# Mitigating Datacenter Incast Congestion Using RTO Randomization

**Conference Paper** · December 2015

DOI: 10.1109/GLOCOM.2015.7417797

Ubaid Hafeez
Stony Brook University
**6** PUBLICATIONS **13** CITATIONS
SEE PROFILE

Aqsa Kashaf
Carnegie Mellon University
**4** PUBLICATIONS **11** CITATIONS
SEE PROFILE

Ihsan Qazi
Lahore University of Management Sciences
**47** PUBLICATIONS **449** CITATIONS
SEE PROFILE

Zartash Afzal Uzmi
Lahore University of Management Sciences
**56** PUBLICATIONS **855** CITATIONS
SEE PROFILE

# Mitigating Datacenter Incast Congestion Using RTO Randomization

Ubaid Ullah Hafeez, Aqsa Kashaf, Qurat-ul-ann Bajwa,
Hassan Zaidi, Ihsan Ayyub Qazi, Zartash Afzal Uzmi
SBA School of Science and Engineering, LUMS

Aisha Mushtaq
Microsoft Corporation
Vancouver, Canada

*Abstract*—**TCP incast congestion happens in many-to-one communication workflow patterns that frequently arise in large-scale datacenter applications such as web search, social networks, and cluster-based storage systems. Incast congestion can severely degrade the performance of applications. This paper studies the effectiveness of randomizing the TCP retransmission timeout (RTO) in mitigating the impact of incast. Our design is based on the observation that under incast, retransmitted packets also get synchronized due to the use of similar RTOs by the senders. Using analysis and experimental evaluation, we show that there exists a tradeoff between the randomization interval (from which the RTO values are picked) and the number of senders involved in incast. Motivated by this insight, we propose three algorithms (TDA, MAA, and FSA) for the dynamic adaptation of the randomization interval that rely on (a) successive timeouts, (b) explicit knowledge of the level of multiplexing, and/or (c) the knowledge of flow sizes (i.e., large interval for long flows and a small interval for short flows), respectively. Our results show that these algorithms improve goodput by 1.5x–11x for up to 64 senders and provide greater improvement for larger number of senders. The proposed algorithms can be readily deployed as they do not require any changes in switches or applications.**

## I. Introduction

Popular datacenter applications (e.g., web search, social networking, and cluster-based storage systems) achieve scalable performance by exploiting massive parallelism [1]. Thus, a single application task such as answering a search query or building a user's social news-feed involves making read requests to many servers concurrently. Such application tasks are often *barrier synchronized* because a client cannot make progress until responses from *all* servers are received. The responses from these servers often arrive synchronously at a common receiver leading to throughput collapse due to TCP timeouts and packet losses. This throughout collapse under many-to-one communication patterns is called *incast congestion* [2] and has been widely observed in many production datacenters (e.g., Bing [3], Facebook [4], Cosmos [5]). Incast congestion can severely degrade performance by increasing response times for the users.

The impact of incast congestion in datacenter networks is exacerbated by (a) the use of Top-of-Rack (ToR) switches with shallow buffers and (b) the presence of long-lived TCP flows that keep high buffer occupancy and thus reduce the ability of switch buffers to absorb large packet bursts due to synchronized server responses. Prior solutions have focussed on either (i) controlling switch buffer occupancy to avoid overflow by using ECN and/or a modified congestion control algorithm at the end-hosts [6], [7], (ii) reducing duplicate ACK threshold, or (iii) reducing the waiting time for recovering from packet losses by decreasing the TCP retransmission timeout (RTO) [2].

This paper studies the effectiveness of *randomizing* TCP retransmission timeouts, first suggested in [2], in mitigating incast congestion. Our approach is based on the observation that under incast, retransmitted packets also get synchronized due to the use of very similar RTOs (we term this phenomena as *RTO synchronization*) as also reported in [2]. This can lead to losses for retransmitted packets, which considerably degrades goodput. RTO synchronization happens because (a) senders involved in incast congestion usually have similar round-trip propagation delays because such communication is mostly within a rack or cluster (comprising of co-located racks) [1] and (b) the minimum TCP retransmission timeout ($\text{RTO}_{min}$) is generally much larger than the round-trip times (RTTs) within datacenters[1]. This causes all senders to use the same timeout value (equal to $\text{RTO}_{min}$) [2], [7], thereby leading to synchronous arrival of retransmitted packets. The loss of retransmissions is particularly detrimental for application performance because TCP increases RTO exponentially on every successive timeout.

