Towards a Redundancy-Aware Network Stack for Data Centers

Ali Musa Iftikhar, Fahad R. Dogar
{musa, fahad}@cs.tufts.edu
Tufts University

Ihsan Ayyub Qazi
ihsan.qazi@lums.edu.pk
LUMS

ABSTRACT

In this paper, we make a case for a redundancy-aware network stack (RANS) for data centers. In RANS, applications expose information about replicas to the network, which in turn, uses *duplicate* requests to improve performance of typical applications by enabling them to effectively avoid *stragglers*. At the heart of RANS is the use of duplicate-aware scheduling, which ensures that duplicate-requests do not overload the system and disturb any primary requests. We highlight the challenges and opportunities present at different layers of RANS, from new interfaces that capture replicas and their semantics, to in-network mechanisms that deal with duplicates. Our preliminary evaluation shows the promise of duplicate-aware scheduling in improving performance of typical data center applications.

1 Introduction

Performance is critical for most data center applications, including both user-facing services (e.g., search), where slower responses result in user dissatisfaction and hence drop in revenue [4, 15], as well as data analytics applications (e.g., MapReduce [13]), where increase in job completion times result in higher cost for the tenant [19]. Unfortunately, meeting the performance requirements of these applications turns out to be challenging. These applications have workflows with *high fanout* – an application operation typically involves contacting multiple servers in parallel. For example, a search query in Bing is processed by thousands of servers before the response is sent back to the user [4]. For such applications, even a single slow component, or *straggler*, ends up delaying the entire application. Studies show that stragglers are a major problem for cloud service providers – they can cause significant increase in application response times, especially

at the tail [4, 15, 6, 30].

Overcoming stragglers is challenging: they are difficult to predict, and a slew of factors can cause stragglers, such as failures, background processes, and garbage collection. In practice, it is difficult to completely eliminate stragglers. Arguably, the most promising approach is to rather *avoid* them by using alternate replicas. Most storage-based applications already have in-built replication. For example, the popular Hadoop Distributed File System (HDFS) stores three copies of a data chunk [10, 2]. Similarly, compute-based applications are also designed to tolerate failures and execute more than a single copy of a task [30, 7]. A common technique to leverage replicas is to issue one or more redundant requests, in addition to the original request, and consider the response of the first one that completes. Various real world systems (e.g., Bigtable [9]) use this approach to reduce their response times [12]. However, issuing redundant requests increases the load on the system so the conventional wisdom is to only use this approach at low loads or to issue duplicate requests only if the original request turns to be a straggler [38, 7, 30]. These approaches tend to be *ad-hoc* (as determining load thresholds for enabling/disabling duplicate requests varies across workloads [38]) or *conservative* (to avoid adjusting thresholds due to variations in workloads) in terms of how they use duplicate requests. Consequently, both the approaches reduce the effectiveness of using duplicate requests.

In this paper, we call for making *duplicate* requests a key network mechanism so that we can use them without having to worry about increased system load. To this end, we propose a simple *duplicate-aware scheduling mechanism* that is implemented by all critical resources in the system (e.g., network, storage, and application) that could potentially become a bottleneck. The key idea is to schedule duplicate requests such that they do not hurt primary requests. We achieve this property by combining two primitives from scheduling theory: *priority queues* which schedule requests in a *strict priority* fashion, and *purging*, which allows requests that are no longer required to be purged from the system.

The key enabler for duplicate-aware scheduling is information about replicas, which allows duplicate-aware network mechanisms at different layers of the stack. To illustrate the rich opportunities (and challenges) in this space, we present a redundancy-aware network stack or RANS, which includes a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets '16, November 9–10, 2016, Atlanta, GA, USA

© 2016 ACM. ISBN 978-1-4503-4661-0/16/11...\$15.00

DOI: <http://dx.doi.org/10.1145/3005745.3005764>

duplicate-aware transport, a network layer design, and a new interface between the applications and the data center network that can be used by applications to expose information about replicas to the network. We provide an initial exploration of the benefits of duplicate-aware scheduling as well as the challenges involved in implementing it for various resources. For example, in case of a network, switches already support priority queues but the challenges include effectively using the priority queues at the transport layer and supporting purging inside the network switches. For applications, supporting priority queues with preemption support is a key challenge, in addition to ensuring that all the different stages of the application are made duplicate aware.

