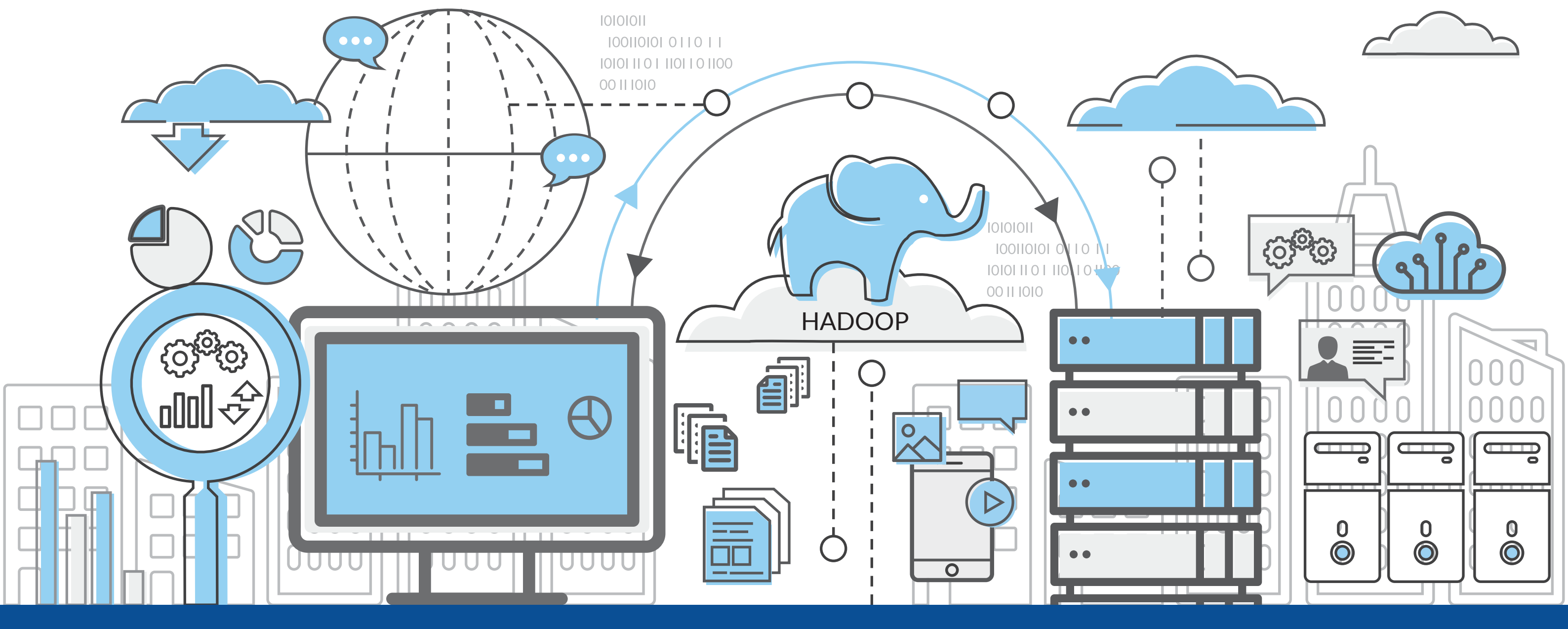
talentica
SPREAD YOUR WINGS

SDN

HADOOP MAP REDUCE SHUFFLE PHASE MEASUREMENT

Author: **Narender Kumar**



Executive Summary

As we move in the era of mobility, virtualization, and cloud, traditional approach of networking is unable to effectively meet the networking needs of our applications. The need of scalability, fault tolerance, and minimum complex and cost effective maintenance of infrastructure is leading to the development of Software Defined Networking (SDN). For big data, SDN provides an oversight of the entire network and enables us to monitor and control data traffic for improved performance.

This white paper proposes that shuffle time in Hadoop can be reduced by using an SDN-enabled infrastructure. Shuffle phase in Hadoop is highly network intensive. With a centralized SDN-controller, we can control each flow in the network and provide alternative routes when there is congestion between mapper and reducer and thus, reduce the shuffle time. This paper highlights the techniques to measure network traffic for shuffle phase of Hadoop MapReduce jobs running over the SDN-enabled topology. Our measurements enable us to find the entire network links utilization metrics and current network congestion state. These network traffic measurements can be used to employ traffic control techniques to improve network performance in SDN-enabled networks in our future work.

Background

“ The key goals of this experiment are:

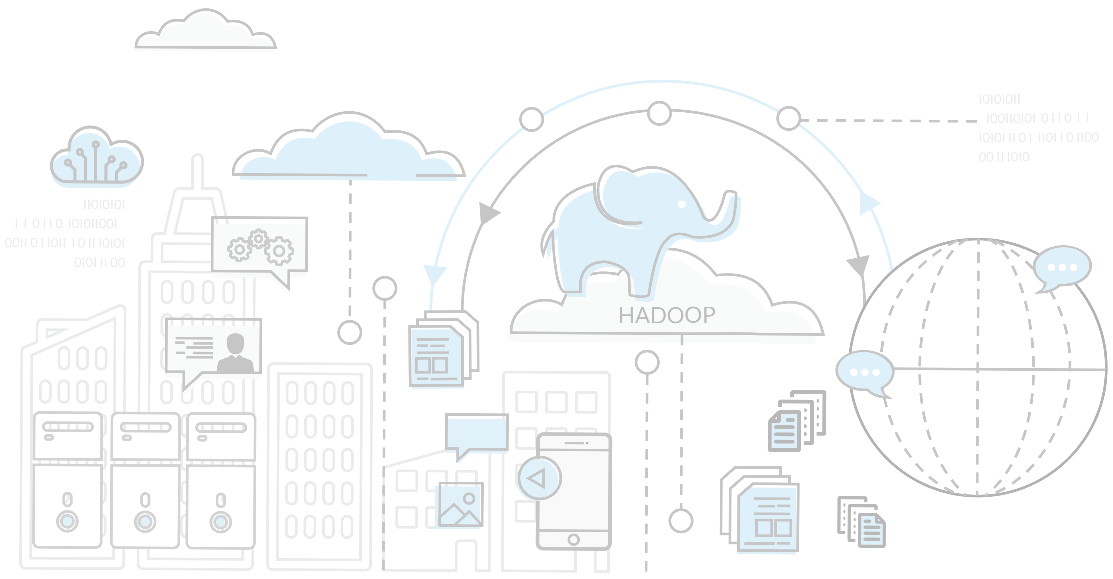
Measure and validate the data and time statistics in shuffle phase in hadoop in a SDN-enabled topology.

Evaluate if there is scope for shuffle phase optimization ”

Hadoop is an open-source framework to write applications that store and process huge amount of data in a large cluster of commodity hardware. A MapReduce application in Hadoop consists of two tasks - map task and reduce task. The input and output of the tasks are stored in a file system called Hadoop Distributed File System (HDFS). Hadoop framework takes care of scheduling, monitoring and executing the tasks on the cluster, at the same time. The user sets job configuration in different parameters and submits the job to Hadoop. Hadoop framework splits the input data into a number of data blocks and copies it in HDFS with replication depending on the job configuration. The framework creates a set of map tasks depending on the number of data blocks and a set of reduce tasks which also may be configured by the user. The framework schedules the map tasks on different nodes in the cluster considering data locality. These nodes execute a user defined map function on the input data block and convert the individual data element into a set of key/value pair. After a map task is completed on a mapper node, the node starts transferring the sorted map output over the network to the reducer node where the reduce task will be running. At the same time, the mapper node might be running other map tasks as well. This process of transferring the data over the network from the mapper node to reducer node as input is called shuffle phase. The mapper nodes in the cluster execute map tasks in parallel and the shuffle data is transferred to the reducer nodes simultaneously over the network. Reduce task takes the shuffled data as input and combines the key/value pairs to generate the final result.