Our results show that the performance gains from RTO randomization critically depend on the choice of the randomization interval (i.e., the interval from which the senders randomly pick a value for the RTO). In particular, we show that there exists a tradeoff between the number of senders in an application task and the length of the randomization interval. While a small interval improves goodput when the number of senders are small, a large interval actually decreases goodput in such cases. Similarly, when the number of senders are large, using a small interval degrades goodput. Thus, no single interval is optimal in *all* cases. Motivated by this insight, we propose three algorithms for the dynamic adaptation of $\text{RTO}_{min}$: (a) *Timeout-Driven Adaptation* (TDA), (b) *Multiplexing-Aware Adaptation* (MAA), and (c) *Flow-Size Aware Adaptation* (FSA). With TDA, the RTO randomization

[1]The default TCP $\text{RTO}_{min}$ in Linux is usually 200 ms whereas typical intra-rack RTTs are less than 0.1 ms [8]. Most operating systems track RTTs and timers at the granularity of 1ms or larger. Thus, it is challenging to reduce $\text{RTO}_{min}$ below 1ms without increasing system overhead due to the use of high resolution timers [2], [9].

978-1-4799-5952-5/15/$31.00 ©2015 IEEE

interval is increased exponentially on every successive timeout. This is similar to the backoff algorithm used in IEEE 802.11 wireless protocols [10]. Under the MAA algorithm, each flow uses the explicit knowledge of the number of flows involved in an application task to dynamically select an interval. In FSA, the randomization interval is chosen based on the size of a flow; long flows use a larger randomization interval than short flows as the former flows tend to saturate the bottleneck and drive the buffer occupancy. Introducing size awareness implicitly prioritizes short flows over long flows during times of network congestion. The size of flows is estimated based on the amount of bytes sent so far as done in [11], [12].

Our evaluation shows that these dynamic algorithms provide goodput improvements that range from 1.5x–11x across a range of senders (e.g., 4–64 senders) involved in an application task. The improvement is even larger when there are more senders involved in a task. Among the proposed dynamic adaptation algorithms, FSA generally performs better than other schemes because (a) it reacts to instantaneous timeout events and (b) treats long flows and short flows differently. While MAA performs well in many scenarios, its performance suffers in the presence of background TCP flows as it does not account for the heterogeneity in flow sizes and thus, their impact on the network.

RTO randomization is complementary to approaches that reduce the *cost* of timeouts by reducing RTO [2] because the latter approaches do not desynchronize transmissions/retransmissions and thus can still lead to goodput degradation under *successive* timeouts. Moreover, unlike several previous solutions, TCP RTO randomization does not require changes in applications or network switches, which is very helpful for ease of deployment and adoption.

Altogether, this paper makes the following contributions.

- We show that there exists a fundamental tradeoff between the number of senders involved in an application task and the length of the randomization interval.
- We propose algorithms for dynamic adaptation of TCP RTO that rely on either observed timeouts, explicit knowledge of the level of multiplexing, and/or knowledge of flow sizes.
- We evaluate the proposed algorithms under typical datacenter specific traffic patterns using extensive packet-level simulations and show that they can provide substantial goodput improvements under incast scenarios. Moreover, these algorithms can also improve performance under non-incast scenarios especially under high network loads.

The rest of the paper is organized as follows. We discuss related work in Section II and the incast impairment in Section III. We analyze the impact of randomization interval on application performance and present dynamic adaptation algorithms in Section IV. The evaluation results are presented in Section V. We offer concluding remarks in Section VI.

## II. Related Work

Prior approaches on mitigating incast congestion can be placed into four categories.

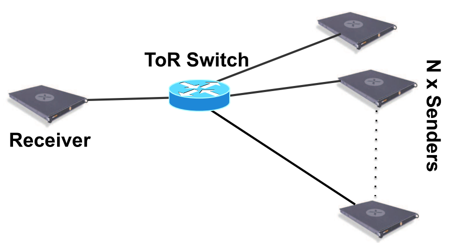

Fig. 1. Intra-rack topology used in our evaluation [1], [12].

