HadoopWatch: A First Step Towards Comprehensive Traffic Forecasting in Cloud Computing

Yang Peng, Kai Chen, Guohui Wang[†], Wei Bai, Zhiqiang Ma, Lin Gu
Hong Kong University of Science and Technology, [†]Facebook

Abstract—**This paper presents our effort towards comprehensive traffic forecasting for big data applications using external, light-weighted file system monitoring. Our idea is motivated by the key observations that rich traffic demand information already exists in the log and meta-data files of many big data applications, and that such information can be readily extracted through run-time file system monitoring. As the first step, we use Hadoop[1] as a concrete example to explore our methodology and develop a system called HadoopWatch to predict traffic demand of Hadoop applications. We further implement HadoopWatch in our real small-scale testbed with 10 physical servers and 30 virtual machines. Our experiments over a series of MapReduce applications demonstrate that HadoopWatch can forecast the traffic demand with almost 100% accuracy and time advance. Furthermore, it makes no modification of the Hadoop framework, and introduces little overhead to the application performance.**

I. Introduction

The explosion of big data applications has imposed significant challenges on the design of network infrastructure in cloud and data center environments. Researchers have proposed various new network architectures [2, 18, 28] and traffic engineering mechanisms [3, 5, 32] to handle the rapid growth of bandwidth requirement in data center networks. Many of these proposals leverage the knowledge of application traffic demands to customize network design. For example, Hedera [3], MicroTE [5] and D3 [32] perform flow-level traffic engineering and rate control based on predicted traffic demands. Helios [12], c-Through [28] and OSA [6] rely on accurate traffic demand estimation to perform dynamic optical circuit provisioning. More recently, researchers have also looked into the tight integration of applications and network to configure the network topology and routing based on application run-time traffic demands [14, 29, 31]. All these systems require comprehensive understanding of application traffic in data center networks – the ability to forecast traffic demand before packets enter the network.

However, it is difficult to predict application traffic demand accurately. All the existing solutions focus on using heuristic algorithms based on the measurement of network level parameters. For example, Hedera [3] and Helios [12] estimate traffic demands using flow counter measurement on switches; c-Through [28] and Mahout [8] use socket buffer occupancy at end hosts to estimate traffic demands for different destinations. However, these schemes fall short. First, most of them cannot predict the traffic demand before the traffic enters the network; Second, parameters observed on network paths cannot accurately reflect the truth of application demands due to the noise of background flows and congestion control at end hosts; Third, they fail to capture the fine-grained traffic dependencies and priorities information imposed by applications. As a result, these network layer solutions are shown to perform poorly in predicting the real application demands [4], which further leads to undesired performance in network provisioning and traffic engineering mechanisms using these demands.

In this paper, we explore an alternative solution to provide comprehensive traffic forecasting at application layer. With application specific information, application layer traffic forecasting is expected to achieve more accurate traffic demand estimation. However, there are several design issues that make this approach challenging and interesting for exploration.

- **Ahead-of-time:** The scheme must be able to predict traffic demand before the data is sent to the network, so that it can be most useful for network configuration and traffic engineering.
- **Transparency:** Many big data applications in data centers are complex distributed applications. Traffic forecasting should be transparent to these applications and do not modify any application codes.
- **Light-weighted:** Traffic forecasting should be light-weighted. Many data center applications are large scale systems with high performance requirements. The traffic forecasting system should not introduce much overhead to degrade the application performance; and it should scale gracefully with the number of computing nodes and concurrent jobs.
- **Fine-grained:** An application layer traffic forecaster is expected to provide more fine-grained traffic demand information than just traffic volume. For example, many distributed applications have computation barriers that cause dependencies among multiple flows, and flows with more dependencies need high priorities in transfer. Such structural information will be useful to enable better network control and optimization.

We observe that rich traffic demand information exists in the log and meta-data files of many big data applications, and such information can be extracted efficiently through run-time file system monitoring. We use Hadoop as a concrete example to explore the design of application layer traffic forecasting using file system monitoring. We develop a system called

[1] One of the most popular and widely-used open-source software framework in cloud computing.

HadoopWatch, which is a passive monitoring agent attached to the Hadoop framework that monitors the meta-data and logs of Hadoop jobs to forecast application traffic before the data is sent. It does not require *any* modification to the Hadoop framework.

We have implemented HadoopWatch and deployed it on a real small-scale testbed with 10 physical machines and 30 virtual machines. Our experiments over a series of MapReduce applications demonstrate that HadoopWatch can forecast the application layer traffic demand with almost 100% accuracy and time advance, while introducing little overhead to the application performance.

Our work is a first step towards comprehensive traffic forecasting at the application layer. Many research problems remain to be explored in future work. However, we believe our work shows early promises of performing comprehensive and light-weighted traffic forecasting through file system monitoring, which could be a useful building block for tight network and application integration in cloud and data center environments.

Roadmap: The rest of the paper is organized as follows. Section § II introduces the background and key observations that enables the forecasting architecture. Section § III presents the design of HadoopWatch. Section § IV discusses the implementation and evaluation results of HadoopWatch. Section § V reviews the related works. We discuss the future work and conclude the paper in Section § VI.

II. Traffic Forecasting via File System Monitoring

Due to a variety of data exchange requirements, there is a large amount of network traffic in the life cycle of a Hadoop job. We first briefly introduce the Hadoop architecture and its dataflow in different stages. Then, to motivate our key idea of forecasting application traffic through external, light-weight file system monitoring, we introduce the observations and opportunities in file systems that provide rich-semantics enabling traffic forecasting.