PROPOSED ARHICTECTURE

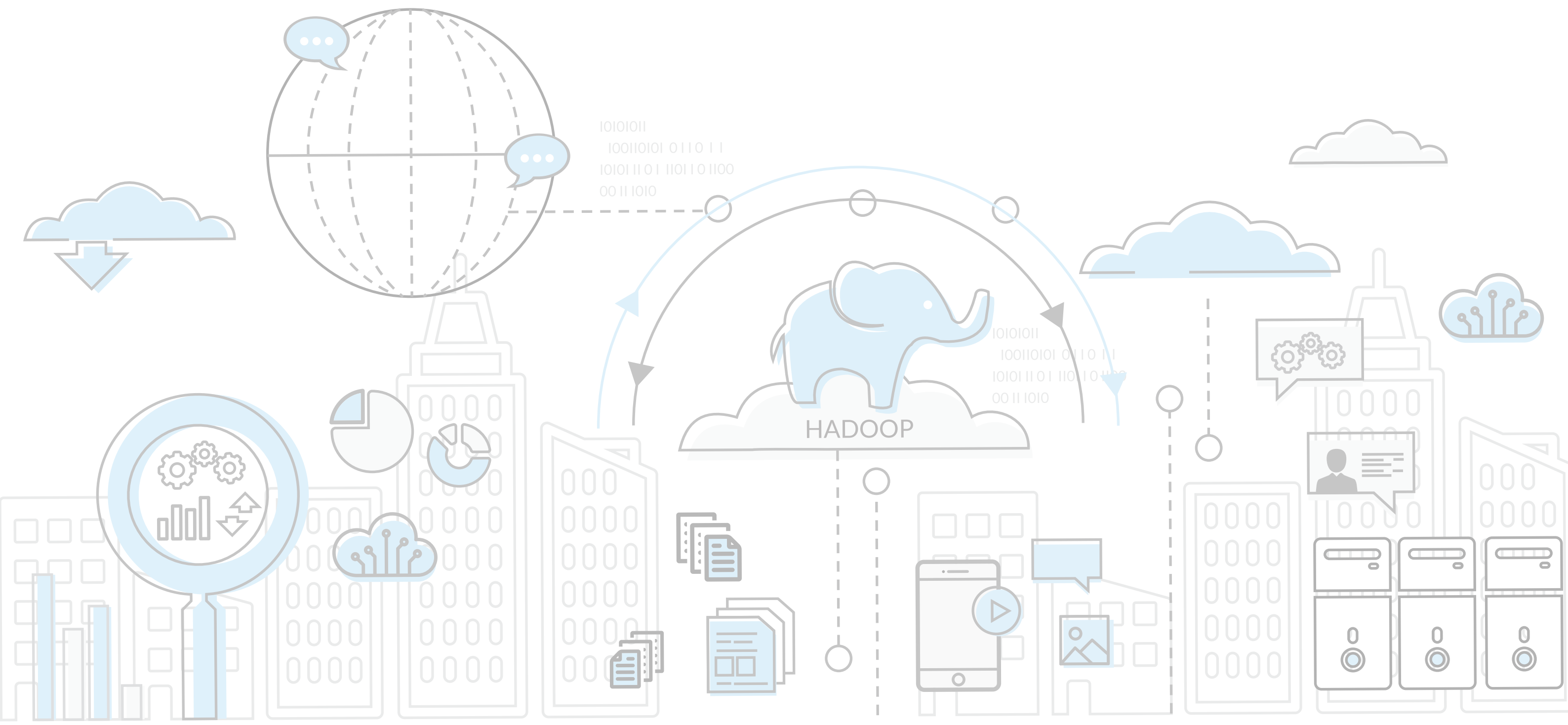
During shuffle phase, as there are simultaneous data transfers occurring from all mapper nodes to the reducer nodes; this may result in the network getting to the state of congestion thereby causing delays in transfers resulting in more shuffle time, and hence slow processing of the job. We can reduce the shuffle time by minimizing the state of congestion. We propose an SDN-enabled Hadoop architecture, where an SDN controller installs the path between mapper and reducer nodes, and monitors the network. SDN is a networking architecture that separates the data plane from the control plane in a network and makes control plane directly programmable. SDN allows users to write logically centralized dynamic control programs, which monitor and control behaviour of the entire network. By using SDN-enabled infrastructure to run Hadoop cluster, we can monitor current network state and deploy traffic engineering strategies to minimize congestion, which in turn will reduce the shuffle time and use distributed resources in a better way.



WHAT TO MEASURE

We want to measure the data transferred and time taken to transfer in shuffle phase in a Hadoop MapReduce job that runs over a SDN-enabled topology. The data from a mapper node to a reducer node is transferred in a number of flows. We are finding near real-time data and time statistics for each flow in our experiment and the current link utilization.

To validate our measurements, we are comparing the total amount of data shuffled and total shuffle time measured in our experiment, against the data shuffled and shuffle time reported by Hadoop. We are using Hadoop logs and job history data to get the shuffle data and time measurements reported by Hadoop.



PROBLEM STATEMENT - TERASORT

We are running Hadoop TeraSort as a Hadoop MapReduce job for flow generation in an SDN-enabled topology. TeraSort is a Hadoop benchmarking tool which can sort any amount of data quickly. A full TeraSort consists of the following steps:

- Generating the input data via TeraGen.
- Running the TeraSort on the input data.
- Validating the sorted output via TeraValidate.

We are running TeraGen and TeraSort as our experiment only needs these two steps

TERAGEN: Generate the input data

TeraGen generates random input data. The syntax to run TeraGen is as follows:

```
hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/
hadoop-*examples*.jar teragen <number of 100-byte rows>
<output dir>
```

The first parameter specifies the number of rows of input data to generate, each of which having a size of 100 bytes. The data format is as follows: <10 bytes key><10 bytes rowid><78 bytes filler>

TERASORT: Sort the input data

The output from TeraGen is used as an input for TeraSort.The syntax to run TeraSort is as follows:

```
hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/
hadoop-*examples*.jar terasort <input dir> <output dir>
```

EXPERIMENT SETUP

We sorted data sets with different size (2 GB, 10 GB, 20 GB and 50 GB) in a cluster of 16 nodes connected through an SDN-enabled fat tree topology as shown in Figure 1 below. In this experiment, we sort 20GB of data by using six nodes (master, slave2, slave5, slave9, slave10 and slave16) out of the 16 nodes from the topology. All the six nodes are running map tasks while reduce tasks are being executed on two nodes (slave5 and slave16).

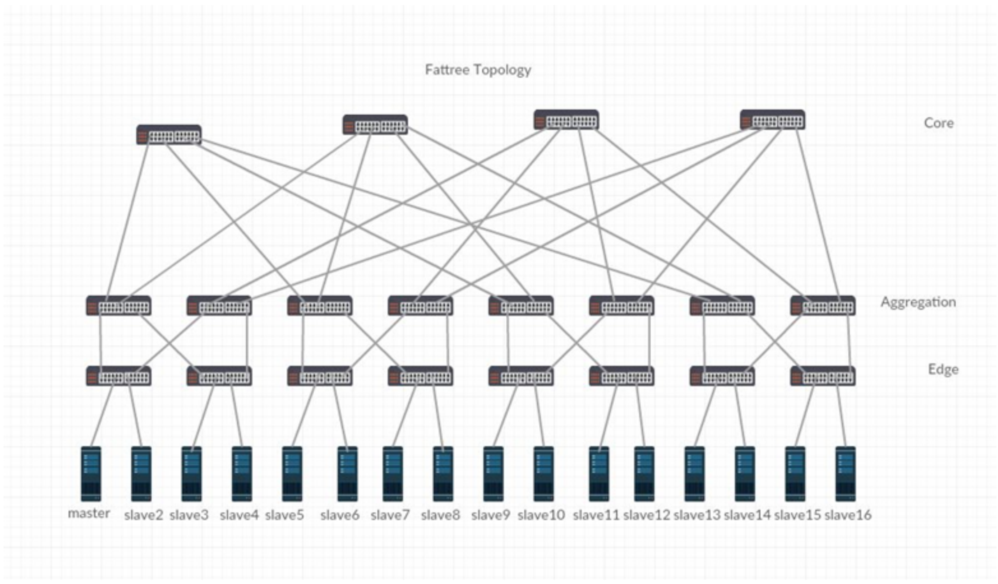


Figure 1: Fat Tree Topology

Topology setup

We used Mininet – an open-source network emulator to create the topology. Mininet runs a collection of end hosts, network devices, and links which are created by using software to make a single system look like a real network. Mininet provides network isolation to the end hosts from the host machine. We created the fat tree topology programmatically by using the Mininet python API. Topology is constructed with reduced link bandwidth capacity so that Hadoop MapReduce jobs can be executed at reduced throughput scale e.g. TeraSort job is executed at the reduced scale of 20 GB.