The first category of approaches aim to *avoid* timeouts. Some notable approaches include (a) *application-level jittering*, in which senders adds random delay in the transmission of packets but requires changes in applications and increases the mean latency [6] and (b) reducing the duplicate ACK threshold from three to one of entering fast retransmission. However, this makes TCP more sensitive to packet reordering thereby making fine-grained load balancing hard in datacenters [13].

The second category of schemes reduce the *cost* of timeouts by reducing RTO$_{min}$. Vasudevan et al. [2] proposes to reduce RTO$_{min}$ to microsecond granularity, however, this requires high resolution timers that are difficult to implement in practice and introduce high system overhead [9]. Moreover, such schemes do not *desynchronize* transmissions. As a result, successive timeouts can still result in performance degradation. Our approach is complementary to such approaches because it reduces the likelihood of successive timeouts by proactively desynchronizing transmissions.

The third category of schemes use a new transport protocol. For instance, ICTCP [7] uses receiver-based flow control to mitigate incast and DCTCP [6] maintains small queues by using an aggressive AQM at the switches. Facebook employs application layer flow control over UDP [4]. While these approaches improve performance, they are hard to deploy in public clouds because the operators do not have control over the end-host stack running in a VM [1], [13].

Finally, one could use Ethernet flow control, however, it does not work well if multiple switches exist between a sender and receiver [9].

## III. TCP Incast Congestion

In this section, we present a study of TCP incast congestion and highlight the benefits of RTO randomization.

### A. TCP RTO Estimation

TCP maintains a smoothed estimate of the RTT and sets the retransmission timeout value as follows:

$$\mathrm{RTO} = \mathrm{SRTT} + (4 \times \mathrm{RTTVAR}) \tag{1}$$

where SRTT is the smoothed RTT and RTTVAR is the linear deviation. There are two factors that place a lower bound on the RTO value used by TCP flows: (a) minimum retransmission timeout ($RTO_{min}$) and (b) granularity of RTT measurement. Most implementations track RTTs at a granularity of 1ms or larger.

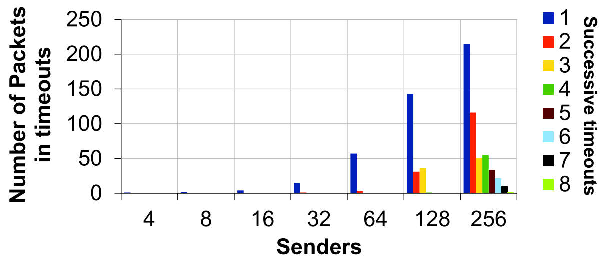

Fig. 2. Number of packets experiencing timeouts as a function of the number of senders in a task. The bars show the number of *successive* timeouts.

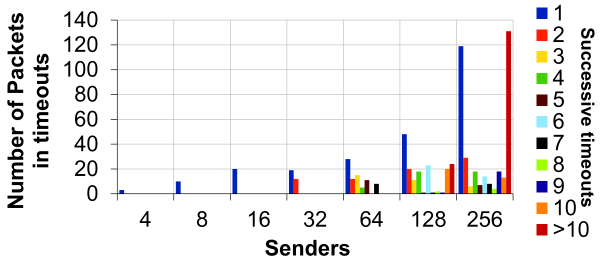

Fig. 3. Number of packets experiencing timeouts as a function of the number of senders in a task. There were two long-lived TCP flows in the background that represents 75th percentile multiplexing in datacenter networks [6].

After each successive timeout, TCP applies an exponential backoff algorithm to the RTO as follows:

$$\text{timeout} = \text{RTO} \times 2^b \tag{2}$$

where $b \geq 1$ is the number of successive timeouts. Whenever an acknowledgement packet is received, the backoff parameter $b$ is reset to zero.

## B. TCP Timeouts under Incast Congestion

We now present a study of incast congestion and show how it leads to TCP timeouts and degrades goodput.