A. Hadoop Background

Hadoop consists of *Hadoop MapReduce* and *Hadoop Distributed File System* (HDFS). Hadoop MapReduce is an implementation of MapReduce designed for large clusters, while HDFS is a distributed file system designed for batch-oriented workloads. Each job in MapReduce has two phases. First, users specify a *map* function that processes the input data to generate a list of intermediate key-value pairs. Second, a user-defined *reduce* function is called to merge all intermediate values associated with the same intermediate key [11]. HDFS is used to store both the input to the *map* and the output of the *reduce*, but the intermediate results, such as the output of the *map*, are stored in each node's local file system.

A Hadoop implementation contains a single *master* node and many *worker* nodes. The master node, called the JobTracker, handles job requests from user clients, divide these jobs into multiple tasks, and assign each task to a worker node for execution. Each worker node maintains a TaskTracker process that executes the tasks assigned to itself. Typically, a TaskTracker has a fixed number of slots for accepting tasks.

B. Hadoop Dataflows

Many Hadoop jobs are communication intensive, involving a large amount of data transfer during their execution. We broadly characterize Hadoop dataflows into three types.

- **Import**: *The map reads data from HDFS*. To increase overall data loading throughput of a job, multiple concurrent map tasks may be scheduled to fetch data in parallel. Specifically, for each map task, the JobTracker will specify its input split. As a map task runs, it will fetch the corresponding data (in the form of key-value pairs) from HDFS and iteratively perform the user defined map function over it. Optimized with various scheduling techniques [30, 33], most map tasks achieve data locality, while there also exist some non-local map tasks reading their input splits from remote DataNodes which involve network transfer.
- **Shuffle**: *Intermediate results are shuffled from the map to the reduce*. Before the reduce function is called, a reduce task requires intermediate results from multiple map tasks as its input. When a map task finishes, it will write its output to local disk, commit to the JobTracker, and reclaim the resources. Meanwhile, the reduce task will periodically query the JobTracker for any latest map completion events. Being aware of these finished map tasks and their output locations, a reduce task initiates a few threads and randomly requests intermediate data from the TaskTracker daemons on these nodes.
- **Export**: *The reduce writes output to HDFS*. Large output data is partitioned into multiple fix-sized blocks[2], while each of them is replicated to three DataNodes in a pipeline [24, 27]. A Hadoop job completes after the outputs of all reduce tasks are successfully stored.

C. Observations and Opportunities

Rich traffic information in file systems: Most cloud big data applications, such as Hadoop, have various file system activities during the job execution. We observe that these file system activities usually contain rich information about the job run-time status and its upcoming network traffic.

First, logging is a common facility in big data applications for troubleshooting and debugging purposes. However, these log file entries can also be used to identify network traffic. For example, the source and destination of a shuffle flow in MapReduce can be determined after locating a pair of map and reduce tasks, while such task scheduling results are written in the JobTracker's log.

Besides, intermediate computing results and meta-data are periodically spilled to disk due to limited capacity of memory allocation. The size of all the output partitions of a map task is

[2] The last block in a file may be smaller.

Event	Description
IN_CREATE	File was created.
IN_ACCESS	File was read from.
IN_MODIFY	File was written to.
IN_ATTRIB	File's attribute was changed.
IN_CLOSE_WRITE	File was closed (opened for writing).
IN_CLOSE_NOWRITE	File was closed (opened not for writing).
IN_OPEN	File was opened.
IN_MOVED_FROM	File was moved away from watch.
IN_MOVED_TO	File was moved to watch.
IN_DELETE	File was deleted.
IN_DELETE_SELF	The watch itself was deleted.

TABLE I
VALID EVENTS IN *inotify*

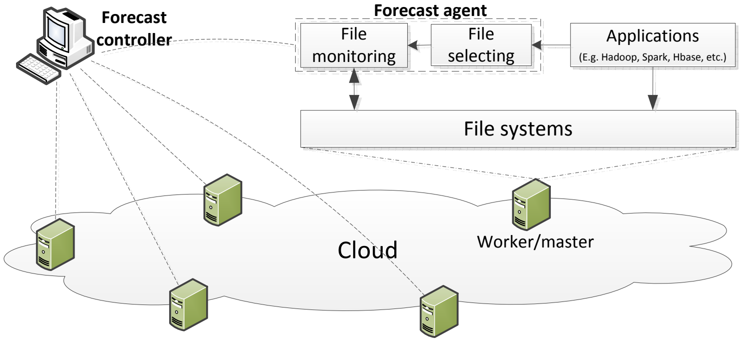


Fig. 1. Architecture of traffic forecasting

saved in a temporary index file, from which we can compute the payload volume of all the shuffle flows from this map.

In addition, the namespace of a distributed file system may also be accessible by parsing its meta-data files on disk. For example, when the HDFS starts up, its whole namespace is updated in a file named *FsImage*. Another file called *EditLog* is used to record all namespace changes thereafter. Although Hadoop only maintains a snapshot of the namespace in memory for frequent remote queries, it can be externally reconstructed with those two files on disks. The reconstructed namespace can be used to recover the network traffic generated by HDFS read and write operations.

Light-weighted file system monitoring. We also observe that these file system activities can be monitored using light-weighted file monitoring facility in modern operating systems. In recent Linux system, there is a file change notification subsystem called *inotify* [21]. The key of inotify is to perform file surveillance in the form of watch, with a pathname and an event mask specifying the monitored file and the types of file change events. A file will be monitored after it is tagged to watch, while all the files in a directory will be monitored if the directory is watched. Table I shows all the valid events in inotify. With inotify, we can dynamically retrieve the footprint of an application with its file system activities. *Inotify* can monitor file change events efficiently in an asynchronous manner to avoid polling. By doing that, the application's file system operations can be executed in a non-blocking mode when inotify is running.