Hadoop setup

To run Hadoop over an SDN-enabled topology we swapped default Mininet hosts with docker containers to obtain isolation of processes running on end hosts. We used docker - an open-source virtualization application which provides lightweight containers that run processes in isolation with each other to create sixteen lightweight containers as end hosts. To run Hadoop TeraSort on end hosts, we created one docker image having Hadoop installed on Ubuntu. While creating docker containers as end hosts in Mininet topology, we are starting each host container from the Hadoop docker image.

Flow Forwarding & Monitoring

We used OpenNetMon – a POX OpenFlow controller module, which monitors per packet flowing in the network. OpenNetMon provides forwarding component, which is used to route the packets and monitoring component responsible for monitoring the flows. Forwarding is done over a spanning tree of topology, and Bellman Ford algorithm is used to calculate the shortest path between the source and destination. Both these components are dependent on POX discovery module, which is responsible for learning the network topology. We used Apache Rumen to extract and analyse Hadoop job history data that provides us time and data statistics measured from Hadoop.

Configuration

Topology	Nodes	Links Bandwidth	CPU	Memory	Block Size
Fat Free	16	1000 mbps - core switches and aggregation switches 100 mbps - aggregation switches and edge switches 100 mbps - edge switches and end hosts	64-bit	16 GB	128 MB

Running the Experiment

We can specify the hosts from the topology that we want to use to run Hadoop job by adding the hosts in /usr/local/hadoop/etc/hadoop/slaves file. As we are starting each host from the Hadoop docker image, each host gets the Hadoop configuration from the image. We may change the Hadoop configuration files in all hosts according to our needs. After formatting the Hadoop distributed file system via NameNode that are running on master node, start the MapReduce daemons, and run Hadoop TeraSort.

A brief overview of the steps is as follows:

TeraGen

Using output directory as teraInput and Hadoop version 2.7.1, run below command to generate 20GB of input data.

```
hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/
hadoop-mapreduce-examples-2.7.1.jar teragen 200000000
/teraInput
```

There are no reducer tasks taking place in TeraGen.

TeraSort:

Using input directory as teraInput, output directory as teraOutput and Hadoop version 2.7.1, run the below command to sort 20GB of input data.

```
hadoop jar $HADOOP_PREFIX/share/hadoop/mapreduce/
hadoop-mapreduce-examples-2.7.1.jar
-Dmapreduce.job.reduces=2  terasort /teraInput /teraOutput
```

There are two reduce tasks which are taking place on reducer nodes (slave5 and slave16).

MEASUREMENT AND ANALYSIS

We used the OpenNetMon to measure data and time over network. It periodically sends probe-packets to edge switches at a pre-defined rate to obtain the flow statistics. With each query, OpenNetMon receives the amount of data sent and time taken for data transfer, and generates csv files that are used to obtain the time and data statistics over network. We analysed per node-to-node communication and measured the amount of data being sent to each other in different flows. We also measured the time taken to transfer the data from a mapper node to a reducer node over network in the experiment. Hadoop logs reports of start and finish time for each mapper task, shuffle finish time, and the amount of data which is ready to shuffle after a map task is completed at each mapper node. When TeraSort is done, we used Hadoop logs and Hadoop job history data to extract task-level details. We extracted information of all the map tasks, reduce tasks, and the nodes where they are getting executed. We aggregated node wise map tasks and the amount of shuffle data transferred to the reducer nodes.

We present detailed comparison of shuffle data and time measured using OpenNetMon against values reported by Hadoop.

Table 1 below presents a comparison of total shuffle data from all the mapper nodes to both the reducer nodes:

Reducer Nodes	Hadoop(in GB)	OpenNetMon(in GB)
Slave5	8.15	8.26
Slave16	8.68	8.76

Table 1: Shuffle Data Comparison

Table 2 below presents comparison of mapper wise shuffle data to slave5:

Mapper Nodes	Hadoop(in GB)	OpenNetMon(in GB)
Master	2.01	2.06
Slave10	1.49	1.50
Slave2	2.07	2.09
Slave9	1.55	1.56
Slave16	1.04	1.05

Table 2: Data Comparison - Slave5

Table 3 below presents comparison of mapper wise shuffle data to slave16:

Mapper Nodes	Hadoop(in GB)	OpenNetMon(in GB)
Master	2.06	2.08
Slave10	1.53	1.55
Slave2	2.09	2.10
Slave9	1.57	1.59
Slave5	1.44	1.45

Table 3: Data Comparison - Slave16

For detailed data and time comparison please refer to Appendix A.

Figure 2 below represents the paths from mapper nodes to reducer node. In our experiment, all flows from a mapper node to reducer nodes are following the same path. We can clearly see the links which are being used the most. The congestion may occur in these links if bandwidth of these links is insufficient to carry out all the transfers. As specified above, OpenNetMon installs path over a spanning tree in this topology, we can clearly see that there are alternative paths available that are not being used to reach from mapper nodes to the reducer node.

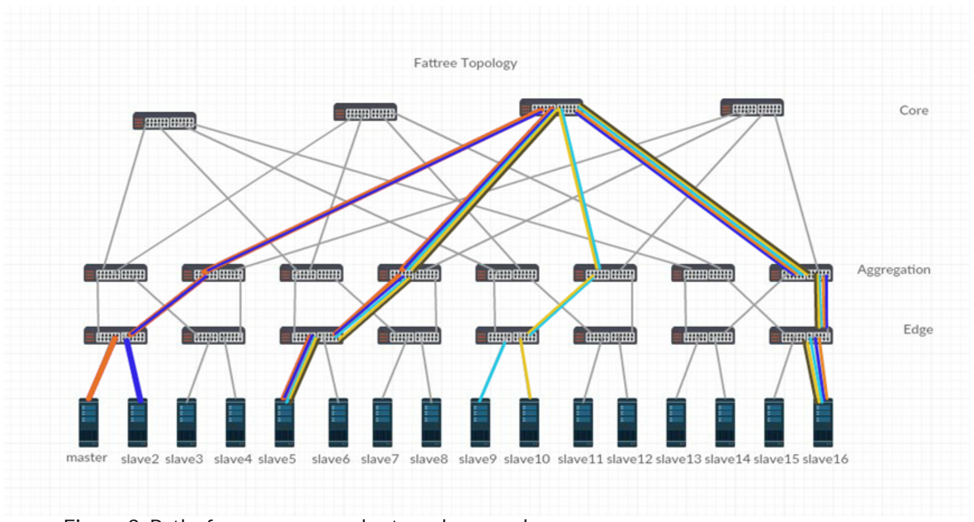
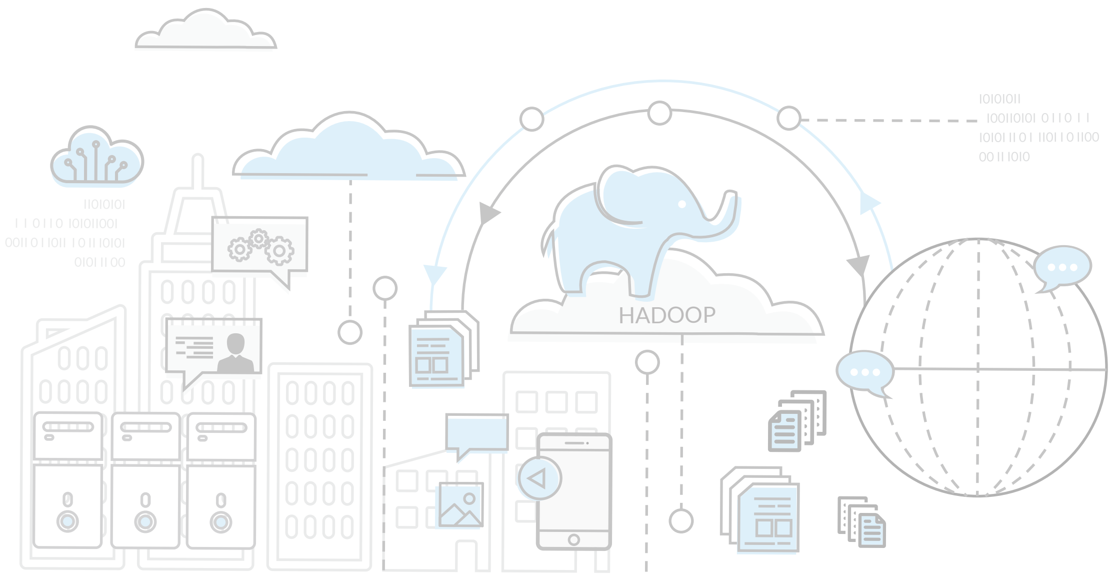