*Experimental Setup:* We conduct our evaluation using ns2 simulations on a single-rooted tree, a commonly used topology for representing a *rack* where a rack contains multiple servers that are interconnected by a ToR switch within a datacenter (see Figure 1) [1], [12]. The round-trip propagation delay is set to $100\mu$s and each server has a 1 Gbps interface and uses a packet size of 1 KB. We configure the switch buffers with a size of 32 KB for every port as done in [2]. Our custom built application issues a request for a block of data (with size $B_s$ bytes) that is striped across $N$ servers. Each server responds with $B_s/N$ bytes of data. This resembles the communication patterns found in applications such as web search and MapReduce. To account for real world end-system scheduling delays, we add a random jitter in the arrival of server responses from the interval $[0, 20]\mu$s picked uniformly at random. We use TCP SACK with drop-tail queues unless specified otherwise. RTO$_{min}$ is set to 10 ms as suggested in [2], [11], [12]. Each experiment is run fifty times and we report the average goodput over the entire duration of the transfer.

Figure 3 shows the number of packets that experience timeouts as a function of the number of senders. Observe that the number of packets undergoing successive timeouts increases with the number of senders. For example, when there are 128 senders, ∼35 packets experience 3 successive timeouts and this number increases to ∼50 for 256 senders (and in addition, many flows now experience more than 3 successive timeouts). This happens because increasing the number of senders increases the number of packets that arrive synchronously at a switch, leading to more packet drops and TCP timeouts. This effect gets exacerbated because hosts use the same value for RTO$_{min}$ and hence for RTO (a phenomena we term as *RTO synchronization*). Consequently, flows that go into timeout, are likely to have synchronized arrivals of retransmitted packets. This increases the chances of successive timeouts and thus degrades goodput as shown in Table I.

| Goodput/Senders | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|
| No long flows | 547 | 592 | 557 | 323 | 173 | 26.9 | 3.3 |
| 2 long flows | 274 | 202 | 177 | 75.9 | 16.2 | 0.04 | 0.01 |

TABLE I
Task goodput (in Mbps) as a function of the number of senders with and without background flows (RTO$_{min}$=10ms).

We now add two long TCP flows in the background as this represents 75th percentile multiplexing in datacenter networks [6]. Observe that successive timeouts increase considerably with the addition of background flows as shown in Figure 3. For example, when there are 128 senders, now ∼25 packets experience more than 10 successive timeouts and this number increases to ∼130 for 256 senders. This happens because the long TCP flows keep high queue occupancy, which reduces the available buffer space for incast flows, thereby degrading the task goodput (see Table I).

## IV. Mitigating Incast: Randomizing RTO

We now analyze the effectiveness of RTO randomization in mitigating incast congestion by reducing successive TCP timeouts and thus, leading to higher task goodputs. In particular, we show that there exists a tradeoff between the length of the randomization interval and the number of senders involved in incast congestion scenarios. Next, we propose algorithms for dynamically adapting the randomization interval.

## A. Impact of the Randomization Interval

When randomizing the RTO of senders, a key design decision is the choice of the *length* of the interval from which the random RTO values are picked. A large interval provides greater degree of RTO desynchronization but can increase the completion time of some flows (*those who happen to pick a large RTO*). While a small interval ensures that no flow picks a large RTO, it reduces the degree of desynchronization.

When a timeout occurs, there are two possible cases which impact the value of the RTO picked:

- Case 1: If a flow does not receive RTT samples before a loss occurs, then the flow uses RTO$_{min}$ as the RTO.
- Case 2: If a flow receives RTT samples before a loss occurs, then the timeout is set to the larger of RTO$_{min}$ and the value determined by Equation 1.

The RTTs in datacenter networks are usually less than 1 ms. They are even smaller within a single rack (typically less than 0.1 ms [8]). On the other hand, existing operating systems track

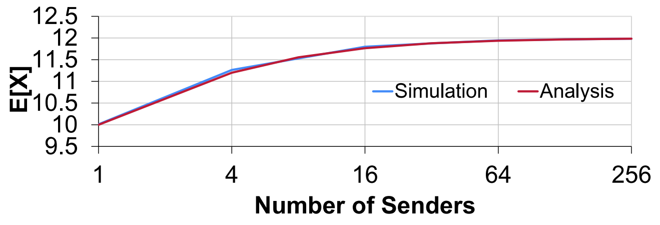

Fig. 4. Comparison of E[X] under simulations and analysis (i.e., obtained via Equation 6) across a range of senders involved in an application task. The RTO$_{min}$=10 ms and $\Delta$=4 ms, thus the maximum value of RTO$_{min}$ is 12.

RTTs at the granularity of at least 1 ms. Thus, RTO is almost always set to RTO$_{min}$ as also indicated by previous works [2], [7], [9].

