Keddah: Network Evaluation Powered by Simulating Distributed Application Traffic

JIE DENG, GARETH TYSON, FELIX CUADRADO, and STEVE UHLIG,
School of Electronic Engineering and Computer Science, Queen Mary University of London, UK

As a distributed system, Hadoop heavily relies on the network to complete data-processing jobs. While the traffic generated by Hadoop jobs is critical for job execution performance, the actual behaviour of Hadoop network traffic is still poorly understood. This lack of understanding greatly complicates research relying on Hadoop workloads. In this article, we explore Hadoop traffic through empirical traces. We analyse the generated traffic of multiple types of MapReduce jobs, with varying input sizes, and cluster configuration parameters. We present Keddah, a toolchain for capturing, modelling, and reproducing Hadoop traffic, for use with network simulators to better capture the behaviour of Hadoop. By imitating the Hadoop traffic generation process and considering the YARN resource allocation, Keddah can be used to create Hadoop traffic workloads, enabling reproducible Hadoop research in more realistic scenarios.

CCS Concepts: • **Networks** → **Network simulations**; *Network measurement*; Cloud computing; Data center networks;

Additional Key Words and Phrases: Hadoop network traffic, network traffic simulation, network behaviour of distributed applications

ACM Reference format:
Jie Deng, Gareth Tyson, Felix Cuadrado, and Steve Uhlig. 2019. Keddah: Network Evaluation Powered by Simulating Distributed Application Traffic. *ACM Trans. Model. Comput. Simul.* 29, 3, Article 16 (June 2019), 25 pages.
https://doi.org/10.1145/3301503

1 INTRODUCTION

Hadoop and related automated parallel data-processing frameworks have become the standard for large-scale data computation. While these platforms abstract developers from the complexity of parallel operations, it is challenging to ensure they achieve good execution performance. Most of the research effort has focused on the application side of Hadoop, aiming to improve its resource management and job-scheduling capabilities, as well as efficiently interpreting programs written in higher-level languages [18]. However, there has been limited attention to the performance impact of the network in Hadoop jobs, despite benchmarking studies showing that communications constitute a significant part of the overall execution time [27].

In parallel, many networking researchers have been working on designing innovations within data centre networks (where Hadoop typically runs). The nature of intra-data-centre traffic is

Authors' addresses: J. Deng, G. Tyson, F. Cuadrado, and S. Uhlig, School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London E1 4NS, UK; emails: {j.deng, g.tyson, felix.cuadrado, steve.uhlig}@qmul.ac.uk.

1049-3301/2019/06-ART16 $15.00
https://doi.org/10.1145/3301503

fundamentally different to Internet application traffic (e.g., such as web requests [16]) and, consequently, there have been various proposals for novel network topologies [2, 6, 11], protocol modifications [3, 37], active management of queues [10], and adaptive forwarding behaviour through SDN [26] that specifically address data centre needs. This, however, creates a simple problem: How do networks researchers evaluate their designs in a realistic environment? This is not trivial, because even though Big Data applications are one of the main contributors to intra-data-centre traffic, their traffic patterns and workloads are still not well understood. This makes it necessary for researchers to evaluate their network innovations within a real data centre. This, however, is very restrictive and limits research to large organisations that have both the required infrastructure for testing and the appropriate jobs to experiment with. As organisations are typically reticent to share their infrastructure and job workloads, this creates a lock-out whereby external researchers cannot develop and evaluate new data centre technologies.

We therefore argue that it is critical that open-source tools are developed to allow networks researchers to evaluate their network innovations in a *simulated* fashion. As part of this, it is paramount that realistic traffic patterns can be reproduced in simulated environments. Thus, in this article, we present *Keddah*, a tool that allows (*i*) Hadoop traffic to be captured within real clusters; (*ii*) anonymous traffic models to be compiled; and (*iii*) the traffic to be replayed in simulated environments. Collectively, this allows any researcher to evaluate their network-level innovations with realistic traffic patterns. Based on the traffic models, Keddah generates traffic by simulating the complete execution flow between compute nodes and the traffic exchanged between them. Keddah models Hadoop traffic at a flow level including the traffic volume, hosts involved, and sessions for each protocol. We have built Keddah as an open-source tool[1] on top of a network simulator (ns3) and integrated publicly available Big Data workloads [5] for network research purposes. We have included various models for immediate use, but also hope that operators using Hadoop will use Keddah to capture and share their own traffic patterns. As well as presenting Keddah, we shed light on the nature of Hadoop traffic, exploring how it is impacted by various job and cluster settings.

This article is organized as follows:

- Section 2 describes the main elements of the Hadoop platform, with an emphasis on network communication between components.
- In Section 3, we outline the architecture of Keddah and specifically illustrate each component in the later sections.
- Section 4 details our methodology to capture Hadoop traffic and extract its distribution. We explore the impact of varying several parameters, including job types, job parameters, and cluster settings.
- Section 5 presents the implementation of Keddah, as well as how it is integrated with ns3.
- Section 6 discusses our validation experiments where we compare Keddah's generated traffic with workloads captured in real environments.
- Section 7 presents and evaluates two example applications of Keddah for data-centre network research. We show the effect of multiple data-centre topologies and transport-level protocols in the performance of Hadoop jobs.
- Section 8 discusses related works in this area.
- Section 9 concludes the article, highlighting remaining lines of future work.

[1] github.com/deng113jie/keddah.

2 BACKGROUND

In this section, we look into how Hadoop traffic is generated during job execution and provide a first characterisation of its nature. We first explain how Hadoop works, before inspecting the types of traffic components that Hadoop generates and diving into individual flow behaviours.

2.1 How Hadoop Works

Hadoop MapReduce is a system designed to support automated parallel data processing across multiple compute servers. It consists of a distributed storage system (the Hadoop Distributed File System—HDFS) and a distributed execution framework called YARN [35]. MapReduce defines a two-stage execution flow where a mapping process is followed by a reduction process. Both mapper and reducer tasks are allocated to Linux containers during task assignment. Jobs start with the YARN Resource Manager allocating containers to nodes. If jobs require input data that needs to be processed by mappers, then the allocation will try to assign containers onto nodes where the data split is stored locally (in the HDFS). Figure 1 presents the structure of a complete Hadoop job containing multiple mappers and reducers. Map containers read the appropriate data from HDFS. Once each mapper has retrieved its allocated data blocks, it begins to execute the computation (e.g., sorting). Results are partitioned by key, with reducers aggregating all emitted values belonging to the same key. Thus, results from mappers are then transferred during the shuffle step to the reducer containers (using HTTP). Each reducer node computes results for each key, and once all computation has been completed, the reducer writes the result back to HDFS. This means that the Hadoop application generates a number of traffic components during job execution.

2.2 Hadoop Traffic Composition

As mentioned above, Hadoop generates several types of traffic, including HDFS control messages, HDFS data transmissions, Shuffle data transfers, and YARN control messages (e.g., keep alives). Hence, it is necessary to understand the importance of these several traffic components. We set up Hadoop clusters with 16 physical work nodes and a master node. We use the default cluster settings (as listed in Table 1) to observe common traffic behaviour.

To capture a range of traffic, we sample a diversity of Hadoop jobs, including machine learning, graph algorithms, and scientific computation. We focus on three types of jobs [22, 30], taken from the benchmarking tool HiBench [13]. The jobs are (*i*) *TeraSort*: the TPCx benchmarking job for Hadoop clusters [25]; (*ii*) *PageRank*: a graph-processing operation for calculating the weight between vertices [17]; and (*iii*) *kmeans*: one of the most popular data-clustering algorithms [1]. We considered other types of jobs in our initial experiments (e.g., word count, join queries, Bayes) and found the selected three provide a good sample of different types of MapReduce computation, as they contain most traffic types. For all jobs, we record the traffic exchanged using sflow.

Using the sflow records, we start by splitting the traffic into its individual types (using the port numbers). Table 2 presents the breakdown of traffic for each type of job. We found the traffic portion is consistent for each job given various job-input datasets. From both virtual and physical clusters, the amount of control plane traffic is negligible, with in excess of 99% of traffic coming from HDFS data transfers and Shuffle. Hence, control traffic can be ignored for network traffic generation, as its impact is negligible. However, for data plane traffic, the diversity across jobs is significant. For example, we observe no Shuffle traffic in map-only jobs such as Bayes and Kmeans compared to 90% in TeraSort (seen in Table 2). Moreover, the PageRank job generates just 1% of its traffic from HDFS read, compared to 54% for the Bayes job. No shuffle traffic can be seen if it is a map-only job (e.g., Bayes and Kmeans). Further, in cases where there is no HDFS replications set, HDFS write traffic is eliminated. Overall, it can be seen that the proportions (and amounts) of traffic generated by these three key protocols is largely based on the job type and dataset.

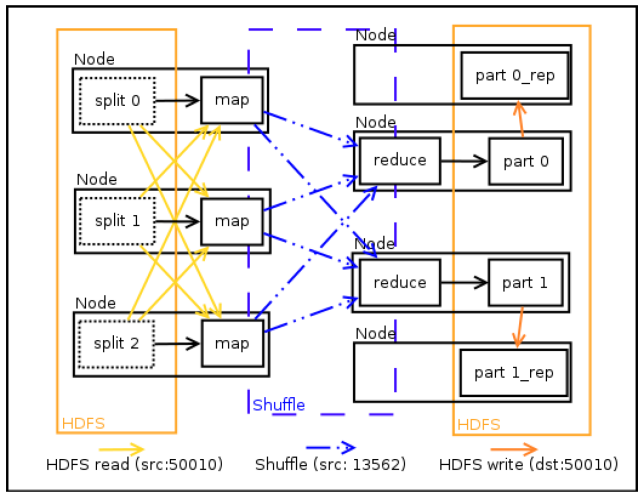


Fig. 1. MapReduce data-processing paradigm.