Figure 2: Paths from mapper nodes to reducer nodes

CONCLUSION

We are able to set up Hadoop on an SDN-enabled topology by using Mininet and lightweight docker containers. This setup can be used as an SDN testbed. Hadoop MapReduce jobs are successfully run on this setup. Using SDN controller, flow tables are installed and data transferred and time for transfer are measured for all flows generated by Hadoop MapReduce Shuffle phase. We are able to find the flow paths between mapper and reducer nodes. Data transferred and time taken to transfer in shuffle phase is calculated by using Hadoop logs. Shuffle phase data transferred and time for transfer closely matches with both Hadoop and SDN controller. We are able to find the near real time data flows in entire network and the links which are being used the most and may be congested. We are able to find the alternative paths available which can be used to reduce shuffle time using one of the effective routing techniques.
We present detailed comparison of shuffle data and time measured using OpenNetMon against values reported by Hadoop.



FUTURE WORK

These measurements can be used by traffic engineering solution to find the optimal path for flows and rerouting the flows to reduce the shuffle phase time. These network traffic measurements can be used to employ traffic control techniques to improve network performance in SDN enabled networks in our future work.

APPENDIX A

Figure 3 represents the shuffle time comparison between the times calculated using Hadoop logs and data transfer time calculated using OpenNetMon. On the time axis, the start and end of shuffle time is relative to Hadoop job submit time (Hadoop job submit time is represented as zero millisecond on time axis). Our time calculations in the experiment are relative to Hadoop job submit time. The amount of data which is being shuffled or transferred has been mentioned on top of each bar. The green bars are representing shuffle time calculated using Hadoop logs. The orange bars are representing shuffle time measured in our experiment. The first green bar is representing shuffle time from mapper nodes (master, slave2, slave9, slave10 and slave16) to reducer node (slave5). The second green bar is representing shuffle from mapper nodes (master, slave2, slave5, slave9 and slave10) to reducer node (slave16). The two other bars are representing time taken for shuffle data transfers from mapper nodes to reducer nodes (slave5 and slave16) calculated using OpenNetMon. Figure 4 is further drill down to these two bars. The first two bars are same as in Figure 3 the remaining bars are representing the shuffle time from each mapper node to each reducer node.

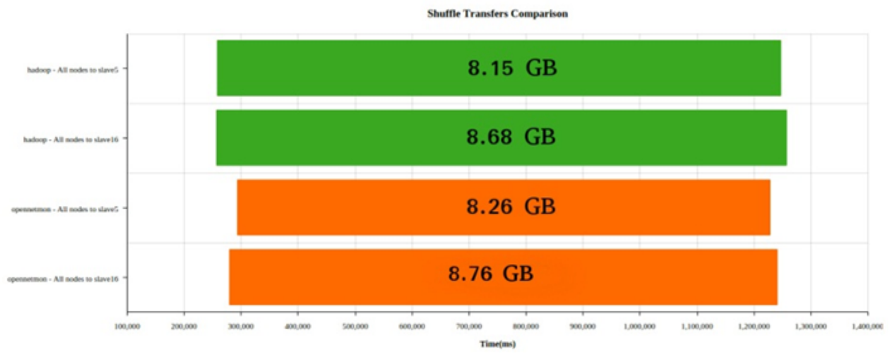


Figure 3: Shuffle time comparison

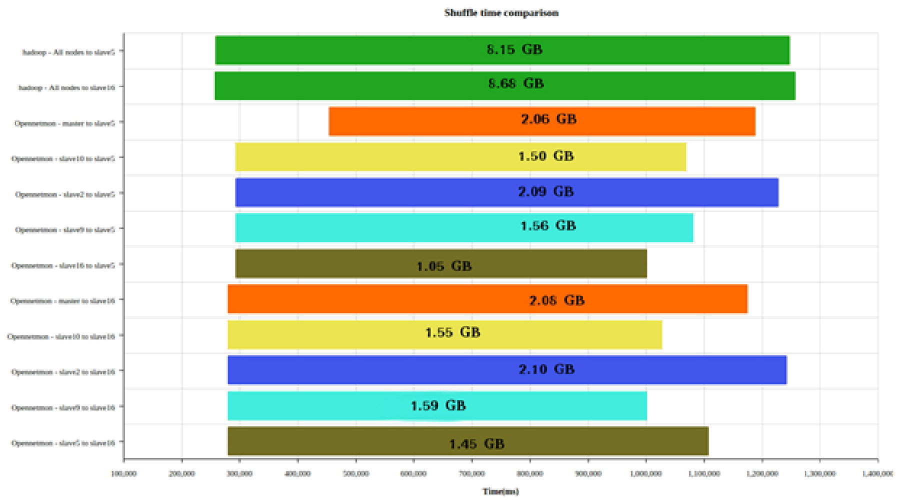


Figure 4: Shuffle time comparison (each mapper node to each reducer node)

We can clearly see that the total time taken for the data transfers measured in our experiment is matching with the shuffle time reported by Hadoop logs.

Figure 5 represents data received on each reducer in shuffle phase. The first two columns represent data received by reducer node (slave5) from mapper nodes (master, slave10, slave2, slave9 and slave16). The other two columns represent data received by reducer node (slave16) from mapper nodes (master, slave10, slave2, slave9 and slave5). The first and third column represents shuffle data measured in our experiment using OpenNetMon and second and fourth column represents shuffle data reported by Hadoop logs.

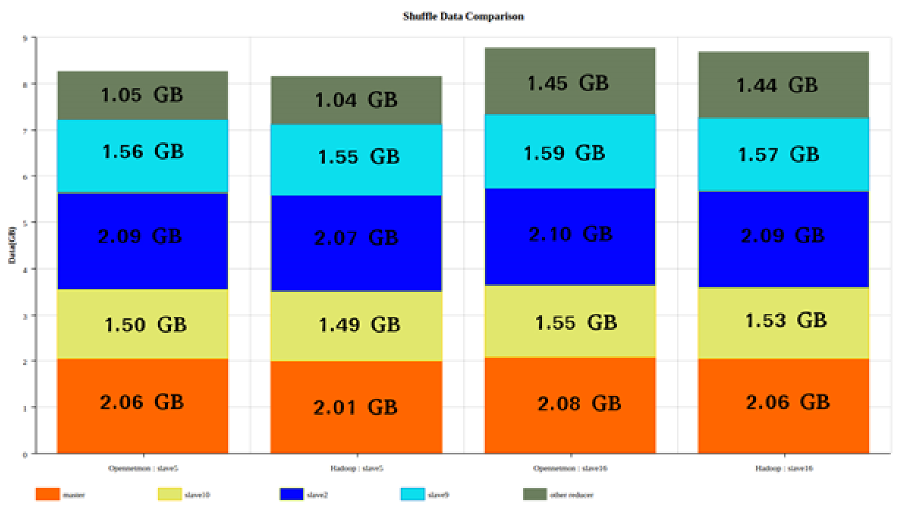


Figure 5: Data transferred from each mapper to reducer

We can clearly see that the shuffle data measured by our experiment using OpenNetMon is matching with the shuffle data reported by Hadoop logs.

