Software Defined Network Support for Real Distributed Systems

Chen Liang

Project for Reading and Research Spring 2012, Fall 2012

Abstract

Software defined network is an emerging technique that allow users to reconfigure queuing, routing and forwarding of the entire network it runs on. Such technology makes it possible for applications to program the network as desired. In this work, we took ZooKeeper and Hadoop as examples to explore how SDN optimization may improve the performance of distributed systems at run-time. Our analysis has indicated that such instrumentation has great potential of improving the performance.

1 Introduction

In recent years, distributed frameworks such as Hadoop and Zookeeper have been emerged as a popular approach for data intensive computing. As a result, effectiveness of distributed systems is becoming a major concern for developers and researchers. A variety of techniques has been applied in attempt to improve its efficiency.

So far, a lot of effort have been focusing on improving the performance by optimizing the structure of the distributed systems. Yet the recent emergence of software-defined network(SDN) provides us the opportunity to improve performance from another perspective. SDN provides applications the interfaces for dynamic reconfiguration over the network at run-time such that applications can change the network to fit its traffic pattern.

Motivated by such new trend, our work tried to evaluate the potential benefit of taking advantage of SDN in real distributed system with our own implementation of SDN controller called PANE. We took ZooKeeper [2] and Hadoop [4] as two examples and instrumented SDN into these two frameworks.We conducted experiments to evaluate how much these systems may benefit from having the network do bandwidth rescheduling at run-time. Our results suggest reduction of execution time.

2 Overview of PANE

Participatory network(PANE) [1] is our SDN controller that provides API to users for dynamic network reconfiguration from application layer. PANE allows applications to contact the controller to request for resources or set up rules for future traffic. The design of the system is motivated by the need of high performance in systems with single logic administrator, such as data center network or campus network. But it can be an optimization for a much wider range of scenarios such as video playing and data transfer.

2.1 SDN Architecture

The most significant change from current networks to SDN is that in classic networking model, data path and control path are placed on the same physical device, however in SDN, only the data path resides on the switch and the control logic is separately placed on a SDN controller, where all the routing decisions are

being made. A protocol for communication between device and controller is deployed to define messages such as flow table modification, packet processing and status query. A SDN switch has a flow table, in which each entry has a set of packet fields to match, and an action for processing. If an incoming packet matches an existing flow entry, it will be processed accordingly. Upon receiving a packet that does not match any flow entry, the switch will send the packet to the controller asking for the routing decision. The controller may consequently create new flow entry on the switch. The controller may also change or remove flow entries at run-time.

Since an SDN controller has the global view and control over the entire network, it also serves as the interface for applications to reconfigure network routing logic according to their need at run-time. Since it works as a resource delegation model, PANE is developed using hierarchical policies.

2.2 Semantics of PANE

In PANE, network resources are described as a hierarchy of ”shares”. A share consists of three major components:

1. Flowgroup

   A flowgroup is a set of rules that flows may match. It is represented as a sequence of paired match rules and actions. It is defined by the TCP's 5-tuple identifier. If a packet's header match a certain rule, the corresponding actions will be applied. And if multiple matches are matched, the one with the highest priority wins.

2. Speaker

   Speakers are the users that have authorities on this share. A share can be configured by multiple speakers, in general, this happens when a parent speaker delegates its authority to its child speakers. In this case, if a conflict happens, the child's action overrides parent's.

3. Privilege

   Privileges are the operations speakers can perform to reconfigure the network. So far, PANE supports following privileges: *Allow*, *Deny* and *Guaranteed Minimum Bandwidth(GMB)*. The former two privileges allow clients to block or unblock certain traffics in the network and the latter one allow clients to reserve bandwidth for certain flows along their path.

2.3 Privilege Delegation

A key feature of PANE is that network administrator may delegate authorities to different users, and each user may further delegate its authority to processes. The whole delegation can be viewed as a tree structure, bandwidth can be delegated by a parent node to its children and the total bandwidth used by all children may not exceed the total bandwidth of parent share.

PANE's ”allow” and ”deny” semantics can be used to control network routing logic and it allows users to control the routing of traffics,with which we can easily implement firewalls that protects hosts from attacks such as denial of service by requesting the network controller to drop the undesired packets at the source. And the GMB semantic allows users to send request asking for a certain amount of bandwidth to be reserved in a certain period of time. This is implemented by creating queues for the flowgroups on its shortest path in the network. If the request for bandwidth can not be satisfied in the desired period, the request will be rejected. But once the bandwidth reservation is established, it is guaranteed that all flows that matches the flowgroup will be given the bandwidth. This becomes particularly useful when the user wants to protect his traffic from suffering the queuing latency due to competing.