Therefore, to desynchronize RTOs of different senders, we randomize RTO as follows:

$$RTO = U(RTO_{min} - \Delta/2, RTO_{min} + \Delta/2) \quad (3)$$

where $\Delta$ is the length of the randomization interval and $U(a, b)$ is a function which returns a value uniformly at random from the interval $[a, b]$. There are two key observations about the above equation: (i) the average value of the interval is $RTO_{min}$ irrespective of its length. This is important to ensure fair comparison when changing the interval length and (ii) increasing the interval length *can* cause some flows to pick small RTO values (e.g., close to $RTO_{min} - \Delta/2$) and some flows to pick large values (e.g., close to $RTO_{min} + \Delta/2$) given sufficiently large number of senders/flows involved in an application task.

*1) Analysis:* Suppose there are $N$ senders in a task undergoing an ensuing incast. Let $X_i$ be a uniform random variable representing the value of RTO picked by sender $i$ and let $X = max(X_1, X_2, .., X_N)$ be the maximum RTO across the $N$ senders. The sender which happens to pick the maximum RTO will wait the longest before retransmitting the packet. As each sender picks the RTO independently of other senders, the probability that $X$ exceeds $k$ is given by[2]

$$P(X > k) = 1 - \left[\frac{k - (RTO_{min} - \Delta/2)}{\Delta}\right]^N. \quad (4)$$

Observe that as the number of senders increases, the probability that $X$ exceeds $k$ increases exponentially.

Let $a = (RTO_{min} - \Delta/2)$ and $b = (RTO_{min} + \Delta/2)$, then the expected value of $X$ is given by

$$E[X] = \int_a^b x \times \frac{N(x - RTO_{min} + \Delta/2)^{N-1}}{\Delta^N} dx \quad (5)$$

$$= RTO_{min} + \frac{\Delta(N-1)}{2(N+1)}. \quad (6)$$

Observe that when $N = 1$, $E[X] = RTO_{min}$ as expected. However, as $N$ increases, the ratio $(N-1)/(N+1)$ converges to 1 and therefore, $E[X]$ approaches $RTO_{min} + \Delta/2$, thereby leading to greater degree of synchronization for retransmitted packets (see Figure 4).

[2]We ignore successive timeouts in this expression.

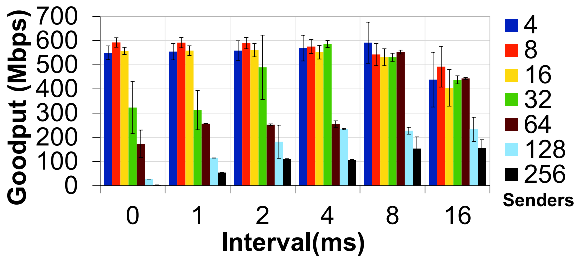

Fig. 5. Goodput as a function of the length of the randomization interval. Note that there are no background TCP flows in these experiments.

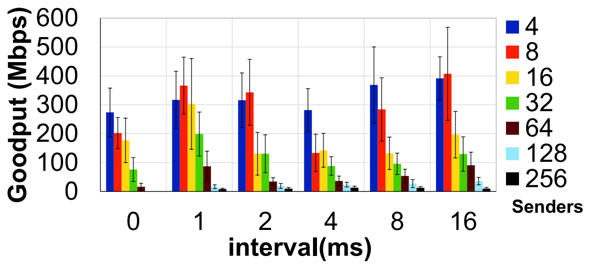

Fig. 6. Goodput as a function of the (randomization) interval length. In these experiments, there are two long-lived TCP flows in the background.

Interestingly, using a large $\Delta$ (for a fixed $N$) also increases the variance of $X$ as

$$Var(X) = \frac{2\Delta^2}{(N+1)(N+2)} - \frac{\Delta^2}{(N+1)^2} \leq \frac{\Delta^2}{(N+1)^2}. \quad (7)$$

Thus, while a larger value of $\Delta$ can lead to greater desynchronization across senders, it may also increase performance variability especially for smaller values of $N$.