2 Duplicate-Aware Scheduling

2.1 Motivation

We discuss the limitations of improved replica selection mechanisms in dealing with stragglers and also highlight the issues with existing techniques that use duplicate requests.

Limitations of intelligent replica selection schemes. Several recent schemes *intelligently* select a replica in order to improve performance [10, 34]. Their high level goal is to intelligently guess which replica will turn to be the best one to use, either based on load or some other heuristic. Specifically, a well-known strategy is to execute the task at the replica with the smallest queue (also known as Join Shortest Queue (JSQ) [17]). There are two limitations with schemes like JSQ. First, load imbalance is only one of the (many) reasons why we may have a straggler – other causes of stragglers (e.g., failures [12]) are not captured by load and may in fact be impossible to predict in advance. Second, it is typically difficult to accurately calculate load and in some cases the overhead of calculating load may be quite high. For example, determining the shortest queue in case of distributed resources incurs a non-trivial overhead.

Limitations of existing uses of duplicate requests. Existing systems that use duplicate requests consider these requests at the *same* priority as normal requests. As a result, a key concern is how to limit the increased load they put on the system. One popular strategy is to *cancel* subsequent requests as soon as the first one finishes [12]. Unfortunately, this strategy is difficult to use for applications that have small requests (e.g., Memcached). For such applications, the cancellation request may arrive too late – the duplicate request may have already consumed resources by that time (at the expense of a primary request). In such scenarios, trying to desynchronize the primary and duplicate requests (by adding a small jitter between them) is also challenging: a value too small would increase system load (as discussed above) and a value too large would limit the benefits of duplicate requests.

2.2 Duplicate-Aware Scheduling - Basic Model

We consider a simple queuing model for duplicate-aware scheduling in the context of jobs arriving at a server farm (or routers). For every request that arrives in the system, we

Figure 1: Role of different layers in RANS.

create multiple copies. Then, for each copy, we randomly pick a primary server and one or more duplicate servers. The system uses two well-known primitives.

1. Priority Queues. Each server maintains two queues, a primary queue and a duplicate queue. Each queue processes requests in a first-in-first-out (FIFO) order. The primary queue gets strict preference over the duplicate queue, i.e., it supports preemption, and the system is work-conserving, i.e., duplicate requests are served if and only if the primary queue is empty. The above behavior ensures that no duplicate request delays a primary request, thus their use is totally opportunistic.

2. Purging. When a request finishes (either the primary or duplicate), we purge its corresponding copies from the system. Purging ensures that a subsequent duplicate request does not wait behind a previous duplicate request which is no longer required. Note that purging is an optimization to lower the waiting time of duplicate requests. This is unlike current systems where purging/cancellation is required to ensure that a request does not delay another primary request.

When is duplicate-aware scheduling useful (and when it is not)? It is most useful in scenarios where the primary turns to be a straggler (for whatever reason) but *at least* one of the replicas is lightly loaded and can process requests from its duplicate queues. We believe such scenarios are common because of: i) skewed workloads which result in load imbalance between nodes (e.g., [29] found job durations at a Google cluster to be heavy-tailed) and ii) random (non-load related) factors that may result in a straggler even under low system loads. On the other hand, if *all* the replicas are heavily loaded, then duplicate-aware scheduling would not provide gains, but it will also not increase the load on the system, i.e., no primary request will be hurt.

3 RANS

We now zoom into the *network* and present the sketch of redundancy-aware network stack or RANS, which provides explicit support for duplicate-aware scheduling. Our goal is to highlight the challenges and opportunities present at different layers of the network stack when we treat duplicate requests as a key network mechanism.