Table 1. Default Hadoop Settings

Name	Value
data replications	3
cluster size	16
block size	128MB
input split	128MB
number of reducers	2
output replication	1

Table 2. Traffic Composition of Hadoop Jobs

Job	**HDFS read**	**HDFS write**	**Shuffle**
sort	0.01	0.7	0.29
wordcount	0.03	0.66	0.31
TeraSort	0.1	0	0.9
Bayes	0.54	0.46	0
Kmeans(itr-1)	0.31	0.67	0
Kmeans(itr-2)	0.07	0.82	0
PageRank(itr-1)	0.10	0.66	0.23
PageRank(itr-2)	0.25	0.09	0.66
Hive Join	1.0	0.0	0.0
Hive Aggregation	0.98	0.02	0.0

It is also possible to inspect how each of these traffic components is generated across the multiple stages of a job. By definition, each stage occurs sequentially: *HDFS Read* loads the data onto mapper nodes, *Shuffle* exchanges data between the mapper/reducer nodes. Finally, *HDFS Write* stores the result. Due to space constraints, we focus on how these traffic components are generated in three example jobs (Kmeans, TeraSort, and PageRank). Figure 2 presents the amount of traffic generated

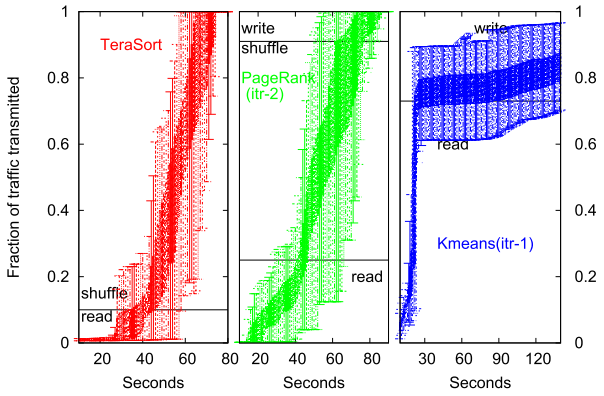


Fig. 2. Fraction of traffic transmitted over time for TeraSort, PageRank, and Kmeans. The horizontal lines demarcate HDFS Read, Shuffle, and HDFS Write traffic. (Note: the TeraSort is HDFS read and shuffle only; the tail of Kmeans is truncated for plotting.)

per second on the physical cluster for: the second iteration of PageRank jobs (with 10M vertices), TeraSort jobs with 6GB data, and the first iteration of Kmeans jobs with 14 clusters and 14M samples per cluster. The amount at each second is shown as a box plot, with data taken from 400 runs.

It can be seen that each job exhibits broadly similar trends, but with key differences. These differences emerge from the varying ratio between the three core traffic components. For instance, TeraSort's first 10% of traffic is HDFS Read, followed by the remaining 90% that corresponds to Shuffle traffic. Similarly for PageRank, there is 25% HDFS Read at the beginning, then 66% Shuffle, and finally the last 9% is HDFS Write. All traffic seen on the network follows this three-stage sequence. Clearly, this stability in the structure gives a clear guidance that HDFS read, shuffle, and HDFS write traffic needs to be investigated separately to get a precise reproduction of the Hadoop traffic. Importantly, this must be performed on a *per-job* basis to reflect the diversity observed.

3 KEDDAH ARCHITECTURE OVERVIEW

This article exploits the above observations to build a tool that can capture and replay Hadoop traffic. *Keddah* offers a full workflow that allows users to (*i*) capture traffic from a real cluster; (*ii*) model it; (*iii*) replay it within a simulation environment. Importantly, this does not involve exporting real traffic traces—instead, Keddah generates compact traffic models that represent the traffic seen. The workflow is shown in Figure 3. On the left, traffic is captured from a real cluster (as shown in Section 4). Following this, the empirical traffic distributions are computed (as shown in Section 4.2). The workflow then exports the distributions into a separate *Keddah file* that can be used to replay the traffic in a simulation environment (as shown in Section 5).

A key goal of this workflow is to allow *any* Hadoop operator to capture their traffic and share it with the community for replay. Importantly, by converting traffic into the appropriate distributions, we ensure anonymity and minimise exposure of sensitive information regarding a particular Hadoop operator. The rest of this article will outline the various components of this workflow.

4 MODELLING HADOOP TRAFFIC FOR REPLAY

The first stage in the Keddah workflow is the capturing and modelling of Hadoop traffic from real environments. This is then used to generate a *Keddah file*, which contains the necessary parameters to replay the traffic within a simulation. In this section, we start with details of how we capture the

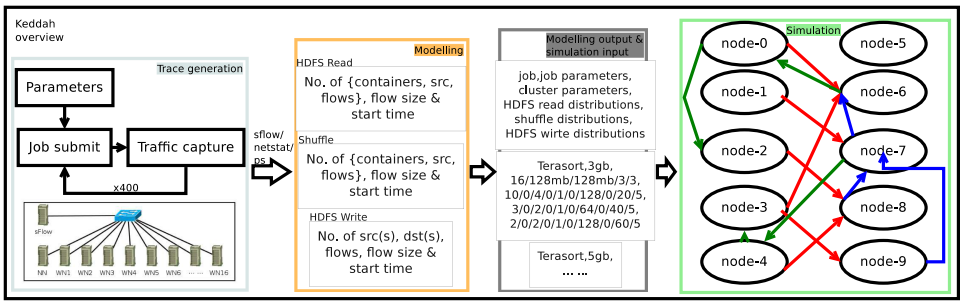


Fig. 3. Architecture of Keddah, starting with execution of real jobs and ending in the generation of the traffic in a simulator.

Hadoop traffic and extract the necessary distributions—first under default settings and then using various jobs, job settings, and cluster settings. The purpose is to then prime our later re-generation of synthetic Hadoop traffic.

4.1 Capturing Hadoop Traffic

To explore the network patterns of Hadoop, it is first necessary to measure its live behaviour. For this, we use well-known benchmarking tools to execute a number of different jobs on a real Hadoop cluster. Packet traces are then collected across the cluster to study, characterise, and model the traffic. Our methodology allows us to sample the network traffic over a set of variables, including cluster settings (number of replications, HDFS block size, cluster size), type of jobs, and job settings (output replication, number of reducers, and job-specific parameters).

To fully capture the Hadoop traffic, we need to map the traffic to each of the containers to acquire a fine granularity about the traffic. Thus, at each of the machines in the cluster, we first collect all the network traffic via sflow with a sample rate of 1. Then, we regularly collect processes and flow information from the operating system on each machine and cross-reference them with the sflow collected. In the end, we are able to map the traffic to the origin process, source node and port, destination node and port, process, and, finally, flow size and duration.

After the traffic is collected, we aim to profile the traffic at the *application level*. This is because we wish to avoid modelling lower-layer factors, thereby freeing up researchers to innovate in layers 2, 3, and 4 while using our synthetic application traffic patterns. Normally, traffic is described at the network level by five tuples: the protocol, source/destination address, and source/destination port. However, more parameters are required to capture the Hadoop traffic when taking the application level behaviour into consideration. Thus, we propose a new set of parameters in an attempt to best capture Hadoop traffic, agnostic to the underlying network features (e.g., transport protocol, topology).

For each traffic component (HDFS read, write, and shuffle), the traffic is described by five parameters: the number of containers, number of source nodes per container, number of flows per source, flow size, and flow start time. We chose this approach because the five-tuple is the minimum required to identify flow-level behaviour. As we wished to achieve high scalability, we wanted to operate on this flow-level to avoid the complexities and overheads introduced by packet-level models. The model is therefore the most lightweight data structure for recording traffic. To aggregate flows into a single model that could be used for traffic replay, we recorded the number of sources, number of destinations, and number of flows to record. This is sufficient for regenerating the number of flows in the simulator. However, it does not capture the size or timings of the flow. Hence, it was necessary to supplement these three parameters with flow

size and flow start times. Collectively, these five parameters are the minimum required to replay flow-level activities in the simulator, thus giving us accuracy when replaying the traffic. The same set of parameters is captured for each component, including HDFS read, shuffle, and HDFS write.

Following this, we then extract the full distribution of parameter values (e.g., flow sizes) across a number of job runs (400 for each parameter set by default), i.e., we repeat the same job 400 times. This is necessary because the traffic generated are non-deterministic and vary per run. Our experiments show that 400 runs gives us enough samples to construct the distribution [7]. And by quantifying the parameters via distribution, we can later compare the parameters from various settings and even extrapolate the parameters in the future. Each run may vary from 2mins to 30mins, depending on the job type and job parameters. In total, we have captured traffic of 2,000 running hours from our cluster, consisting of various jobs, cluster parameters, and job parameters.

As Keddah primarily performs traffic replay-like functionality, we assume that the simulated cluster is broadly similar to the live cluster where the initial traffic traces were captured. For tractability, when capturing the traffic, we only focus on the parameters that people often change when initialising their cluster. That said, it is possible to record various different settings in the live cluster and replay them in the simulator—we make our simulator open-source to allow others to experiment with these possibilities.

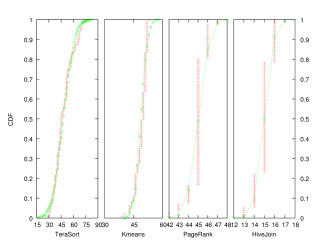
4.2 Extracting the Traffic Distributions

The previous section shows how we capture traffic from a live cluster. We next explore how to model the Hadoop traffic, so it can be later replayed within a simulator. Here, we run TeraSort, PageRank, Kmeans, and Hive Join with the default cluster settings listed in Table 1. We emphasise that our methodology works across *any* job; we use these three simply as exemplars.

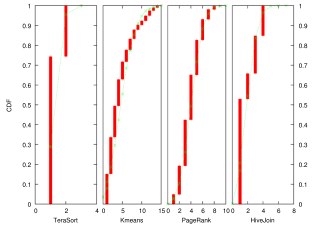
The approach taken to traffic modelling is straightforward. We collect the empirical traffic over 400 runs and, for each run, capture the parameters listed in the previous subsection (e.g., flow sizes observed). We then perform a maximum-likelihood fitting of univariate distributions, evaluate the fitting by using the Kolmogorov-Smirnov test (ks.test), and then select the distribution with the highest p-value. Figure 4 presents the CDF of empirical samples (red) and fitting results (green) for each of the parameters that we used to describe the Hadoop traffic. The shuffle part has only TeraSort and PageRank as Kmeans, as Hive jobs do not have shuffle traffic. These plots give us an intuitive visualization of the distribution extracted by comparing the similar of two CDFs; though some of the distributions may not look so similar (Figure 4(g), for example), which is due to the discreteness of real traffic. Once the fitted distributed is used in the simulation, the generated random number will be rounded for a better imitation of real traffic.