*2) Evaluation:* Using the same experimental setup as described in Section III-B, we now carry out ns2 simulations to quantify the tradeoff between the length of the randomization interval and the number of senders. Figure 5 shows goodput as a function of the number of senders across different interval lengths. Observe that for small number of senders (e.g., 4, 8, and 16), the goodput decreases when the interval length becomes greater than or equal to 8 ms. This happens because in such cases senders experience relatively small number of timeouts (see Figure 2) and picking a large RTO unnecessarily increases the task duration and thus decreases goodput. When there are 32 senders, goodput first increases with the interval length because of higher degree of desynchronization enabled by a large interval, however, beyond a certain value, increasing the interval unnecessarily delays the task resulting in loss of goodput. For larger number of senders (e.g., 64, 128, 256), increasing the interval provided substantial improvement in throughput. For instance, using a interval of 16 ms improves goodput for 256 senders by more than 150x.

*Impact of Background Flows*: We now maintain two long TCP flows in the background and generate a task as before. These long flows maintain high buffer occupancy which reduces the ability of the buffer to absorb synchronous packet bursts. Figure 6 shows the goodput as a function of the number of senders. Observe that in several cases (and unlike previously), using the largest randomization interval provides the most significant improvements in goodput.

These results show that while randomization *can* improve goodput, the improvement depends on the number of senders and the randomization interval. Thus, no one randomization interval is optimal for all number of senders. This motivates the need for the dynamic adaptation of the randomization interval.

## *B. Dynamic Adaptation of Randomization Intervals*

In the previous section, we showed that a small interval improves goodput when the number of senders $N$ is small but results in goodput degradation for large $N$. On the other hand, a large interval improves goodput for large $N$ but degrades goodput otherwise. Towards this end, we propose algorithms for the dynamic adaptation of the randomization interval.

(a) ***Timeout-Driven Adaptation (TDA)***: In this approach, each sender picks a RTO value uniformly at random from an interval and on each successive timeout, exponentially increases the randomization interval. A similar approach is employed in randomized backoff protocols in IEEE 802.11 based wireless networks for collision avoidance. This approach is based on the observation that as the number of senders increases, the likelihood of successive timeouts also increases. Thus, we adapt the RTO as follows:

$$RTO = U\left(RTO_{min} - \frac{\omega^s\Delta}{2}, RTO_{min} + \frac{\omega^s\Delta}{2}\right) \quad (8)$$

where $\Delta$ is the *initial* length of the randomization interval, $\omega$ is the factor by which $\Delta$ is increased on every successive timeout, and $s \in \{1, 2, 3, ..., M\}$ denotes the number of successive timeout. For example, if $\omega = 2$ then on the third consecutive timeout (i.e., $s = 3$), the randomization interval would be $2^3 \times \Delta$. Note that the choice of the initial $\Delta$ would depend on the number of packets that can be transmitted within the interval e.g., if the packet transmission time is $10\mu s$ then 100 packets can be transmitted when $\Delta = 1\,ms$.

(b) ***Multiplexing-Aware Adaptation (MAA)***: In this approach, each sender is aware of the number of senders $N$ (i.e., level of multiplexing) that are part of an application task. Thus, each sender uses a common function $f(N)$ to pick a randomization interval based on the knowledge of $N$ i.e., a small interval for small $N$ and a large interval when $N$ is large. While determining an optimal function is beyond the scope of this work, insights drawn from our evaluation suggest that a sub-linear function is suitable e.g., $f(N, \sigma) = \left\lfloor \frac{N}{\sigma log N} \right\rfloor$, where $\sigma$ is a parameter that controls the aggressiveness of growing the randomization interval with increase in $N$.

(c) ***Flow-Size Aware Adaptation (FSA)***: Under incast congestion scenarios, the presence of long flows increases the chances of successive timeouts as they reduce the ability of switch buffers to absorb packet bursts. With FSA, the randomization interval is adapted based on flow sizes i.e., long flows pick a larger interval (e.g., by using TDA with $\omega = 4$) compared to short flows (e.g., by using TDA with $\omega = 3$). This is beneficial for two reasons: (a) it reduces the buffer occupancy due to long flows' packets, which helps in reducing timeouts for the incast flows and (b) it reduces the level of multiplexing at the link, which helps short flows to attain higher throughput.