Figure 6, Figure 8 and Figure 10 are representing per mapper node map tasks (shown in green bars) reported by Hadoop and shuffle flows (violet bars are representing shuffle flows to reducer node-slave16 and brown bars represent the shuffle flows to reduce node-slave5) found in our experiment. From the time values we can see that there are simultaneous data transfers from a mapper node to both the reducer nodes.

Figure 7, Figure 9 and Figure 11 are representing the dynamics of shuffle flows from mapper nodes to each reducer node found in our experiment. Each line in the figures represents a shuffle flow. Each figure represents the amount of data transferred from a mapper node to the reducer node at a point of time for a particular flow.

Referring Figure 6, there are shuffle data transfers to both the reducer nodes from master node. Data transfer to reducer node-slave16 starts after finishing of first map task. In our experiment, we found that there are four flows in each transfer from master to slave5 and slave16.

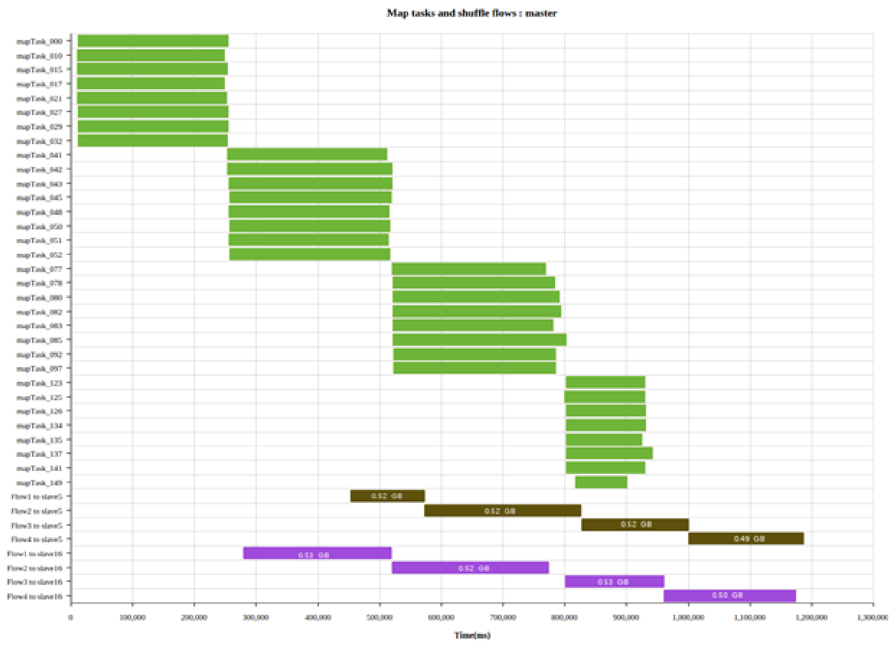


Figure6: Map tasks and shuffle flows from master

Figure7 represents the dynamics of shuffle flows from master to the reducer nodes. Master node is sending data to both the reducer nodes simultaneously. As mentioned above, the time calculations are relative to job submit time. The slope of line is an indicator of data transfer rate. Steeper slope means higher value of data transfer rate. We can clearly see that when the link is occupied by a single flow, data transfer rate is higher.