For the distributions found, we can see that for the number of containers, number of sources/destinations per containers, number of flows, and flow starting time, all have one thing in common: they follow the Normal distribution. However, for the HDFS flow size, we find two other types of distribution that fit different jobs: for mappers requiring the data from one block only (e.g., TeraSort, Kmeans, and PageRank) the HDFS read flow size is binary; for mappers that require the data from more than one block (e.g., Hive, Join) the fitted distribution is Weibull. Keddah automatically extracts these parametrised distributions using its maximum-likelihood fitting.

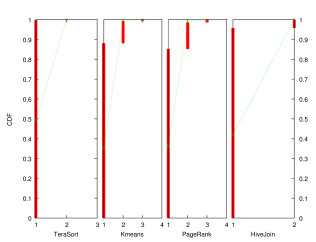
Apart from the visualisation, we list the numerical distribution parameters found for various TeraSort jobs with different job settings in Table 8 (*cf.* Appendix A). This includes the mean and standard deviation for the Normal distribution, the probability for Bernoulli or shape, and scale for the Weibull. The table is the core output of the modelling section mentioned in the Keddah architecture 3.



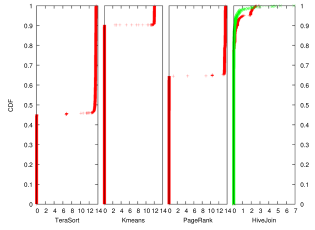
(a) Number of mapper containers.



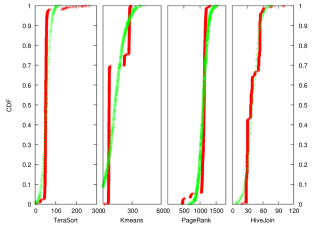
(b) Number of sources per mapper.



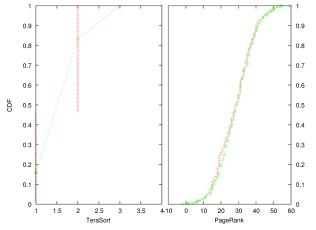
(c) Number of mapper flows.



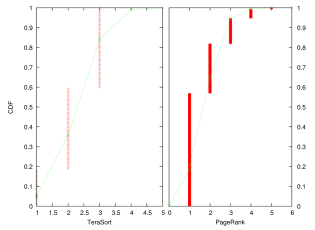
(d) HDFS read flow size(10MB).



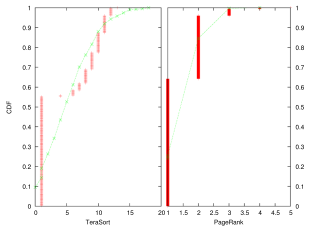
(e) HDFS read flow starting time.



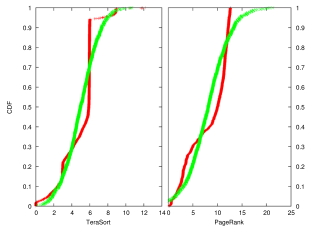
(f) Number of reducer containers.



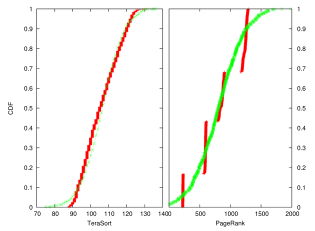
(g) Number of sources per reducer.



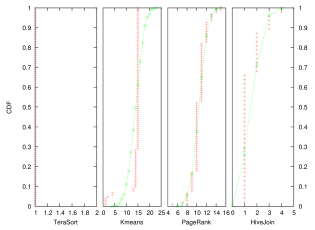
(h) Number of flows per reducer source.



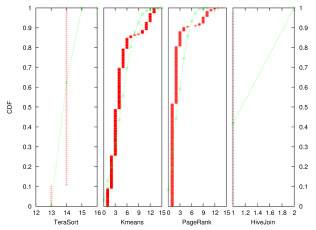
(i) Shuffle flow size(10MB).



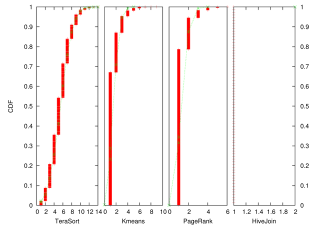
(j) Shuffle flow starting time.



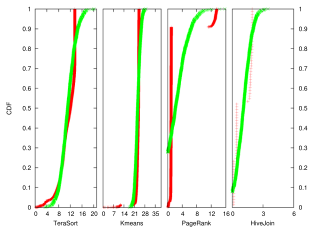
(k) Number of HDFS write source.



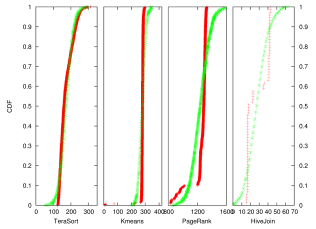
(l) Number of HDFS write destination per source.



(m) Number of flows per HDFS write source destination.



(n) HDFS write flow size(10MB).



(o) HDFS write flows starting time.

Fig. 4. Fitting the traffic distribution over various jobs.

4.3 Modelling Impact of Cluster Setting

The previous subsection has shown the distribution of traffic metrics across a variety of jobs. Importantly, it has used simple best-fitting to extract the probability distributions of these metrics to allow later replay. However, we have also observed that individual cluster settings with Hadoop can heavily impact these values. Hence, it is necessary to capture the impact that these different cluster settings have on the traffic make-up. To achieve this, we repeat the same methodology while varying several key cluster parameters; under each parameter setting, we extract the five-tuple model. This is then embedded within the Keddah file, allowing an experimenter to select the cluster settings they wish to generate traffic for.

We start with a set of parameters that are commonly used in practice to tune cluster performance, namely data replication, cluster size, block size, input split size, number of reducers, and output replications. Clearly, it is not feasible to fully explore this large sample space by executing millions of jobs. Thus, we use the default settings as a baseline (as shown in Table 1), and change only one setting at a time to explore the difference. As well as executing these tests to extract the models for the Keddah file, we explore the impact the differing settings have to better elucidate the nature of Hadoop traffic.

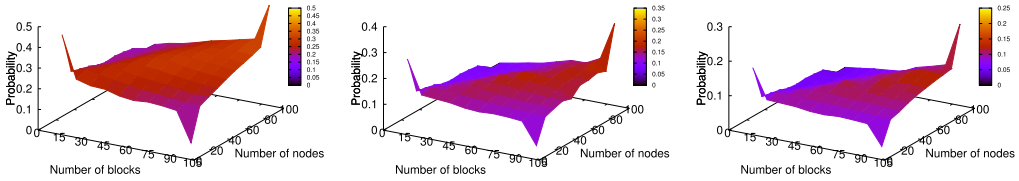
4.3.1 Replication and Cluster Size. Among the most important decisions made when operating a Hadoop cluster are the *replication rate* (i.e., number of data replicas) and *cluster size* (i.e., number of nodes). When data is stored on HDFS, the data is split into file blocks. Each block may be stored on multiple nodes for fault tolerance; this is controlled by the replication setting. With more replicas (or fewer work nodes in the cluster), each node will clearly have to store a greater number of blocks. As we have discussed previously, the mapper reads a block at the beginning of execution, and traffic will be generated if the block is not locally stored.

In essence, the above can be modelled using the probability by which a mapper will have a local copy of the required block. If a local copy is not available, then traffic will be generated. To understand this probability, we have extracted the algorithm used for replica management from the Hadoop source code. We find that blocks are randomly assigned to nodes within the cluster. Following this, when a mapper container is assigned to a node, it is, by preference, allocated to the node where the data is already stored. However, this does not always happen. Using Hadoop's source code, we have extracted the assignment algorithm[2] such that we can predict whether or not a mapper container will be placed on the same node as its data.

Rather than exploring the impact of replication by executing real jobs, we simply use this (deterministic) assignment algorithm, which produces the probability by which a mapper will be co-located with its data. To compute this, it takes four input parameters: number of nodes, number of blocks, number of replications, and number of mappers. The algorithm first randomly assigns blocks on each node. Note the total number of blocks will be the number of blocks input multiplied by the number of replicas. The algorithm then starts to assign the mappers to each node. It then simply counts the number of mappers running on a node without the requested block stored. Finally, it outputs the counted number divided by the total number of mappers: this is the probability of have traffic generated during the mapper process. We find that the empirical probability that a mapper reads a remote block follows the Bernoulli distribution (parameters listed in Table 8).

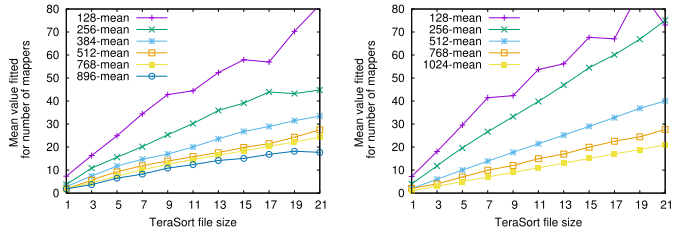
To confirm the above is correct, we compare our predicted results against the behaviour of our real Hadoop cluster. For each instance, we emulate the process 1,000 times to find the mean value. Figure 5 shows the value found for various numbers of blocks and numbers of nodes combinations under different replication settings. Not surprisingly, we found the same result from

[2] https://github.com/deng113jie/keddah/locationemu.



(a) Cluster replication set as 1. (b) Cluster replication set as 2. (c) Cluster replication set as 3.