Figure 1 provides an overview of the role of different layers in RANS. Applications implement duplicate-aware schedul-

Figure 2: A simple DAG for a `get()` operation at a client (C) which involves reading from any of the two servers (S1 and S2).

ing and use a high level *interface* based on `put()` and `get()` to communicate with the network (i.e., transport layer). This interface allows applications to expose information about replicas, their associated semantics, and performance requirements in the form of a directed-acyclic graph (DAG). The data granularity is at the level of an application data unit (ADU) [11, 14] or chunks (i.e., units meaningful for the application).

The RANS transport is responsible for implementing the `put()` and `get()` operations, with the help of the information in the DAG. Unlike today’s point-to-point transport, the RANS transport is point to multi-point: each application operation will generate multiple transport layer requests (or flows). For example, a `get()` may result in one request to the primary replica and one or more duplicate requests to the other replicas. Note that in the case of a `put()`, the benefits of duplication would come if the application has multiple choices in terms of where to place a replica, as is the case in distributed file systems (e.g., CFS [10]). RANS transport also supports purging, so data that is in the send buffers and ready to go out may be purged.

Primary and duplicate flows traverse the duplicate-aware network layer, which supports priority queues and purging. In case of a `put()`, if the semantics allow, the transport may complete the operation when a write completes at any of the candidate machines that can store the data, and the application is informed about where the data was written. For `get()` requests, the transport can combine responses from multiple replicas and complete the operation when the entire chunk is retrieved – depending on the semantics, the application need not know which replica(s) were used to construct the response.

Assumptions. We highlight the three key assumptions underlying the design of RANS.

1. Application semantics allow execution of a request more than once, either because the operation is idempotent (e.g., read operations) or the application can implement stricter semantics on top of the “at-least-once” semantics provided by RANS. Many popular cloud systems (e.g., MapReduce [13]) provide similar semantics under failure scenarios.
2. Network switches support *priority queues* – most modern data center switches are already equipped with up to eight hardware priority queues [3].
3. Creating multiple copies of the request has negligible overhead on the client and this overhead does not become a bottleneck in the system. This assumption holds true

for many popular cloud systems where the bottlenecks are typically on the server side, or inside the data center network, or when the client processes the responses (i.e., the return path) [23, 20, 10, 5, 24]

3.1 Interface

RANS requires a new interface between the applications and the network (i.e., transport protocol). Today’s interface – the socket API – is at the flow-level and reveals little application information to the network. The RANS interface between applications and the network needs to meet three requirements. First, it needs to be at a higher level compared to today’s socket-based flow-level interface. For storage-based applications, the right level is specifying `put()` and `get()` operations. Second, the application should be able to reveal the redundancy information to the network, including information about replicas. Third, the interface should capture the semantics and performance requirements of the application. For example, the application should be able to specify whether it wants a `put()` to be synchronous or asynchronous.

DAG-based Interface. Our proposed DAG-based interface supports two basic operations: `put()` and `get()`. Nodes will be compute and storage nodes while edges will correspond to the flow of information (control information or data). For each operation, the application can specify exact storage nodes to be used (e.g., machine A, machine B) or can also specify high level requirements based on typical data center topologies, allowing the network to choose an appropriate machine. For example, the application can specify a policy like *choose a replica which is in a different rack compared to the primary replica* (i.e., fault tolerance requirement). The application can also specify the timing requirement for each operation, allowing the network to optimize in terms of the level of duplication it wants to use.

Figure 2 shows an example of a `get()` operation with timing requirements of <10msec. The application specifies the replicas as well as the overall timing requirement. In this case, the application can choose any replica, but the interface will allow flexible options. The network can also choose the number of duplicate requests to make based on the timing requirement. The main challenge in designing the DAG based interface will be to support all the flexible options to specify replicas, their semantics, and performance requirements while keeping the interface as simple as possible.

3.2 Transport

Compared to TCP, the RANS transport provides a higher level abstraction to the applications. It supports two operations: `put()` and `get()`. Each operation will result in one or more flows between the client and servers, thus requiring a point-to-multipoint transport (unlike TCP’s point-to-point abstraction). A typical goal of the transport will be to minimize the completion times of these operations. It will also need to address challenges related to implementing the required semantics of the operation (e.g., strict consistency vs. eventual consistency), dealing with primary and duplicate flows,

Figure 3: RANS transport showing dynamic change in flow priorities and requesting non-overlapping byte ranges from different nodes.

the control laws that different types of flows will use, what to request on each of the flow, and how to decide priorities of the flows.