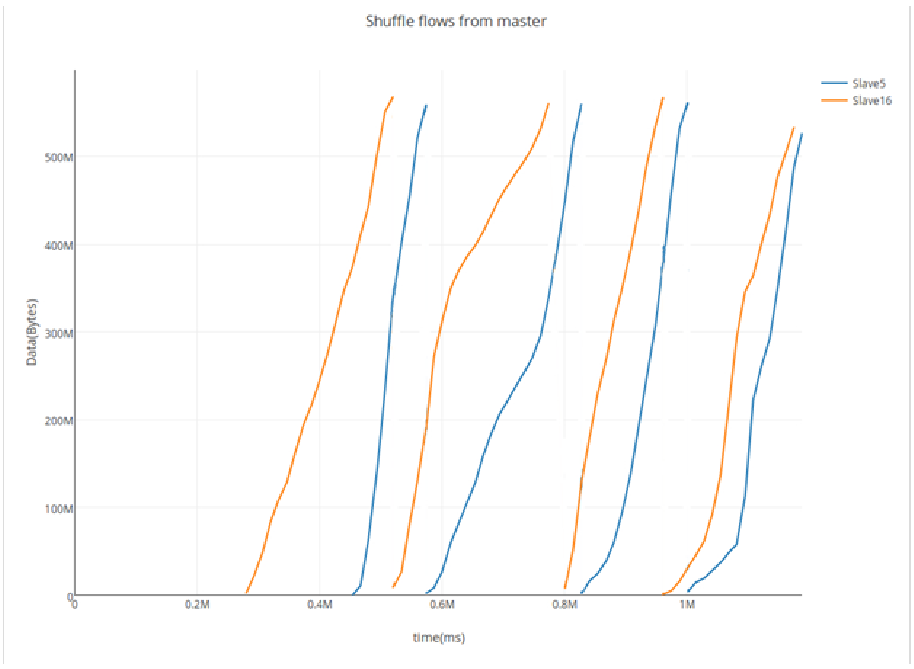


Figure7: Shuffle flows from master node

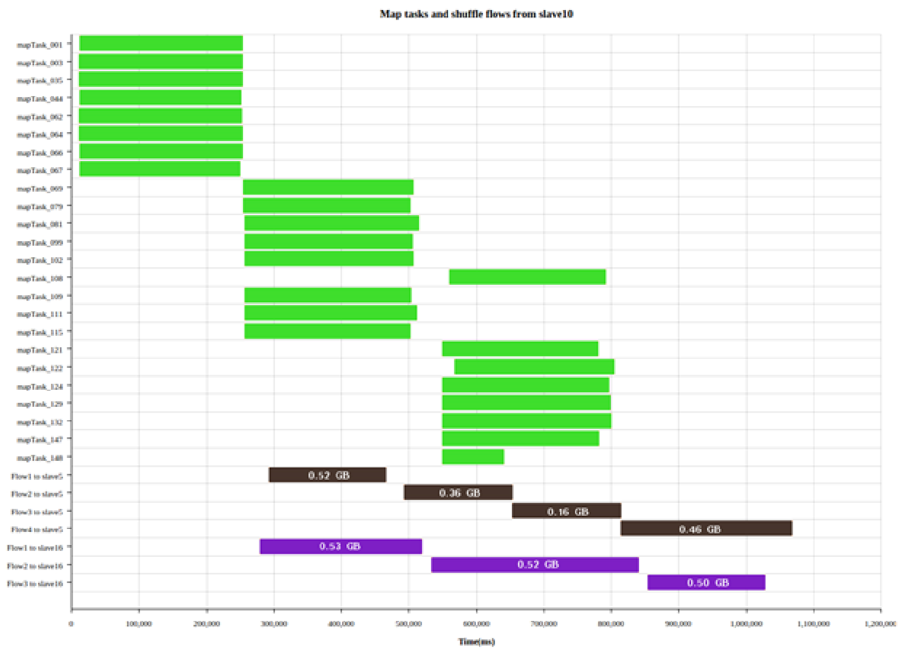


Figure 8: Map tasks and shuffle flows from slave10

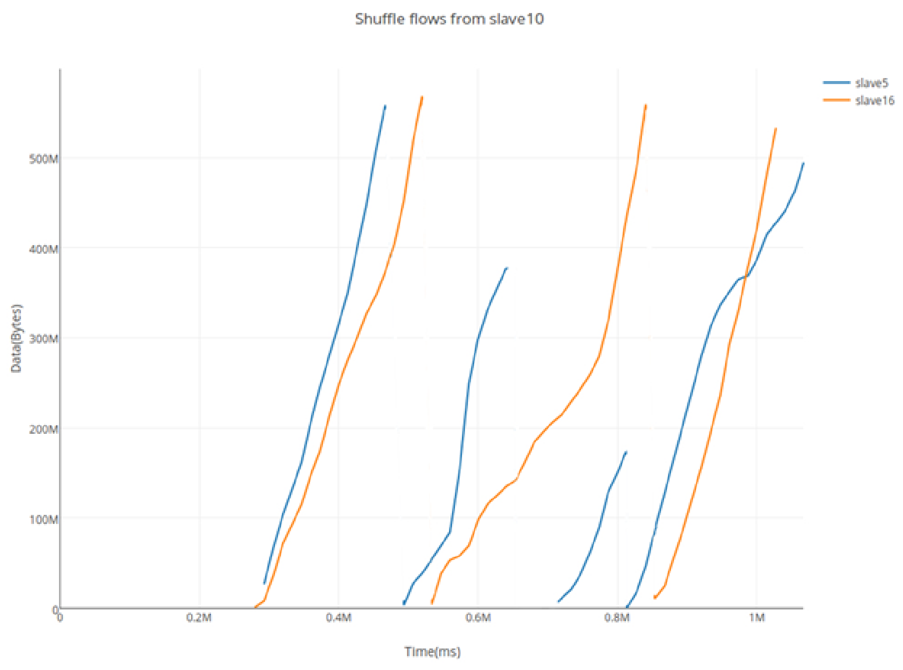


Figure 9: Shuffle flows from slave10

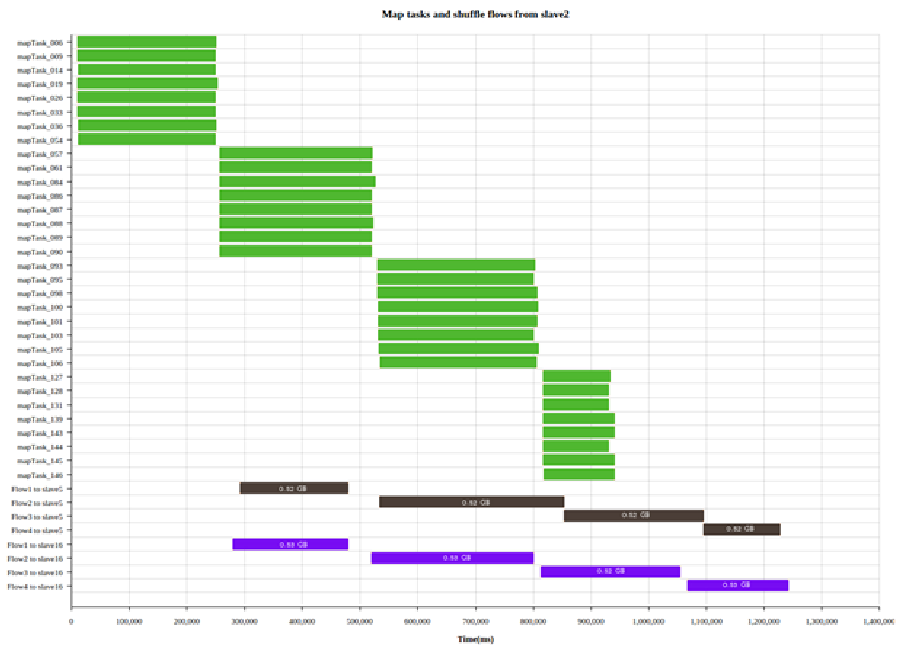


Figure 10: Map tasks and shuffle flows from slave2

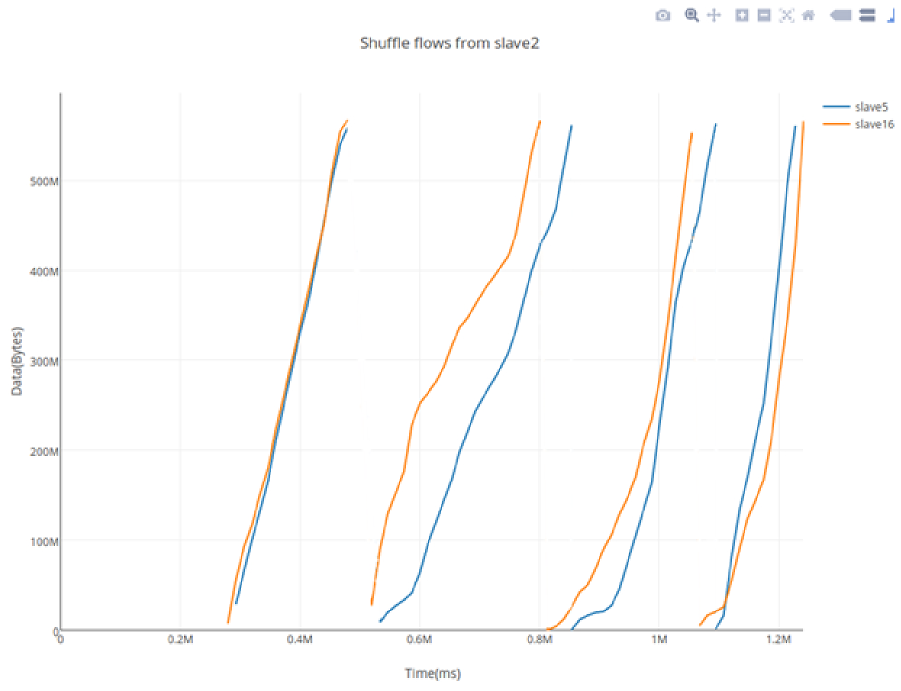


Figure 11: Shuffle flows from slave2

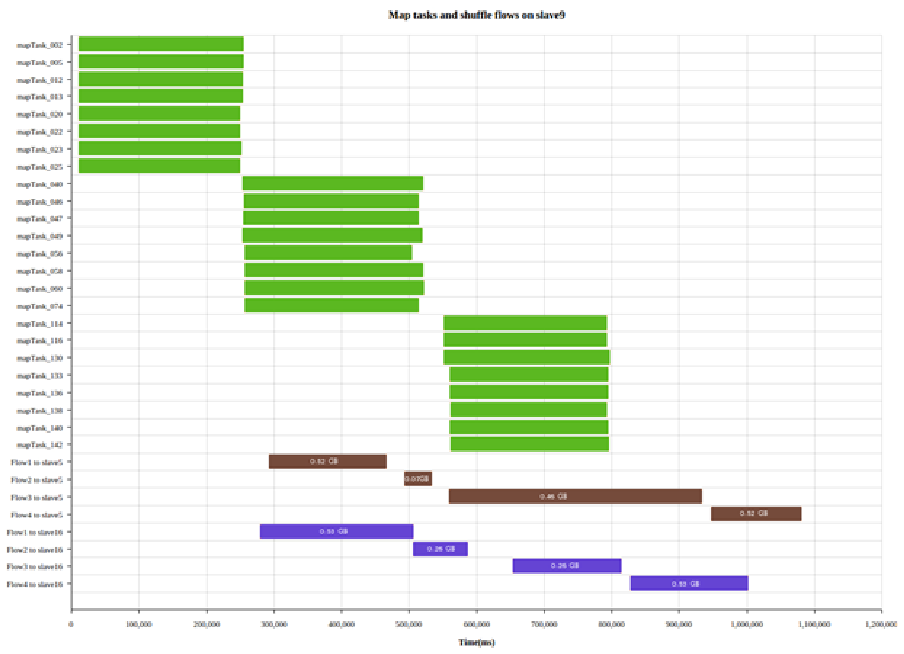


Figure 12: Map tasks and shuffle flows from slave9

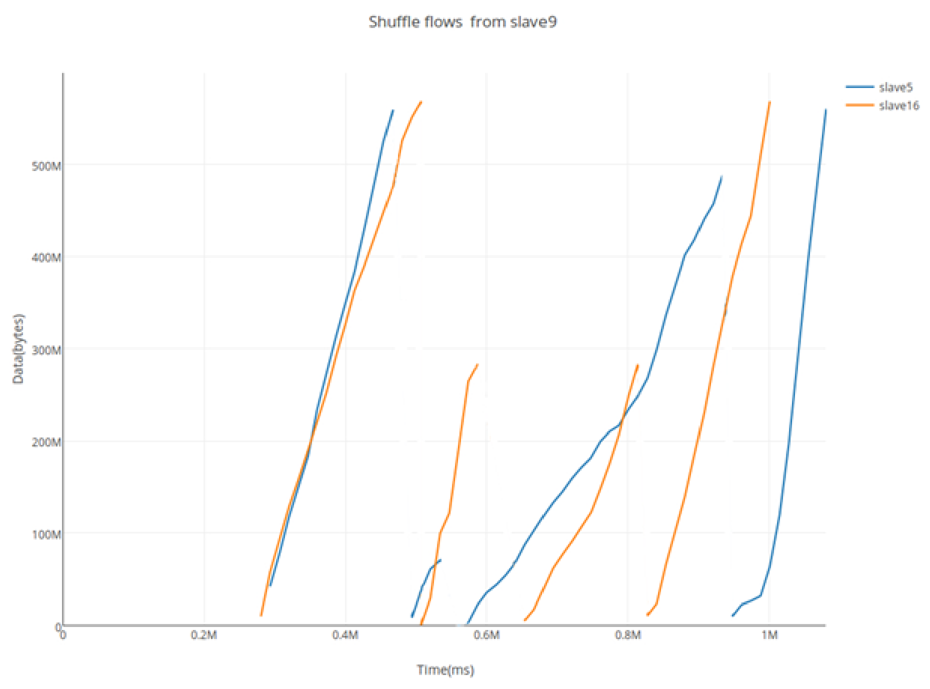


