, then connecting the insights to real life by introducing increasing levels of complexity. We hope our experiences tackling the incast problem demonstrates the value of a design process rooted in empirical measurement and evaluation.

6. Acknowledgements

This research is supported in part by the UC Berkeley AMP Lab (`https://amplab.cs.berkeley.edu/sponsors/`), and the DARPA- and SRC-funded MuSyC FCRP Multi-scale Systems Center. Thank you to Rik Farrow and Sara Alspaugh for proof reading a draft of the article. Thank you also to Keith Sklower for assistance with the DETER Testbed logistics.

References

[1] Apache Hadoop Documentation. `http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html#Configuring+the+Hadoop+Daemons`.

[2] Hadoop World 2011 Speakers. `http://www.hadoopworld.com/speakers/`.

[3] IEEE 802.1Qau Standard - Congestion Notification. `http://www.ieee802.org/1/pages/802.1au.html`.

[4] Personal communications with Cloudera engineering and support teams.

[5] M. Alizadeh et al. Data center TCP (DCTCP). In *SIGCOMM 2010*.

[6] M. Alizadeh et al. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Annual Allerton Conference 2008*.

[7] Amazon Web Services. Amazon elastic compute cloud (amazon ec2). urlhttp://aws.amazon.com/ec2/.

[8] G. Ananthanarayanan et al. Disk-locality in datacenter computing considered irrelevant. In *HotOS 2011*.

[9] G. Ananthanarayanan et al. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI 2012*.

[10] G. Ananthanarayanan et al. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Eurosys 2011*.

[11] Apache Foundation. Apache Hadoop. `http://hadoop.apache.org/`.

[12] T. Benzel et al. Design, deployment, and use of the deter testbed. In *DETER 2007*.

[13] D. Borthakur et al. Apache Hadoop goes realtime at Facebook. In *SIGMOD 2011*.

[14] D. Borthakur et al. HDFS RAID. Tech talk. Yahoo Developer Network. 2010.

[15] Y. Chen et al. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *EuroSys 2012*.

[16] Y. Chen et al. SWIM – Statistical Workload Injector for MapReduce. `https://github.com/SWIMProjectUCB/SWIM/wiki`.

[17] Y. Chen et al. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS 2011*.

[18] Y. Chen et al. Understanding TCP incast throughput collapse in datacenter networks. In *WREN 2009*.

[19] EMC and IDC iView. Digital Universe. `http://www.emc.com/leadership/programs/digital-universe.htm`.

[20] M. Isard et al. Quincy: fair scheduling for distributed computing clusters. In *SOSP 2009*.

[21] E. Krevat et al. On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems. In *PDSW 2007*.

[22] T. Lipcon and Y. Chen. Hadoop and performance. Hadoop World 2011. `http://www.hadoopworld.com/session/hadoop-and-performance/`.

[23] M. Mathis et al. Request for Comments: 2018 - TCP Selective Acknowledgment Options. `http://tools.ietf.org/html/rfc2018`, 1996.

[24] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. In *VLDB 2010*.

[25] A. Phanishayee et al. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*.

[26] P. Prakash et al. The TCP Outcast Problem: Exposing Throughput Unfairness in Data Center Networks. In *NSDI 2012*.

[27] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM 2009*.

[28] H. Wu et al. ICTCP: Incast Congestion Control for TCP in data center networks. In *Co-NEXT 2010*.

[29] M. Zaharia et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 2010*.

[30] C. Zedlewski. An update on apache hadoop 1.0. `http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/`.