Fig. 5. The probability that a mapper requires a block that is not locally stored. Varies over cluster replication setting, number of nodes, and number of blocks required to store the file. Results obtained using the algorithm taken from Hadoop source code.



(a) Number of map containers found over various block sizes. (b) Number of map containers found over various input splits.

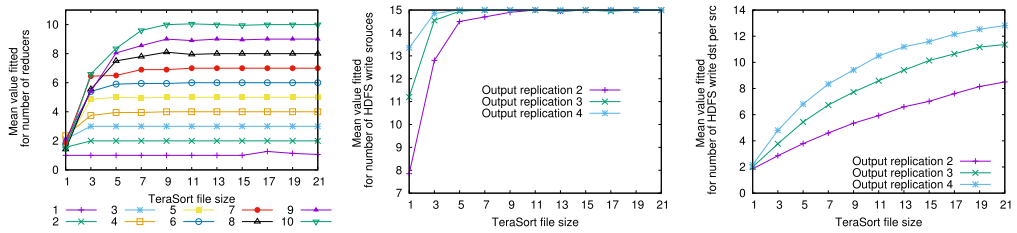
Fig. 6. Traffic variation over block size and input split.

running empirical traces in our real cluster. We can see that the probability varies over different combinations but also decreases with more replication set in general.

4.3.2 Block Size and Input Split. The block size decides how many blocks each file is fragmented into when stored in HDFS. The input split size is the data file size that each mapper processes. By default, the block size is treated as the input split size. Next, we run jobs over various block and input split sizes and see how the traffic is affected. Each run results in a new five-tuple model, which is included in the Keddah file (allowing people wishing to replay the traffic to select their preferred block and input size).

To explore how these parameters impact traffic generation, Figure 6 plots the number of map containers used when experimenting with various block sizes. The Y-axis shows the fitted mean to aggregate the samples. When we alter the block size or input split size, the number of mappers decreases linearly. This is expected, as the change of block size will reduce the number of blocks stored, thus the mapper has a higher chance of running on a node with blocks locally stored. The input split size will reduce the number of mappers directly, thus the number of mappers found from the traffic wise will be decreased accordingly.

4.3.3 Number of Reducers. Another key metric is the number of reducers, which will clearly affect the reduce phase of MapReduce. The number of reducers can be controlled with mapreduce.job.reduces setting. Hence, we vary this parameter and generate the five-tuple model for each parameter setup. We find that the number of reducers is affected by such a configuration when running TeraSort, PageRank, and Hive Join jobs. Surprisingly, this is not always the case, though. Figure 7(a) presents the fitted mean value found for the number of reducers from the 400 empirical runs against various TeraSort job parameters. It shows that the number of reducers increases with the setting, but not when the input data file size is small. For example, even though it is set



(a) The number of reducers containers found, varies over the number of reducers setting.

(b) The number of HDFS write traffic sources affected by various output replications.

(c) The number of HDFS write traffic destinations affected by various output replications.

Fig. 7. Traffic variation over number of reducers and output replications.

Table 3. Hadoop Traffic Distributions

	Number of containers	Number of srcs per container	Number of flows per source	Flow size	Start time
HDFS Read	Normal	Normal	Normal	Bernoulli/Weibull	Normal
Shuffle	Normal	Normal	Normal	Normal	Normal
HDFS Write	Normal	Normal	Normal	Block size	Sequential

to 3, only 1 reducer is generated with TeraSort 1G job. Hence, the mapreduce.job.reduces setting constitutes a limit, not a lower-bound. By executing and capturing jobs with these different settings, we automatically capture the number of reducers seen in this Keddah file.

4.3.4 Output Replication. The final parameter we inspect is the output replication level; this dictates the number of replicas stored for the output of the job. Clearly, this will affect the amount of HDFS write traffic. Again, we execute a number of jobs while varying the replication setting. In each case, we record and model the traffic for storage within the Keddah file.

Figure 7(b) and Figure 7(c) show the variations by presenting the mean value fitted for number of HDFS write traffic sources and number of HDFS write traffic destinations per source. We find that this increases the number of source and destinations, as increasing the replica degree by one with naturally result in one more source-destination pair. That said, the distribution of other parameters such as flows per source-destination pair does not change. Again, we can see that the value increases linearly with yields to the total number of nodes in the cluster.

4.4 Summary

In this section, we have empirically captured, modelled, and explored the nature of Hadoop traffic. The purpose behind this has been to underpin the recreation of synthetic Hadoop traffic. Using a variety of jobs and Hadoop parameters, we have quantified the resulting traffic that is generated. For each parameter under consideration, we have investigated the distribution of its values across 400 runs to capture its properties. A summary of all measurements and fittings can be found in Table 3.

This entire process is embedded within the Keddah toolkit, allowing Hadoop operators to capture and model traffic. Importantly, the toolkit generates all job configurations such that multiple parameters can be experimented with. The output of this process is a Keddah file: a configuration file containing five-tuple descriptions of each traffic type observed (HDFS Read, Shuffle, HDFS write). This set of 15 parameters is stored for each cluster setting that has been recorded, e.g., for

each replication level. By doing this, the Keddah file can be shared with experimenters who wish to replay the same traffic in a simulated environment.

5 SIMULATING HADOOP TRAFFIC USING KEDDAH: AN IMPLEMENTATION PERSPECTIVE

The previous section has captured and computed empirical models for the distribution of various key traffic metrics. These are embedded within a Keddah file that can be publicly shared with others wishing to replay the traffic. In this section, we implement the last part of Keddah, which uses a Keddah file to replay traffic within a simulated environment. Keddah is built on top of ns3, so the traffic can be reproduced while experimenting with different network possibilities (e.g., topology, routing protocols). Note that this is limited to explorations *below* the application layer, such that the application layer behaviour does not vary.

5.1 Implementation within ns3

5.1.1 Implementation Overview. The previous section has detailed the first stage in the Keddah workflow (*cf.* Figure 3), where real traffic is captured and modelled to create a Keddah file. The second stage is therefore the use of this file to *replay* traffic in a simulated environment. Thus, Keddah also consists of a series of new ns3 software classes that represent the entire set of simulated Hadoop functionality, e.g., mappers, reducers, and the YARN scheduler. Although we have chosen ns3, due to its wide availability, Keddah could be implemented in any open-source simulator. Figure 8 shows the class diagram of the Keddah implementation. The Keddah ns3 plug-in has three main components that represent the cluster topology and end hosts (green and blue), the Hadoop scheduler (black), and the specific Hadoop jobs (orange). Each component consists of multiple classes, represented by their different colors. The arrows show the calling relations between classes. The rest of this section outlines their operation.

When running the simulation, users can choose from different cluster topologies and Hadoop jobs (just as Hadoop works in reality). Once given a certain topology and job, Keddah starts by building the cluster with a set of nodes interconnected by the stipulated topology (within ns3). Following this, then jobs are also created, which involves request placement on each node based on its available CPU and memory resources. Note that the placement is actually modelled as a container (i.e., one physical node in the simulation can host multiple containers running mappers or reducers). Once the resources are allocated by the simulated YARN class, the containers are executed and they begin to generate traffic.

5.1.2 Implementing the Cluster. When the simulation starts, it is first necessary to simulate the underlying cluster (data centre) topology and attach the "physical" nodes at the appropriate locations. As Keddah is built within ns3, it is not necessary to implement most functionality involved in this, e.g., TCP/IP. Here, we only detail the ns3 components used to construct Keddah. The *Cluster* contains both the nodes (as an *ns3::NodeContainer*) and the *DCTopology*, which defines how the nodes are connected. DCTopology is a parent class that must be extended by specific classes for each type of topology. As can be seen in Figure 8, we have implemented a number of example topologies already: *Star*, *FatTree*, *CamCube,* and *DCell*. Other topologies can be implemented by following the same procedure. Each DCTopology class is expected to implement its own topology within the *setup()* method. For example, FatTree class defines the nodes, their IP addresses, and their interconnections via its setup() method.

5.1.3 Implementing YARN Resource Allocation. YARN is the component that manages the Hadoop cluster, i.e., allocation of containers to nodes. We separate YARN's responsibilities into two classes, shown in Figure 8: *YARN* schedules the job submissions and *YarnScheduler* handles the resource allocation for scheduled jobs. The *YarnScheduler* is the equivalent of the Resource

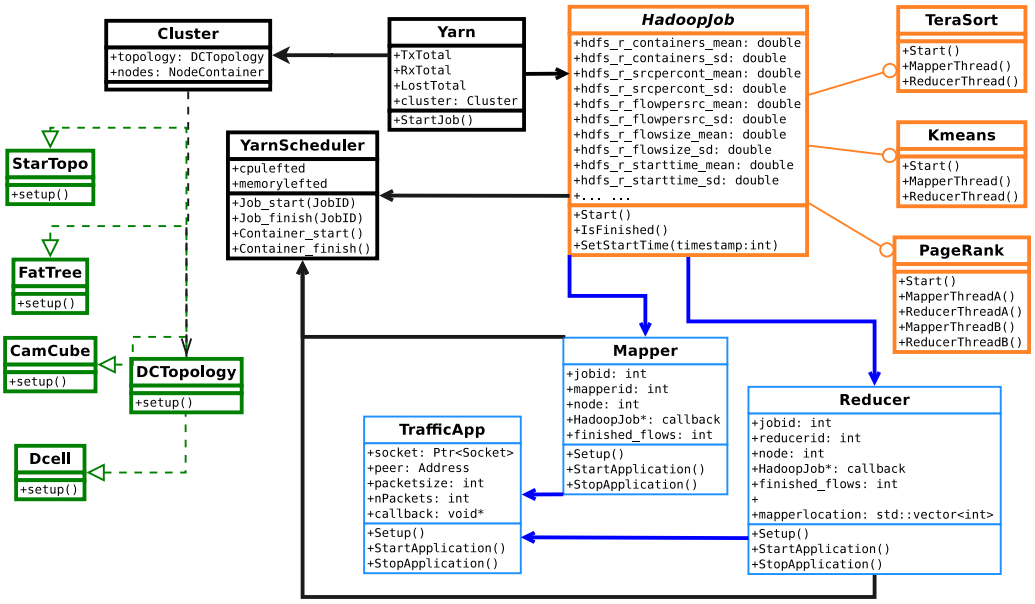


Fig. 8. Main classes implemented in Keddah, including cluster (green), YARN (black), Hadoop jobs (orange), and containers (blue).