Strawman Solution. The key RANS transport functionality for leveraging duplicates and dealing with the associated challenges can be implemented on top of DCTCP [4, 22]. The two key features of the transport are: i) dynamically changing the priority of flows based on the available throughput, and ii) requesting non-overlapping byte ranges from primary and duplicate flows (in case of reading a large block). The former ensures that the best replica is used while the latter allows bandwidth aggregation, if the bottlenecks are close to the individual servers.

Priorities. The client transport *dynamically* decides the priority of each flow, the response flow also reflects this same priority. As a starting point, the client can use some hint from the application to decide which flow will be the primary and which flow will be the duplicate (this could be based on replica selection algorithms [10, 34]). Once it starts getting data, it can change the priority of the flows. For example, in Figure 3, if server 1 turns out to be a straggler and doesn't respond, while server 2 responds, the transport will switch the priority of the flows. A more challenging case is deciding to switch *during* a transfer because it is difficult to compare the throughput of a primary and duplicate flow as they are mapped to different priority queues. We can consider the use of sharing available bandwidth information *across* requests – this is based on the observation that a server sending a duplicate flow may also have a primary flow belonging to a different application operation which is mapped to the high priority queue.

Non-Overlapping Byte Ranges. Figure 3 also shows a simple example to illustrate how primary and duplicate flows can transmit non-overlapping bytes streams. Both flows need to transmit the same data chunk but rather than transmitting the same bytes, we make them send *different* parts of the chunk. For the primary flow, we will send the bytes in the typical order, starting with the first byte and transmitting as much as our congestion windows allows. For the duplicate flow, we transmit the bytes in the *reverse* direction, i.e., from the last byte of the chunk to the first byte. This ensures that the client is able to *aggregate* the bandwidth of these flows when the server uplink is the bottleneck. The byte range selection

Figure 4: Motivating example for why purging is required. Server 2 turns out to be a straggler but Server 1 cannot process its request because it is queued behind another request which is no longer required.

depends on the congestion window of the flow as well as the number of flows contributing to the request. An important challenge will be to figure out the suitable byte ranges that will be requested from each flow under different scenarios.

3.3 Network Purging

Prioritization ensures that packets belonging to duplicate requests do not harm the primary requests. However, duplicate queues could still fill up with packets that are no longer required. In such scenarios, *network purging* is particularly useful as it removes duplicate requests that are no longer required so they do not end up delaying *important* duplicate requests (whose primary request is a straggler). Figure 4 shows an example to illustrate this scenario. Request A is finished but its duplicate is still in the queue at Server 1 and blocking Req B's duplicate request whose primary request turns out to be a straggler (at Server 2). In this case, Req B's duplicate request becomes important but because of the FIFO nature of the queues, the duplicate request will only get a turn after Req A's duplicate request is processed. Unfortunately, network switches, do not support purging. Once a packet is queued, it can only be transmitted on the link.

Below, we present two possible approaches to support a `purge()` operation for network switches.

Efficient Purging. The key challenge with supporting purging is to do it efficiently – the primary overhead being *searching* for the right packet. A solution that looks at all the packets in the queue to identify the correct packet to purge is too costly to implement. Recent work on programmable schedulers show how to efficiently insert a packet in its suitable location, but it is unclear how we can extend this work to efficiently search for packet(s) that needs to be purged [32]. One specific idea we can consider is the use of *multiple* weighted fair queues for the duplicate traffic (instead of a single duplicate queue) and doing purging at the granularity of a *queue* rather than a flow. Modern switches already have multiple queues and can support weighted fair queueing. This approach offers three potential benefits. First, having multiple queues will reduce the head-of-line blocking problem as a packet will only wait behind packets in its own queue rather than a single long queue. Second, we can adjust weights for each queue based on the number of redundant duplicate flows mapped to it. Queues with more “purgeable” flows will have lower weight compared to a queue with fewer purged flows. Third, we can purge an entire queue if all its flows need to be purged, without requiring per-packet look-ups.