Figure 13: Shuffle flows from slave9

Figure14 represents map tasks and shuffle flows from slave16. The shuffle transfer from slave16 is to slave5 only. There are three shuffle flows from slave 16.

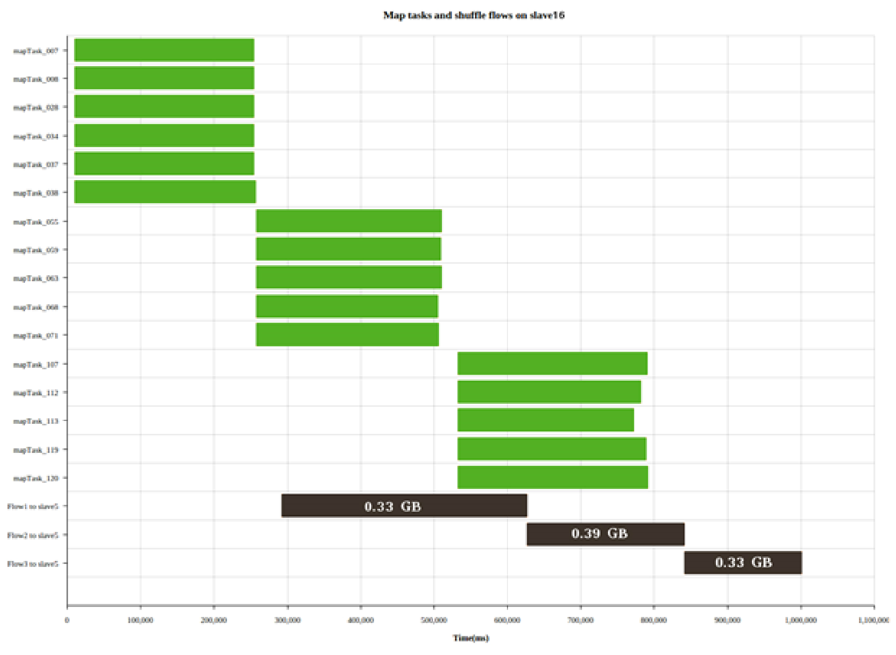


Figure 14: Map tasks and shuffle flows from slave16

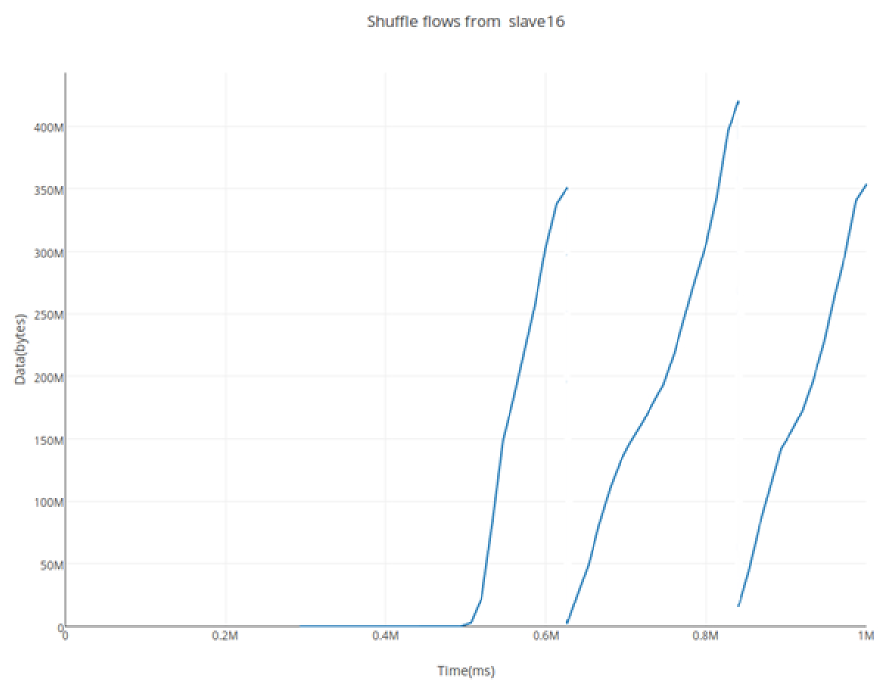


Figure 15: Shuffle flows from slave16

Figure16 represents map tasks and shuffle flows from slave5. The shuffle transfer from slave5 is to slave16 only. There are four shuffle flows from slave5.