Manager in Hadoop: it assigns the CPU and memory resources to the container when requested, i.e., it allocates mapper and reducer containers to nodes within the simulator. CPU and memory are modelled for each node as a series of units; each node has a certain number of units that are depleted by the placement of a container on the node. The YarnScheduler therefore works as a queue that allocates containers (mappers and reducers) onto nodes that have sufficient resources. Though resource allocation is implemented via various schemes in Hadoop (including FIFO, fair, and capacity), we have implemented a FIFO queue as a proof of concept. We have left the interface available for other schemes to be added. Thus, when jobs are submitted to YARN, it deals with them in a FIFO queue, allocating their mapper and reducer containers to specific nodes in the simulator one-by-one.

5.1.4 Implementing Jobs. The last part is executing the Hadoop jobs, where the traffic is generated. As an abstract class, *HadoopJob* declares the core attributes and methods needed within jobs. HadoopJob is then extended by any specific job class. This is because each job has its own set of distribution parameters according to Table 3. Thus, anybody can write a new job by extending this class.

5.1.5 Implementing Mappers and Reducers. Finally, we follow the ns3 design principle and implement Mapper and Reducer containers each as an *ns3::Application*. Each container is placed on a node within the simulator and consumes a certain number of resource "units"; this is used to limit the number of containers per node. In other words, YARN will only execute jobs and push the containers onto nodes once there is sufficient (modelled) resources to handle the complete job (i.e., all containers). Each container is also equipped with a callback function so the main job thread is notified when the container finishes; this imitates the resource-allocation process of Hadoop in reality. Overall, by implementing the mapper and reducer containers as an *ns3::Application*, we can leverage ns3 to have an independent running process for each container, so they can be simulated

in a parallel fashion. To avoid repeating code within both mapper and reducer classes, we also implement a third class: *TrafficApp*. This is responsible for "orchestrating" all flows generating within the simulator. It reads in the Keddah file and uses its parameters to generate a timeline of flows to be initiated between the containers (*cf.* Section 5.2).

5.2 Simulating Single Jobs

Collectively, the above classes are used to generate traffic within the simulator. Because each job operates in a different fashion, the start method in each job implementation is customised. For example, the TeraSort starts the mapper threads and reducer threads only once, while the PageRank has two calculation stages and each stage requires a new set of mappers and reducers. This highlights the need for different jobs to be implemented separately. Keddah currently has implementations for TeraSort and Kmeans; it is possible for others to implement their own jobs.

Figure 9 shows the traffic generation process in each job, including the job thread (black), mapper thread (green), and reducer thread (yellow). When the job starts in the simulator, the main thread first gets the number of mappers (from the Keddah file's five-tuple model). Note that the Keddah file might contain multiple five-tuple models, each for a different cluster configuration; hence, when running simulations it is necessary to select the preferred configuration. It then starts the individual mapper threads. In the mapper threads, the process asks the *YarnScheduler* for the required resource (i.e., a working node to host itself). Once acquired, the node decrements the appropriate resource unit count from itself (using the *YarnScheduler)* and initiates the container. Note, CPU and memory resources are modelled (on each node) as possessing a certain number of "units" that can be stipulated by the experimenter (e.g., 1GB of memory). Each container consumes a certain number of these units; by default, each container consumes 2xCores and 1GB of memory.

Once the mapper containers have been started, it follows the procedure below to generate the traffic:

(1) The *number of sources* (NS) and *number of flows per source* (NFPS) parameters are loaded (from the Keddah file).
(2) The two parameters are used to parametrise two ns3's Normal Distribution implementation ($\mathcal{N}(NS)$ and $\mathcal{N}(NFPS)$). The implementation contains a GetValue() method, which returns the next value from the distribution. These will be used to extract values from the distributions within the remainder of the simulation for selecting the number of sources each container receives flows from and the number of flows sent by that source.
(3) Each mapper container randomly selects a set of source nodes; the size of the set is taken from $\mathcal{N}(NS)$. Each mapper container then iterates through these nodes to generate a set of flows. For each node, the number of flows to generate is taken from $\mathcal{N}(NFPS)$:
    (a) Each flow is handled by a TrafficApp instance (*cf.* Figure 8) separately, which is initialized by the last two parameters from the Keddah file: flow size (FS) and flow start time (FST). This initialisation is identical to Step 2: the ns3 random number generator is used to parametrise the appropriate distributions, such that values can be extracted following the same pattern seen in the original cluster traces. The random number generated is fed by different seeds based on the simulator system timestamps, thus a more random behaviour is ensured to imitate the real traffic generation. Note that flow start time follows a Normal distribution ($\mathcal{N}(FST)$) and flow size follows either a Normal ($\mathcal{N}(FS)$), Binary ($\mathcal{B}(FS)$), or Weibull ($\mathcal{W}(FS)$) distribution based on job type.
    (b) After initialization, the TrafficApp then triggers a TCP flow from the source to the mapper container at the flow start time. The flow start time is extracted from $\mathcal{N}(FST)$ This flow size is, again, taken from the appropriately parametrised distribution, e.g., ($\mathcal{W}(FS)$).

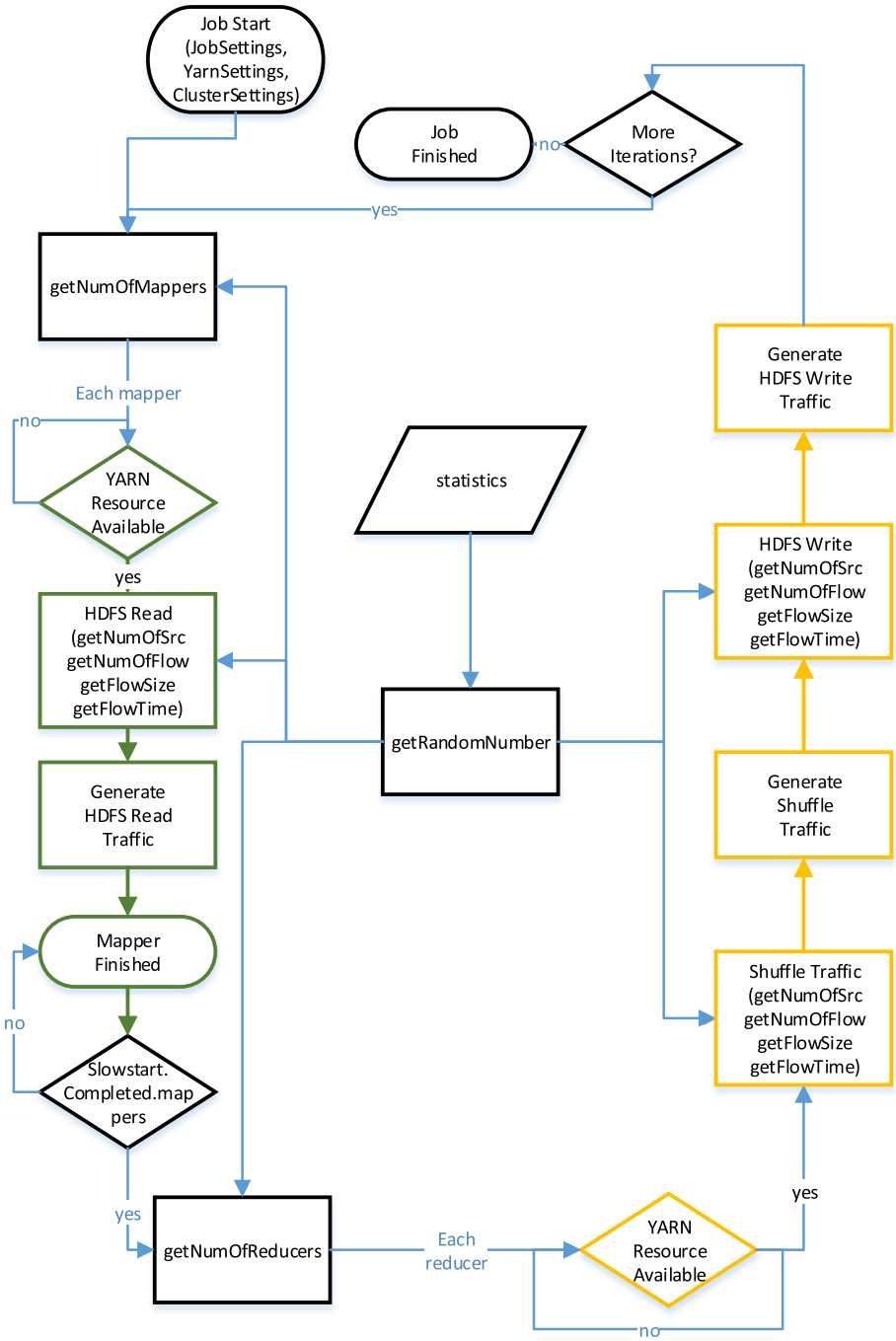


Fig. 9. The flow chart for generated Hadoop traffic in ns3.

After all the mapper containers have been started, the job thread keeps track of running/finished mappers. When the number of completed mappers exceeds the threshold (set by mapreduce.job.reduce.slowstart.completedmaps value in the Keddah source file), the job thread starts the reducers, which the total number of reducers yields to the distribution parameter in the Keddah file. Similar to the mapper, the reducer threads start with requesting resources, then generate shuffle traffic from the container by using TrafficApp as the same procedure as listed before. Finally, the HDFS write traffic is generated by the end of each reducer thread, and resources assigned to the container at the node are returned back to the YarnScheduler for new assignment.

5.3 Simulating Multiple Jobs

Keddah can also support running multiple jobs as a workload (within one simulation). This is typical in clusters, where multiple datasets will be processed in parallel across several jobs. However, simulating multiple nodes and jobs in parallel has always been challenging, due to the difference between parallel processing with the discrete event simulations [9, 21]. We have explicitly built support for running multiple jobs with Keddah.