Figure 5: Preliminary simulation results for Duplicate-Aware Scheduling.

Figure 6: Benefits of different Duplicate-Aware Scheduling features.

Approximating Purging. The above approach of supporting a `purge()` operation requires changes to existing switches. An interesting research direction is to explore how we can support purging without modifying the switches. One possible approach is to minimize the duplicate queue size within the network and move the queues to the end-hosts where it is easier to support the purge operation. To this end, we can consider the possibility of using the priority flow control (PFC) feature supported in modern data center Ethernet switches [1]. PFC uses per-queue back-pressure to achieve zero loss. We can use PFC only for the duplicate queue and the primary queue can continue to have the traditional end-to-end congestion control (in order to avoid the issues with PFC, namely fairness and deadlocks [33]). The goal is to keep the duplicate queue size small while ensuring that whenever there is spare capacity it is quickly used through hop-by-hop signaling. To achieve this, we can use PFC with a small queue size threshold; this will keep the duplicate queue size within the network small, but will result in queue buildup at the end-hosts. At the end-host, we can implement support for purging (as described earlier). The key challenges for this technique include careful selection of the queue thresholds (to ensure stability and performance) and adequate support at the end-hosts (purging as well as integration with the transport protocol).

4 Preliminary Evaluation

We use simulations to evaluate the potential benefits of duplicate-aware scheduling and to understand the benefits of different mechanisms in RANS. Our simulations are done in

the ns-2 simulator and capture all the flow level details relevant in a practical scenario. We consider a strawman duplicate-aware solution that uses priority queues inside the network along with transport and network level purging support.

We model a storage scenario with a single client and ten storage servers. The network is the only bottleneck in this scenario. A read at the client results in randomly picking two servers – one acts as the primary while the other acts as the duplicate. The duplicate server assigns a lower priority to its flow. This duplicate flow fetches a non-overlapping byte range. Request arrivals are Poisson and each request is for a 10MB chunk size. Overall, the scenario represents an M/D/C queuing system.

Benefits of Duplicate-Aware Scheduling. We first compare duplicate-aware scheduling (with all the features enabled) against two other schemes: i) single-request, and ii) two-requests, where there is no support for duplicate-aware scheduling (so both requests are mapped to the same queue). Figure 5 shows the average request completion time as a function of load. The results show that duplicate-aware scheduling provides gains at all load levels whereas the two-copy technique provides gains only at low loads and as soon as the load increases, it overloads the system and performs worse than the single-request scheme. Observe that at high loads, duplicate-aware scheduling improves request completion time by almost 75% and we observe even higher gains if we look at the tail request completion times (not shown here). Also, note that this experiment only captures load imbalance as the potential source of a straggler, whereas in a practical scenario there are various other sources of stragglers as well. In such scenarios, we can potentially get gains at lower loads too.

Where are the benefits coming from? Next, we isolate the benefits of specific features of duplicate-aware scheduling. Figure 6 summarizes the results, showing that each feature of RANS is important and contributes towards the benefits provided by duplicate-aware scheduling. First, we focus on the benefits of *only* using priority queues, shown as “two-requests + priority queues” in the figure. We can see that by de-prioritizing duplicates the flow completion times improve and are strictly better than those of *single-request* at all loads even though at high loads, the gains are minimal. However, if we introduce purging (at both the transport and network levels) shown as “two-requests + priority queues + purging” in the figure, we see gains at the higher loads too. Finally, we can get additional gains if we also add support for aggregating bytes across flows – which is included in the *duplicate-aware scheduling* scheme (along with all the other features).

Need for in-network Purging. In the above experiment, the large request size means that most of the benefits of purging come from transport layer purging of flows as most of the remaining bytes are on the server rather than inside network switches. For experiments with smaller request sizes, we observe that network-level purging is important because most of the bytes can be inside the queues of switches. Specifically, for a 10KB request size, network level purging provides an

additional reduction of 10% in the average and nearly 20% in the 99th percentile request completion time compared to only transport layer purging.