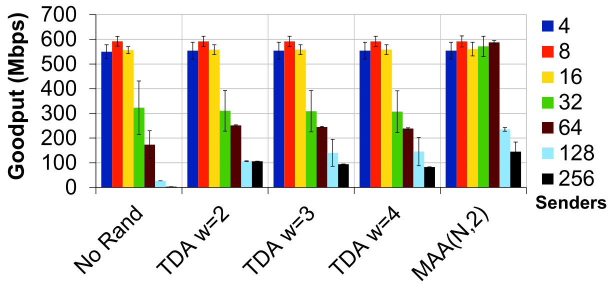

Fig. 7. Goodput of an application task under TDA and MAA in the absence of background flows for senders ranging from 4 to 256.

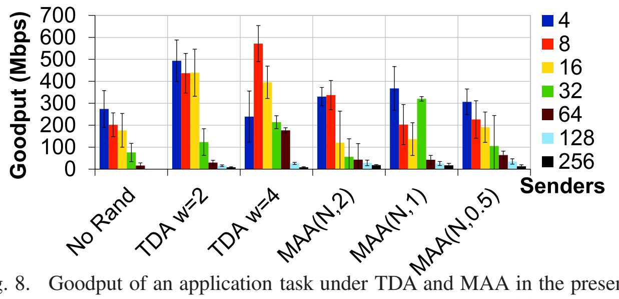

Fig. 8. Goodput of an application task under TDA and MAA in the presence of two long-lived TCP flows for senders ranging from 4 to 256.

## V. Evaluation

We now evaluate the impact of RTO randomization and the proposed algorithms using extensive packet-level ns2 simulations. We have implemented all the algorithms in ns2. First, we evaluate the performance of the proposed algorithms under incast congestion scenarios for varying number of senders using the same experimental as described in Section III-B. Finally, we evaluate the proposed solution under non-incast scenarios for a range of network loads. Each data point represents the average of twenty runs.

## *A. RTO Adaptation Algorithms*

*1) TDA and MAA:* Figure 7 shows the goodput of the application task for TDA and MAA for different number of senders. Observe that randomization (for both TDA and MAA) provides substantial improvement in goodput when the number of senders are more than 64. For example, TDA improves by ~1.5x, ~4x, and ~32x when the number of senders are 64, 128, and 256, respectively. This happens because in the case of timeouts, TDA increases the backoff interval exponentially, which helps in desynchronizing retransmissions and thereby improves goodput. MAA provides the most improvement in goodput. In particular, it improves goodput by ~1.8x, ~3.4x, ~8.7x, and ~44x for 32, 64, 128, and 256 senders. This happens because MAA keeps track of the active number of senders at all times. Thus, it is able to effectively prevent many timeouts. This is unlike TDA, which is a *reactive* approach.

*Impact of Background Flows*: We now introduce two background TCP flows [6]. Figures 8 and 9 show the goodput and the average number of timeouts/packet, respectively, for

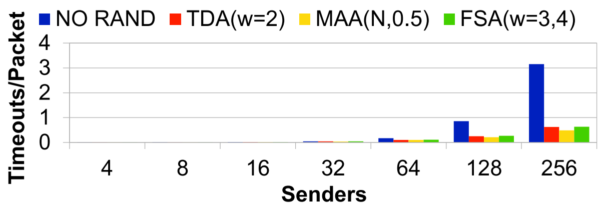

Fig. 9. Average number of timeouts per packet for different schemes (with two long-lived flows in the background).

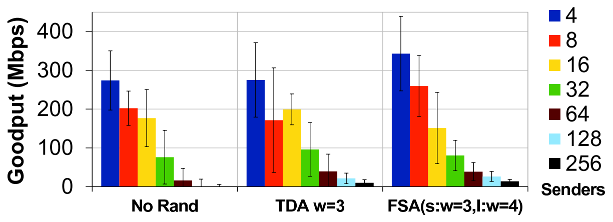

Fig. 10. Goodput of an application task under TDA and FSA in the presence of two long-lived TCP flows for senders ranging from 4 to 256.

different schemes. Observe that randomization provides substantial goodput improvement across a range of senders. For example, under TDA ($\omega = 2$), goodput improves by ∼2.8x-11x for senders in the range 4-64 and by more ∼629x for 128 and 256 senders. Under MAA, the goodput improvement is less than under TDA when the number of senders are 64 or less. This happens because the presence of two background flows increases timeouts significantly (see Figure 3) but MAA does not react to these timeouts as it treats all flows alike (whether short or long). However, for large number of senders, it provides goodput improvement that is comparable to or better than TDA. This happens because MAA picks a very large randomization interval (due to large number of senders) and dynamically reduces it when flows depart the network.