In summary, the above two key observations enable us explore the idea of forecasting application traffic through external, light-weight file system monitoring.

III. HADOOPWATCH

As the first step, we use Hadoop as a concrete example to explore the detailed design of our traffic forecasting approach. We choose Hadoop because it is one of the most popular big data applications running in today's cloud and data centers, and its execution structure represents several typical traffic patterns in data center applications. We develop a system called HadoopWatch and show how we can forecast traffic demand of Hadoop jobs by only monitoring the file system activities. In the following, we first introduce the HadoopWatch architecture, then illustrate the monitor file selection, and finally show how to perform traffic forecasting.

A. Architecture

Based on the above observations, we propose to forecast the traffic demand of big data applications by monitoring their file system activities. Figure 1 shows the architecture of this traffic forecasting framework. This framework is a passive monitoring engine attached to a big data cluster, which does not require any modification to the applications. It continuously monitors the file system activities underneath big data applications and predict their traffic demands at job run-time. The traffic forecasting framework has two components, *Forecast Agents* and *Forecast Controller*. We implant a Forecast Agent in each worker/master node to collect traffic information and report to the centralized Forecast Controller.

- **Forecast Agent:** Forecast Agent is a daemon implanted on each node that collecting run-time file system activities. To keep it light-weighted, we selectively monitor the specific files and principal activities with *inotify*. To monitor file accessing details, some system call parameters are also collected and encoded in the redundant space of *inotify* interface. To get the content of other bulk data and metadata on disk, the agent will read them directly. Information collected by Forecast Agent will be continuously reported to the Forecast Controller.
- **Forecast Controller:** The main function of the Forecast Controller is to collect reports from all the Forecast Agents and generate comprehensive traffic demand results. The reported traffic demand information may include the basic information such as source, destination and volume and more fine-grained information such as flow dependencies and priorities.

Note that this centralized model is also inspired by the success of several recent large-scale infrastructure deployments such as GFS [17] and MapReduce [11] which employ a central master to manage tasks at the scale of tens of thousands of worker nodes.

B. Monitor File Selection

In Table II, we summarize all the files that should be monitored to forecast the three types of dataflows in Hadoop.

Application	Flow type	Required information	Monitoring files & events		Response action
Hadoop	Shuffle	location of a complete map task, size of each map output partition	file.out.index	IN_CLOSE_WRITE	parse the sequence file format, collect partLengths of all partitions
		location of a new reduce task	JobTracker's log	IN_MODIFY	scan next entry for the TaskTracker that launches a reduce task
		when a flow starts	file.out	IN_SEEK [3]	determine which reduce task are fetching the intermediate data
		when a flow terminates	TaskTracker's log	IN_MODIFY	scan next entry indicating success of the shuffle flow
	Export	block allocation results	NameNode's log	IN_MODIFY	scan next entry for a new block allocation result
		where a pipeline establishes and when a flow starts	DataNode's logs	IN_MODIFY	scan next entry for remote HDFS writing request
		when a flow terminates	DataNode's log	IN_MODIFY	scan next entry indicating success of the HDFS writing flow
	Import	locality of a map task	JobTracker's log	IN_MODIFY	scan next entry for locality of a map task
		input split (blocks) and flow size	split.dta	IN_CLOSE_WRITE	parse split(HDFS path/start/offset), query for corresponding blocks
		when a flow starts	block file	IN_SEEK	match the probable map task fetching the data block
		when a flow terminates	DataNode's log	IN_MODIFY	scan next entry indicating success of the HDFS reading flow

TABLE II
THE FILES FOR TRAFFIC FORECASTING IN HADOOPWATCH

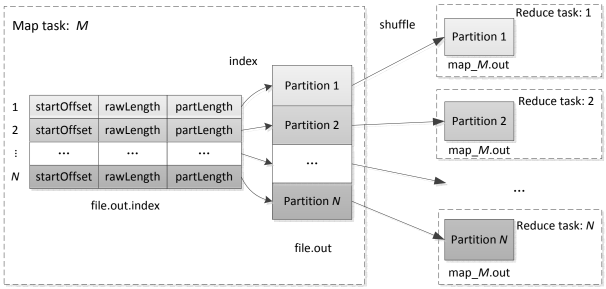
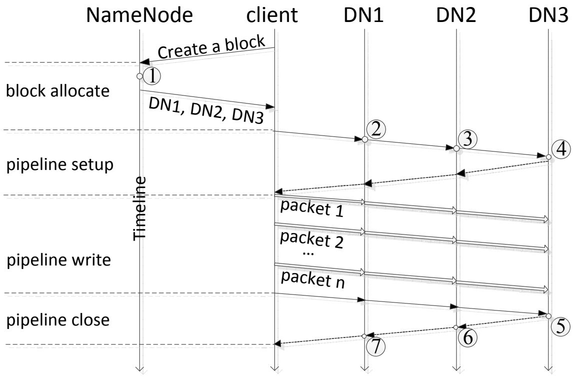


Fig. 2. Data flows in shuffle phase



Log files and information:
1. { blockID, DN1, DN2, DN3, HDFS filename and offset };
2. { blockID, client, DN1}; **3**. { blockID, DN1, DN2 }; **4**. { blockID, DN2, DN3};
5. { blockID, size }; **6**. { blockID, size }; **7**. { blockID, size }

Fig. 3. Pipelined HDFS writing

Note that the selection of these files is based on the detailed analysis and observation of the Hadoop framework and job execution semantics as follows.

Import: The map needs to read input from HDFS. From the JobTracker's log, we can easily identify whether a map task is data-local, rack-local or non-local. For rack-local and non-local maps, they introduce data import from remote nodes through networks. However, the JobTracker's log does not tell from which DataNodes a map will read its input data split. To forecast this information, we pick out the *split.dta* file. This file contains a map task's input split information, i.e., the input file location, offset and length in HDFS.