Simulating the traffic from multiple jobs in Keddah is similar to a single job: We generate the traffic from the application level, so network layer behaviour can be exposed for study. We first tried to integrate OpenMP with ns3 to run jobs in parallel; however, OpenMP complicates the code by specifying the behaviour for each computer core. We then tried pthread; however, the use of pthread in ns3 requires the real-time simulator implementation, and the simulation of Hadoop traffic is often too heavy to be running on real time. So, finally, we come up with a solution of: each job implementation is expected to extend (inherits) from the ns3 *Application* class. Upon executing the simulation, each job will enter a queue and the job is only executed once sufficient resources (i.e., simulated CPU and memory on the nodes) is available. At this point, the next job within the queue will begin execution. This reflects the process by which jobs are queued in a real Hadoop schedule.

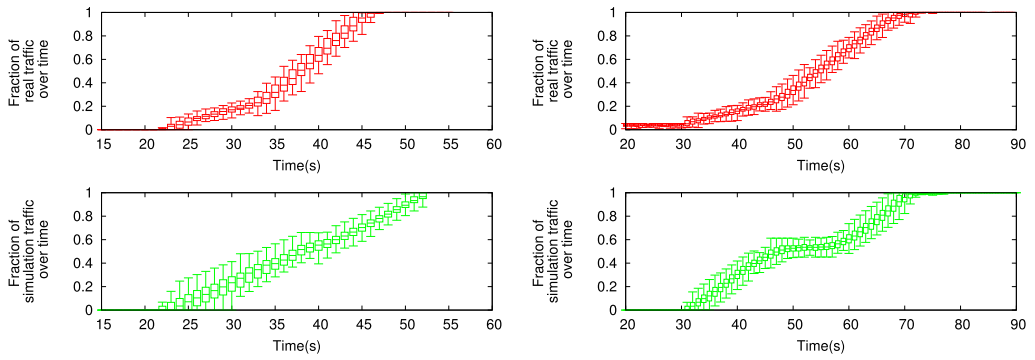
Researchers can stipulate their own arrival rate for the multiple jobs. However, for convenience, we also make available the arrival rates based on the SWIM [5] dataset (which contains multiple jobs). To support multiple jobs, the YarnScheduler we have implemented (which deals with resource management) allows multiple containers from multiple jobs to compete for the nodes within the simulator. YARN will allocate containers to nodes one-by-one from each job in a round-robin fashion based on the resources available. Each job therefore executes as a single job, generating its own traffic within the simulator. We note that Keddah has an API allowing developers to implement any other scheduling mechanism they desire. The traffic is multiplexed within the IP-layer implementation of ns3.

6 VALIDATION

It is important to validate that the traffic generated by Keddah is representative of real traffic. To achieve this, we compare the original traffic against that replayed by Keddah. Ideally, these two sets should be statistically similar across the duration of the job execution. We first compare the traffic of a single job, then extend to a workload consisting of multiple jobs.

6.1 Single Job

To validate the traffic generated, we simulate the topology of our cluster testbed in ns3, using the traffic distributions computed in Section 4.2. However, there is no existing method on comparing Hadoop traffic or the exact similarity of network traffic. We note that cumulative traffic over time is often used in characterising Internet or datacentre traffic. Here, we adopted the idea and used it



(a) TeraSort 3GB. (b) TeraSort 5GB.

Fig. 10. Comparison of the simulated traffic (lower/green) to the realistic traffic (upper/red), by the fraction of accumulated traffic transmitted at each second.

Table 4. Correlation of Simulated Traffic with Real Traffic for Single Job

	Minimum	1st quartile	Median	Mean	3rd quartile	Maximum
TeraSort 3GB	0.91	0.93	0.94	0.92	0.91	0.92
TeraSort 5GB	0.92	0.93	0.98	0.91	0.93	0.92
TeraSort 11GB	0.92	0.93	0.96	0.94	0.93	0.92

to compare the Hadoop traffic. This is because the same traffic patterns require the same amount of traffic to be delivered at the same time, which is a high demand on simulation accuracy.

With the same analysis done to the sflow recorded on our testbed, we compare the traffic transmissions against each other. As shown in Figure 10, we plotted the accumulated traffic transmission over time of two jobs: TeraSort 3G and TeraSort 5G. Although the simulated traffic is not as linear as the real traffic in terms of transmission, they show the same patterns at each second and similar start/stop time of the traffic.

To quantify the simulation accuracy, we use two metrics to measure the accuracy of simulated traffic: the correlation and the mean absolute percentage error (MAPE). The correlation can measure the trend between two transmissions (*S*imulated *vs. R*eal):

$$C_{sr} = \frac{\sum_{k=start}^{end}(s_k - \bar{s})(r_k - \bar{r})}{\sqrt{\sum_{k=start}^{end}(s_k - \bar{s})^2 \sum_{k=start}^{end}(r_k - \bar{r})^2}}, \tag{1}$$

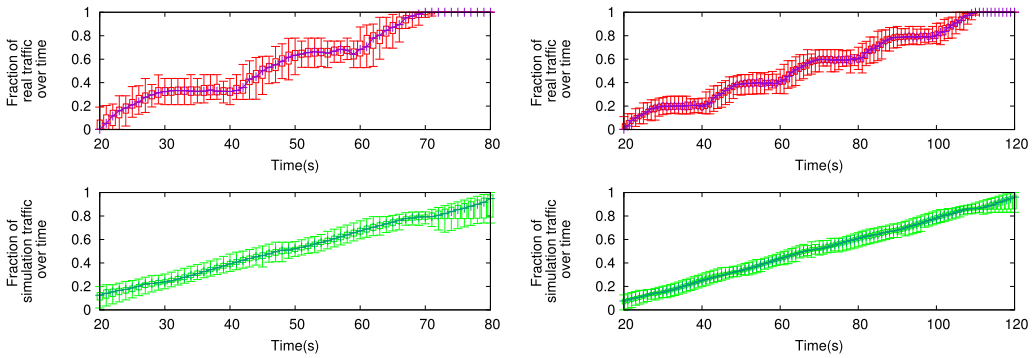
where k is the time, r_k is the fraction of traffic transmitted at time k in the real testbed, and s_k is the fraction of traffic transmitted at time k in the simulation. Due to the randomness in traffic, for each r_k and s_k, we use multiple values including minimum, first quartile, median, mean, third quartile, and maximum from grouped statistics of all experiment batches and compare the correlation, respectively. Table 4 shows the correlation calculated in each scenario. We can see that all the correlation is higher than 0.9, which means the simulated traffic is very similar to the real traffic.

To further justify the similarity in the simulated traffic, we use another metric MAPE to measure the specific distance between two transmissions (*S*imulated *vs. R*eal):

$$MAPE = \sum_{k=start}^{end} \frac{|r_k - s_k|}{r_k}. \tag{2}$$

Table 5. MAPE of Simulated Traffic Transmission for Single Job

	Minimum	1st quartile	Median	Mean	3rd quartile	Maximum
TeraSort 3GB	0.32	0.11	0.13	0.31	0.16	0.11
TeraSort 5GB	0.56	0.12	0.22	0.28	0.28	0.22
TeraSort 11GB	0.09	0.09	0.14	0.66	0.23	0.21
Kmeans 100k*50	0.1	0.56	0.34	0.63	0.84	0.19
Kmeans 400k*50	0.75	0.13	1.89	$\gg 1$	0.32	0.19
Kmeans 1m*50	0.36	0.48	0.26	0.56	1.08	0.36



(a) Executing 3 TeraSort 3GB jobs. (b) Executing 5 TeraSort 3GB jobs.

Fig. 11. Comparison of traffic workload in reality and simulation.

Table 5 shows the MAPE value calculated for TeraSort with different parameters. To gain a better confidence, apart from TeraSort, we also implemented Kmeans in Keddah and listed the accuracy in Table 5, too. We can see that for 75% of the time, the MAPE is smaller than 0.5 for most of the time, which means the difference between simulated traffic and realistic traffic is below 50%. That said, the simulated traffic successfully regenerates the real traffic with small enough bias.

6.2 Multiple Jobs

Keddah can also simulate multiple job executing within a single cluster. To validate the accuracy of Keddah, we run a set of TeraSort jobs with a fixed interval of 20s between each job. We execute these in both the real testbed and the simulator environments and compare the traffic transmission. Figure 11 shows the comparison of traffic workload between reality and the simulation. We can see that the simulated workload shows a very similar trend over time.

To quantify this similarity, we perform the same MAPE calculation. The results are shown in Table 6. The distance between the simulated traffic and the real traffic is below 1 for 15 out of the 20 results (75%), and if we look at the median values of traffic, which is statistically more intuitive, 75% of the time the MAPE is below 0.2. Thus, we consider the difference small enough to generate valid Hadoop traffic from the simulation process.

7 USE CASES

The goal behind Keddah is to allow researchers to evaluate network innovations under realistic application-layer Hadoop traffic. These include innovations such as novel data-centre topologies, queuing mechanisms, routing protocols, or congestion-control algorithms. In this section, we

Table 6. MAPE of Simulated Traffic Transmission for Workload Consists of Multiple Jobs

	Minimum	1st quartile	Median	Mean	3rd quartile	Maximum
TeraSort 3GB x 3	0.77	0.38	0.09	$\gg 1$	0.98	0.98
TeraSort 3GB x 5	0.25	0.13	0.10	$\gg 1$	1.02	0.56
Kmeans 100k*50 x 5	0.89	0.67	0.16	0.12	0.90	0.78
(TeraSort + Kmeans) x 5	0.87	0.49	0.78	$\gg 1$	1.12	0.72

Table 7. Experiment Traffic Model Parameters

Name	Description	Value
n	cluster size	$\begin{cases} FatTree: & 64 \\ CamCube: & 64 \\ DCell: & 42 \end{cases}$
b	mapred.max.split.size	128MB
s	TeraSort file size	3GB
clusters	Number of Kmeans clusters	50
samples	Number of Kmeans samples	50K

demonstrate how Keddah can be used in profiling different network topologies and TCP protocol flavours.