2.4 Client Library

A client library in Java is implemented to simplify the instrumentation into distributed applications. The client library encapsulates all the PANE semantics into Java objects. PANE's notion such as share and client are object-orientated by their nature. And operations *allow* and *deny* are also exposed as objects for applications to easily manage interaction with PANE. The client library also hides the details of setting up connections. As mentioned above, in SDN, applications need to send their requests to the SDN controller to ask for services. PANE uses TCP is used and the controller always waits on a known port waiting for incoming connections. The connection is used for accepting clients' requests and sending back the information about whether the operation has succeeded. The functions exposed to clients are used to send request and only return after the controller sends back respons. On the other hand, command sent to PANE needs to follow certain format to be recognized by PANE and to specify flowgroup. The library encapsulate all these executions and exposes a clean and simplified interface.

3 PANE and ZooKeeper

3.1 Overview of ZooKeeper

ZooKeeper is a distributed coordination service developed by Yahoo. It allows large scale distributed application to implement high-performance, reliable coordination tasks such as leader election, run-time reconfiguration and ensemble management. To support high performance services, ZooKeeper is designed to avoid using locks, and instead uses atomic broadcast is to implement wait-free and reliable service.

The ZooKeeper service is implemented as a hierarchical space of *znodes*. By design, a znode is an abstract data node that generally used by application to store small data(less than 1MB in general). Since it is supposed to be small, each server keeps a complete copy of all znodes and their hierarchy in memory. Application nodes perform coordinations by reading and writing znodes, and ZooKeeper guarantees the synchronization of updates and reliability of all znodes on the cluster. This is guaranteed by an atomic broadcast protocol called Zab [3]. Zab is embedded into ZooKeeper and is used to process all message exchange between ZooKeeper hosts. By taking advantage of znodes, application can implement flexible and complicated coordination semantics.

3.1.1 Leader Election

ZooKeeper aims at providing distributed coordination service, its cluster is composed by two types of host: *Leader* and *Follower*. There can only be one leader in a ZooKeeper cluster and all other hosts are followers. It is crucial for ZooKeeper to have all hosts reach agreement on the leader before it can process any client's update. A Paxos-like algorithm ZAB is applied to elect a leader. The leader needs to be one of the hosts that have seen the most recent update in the system. Hosts exchange proposals and acknowledgments to elect a leader. Exactly one leader will be elected as long as there are more than half hosts that are active in the cluster. The role of the leader is to guarantee total order delivery. But clients can still send their request to any server in the cluster. If it is a follower that receives the request, if it is a read request, since all servers keep a complete copy of the data tree, the follower will handle it locally, if the request involves update of znodes, follower will always forward it to the leader. The leader will handle it using a two phase commit protocol as described below.

3.1.2 Two Phase Commit

Once a leader is elected, ZooKeeper is ready to provide service. During the execution, since the coordination is on a distributed system, the most important property ZooKeeper needs to maintain is total order

delivery. This is done by a two phase commit protocol: upon receiving a update request, a ZooKeeper server will always forward it to the leader, and leader makes one proposal for each request and broadcasts proposals to all followers in FIFO order. Upon receiving an proposal from leader, follower sends back an acknowledgment. If acknowledgments of more than half followers reach the leader, the leader will send a commit message to all followers.The update will then be delivered on followers.

As can be seen from above, to achieve wait free coordination with high performance, ZooKeeper replies on message exchange to elect leader and process all client requests. As a result, the latency of message transmit will have a negative impact on ZooKeeper performance. Especially when the volume of clients' requests gets large. On the other hand, message exchange can be decelerated due to competing with other traffic in the same network. This is common for ZooKeeper to share the network with other application because that the purpose of using ZooKeeper is to provide service for other distributed systems. To provide guaranteed performance for ZooKeeper, we resort to SDN technique. We used PANE's GMB" semantic to protect ZooKeeper's traffic from being influenced by other traffic.

3.2 Instrumentation

3.2.1 Network Traffic Pattern

During the service, ZooKeeper servers always follow a many-to-one traffic pattern: all followers forward the requests they received to the leader, leader broadcasts proposals to all followers, follower sends back acknowledgment and the leader responds by sending commitment. But when leader election happens, it becomes many-to-many pattern, every server will exchange messages with all other servers in order to reach agreement on the leader.

Besides the communication between ZooKeeper servers, another type of traffic is between a ZooKeeper server and a client. Clients may send their requests to any of the servers in the system, regardless of whether it is the leader or not. Operations exposed by ZooKeeper are all based on znodes, clients may request for their status, create new one or modify their value. It is a typical client-server model and the data directly involved on this link is relatively small. But all the requests that will change status of existing znode hierarchy will result in broadcasting in the system.