Shuffle: To forecast the shuffle traffic from the map to the reduce, we observe that the output of a map task is stored in a file named *file.out* as shown in Figure 2. To support direct access to all data partitions in it, there is an index file named *file.out.index* that maintains all their offsets and sizes. Since each partition in *file.out* corresponds to the payload of a shuffle flow sent to a reduce, we can forecast its volume based on the size of the corresponding partition. On the other hand, to infer the source and the destination of a shuffle flow, we use the scheduling result in the JobTracker's log which includes the information where a map task or a reduce task is launched.

Export: The reduce may export its output to HDFS, which entails HDFS writing. In HDFS, each data block is replicated with multiple replicas (3 replicas by default). The HDFS writing is implemented using a pipeline set up among replica nodes to maintain data consistency. Figure 3 shows the diagram of a pipelined HDFS writing. First, when a HDFS client requests to write a block, the NameNode allocates 3 sequential DataNodes and writes this allocation in its log (1). Then, before the block is transferred between each pair of DataNodes, the receiver side writes a log indicating the upcoming HDFS writing (2-4). Finally, when the write completes, its volume is saved in the DataNode's log (5-7). Through these log files, we can forecast the communication matrix according to the pipeline and predict the volume of upcoming flows based on the HDFS block size (e.g., 64MB).

C. Traffic Forecasting

As above, HadoopWatch can provide accurate traffic forecasting for every data flow, including its source, destination and volume. As shown in Table III, the source and destination are two identities, which can uniquely identify a flow in a

[3] This event type is not implemented in primitive inotify.

Dataflow	Source	Destination	Volume
Import	mapID, input blockID	mapID	blockID's size
Shuffle	mapID	reduceID	partition size
Export	reduceID	reduceID, output blockID	blockID's size

TABLE III
PER FLOW METRICS

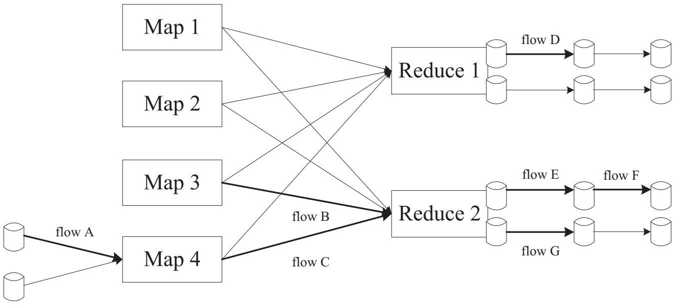


Fig. 4. Flow patterns in Hadoop

Algorithm 1 Traffic matrix calculation

```
Input: every $f$: ($src$, $dest$, $vol$)
for each $f$ do
    ip1 = location($f$'s $src$)
    ip2 = location($f$'s $dest$)
    $TM$(ip1, ip2) $\leftarrow$ $TM$(ip1, ip2) + $f$'s $vol$
end for
```

Hadoop job. With these per flow metrics, we can not only compose the overall traffic matrix in the cluster, but also identify fine-grained flow relations such as dependency and priority.

The source and destination in Table III are logical identities. To determine the physical locations, we just map the logical source and destination to the physical nodes. The mapping requires knowledge of task scheduling results and block locations. Most of these information is plainly accessible by monitoring files listed in Table II. However, the source locations of map input blocks and the volumes of reduce export flows cannot be explicitly captured. Therefore, HadoopWatch develops the following two heuristics to get these information.

- **Source location of a map input block:** When the JobTracker schedules a rack-local or non-local map task, it will independently choose the closest block replica to fetch. For these map tasks, we can translate their input splits to blockIDs through querying the NameNode. Since there are rarely two tasks processing a same input dataset simultaneously, the node where we captured such block file access event probability is the chosen source of the data import flow.
- **Volume of a data export flow:** When the reduce task output is written into HDFS, the data will be divided and stored into multiple blocks (typically 64MB) and replicated to several DataNodes. Because the block size is fixed, in most cases, the size of an export flow is fixed, i.e., 64MB. However, the last block size is uncertain. Considering that the output size of a fixed user-defined reduce function is approximately proportional to its input size, we maintain a selectivity, $s(r)$, for reduce tasks in a job, which is defined as the output size to its input size. For an upcoming reduce operation, we estimate its selectivity based on the selectivities of the reduce operations in recent past using exponentially weighted moving average (EWMA).

  $$s_{new}\text{ (r) = }\alpha s_{measured}\text{(r) + (1-}\alpha\text{) }s_{old}\text{(r)}$$

  where, α is the smoothing factor (HadoopWatch uses $\alpha = 0.25$). On the other hand, a reduce task r's input size, $I(r)$, is the sum of shuffle flow volumes from all map tasks. We get the estimated reduce output using:

  $$O(r) = I(r) \times s(r)$$

  For the ith export flow of the reduce task r, its volume $vol(i)$ is calculated by checking whether the maximum block size is enough to save the remaining bytes:

  $$vol(i) = \begin{cases} BLK_{max}, & if\ O(r) > i \times BLK_{max} \\ O(r) - (i-1)BLK_{max}, & otherwise \end{cases}$$

  Here, BLK_{max} stands for the maximum size of a HDFS block (e.g., 64MB).

With the above basic data flow information inferred, we next can compose the traffic matrix and identify the flow dependency and priority. While easy to achieve, we anticipate that these information is very useful for a variety of data center network technologies to achieve a fine-grained traffic engineering, network profiling, to transport protocol design. For example, traffic distribution information is critical for fine-grained flow scheduling in Hedera [3] and traffic engineering in MicroTE [5], while dependency and priority information can be incorporated into several recent deadline-aware transport designs like D^{2}TCP [26] and D^3 [32] for more intelligent congestion control.