7.1 Base Setup

The workload we generate consists of an equal number of TeraSort (3G dataset) and Kmeans (50 cluster and 50K samples) jobs, launched in a random sequence. The job parameters can be tuned for better flexibility, but we use a fixed job setup as shown in Table 7 for testing purposes. We perform sensitivity analysis across the load by selecting different parameters for the job arrival process, modelled using an exponential distribution.

7.2 Evaluating Network Topologies

A number of novel data-centre topologies have been proposed in recent years, e.g., FatTree [2], Dcell [11], and Camcube [6]. Thus, researchers developing such topologies are required to evaluate the performance of applications (like Hadoop) operating over them. Previously, people have studied datacentre topologies to see how the network changes as the data centre scales up [32]. However, doing so requires flexible environments (e.g., simulations) that can easily vary these scale parameters. Keddah offers a powerful tool for such experiments.

Topology design involves a number of considerations, including performance, reliability, scalability, security, and cost, among which, the performance is usually evaluated by the throughput, packet loss, and latency (under various traffic loads). However, for tractability, only few types of traffic are typically used in real setups [15]. Thus, simulation methodologies are often used [4, 8, 14, 19] to investigate the network performance. Even so, the traffic is still so important, because the performance of certain topologies may vary across different workloads and application behaviours [4, 15].

To highlight the efficacy of Keddah, we next present experiments exploring the performance of various data-centre topologies with Hadoop traffic. The topologies are FatTree [2], Dcell [11], and Camcube [6], as these three are very typical network topologies (representative of clos, star,

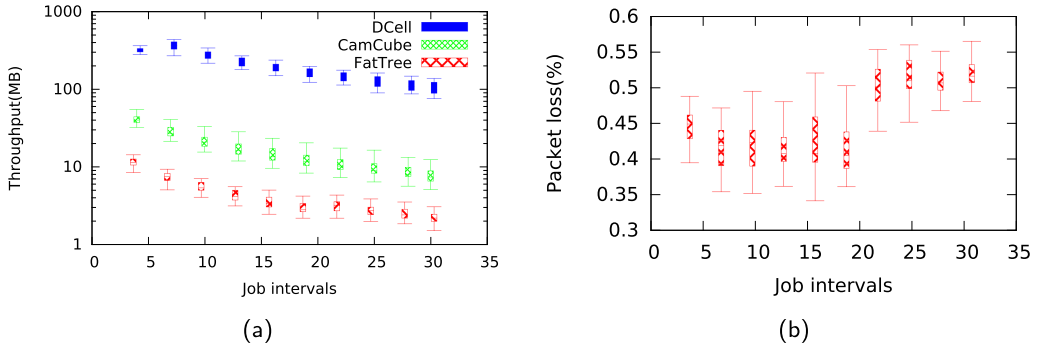


Fig. 12. (a) Comparison of the average throughput of three topologies in different workloads. The same result can be found from previous work. (b) Comparison of the packet dropped in three different topologies.

and torus). The topology instances we use in the experiments are FatTree (8x8), CamCube (4), and DCell (2,2) which contain 64, 64, and 42 nodes, respectively. Of course, Keddah could operate within any configuration the experimenter wishes.

Figure 12(a) shows the throughput measured from all three topologies under different workloads. Each instance is run over 100 times so a range of measures is presented. We can see that, from the throughput point of view, topology is a determinate factor rather than workload intensity: DCell can achieve 10× high throughput than CamCube and nearly 50× more than FatTree. Though the difference is getting smaller with low-intensity workloads, DCell still holds more advantages than CamCube and FatTree. It is worth noting that the throughput result shown in our simulation matches the result of work [4, 15] in terms of the all-to-all or random selected traffic matrix.

Moreover, we also evaluate the packet loss over various workloads. Figure 12(b) shows the percentage of packet lost compared to the total traffic delivered in FatTree. Strangely, though the figure is floating within a certain range, the percentage of packet lost is increased when the workload intensity decreases. We failed to find any previous study on this as reference.

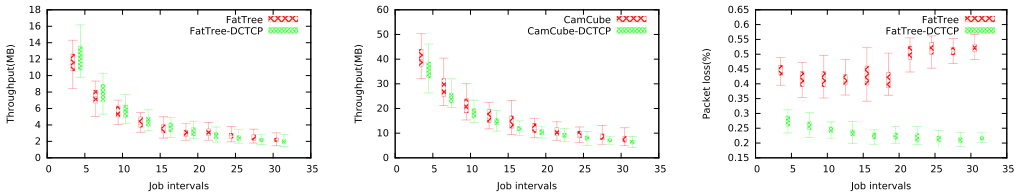
7.3 TCP vs DCTCP

Our next experiment explores the performance benefits of DCTCP—a flavour of TCP specifically designed for use in data centres. Unlike the traffic patterns that DCTCP was originally evaluated using, we have found that Hadoop generates many short flows with high burstiness. We use the DCTCP implementation in ns3[3] and RED queues with a marking threshold at 100 and 40 packets, respectively. The workload consists of TeraSort and Kmeans jobs, running on the FatTree and CamCube.

We compare the performance achieved by TCP NewReno against DCTCP with various topologies and workload intensities. Figure 13 shows the throughput and packet loss measured for FatTree and CamCube under various workloads. Again, as each experiment is run over 100 times, the measure is shown as a range.

We found that DCTCP, indeed, attains improved throughput; however, it depends on the type of topology. As shown in Figure 13(a), DCTCP increased the throughput beyond TCP NewReno by between 1% and 14% (best case was under a heavy workload, where TCP NewReno performed poorly). However, when it comes to CamCube, as shown in Figure 13(b), the throughput of normal

[3]https://github.com/i-maravic/ns-3.



(a) Throughput, higher the better.

(b) Throughput, higher the better.

(c) Percentage of packet loss, lower the better.

Fig. 13. TCP vs DCTCP using Hadoop workload.

TCP can be higher than the DCTCP, especially when the workload is high intensity. In terms of packet loss, as shown in Figure 13(c), DCTCP achieves a lower packet loss across all workloads around 0.25%, whereas for TCP NewReno it is around 0.45%. We also note that the packet loss is particularly lower during heavy workloads.

In conclusion, we have shown two examples of how researchers might use Keddah to profile network innovations in Hadoop cluster, i.e., novel topologies and protocols. Previously, due to the limited experimental methods and traffic generators, people could not investigate how Hadoop might interact with the network via simulations. However, as the two examples have highlighted, Keddah allows researchers to explore their network-level innovations with realistic Hadoop traffic running over them. We emphasise that this is not meant as an exhaustive list of experiments that Keddah supports—simply two exemplars of what Keddah *can* support.

8 RELATED WORK

Various studies have looked at Hadoop workloads primarily for the purposes of cluster benchmarking (e.g., comparing job completion times across clusters). Hadoop provides a number of benchmarking tools. TeraSort [28], later standardised as TPCx-HS [25], is the most popular benchmarking job for cluster comparison. There are also more diverse Hadoop benchmarking projects such as the Big Data benchmark [29] and HiBench [13]. These provide a more diverse set of Big Data processing use-cases, such as machine learning or graph processing (e.g., HiBench uses Pegasus [17] to benchmark PageRank).

Just a small number of studies have investigated the network behaviour of mainstream "Big Data" platforms such as Hadoop. For instance, Reference [27] discusses the performance of Spark, finding that job completion time can be reduced by only 2% through network improvements. A more recent article [24] pointed out that these results are partially an artifact of specific platform optimisations of Spark (namely, heavy data replication to avoid network traffic). In contrast, they found that PageRank jobs can be improved by up to 3× by increasing the network capacity from 1Gbps to 10Gbps. Other works have tried to improve Hadoop performance by, for example, performing aggregation on network devices in the cluster [22]. These works have confirmed the vital role of the network in the performance of distributed processing platforms such as Hadoop MapReduce.

There are few studies that investigate the behaviour and role of the network in Hadoop jobs. Previous works on Hadoop workloads mainly focus on the resource scheduling, including task prediction [12, 20, 40], computation resources (CPU, memory, storage [39]), and computation time prediction [33, 36]. In general, the simulation approach has been widely used when studying the performance of distributed systems [8, 40]. However, none of these works can provide any solution for network-related evaluations, such as network traffic characterization, network performance

evaluation, or network resource management. Though some work has addressed the importance of networking in distributed systems [8, 19, 31], the attention on the network aspect has not always been enough. Recently, researchers have developed traffic modelling tools for Hadoop [38]; however, these tools cannot precisely reflect Hadoop behaviour given that resource scheduling is not considered [7].

These tools also do not provide any flexibility in changing the network components to evaluate with different Hadoop traffic [38]. To the best of our knowledge, we are the first to provide a standard simulation environment to profile networking performance for Hadoop.

9 CONCLUSION

We have presented a methodology and tool for capturing the traffic of Hadoop jobs to generate traffic workloads for network simulations. Our empirical measurements show how different jobs have unique traffic patterns. We have also characterised the effect of cluster and job parameters in the generated traffic, as well as the random variance observed in different job executions. Keddah uses the captured information to generate traffic workloads parametrised to the main characteristics of the job and the dataset.

Keddah greatly simplifies the simulation of Big Data workloads, which we believe can help with the initial evaluation of data-centre network research initiatives. We have shown in this article how the integration with a network simulator enables researchers to evaluate very different network components such as data-centre topologies and congestion control protocol variants with respect to the workload of a distributed data-processing job.

While Keddah only supports Hadoop MapReduce workloads, we demonstrate how it can be applied to different execution flows, ranging from single map-only jobs to iterative computations. The methodology we followed for capturing traffic would also be adaptable to other distributed data-processing platforms that follow Valiant's Bulk Synchronous Parallel computing model [34] (including systems such as Spark [41] and Pregel [23]). We intend to extend Keddah to additional distributed processing frameworks as future work.