5 Related Work

Transport. Several recent systems (e.g., [22, 8]) use multiple priority queues inside the network, where flows are mapped to different priorities based on their size, deadline, etc. So all flows irrespective of their priority are required by the application. In case of RANS, a flow may have multiple duplicate flows, and only one of them needs to finish. Recently, Mittal et al. use lower priority queues to transmit extra data – above and beyond what is allowed by TCP congestion control [21]. This is an example of *intra-flow* redundancy whereas in our case the duplicate data belongs to different flows with independent control laws. RepFlow [40] considers replicating certain flows (but at the same priority) while Vulimiri et al. [38] conduct some preliminary experiments using redundant flows with lower priority but do not consider the full set of features we propose in RANS (e.g., purging and aggregating bytes). Finally, multipath TCP sets up multiple sub-flows between the same source-destination pair whereas our transport is point-to-multipoint [27]. All these differences create new challenges and opportunities for RANS.

Applications. Recent work considers intelligent scheduling or duplicate/speculative execution for improved application performance. The scheduling techniques consider application level semantics, such as the relationship between flows, considering the deadlines of the tasks, and so on [4, 15, 22, 8, 32]. Our work on duplicate-aware scheduling is complementary to these techniques – within the primary request queue we can consider scheduling options other than FIFO. Speculative-aware techniques make use of duplicate requests in an intelligent fashion, so as not to issue too many duplicate requests [30, 7]. Our goal is to make the overhead of duplicates so low that we do not need to be speculative – we can treat duplicates as a first-order primitive.

Interface. Several proposals have considered high level interfaces based on `put()` and `get()` [26, 14, 18]. There are also proposals that capture the application semantics for storage-based applications [37, 35, 36]. These proposals do not consider redundancy as a first-order primitive. A preliminary proposal for capturing the redundancy is through replication topology [37] which captures the relationship between compute and storage nodes, including replicas and the semantics.

Scheduling Theory. There is a rich body of work on scheduling theory that is relevant to duplicate-aware scheduling. The recently proposed redundancy models are the closest to our setting [31, 16, 38]. They consider redundant requests that share the same queue with the primary requests whereas we consider isolating the duplicate requests into a separate lower priority queue.

6 Discussion

Redundancy-based Storage Systems. Quorum-based and redundancy-based storage systems can also naturally deal with stragglers [28, 25], but they incur high reconstruction cost because they need to make more requests than required. Such systems can potentially use RANS to reduce the overhead. For example, K out of N requests can be high priority while the other $N - K$ can be lower priority.

Non-Network Resources. In addition to the network, we also need to make the applications and storage system aware of duplicates. First, for most of these applications, there is no single point of queueing. Typically these applications are designed as a set of stages and queueing can happen in any of these stages [41, 39, 36]. We need to first identify all the stages in these applications and for each stage quantify its contribution towards the overall response times. Similar to IOFlow [36], we can modify only those stages that are critical and leave the other stages untouched or with as few modifications as necessary. Second, for some applications, preemption is difficult to achieve. If a request is being processed, it is difficult to move on to a different request without finishing the first one. For such applications, we can consider breaking up the request in smaller parts [24]. The goal would be to go to a smaller unit for which preemption is not required (as is the case for the network where individual packets need not be preempted).

Energy Considerations. While making additional requests in RANS may increase the energy consumption, there are two factors which can limit this overhead or may even reduce the overall energy consumption: i) we use purging and byte-aggregation, so the overall work done by the system (in terms of executing a request) may not necessarily increase¹, and ii) the overall reduction in response time imply that requests stay in the system for less time, possibly consuming fewer system resources and thus improving the energy consumption.

7 Conclusions

In this paper, we take the position that duplicate requests should be a key network mechanism for data center stacks. We sketch the design of RANS and highlight that there are rich opportunities (and challenges) at different layers of the stack, from the interface between the application and the network, to transport and network layer mechanisms. Our preliminary evaluation shows that the benefits of duplicate-aware scheduling are promising, but we also need to solve some key challenges in order to get these gains.