- **Traffic matrix**: We can easily calculate the traffic matrix in a cluster with every data flow information. As shown in Algorithm 1, to calculate the traffic volume between any two physical nodes ip1 and ip2, we just need to sum up the volumes of individual flows between them.
- **Dependency**: We define two types of dependencies, i.e., causal-dependency and co-dependency. The causal-dependency $f_1 \rightarrow f_2$ means that the initiation of f_2 depends on the completion of f_1. For example, $flowA \rightarrow flowC$ and $flowC \rightarrow flowE$ in Figure 4. The co-dependency $f_1 \leftrightarrow f_2$ specifies that both f_1 and f_2 share a common barrier. The barrier cannot be passed through until the completion of all these co-dependent flows. One such example is the shuffle flows that serve the same reduce task (e.g. $flowB \leftrightarrow flowC$), and another example is the pipeline flows replicating the same block (e.g. $flowE \leftrightarrow flowF$).

Algorithm 2 Dependency

```
Input: $f_1$: (src, dest) and $f_2$: (src, dest)
Output: 1 ($f_1 \leftarrow f_2$); 2 ($f_1 \rightarrow f_2$); 3 ($f_1 \leftrightarrow f_2$); 0 (otherwise).
if $f_1$'s dest == $f_2$'s src then
    return 1
else if $f_1$'s src == $f_2$'s dest then
    return 2
else if $f_1$'s dest == $f_2$'s dest then
    return 3
end if
return 0
```

We can infer the dependency of two flows based on their logical source and destination identities in Table III. Algorithm 2 determines the dependency between f_1 and f_2. For example, we know $flowA \rightarrow flowC$, since the destination of $flowA$ and the source of $flowC$ are both Map 4. And $flowB \leftrightarrow flowC$, since they share a same destination, Reduce 2.

- **Priority**: Unlike flow dependency that reflects the inherent structure of Hadoop jobs, flow priority is a metric that depends more on the optimization objectives when the forecasted traffic is used in network planning. For different optimization objectives, we can define different policies to assign flow priorities. In a normal use case that we want to boost the execution of a Hadoop job, a flow in an earlier phase should be assigned to a higher priority. Because most Hadoop jobs are composed of multiple tasks (e.g., import ⇒ map ⇒ shuffle ⇒ reduce ⇒ export), and a job completes when the slowest task is finished. As shown in Figure 4, import, shuffle and reduce flows should be prioritized accordingly (e.g., $flowA > flowB > flowD$) to ensure the slowest task finishes as fast as possible. Another example of priority assignment policy is that shorter flows in larger co-dependent groups should be assigned with higher priorities. Using the "shortest job first" strategy, finishing shorter flows in larger co-dependent group first can quickly decrease the number of blocking flows and speed up the execution of a Hadoop job.

IV. Evaluation

Testbed: We deployed HadoopWatch on 30 virtual machines (VMs) running on 10 physical servers. All the 10 physical servers are HP PowerEdge R320 with a quad core Intel E5-1410 2.8GHz CPU, 8GB memory and a 1Gbps network interface. On each physical server, we set up 3 Xen VMs (DomU) and each of the VMs is allocated with 1 dedicated processor core (2 threads) and 1.5GB memory. We run Hadoop 0.20.2 on all the 30 VMs for the traffic forecasting experiments.