APPENDIX

A TERASORT TRAFFIC DISTRIBUTION EXAMPLE

Here, we list the traffic distributions found for the TeraSort job with different input sizes in Table 8. The distributions are extracted from a 400-run sample, through maximum-likelihood estimation in R and evaluated by the Kolmogorov-Smirnov test. By generating the traffic with parameters following the same distribution, we can reproduce the Hadoop job traffic in simulator.

Table 8. TeraSort Traffic Distribution

	HDFS read				Shuffle										HDFS write							
TeraSort file size	number of containers		start time		number of containers		number of src		number of flows per src		size per flow(MB)		start time of flows		number of src		number of dst per src		number of flows		start time	
1.00	7.36	0.57	12.90	1.76	1.56	0.50	3.96	1.70	1.00	0.00	22.42	10.34	21.98	1.11	7.50	0.98	1.90	1.23	1.08	0.29	27.62	1.80
3.00	16.33	2.85	14.75	4.28	1.97	0.17	8.49	2.30	1.11	0.32	21.03	11.99	25.60	3.33	12.72	1.11	2.84	2.51	1.25	0.56	40.13	4.80
5.00	24.92	6.02	16.91	7.02	2.00	0.14	11.17	2.53	1.28	0.53	21.63	13.53	30.98	4.94	14.12	0.87	3.71	3.13	1.41	0.77	55.68	9.17
7.00	34.42	11.00	18.65	8.40	2.00	0.00	12.80	1.55	1.51	0.78	21.45	12.79	35.72	6.44	14.33	0.76	4.40	3.40	1.59	1.01	71.73	15.79
9.00	42.82	13.80	20.14	12.78	2.00	0.00	13.25	1.17	1.69	0.97	23.43	14.14	40.09	7.79	14.82	0.41	5.00	3.58	1.73	1.18	86.10	20.11
11.00	44.45	20.51	22.54	16.57	1.99	0.10	13.34	1.30	1.86	1.13	21.58	13.88	45.13	9.47	14.92	0.27	5.93	3.34	1.84	1.36	104.11	27.00
13.00	52.38	24.25	27.38	21.89	2.00	0.00	13.35	1.50	2.08	1.43	23.61	14.14	51.22	11.67	14.98	0.14	6.56	3.27	1.96	1.54	121.88	32.00
15.00	57.95	28.02	29.89	21.28	2.01	0.10	13.44	1.60	2.22	1.55	23.00	11.95	57.16	14.50	15.00	0.00	7.12	3.12	2.08	1.70	136.58	36.04
17.00	56.97	29.77	32.62	25.54	2.01	0.10	13.62	1.52	2.32	1.74	23.18	12.84	62.47	14.75	14.99	0.10	7.62	2.98	2.21	1.88	163.96	47.80
19.00	70.29	36.61	28.95	29.59	2.00	0.00	13.78	0.84	2.56	1.90	23.02	12.45	64.12	15.74	15.00	0.00	8.07	2.89	2.33	2.07	191.03	65.12
21.00	81.77	43.43	28.97	28.43	2.00	0.00	13.80	0.76	2.62	2.07	23.52	12.64	69.17	17.65	15.00	0.00	8.56	2.68	2.44	2.22	221.11	77.13

ACKNOWLEDGMENTS

The authors would like to thank Dr. Timm Boettger for his help on managing distributed computing infrastructures; also Dr. Richard Clegg for his valuable knowledge on network traffic modeling.

REFERENCES

[1] Apache Software Foundation. 2017. The Apache Mahout project. Retrieved from http://mahout.apache.org/.
[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM Comput. Commun. Rev.* 38, 4 (2008), 63–74.
[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data center TCP (DCTCP). *ACM SIGCOMM Comput. Commun. Rev.* 41, 4 (2011), 63–74.
[4] Kashif Bilal, Samee U. Khan, Limin Zhang, Hongxiang Li, Khizar Hayat, Sajjad A. Madani, Nasro Min-Allah, Lizhe Wang, Dan Chen, Majid Iqbal, et al. 2013. Quantitative comparisons of the state-of-the-art data center architectures. *Concur. Computat.: Pract. Exper.* 25, 12 (2013), 1771–1783.
[5] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. 2011. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS'11)*.
[6] Paolo Costa, Austin Donnelly, Greg O'Shea, and Antony Rowstron. 2010. *CamCube: A Key-based Data Center*. Technical Report MSR TR-2010-74, Microsoft Research.
[7] Jie Deng, Gareth Tyson, Felix Cuadrado, and Steve Uhlig. 2017. Keddah: Capturing Hadoop network behaviour. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. IEEE, 2143–2150.
[8] Svend Frølund and Pankaj Garg. 1998. Design-time simulation of a large-scale, distributed object system. *ACM Trans. Model. Comput. Simul.* 8, 4 (1998), 374–400.
[9] Richard M. Fujimoto. 2016. Research challenges in parallel and distributed simulation. *ACM Trans. Model. Comput. Simul.* 26, 4 (2016), 22.
[10] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues don't matter when you can JUMP them! In *Proceedings of the ACM Symposium on Networked Systems Design and Implementation (NSDI'15)*.
[11] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. 2008. Dcell: A scalable and fault-tolerant network structure for data centers. *ACM SIGCOMM Comput. Commun. Rev.* 38, 4 (2008), 75–86.
[12] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah Khalid Alham, and Zelong Liu. 2010. MRSim: A discrete event based MapReduce simulator. In *Proceedings of the IEEE International Conference on Fuzzy Systems and Knowledge Discovery (FSKD'10)*.
[13] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proceedings of the IEEE Data Engineering Workshop*.
[14] Hao Jin, Tosmate Cheocherngngarn, Dmita Levy, Alex Smith, Deng Pan, Jason Liu, and Niki Pissinou. 2013. Joint host-network optimization for energy-efficient data center networking. In *Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13)*. IEEE, 623–634.
[15] Sangeetha Abdu Jyothi, Ankit Singla, P. Godfrey, and Alexandra Kolla. 2014. Measuring and understanding throughput of network topologies. Retrieved from: *arXiv preprint arXiv:1402.2531*.
[16] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. ACM, 202–208.
[17] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the IEEE International Conference on Data Mining (ICDM'09)*.
[18] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. 2012. Parallel data processing with MapReduce: A survey. *ACM SIGMOD Record* 40, 4 (2012), 11–20.
[19] Ting Li and Jason Liu. 2015. Cluster-based spatiotemporal background traffic generation for network simulation. *ACM Trans. Model. Comput. Simul.* 25, 1 (2015), 4.
[20] Ning Liu, Xi Yang, Xian-He Sun, Johnathan Jenkins, and Robert Ross. 2015. YARNsim: Simulating Hadoop YARN. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*. IEEE, 637–646.
[21] Diego Lugones, Kostas Katrinis, Martin Collier, and Georgios Theodoropoulos. 2012. Parallel simulation models for the evaluation of future large-scale datacenter networks. In *Proceedings of the IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications (DS-RT'12)*. IEEE, 85–92.

[22] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. 2014. NetAgg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the ACM Conference on Emerging Network Experiment and Technology (CoNEXT'14)*.

[23] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'10)*.

[24] Frank McSherry. 2015. The impact of fast networks on graph analytics. Cambridge Systems at Scale Blog. Retrieved from: http://www.cl.cam.ac.uk/research/srg/netos/camsas/blog/2015-07-08-timely-pagerank-part1.html.

[25] Raghunath Nambiar. 2015. Benchmarking big data systems: Introducing TPC express benchmark HS. *Workshop on Big Data Benchmarks*. Springer, Cham.

[26] Sandhya Narayan, Susan Bailey, Matthew Greenway, Robert Grossman, Allison Heath, Ray Powell, and Anand Daga. 2012. Openflow enabled Hadoop over local and wide area clusters. In *Proceedings of the High Performance Computing, Networking, Storage and Analysis (SCC'12)*.

[27] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks. In *Proceedings of the ACM Symposium on Networked Systems Design and Implementation (NSDI'15)*.

[28] Owen O'Malley. 2008. Terabyte sort on Apache Hadoop. Retrieved from: http://sortbenchmark.org/Yahoo-Hadoop.pdf.

[29] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'09)*.

[30] Yang Peng, Kai Chen, Guohui Wang, Wei Bai, Zhiqiang Ma, and Lin Gu. 2014. Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing. In *Proceedings of the IEEE INFOCOM*.

[31] Yuansong Qiao, Xueyuan Wang, Guiming Fang, and Brian Lee. 2016. Doopnet: An emulator for network performance analysis of Hadoop clusters using Docker and Mininet. In *Proceedings of the IEEE Symposium on Computers and Communication (ISCC'16)*. IEEE, 784–790.

[32] S. V. Rao, Debkalpa Goswami, et al. 2013. NS3 simulator for a study of data center networks. In *Proceedings of the IEEE 12th International Symposium on Parallel and Distributed Computing (ISPDC'13)*. IEEE, 224–231.

[33] Jian Tan, Xiaoqiao Meng, and Li Zhang. 2012. Performance analysis of coupling scheduler for MapReduce/Hadoop. In *Proceedings of the IEEE INFOCOM*.

[34] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[35] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'13)*.

[36] Guanying Wang, Ali R. Butt, Prashant Pandey, and Karan Gupta. 2009. A simulation approach to evaluating design decisions in MapReduce setups. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS'09)*.

[37] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. 2013. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Trans. Netw.* 21, 2 (2013), 345–358.

[38] Zhen Xie, Zheng Cao, Zhan Wang, Dawei Zang, En Shao, and Ninghui Sun. 2016. Modeling traffic of big data platform for large scale datacenter networks. In *Proceedings of the IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS'16)*. IEEE, 224–231.

[39] Hailong Yang, Zhongzhi Luan, Wenjun Li, and Depei Qian. 2012. MapReduce workload modeling with statistical approach. *J. Grid Comput.* 10, 2 (2012), 279–310.

[40] Srikanth B. Yoginath and Kalyan S. Perumalla. 2015. Efficient parallel discrete event simulation on cloud/virtual machine platforms. *ACM Trans. Model. Comput. Simul.* 26, 1 (2015), 5.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

Received August 2017; revised October 2018; accepted December 2018