3.2.2 Implementation

PANE needs to know the TCP 5-tuple identifier to make a reservation. In ZooKeeper, all the needed parameters can be easily obtained as they are all constant during the execution. On each server, two different ports are used for different purposes: one port is used for leader-follower communication and the other is for two phase commit. These ports are configured in system configuration file and after the system starts, they do not change. Thus it is very straightforward to get the ports and make the bandwidth reservation. Reservations need to be made on both directions because the are two different flowgroups.

For simplicity reason, we did not distinguish between leader and follower in terms of making reservation. For each server, reservations are made for all other servers. Therefore, no matter how the leader changes over time, the reservations will always be the same. But the obvious downside is that many of the reservations are idle, because in ZooKeeper, followers do not talk to each other except in leader election phase.

Another thing that PANE needs to know for making reservation is the bandwidth. On one hand, since ZooKeeper is used to provide coordination service for other distributed applications, latency in its execution will have a dramatic impact on not just itself, but also all distributed applications that reply on ZooKeeper. Therefore we assumed that ZooKeeper's traffic should get the highest priority over the traffic of the distributed applications using ZooKeeper, and it should always be satisfied as possible as it could be, with the assumption that we have sufficient available network bandwidth. Which means we should always give

it bandwidth as much as it needs. In our implementation, ZooKeeper keeps requesting for reservation a bandwidth of 10Mbps.

One last component of a reservation is how long it takes. As discussed above, since ZooKeeper is generally used to serve other distributed applications, latency in its traffic will have a particularly negative impact. And on the other hand, the primary part of requests in ZooKeeper is update request which completely relies on clients' behavior. To ensure that no matter when a client request comes, the reservations are always available. We always maintain a reservations on switches between all ZooKeeper servers. Since PANE needs a time span for setting up a reservation, in our implementation, ZooKeeper servers repeatedly send requests for reservations such that when the previous one expires, a new reservation will immediately be reestablished. One issue here is that the next reservation should only be requested after the previous one has expired. This is because PANE semantic would treat these two requests as two different reservations and they are both established on the switch, they would overlap and waste bandwidth.

To minimize the modification we need to make in original ZooKeeper implementation, when the server starts, a separate thread is created to interact with PANE. The thread reads all the arguments from ZooKeeper configuration file and repeatedly requests for bandwidth reservations. The thread is separated from ZooKeeper's protocol and only replies on the host, meaning that as long as the server is up, the reservations will always be requested, regardless of the role of the server and in what phase it is. And we also modified ZooKeeper client such that it will make reservation to the server immediately upon connecting to it.

3.3 Evaluation

We installed the modified ZooKeeper on a testbed with 5 machines, with a client running on a 6th machine. All the machine are connected by an OpenVSwitch. We wrote a benchmark tool to measure the performance of ZooKeeper with and without support of PANE. To accurately measure the time spent on each operation, we chose to use ZooKeeper's synchronous operations, for which a request will only return to client after it has actually been completed by server. In the experiment, our client repeatedly sent CREATE request to server asking for creating new znodes during certain time period. The time of sending the request and receiving the return value are recorded to calculate the latency of this request. Figure 1 shows the latency we measured, the line marked as ”Pre” is when ZooKeeper is the only traffic in the network. The ”Post” line is when we used iperf as competing traffic in the same network, but without PANE. And the line ”PANE” is after we setup bi-direction PANE reservations to protect the ZooKeeper traffic. As can be seen, the iperf traffic has a dramatic negative impact on ZooKeeper, but after PANE is being applied, the performance of ZooKeeper goes back to a high point. From our experiments, we also observed similar scenario for other write requests such as writing and deleting znode. And for read request, as explained above, since it is handled locally, it does not involve large amount of network traffic, the improvement we gain from making reservations for read request is not very obvious.

4 PANE and Hadoop

4.1 Overview of Hadoop

Hadoop is an open-source framework designed to simplify storing and processing large data on a cluster of computers. By using Hadoop, clients may design and implement highly reliable and usable distributed applications that can scale up to thousands of computers, and also, clients do not need to understand the lower level architecture of the distributed system. Hadoop system has two major components: Hadoop distributed file system(HDFS) aims at providing highly reliable distributed file system that supports fast and large sequential read and write. The other one is Hadoop MapReduce, a simplified and unified distributed