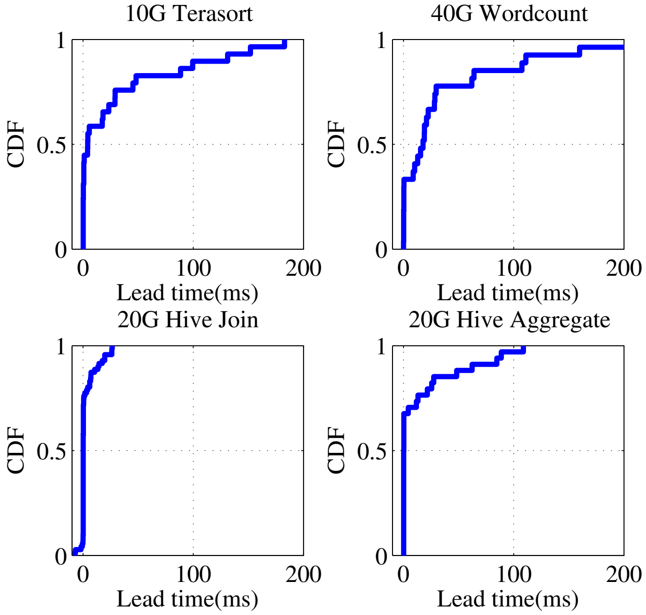


Fig. 6. Time advance in remote import flow forecasting

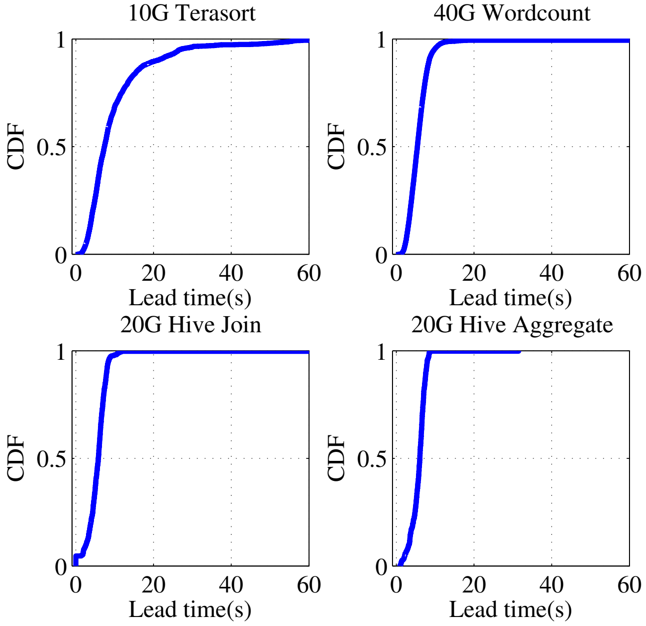


Fig. 7. Time advance in shuffle flow forecasting

Evaluation metrics: In our evaluation, we study the accuracy of HadoopWatch in prediction flow volumes, the time advance of these predictions and the overhead of HadoopWatch. To collect the ground truth of Hadoop traffic volume and timing, we use TCPdump to capture the actual time and volume of data flows. We extract the source and destination of each flow by parsing the application-level requests and responses in Hadoop flows. We use the absolute difference between the predicted volume and actual volume to evaluate the forecasting accuracy of HadoopWatch. We use the lead time metric to evaluate the time advance, which is defined as $(actual\_time - predicted\_time)$ of Hadoop flows.

Accuracy: Figure 5 shows the traffic volume and forecast accuracy for four representative Hadoop jobs: Terasort, Word-

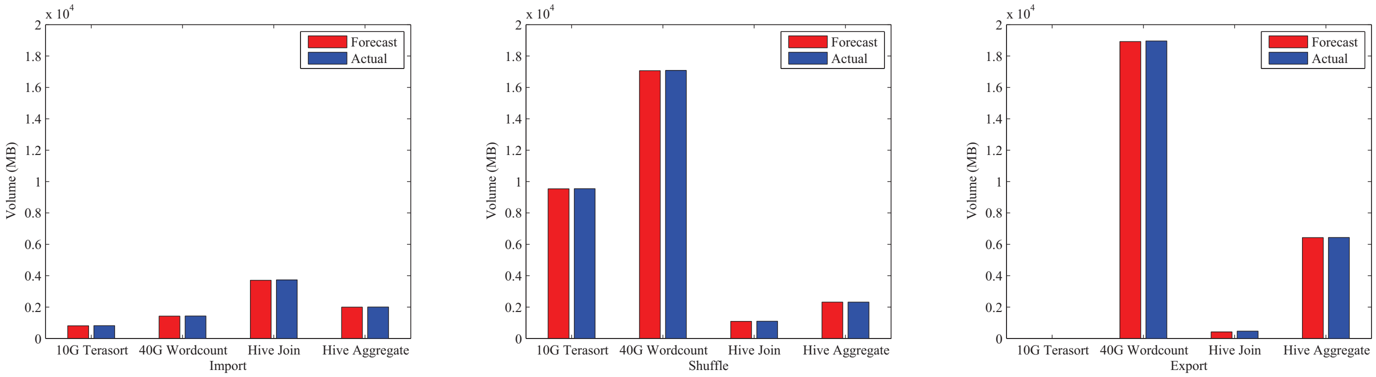


Fig. 5. Accuracy of traffic volume forecasting

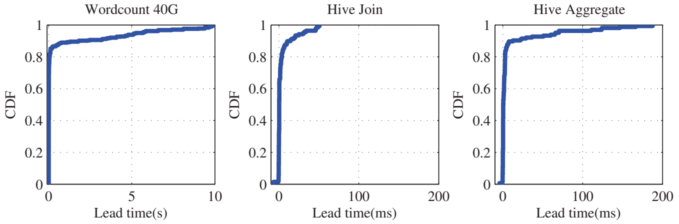


Fig. 8. Time advance in remote export flows forecasting

count, Hive Join and Hive Aggregation [19]. Overall, it can be seen that the shuffle and export phase introduced most of the traffic for these jobs[4], and we achieve high accuracy for all types of traffic. The slight difference between the forecast results and the actual ones is mainly caused by the control signals and TCP retransmission packets. Besides, there are occasionally a few dead export flows, since a slower reduce task will be killed between a normal task and its backup instance.

Time advance: Figure 6, 7 and 8 show the forecasting lead time for data import, shuffle and export flows, respectively. We use NTP [22] to synchronize the clock on these nodes. The results show that almost 100% traffic flows are successfully forecasted in advance. Most data import and export flows occur soon after the corresponding traffic forecasts ($\leq$100 ms), while most shuffle flows are forecasted much earlier in advance (5s - 20s). Because a reduce task only initiates 5 shuffle flows fetching intermediate data and other shuffle flows are pending. In a small percent ($< 3\%$) of flows, we do observe some forecast delays that flows are forecasted after they are actually sent out. They are either caused by deviation of clock synchronization or monitoring delay of inotify events.

Overhead: Figure 9 compares the execution time of Hadoop jobs with and without HadoopWatch to understand the overhead introduced by HadoopWatch. Among all the 4 jobs we tested, their execution time only increases by 1% to 2% with HadoopWatch running in the cluster.