Acknowledgements

We thank Owais Ahmad, the HotNets reviewers, and our shepherd James Mickens, for useful feedback on this work. This work is partially supported by NSF CNS award 1618321.

¹The additional queue management overhead of RANS is likely to be negligible in comparison with the overhead of executing a request.

References

- [1] Datacenter Bridging. <http://tinyurl.com/hkrvvyk>.
- [2] Apache Hadoop. <https://hadoop.apache.org/>.
- [3] Switch Datasheet. <http://www.pica8.com/documents/pica8-datasheet-picos.pdf>.
- [4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proc. ACM SIGCOMM*, 2010.
- [5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proc. USENIX OSDI*, 2010.
- [6] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proc. USENIX NSDI*, 2013.
- [7] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. In *Proc. USENIX NSDI*, 2014.
- [8] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *Proc. USENIX NSDI*, 2015.
- [9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wal-lach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
- [10] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. ACM SIGCOMM*, 2013.
- [11] D. D. Clark and D. L. Tennenhouse. Architectural consid-erations for a new generation of protocols. In *Proc. ACM SIGCOMM*, 1990.
- [12] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.
- [13] J. Dean and S. Ghemawat. Mapreduce: simplified data pro-cessing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
- [14] F. R. Dogar and P. Steenkiste. Architecting for Edge Diversity: Supporting Rich Services Over an Unbundled Transport. In *Proc. ACM CoNext*, 2012.
- [15] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. De-centralized Task-aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.
- [16] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, E. Hyttiä, and A. Scheller-Wolf. Reducing latency via re-dundant requests: Exact analysis. In *Proc. ACM Sigmetrics*, 2015.
- [17] V. Gupta, K. Sigman, M. Harchol-Balter, and W. Whitt. In-sensitivity for ps server farms with jsq routing. *SIGMETRICS Perform. Eval. Rev.*, 35(2):24–26, 2007.
- [18] D. Han, A. Anand, F. R. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, et al. XIA: Efficient Support for Evolvable Internetworking. In *Proc. USENIX NSDI*, 2012.
- [19] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proc. ACM SoCC*, 2012.
- [20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proc. ACM SoCC*, 2012.
- [21] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively cautious congestion control. In *Proc. USENIX NSDI*, 2014.
- [22] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.
- [23] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proc. USENIX NSDI*, 2013.
- [24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proc. ACM SOSP*, 2013.
- [25] G. Parisi, T. Moncaster, A. Madhavapeddy, and J. Crowcroft. Trevi: Watering down storage hotspots with cool fountain codes. In *Proc. ACM HotNets*, 2013.
- [26] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th USENIX NSDI*, Apr. 2007.
- [27] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duch-ene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proc. USENIX NSDI*, 2012.
- [28] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. ACM SIGCOMM*, 2014.
- [29] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. ACM SoCC*, 2012.
- [30] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hop-per: Decentralized speculation-aware cluster scheduling at scale. In *Proc. ACM SIGCOMM*, 2015.
- [31] N. B. Shah, K. Lee, and K. Ramchandran. The MDS queue. *CoRR*, abs/1211.5405, 2012.
- [32] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. Towards programmable packet scheduling. In *Proc. ACM HotNets*, 2015.
- [33] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical dcb for improved data center networks. In *Proc. IEEE INFOCOM*, 2014.
- [34] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. USENIX NSDI*, 2015.
- [35] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. ACM SOSP*, 2013.
- [36] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proc. ACM SOSP*, 2013.
- [37] B. Tiwana, M. Balakrishnan, M. K. Aguilera, H. Ballani, and Z. M. Mao. Location, location, location!: modeling data proximity in the cloud. In *Proc. ACM HotNets*, 2010.
- [38] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proc. ACM CoNext*, 2013.
- [39] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.
- [40] H. Xu and B. Li. Repflow: Minimizing flow completion times with replicated flows in data centers. In *IEEE Conference on Computer Communications, INFOCOM 2014*, pages 1581–1589, 2014.
- [41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mc-Cauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX NSDI*, 2012.