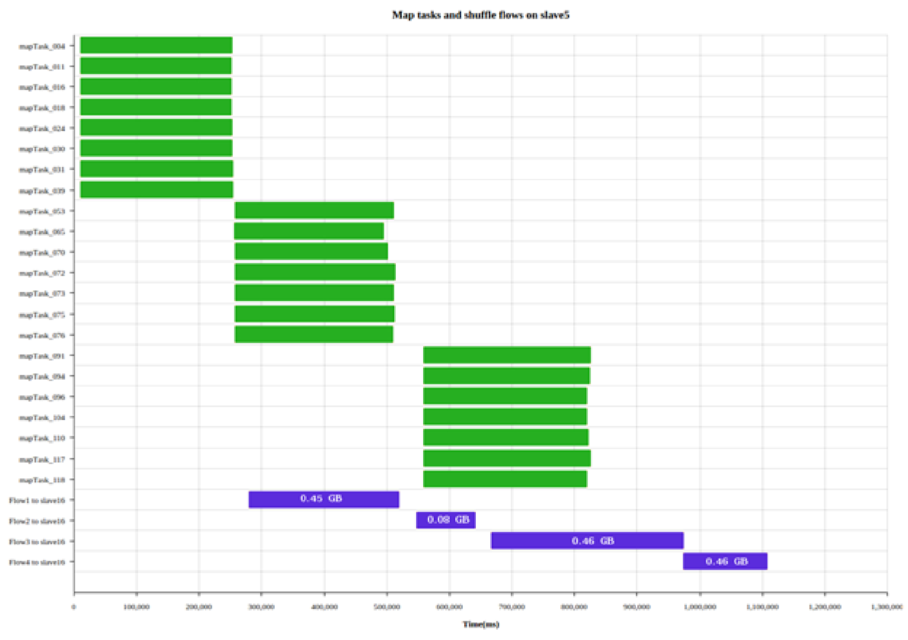


Figure 16: Map tasks and shuffle flows from slave5

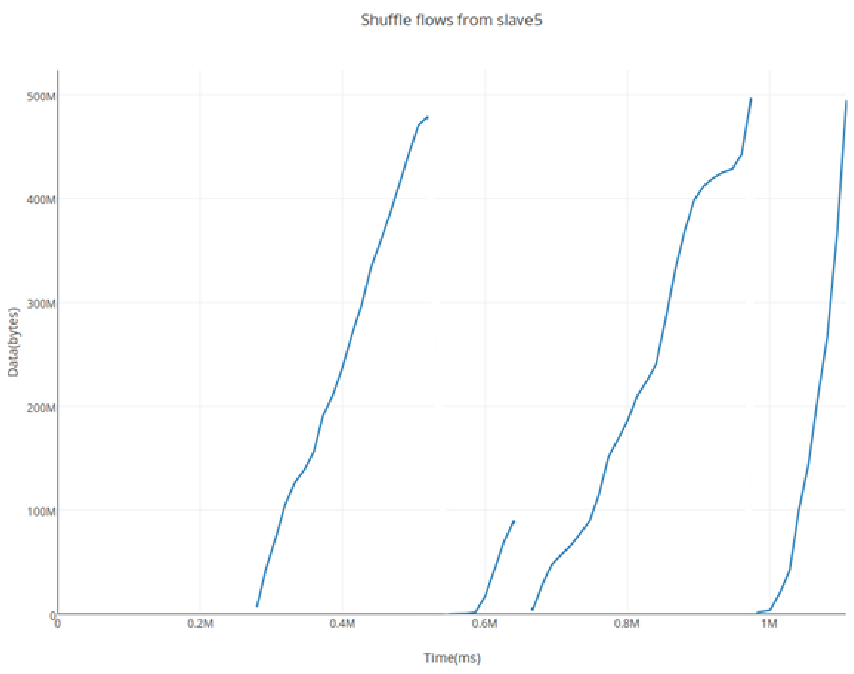
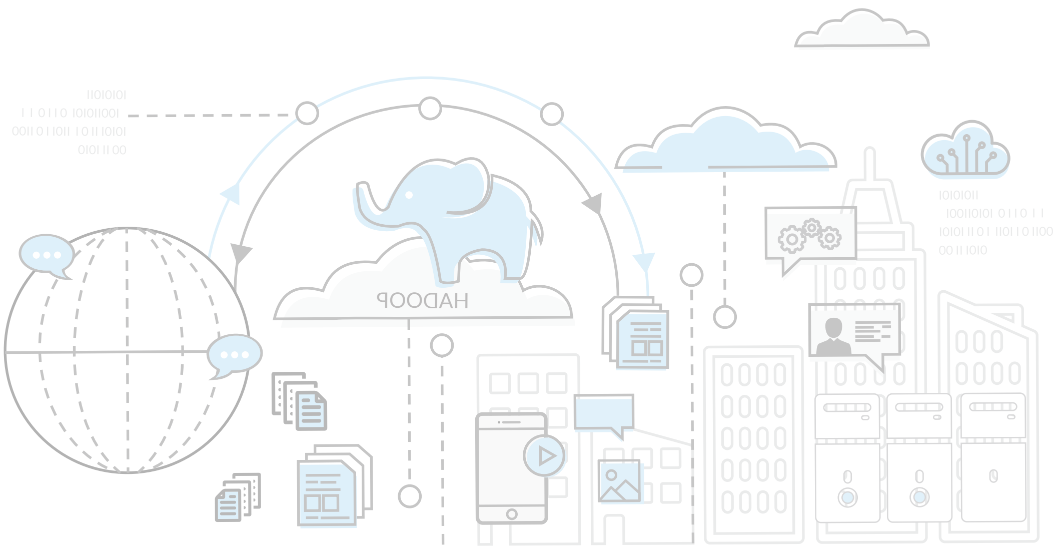
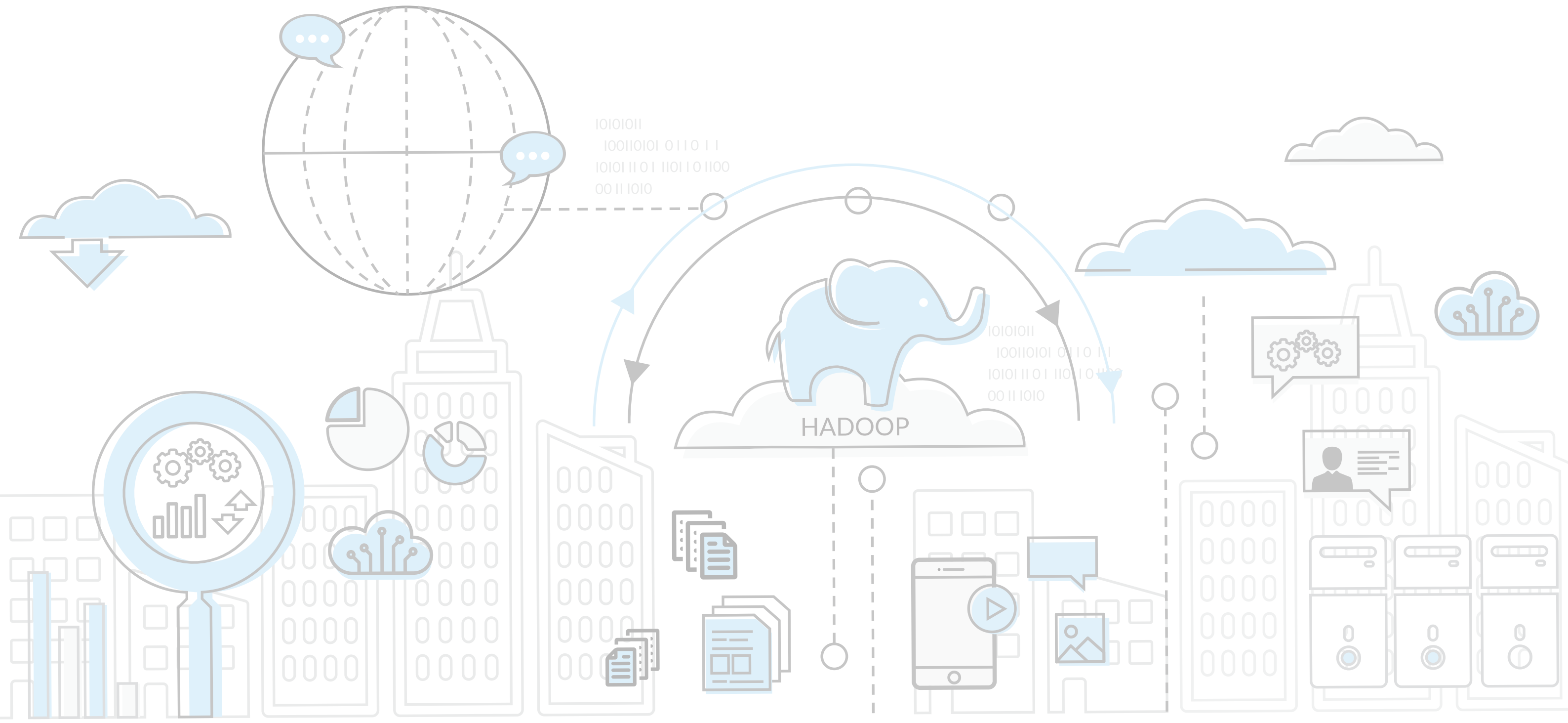


Figure 17: Shuffle flows from slave5



ADDITIONAL RESOURCES

- https://en.wikipedia.org/wiki/Software-defined_netwo rking
- https://github.com/TUDelftNAS/SDN-OpenNetMon
- https://openflow.stanford.edu/display/ONL/POX+Wiki
- https://hadoop.apache.org/docs/r1.2.1/rumen.html
- https://github.com/docker/docker



About Talentica

Talentica Software is an innovative outsourced product development company that helps startups build their own products. We help technology companies transform their ideas into successful products by partnering in their roadmap from pre-funded startups to a profitable acquisition.

We have successfully built core intellectual property for more than 60 customers so far. We have the deep technological expertise, proven track record and unique methodology to build products successfully. Our customers include some of the most innovative product companies across USA, Europe and India.

Office No. 501, Amar Megaplex
Baner, Pune 411045

Tel: +91 20 4660 4000 | Fax: +91 20 4075 6699

www.talentica.com