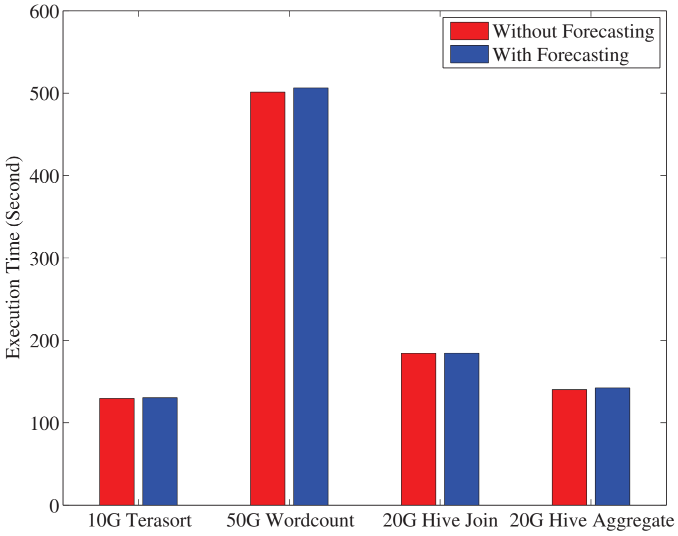


Fig. 9. Execution time (Second)

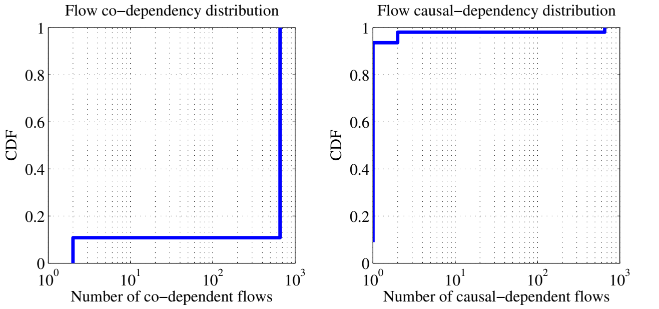


Fig. 10. Distribution of codependent and dependent flows

Dependency: Figure 10 shows the distribution of flow's co-dependent flow numbers and casual-dependent flow numbers. We take the 40GB Wordcount job as an example, which consists of 658 map tasks and 20 reduce tasks. Therefore, shuffle flows can be divided into 20 co-dependent groups. Each group contains 658 co-dependent flows initiated by the same reduce task. On the other hand, because of the data locality of map tasks, a large number of them are importing data from local disks. Thus, only a small number of shuffle

[4] Note that the output of Terasort is not replicated in remote DataNodes, so it does not introduce any export flows.

flows are casual-dependent on import flows. Meanwhile, the output export flows of each reduce task are casual-dependent on its input shuffle flows.

Scalability: Due to the limitations of testbed size, our evaluation results of HadoopWatch are limited to tens of nodes. We use simulation analysis to understand the scalability of HadoopWatch. We first analyze the major determinants of monitoring overhead in Forecast Agent and Forecast controller, then estimate the HadoopWatch overhead in large production settings.

On a worker node, our agent iteratively processes the inotify events that are caused by multiple tasks. It just parses the traffic related information and sends to the controller. The memory usage is fixed, since no extra data is stored locally. In addition, the CPU usage scale linearly with the number of active tasks (n_{task}) on each node, since the execution of each agent is strictly driven by these tasks' file system events. On the controller end, the forecasting controller continuously receives event reports from multiple agents and generates traffic forecasts accordingly. As a result, its CPU usage scales linearly with the number of total active tasks (N_{task}). Meanwhile, a lot of memory is required to store the volumes of shuffle flows between these map tasks and reduce tasks. The total memory usage exhibits a quadratic growth as N_{task} increases.

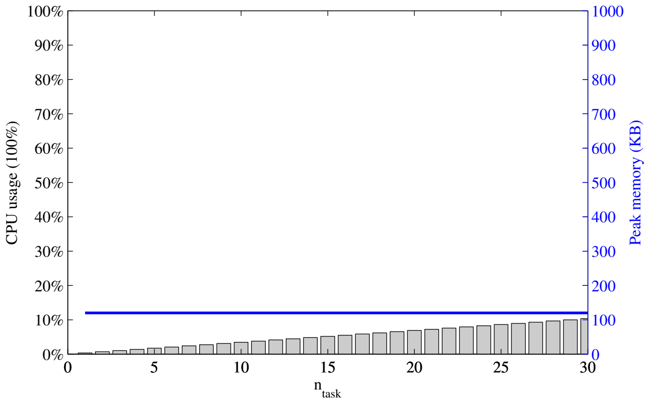


Fig. 11. Agent overhead prediction

Based on the performance metrics collected on our testbed, we estimate the potential overhead that HadoopWatch introduces on large production clusters. Figure 11 shows the estimated overhead on clusters with sizes release by Google and Facebook. We observed that the agent only consume 6.89‰ of a CPU core under hours of heavy workload. To support 30 concurrent tasks one each node in the Google cluster [7, 35], HadoopWatch agent may take 10% consumption of a CPU core and fixed memory. Similarly, we estimate the overhead of HadoopWatch central controller. It will only consume a CPU core's 30% resources and around 50MB memory to support hundreds of thousands of tasks running concurrently. In summary, we believe HadoopWatch is scalable to support traffic forecasting in large MapReduce clusters.