*2) FSA:* Long flows and short flows impact network congestion differently. Long TCP flows are able to saturate the network and tend to dominate the buffer space. As a result, they have a greater impact on application performance. Thus, it is useful to penalize these flows more than the relatively short flows of applications tasks when congestion occurs. Figure 10 shows the goodput under FSA for different number of senders. Observe that FSA can be better than TDA in many scenarios.

## *B. Non-incast Scenario*

We now evaluate the impact of RTO randomization on *non-incast* scenarios, where flows do not necessarily arrive synchronously. Such scenarios are also fairly common in datacenter networks [1], [11]. To emulate such a scenario, we generate short query traffic with flow sizes drawn from from the interval [10 KB, 50 KB] using a uniform distribution. These flows arrive according to a Poisson process as done in prior works [11], [12]. We also generate two long-lived TCP flows in the background. The resulting workload resembles that of web search as reported in a prior study [6].

Figure 11 shows the average flow completion time (AFCT) of short (or mice) flows as a function of the their offered load. Observe that TDA results in similar AFCTs across a range of offered loads. In fact, at high loads ($\geq$ 80%) TDA improves

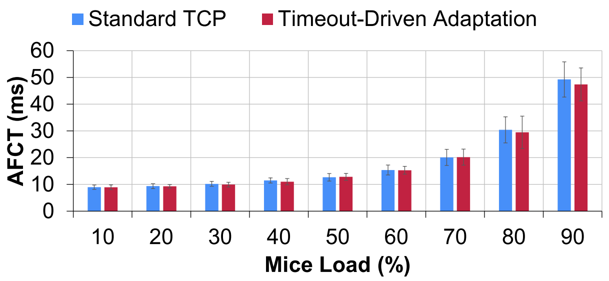

Fig. 11. Average flow completion times (AFCT) as a function of mice load. We maintain two long TCP flows in the background.

AFCTs by ∼9%. This happens because at high loads, large number of short flows are simultaneously active and incur timeouts. TDA randomization actually reduces the level of multiplexing by delaying retransmissions from some flows, thus helping flows to complete faster.

## VI. Conclusion

In this paper, we studied the efficacy of RTO randomization in mitigating incast congestion. We showed that the ideal randomization interval depends on the number of senders involved in incast and the flow sizes. Towards this end, we proposed three algorithms that dynamically adapt the randomization interval based on various metrics including successive timeouts, level of flow multiplexing, and the size of flows. These algorithms substantially improve goodput and can be easily deployed without requiring changes in switches or applications.

## References

[1] D. Abts and B. Felderman, “A guided tour of data-center networking,” *Commun. ACM*, vol. 55, no. 6, pp. 44–51, Jun. 2012.
[2] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, “Safe and effective fine-grained tcp retransmissions for datacenter communication,” in *SIGCOMM’09*.
[3] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, “Speeding up distributed request-response workflows,” in *SIGCOMM’13*.
[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, and et al., “Scaling memcache at facebook,” in *NSDI’13*.
[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, “Reining in the outliers in map-reduce clusters using mantri,” in *OSDI’10*.
[6] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *SIGCOMM’10*.
[7] H. Wu, Z. Feng, C. Guo, and Y. Zhang, “Ictcp: Incast congestion control for tcp in data center networks,” in *Co-NEXT’10*.
[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, “pfabric: Minimal near-optimal datacenter transport,” in *SIGCOMM’13*.
[9] J. Zhang, F. Ren, and C. Lin, “Modeling and understanding tcp incast in data center networks,” in *INFOCOM’11*.
[10] “IEEE Standard for local and metropolitan area networks part 11; amendment 5: Enhancements for higher throughput,” 2009, 802.11n.
[11] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, “Minimizing Flow Completion Times in Data Centers,” in *INFOCOM’13*.
[12] C.-Y. Hong, M. Caesar, and P. B. Godfrey, “Finishing flows quickly with preemptive scheduling,” in *SIGCOMM’12*.
[13] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, “Conga: Distributed congestion-aware load balancing for datacenters,” in *SIGCOMM’14*.