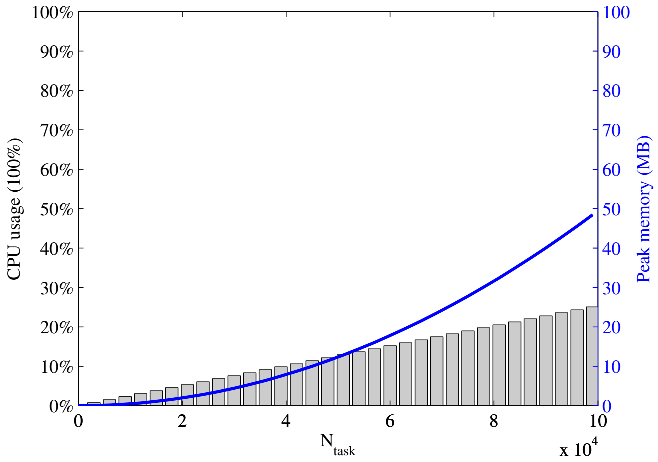


Fig. 12. Central control overhead prediction

V. Related works

Networking researchers have been exploring ways to predict and estimate network traffic demands in different environments. For example, ISPs have employed various traffic demand matrices in their WAN traffic engineering and capacity planning [13, 23]. However, these methods required a large number of past statistics, such as server logs or link data, to provide reliable estimation of traffic demand in the next period of time. Such techniques are not feasible in data centers, where most of the longest-lived flows last only a few seconds [9] and the traffic is elastic. To gain more instant information for traffic demand forecasting, many researchers proposed to estimate the traffic demand based on real-time measuring socket buffers in end hosts [8, 28], or counters in switches [3, 9, 12]. Such techniques are designed for general traffic load prediction in data center, while our method can generate a more accurate traffic forecast with application-level semantics captured in real time.

Various tracing and profiling tools have been proposed to collect execution information of Hadoop. X-Trace [15] is integrated into Hadoop to collect cross-layer event traces for performance diagnosis. To avoid modifying Hadoop, researchers proposed to perform off-line analysis on Hadoop log files for performance tuning and anomaly detection [16, 25]. However, HadoopWatch is focused on forecasting traffic demands based on real-time file system monitoring. Compared with a recent attempt which focused on predicting shuffle flows by periodically scanning Hadoop logs [10], HadoopWatch can provide more comprehensive traffic forecast and more scalable event-driven monitoring.

VI. Conclusion and Future Work

In this paper, we have proposed to use file system monitoring to provide comprehensive traffic forecasting for big data applications. We develop HadoopWatch, a traffic forecaster for Hadoop, that can forecast Hadoop traffic demand accurately and efficiently without modifying the Hadoop framework. We believe application layer traffic forecasting is a key building block to enable workload optimized networking in cloud data centers and tightly integrate network design with applications.

We have implemented HadoopWatch and deployed it on a small-scale testbed with 10 physical machines and 30 virtual machines. Our experiments over a series of MapReduce applications demonstrate that HadoopWatch can forecast the application layer traffic demand with very high accuracy and time advance, while introducing little overhead to the application performance.

Our work is a first attempt exploring the rich research space of comprehensive traffic forecasting at application layer. With the surge of software defined networking in cloud data centers, we believe HadoopWatch can be useful in a many scenarios that target to jointly optimize application performance and network configuration, e.g., from topology optimization, traffic engineering, flow scheduling to transport control, etc. In the meanwhile, we also acknowledge that there are many research problems remain to be explored in our future work. For example, HadoopWatch is now a traffic forecaster particularly designed for the Hadoop framework. The file selection and monitoring mechanisms are tied to the specific parameters of the Hadoop system. An important direction of our future work is to explore the generic design principles to apply the forecasting mechanisms to different big data applications, such as Dryad [20], HBase [1], Spark [34], etc.

VII. Acknowledgements

We thank the INFOCOM anonymous reviewers for their comments, HKUST REC12EG07 funding, and National Basic Research Program of China (973) under Grant No.2014CB340303.

References

[1] HBase, http://hbase.apache.org/.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM '08*.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI '10*.

[4] H. H. Bazzaz, M. Tewari, G. Wang, G. Porter, T. S. E. Ng, D. G. Andersen, M. Kaminsky, M. A. Kozuch, and A. Vahdat. Switching the optical divide: fundamental challenges for hybrid electrical/optical datacenter networks. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11.

[6] K. Chen, A. Singlay, A. Singhz, K. Ramachandranz, L. Xuz, Y. Zhangz, X. Wen, and Y. Chen. OSA: an optical switching architecture for data center networks with unprecedented flexibility. In *NSDI '12*.

[7] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *EuroSys '12*.

[8] A. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM '11*.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM '11*.

[10] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu. Transparent and flexible network management for big data processing in the cloud. In *Proceedings of the 5th USENIX conference on Hot topics in cloud computing*, HotCloud '13.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI '04*.

[12] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM '10*.

[13] A. Feldmann, N. Kammenhuber, O. Maennel, B. Maggs, R. De Prisco, and R. Sundaram. A methodology for estimating interdomain web traffic demand. In *ACM IMC '04*.

[14] A. D. Ferguson, A. Guha, J. Place, R. Fonseca, and S. Krishnamurthi. Participatory networking. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12.

[15] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. In *NSDI' 07*.

[16] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP '03*.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *ACM SIGCOMM '09*.

[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops*, ICDEW '10.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07*.

[21] R. Love. Kernel korner: intro to Inotify. *Linux J.*, 2005.

[22] D. L. Mills. RFC 1305. Network Time Protocol (Version 3). 1992.

[23] M. Roughan, M. Thorup, and Y. Zhang. Traffic engineering with estimated traffic matrices. In *ACM IMC '03*.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10.

[25] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Mochi: visual log-analysis based tools for debugging Hadoop. In *Proceedings of the first USENIX conference on Hot topics in cloud computing*, HotCloud '09.

[26] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter TCP (D2TCP). In *ACM SIGCOMM '12*.

[27] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI '04*.

[28] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. In *ACM SIGCOMM '10*.

[29] G. Wang, T. E. Ng, and A. Shaikh. Programming your network at runtime for big data applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12.

[30] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. Map task scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality. In *INFOCOM '13*.

[31] K. C. Webb, A. C. Snoeren, and K. Yocum. Topology switching for data center networks. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE '11.

[32] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *ACM SIGCOMM '11*.

[33] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys '10*.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud '10.

[35] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *EuroSys '13*.