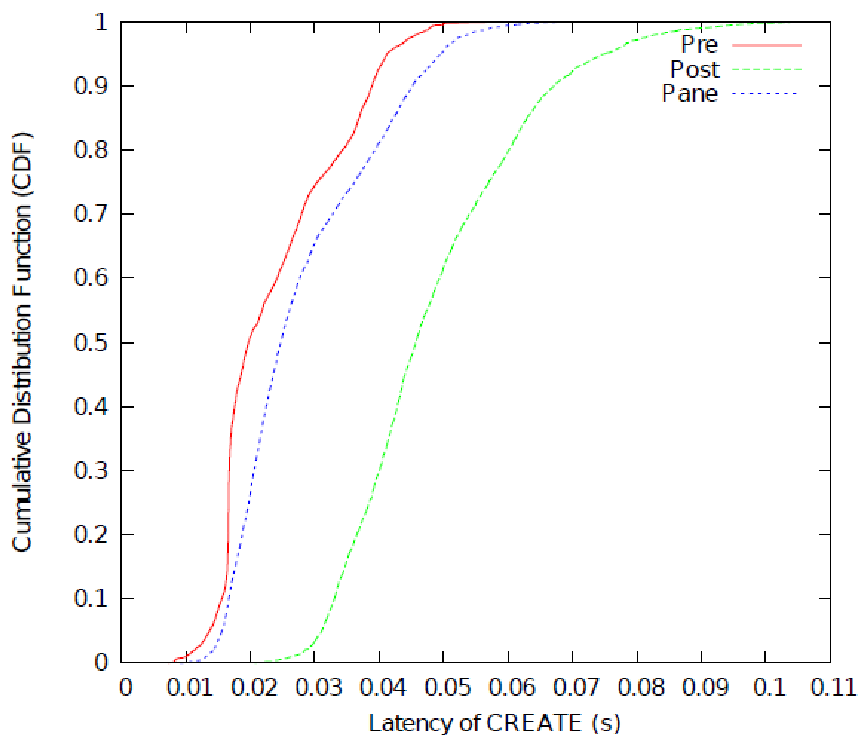

Figure 1: Latency of ZooKeeper CREATE requests with and without PANE support

programming paradigm. In order to explore how to improve performance of distributed applications, we were more focusing on MapReduce because it is the task execution part of Hadoop.

MapReduce program consists of two functions: map and reduce. In Hadoop, clients submit their job by submitting the input data to HDFS and the map and reduce functions to the server. The server will first run map function on each server in parallel and read the data from HDFS, and the intermediate output of each map will be shuffled and aggregated to different reduces. Eventually, the output of all reduces will be merged and the final data will also be stored on HDFS.

4.1.1 Input/Output

Prior to starting the job, clients need to put the input data on HDFS, since Hadoop MapReduce will always read from HDFS. Then the input data has to be assigned to all map tasks. In general, the entire input data will be evenly divided into small datasets called splits. One feature of HDFS is that, to guarantee reliability, data stored on HDFS will be replicated on several machines. Each split is assigned to a single map. Upon receiving the split, a map task will try to further divide the split into key/value pairs according to a format specified by clients. In MapReduce paradigm, both input and output of map and reduce function need to be in key/value pairs. After all reduces have finished their task, the job is finished, and the result will again be written back to HDFS. One task of this phase is to create all copies of the result. The server will find multiple machines in different racks, each of them will be holding one replica.

4.1.2 Shuffle

The heart of MapReduce paradigm is the shuffle phase. Maps' intermediate output will be fetched to different reduces: all values of one key will be aggregated to one single reduce task. Generally, this is an

all-to-all communication.

4.2 Instrumentation

4.2.1 Network Traffic Pattern

Execution of a hadoop job involves 3 phases of network communication. The first is when maps fetch data as input. As mentioned above, maps only read from HDFS and HDFS holds multiple copies for all data on different machines. To minimize the time for fetching, the Hadoop server will always try to assign a map to a machine that holds the corresponding data split. Consequently, it is possible that this phase does not involve any network traffic because all maps are reading locally. But there is no policies for locating reduces because reducers will be reading from maps on many different machines, so generally a Hadoop job will always run into an all-to-all communication in shuffle phase, and the distance from a map to a reduce can be arbitrarily long across the cluster. In the end, when reduce writes the result to HDFS, copies will be made in this phase, as mentioned earlier. Specified number of machines will be found and a one-to-one transmission pipeline will be created across all of them such that copies can all be written at the same time.

4.2.2 Implementation

Our instrumentation attempts to minimize the reservation for Hadoop on other traffic in the same network. More specifically, if we setup a reservation on a switch too aggressively such that they are installed much earlier than the time the traffic actually happens, the reservations would stay idle for some time, and prevent other applications from using the bandwidth. We would like to minimize the idle time of all reservations. We address this by improving the granularity of our implementation, we look for the place in the code where the flows are exactly being created and our instrumentation makes reservation right before the traffic is about to happen. The details is presented below.

Shuffle Among all three phases mentioned above, the shuffle phase has the heaviest network traffic load because each reduce fetches data from each map. Just like what we did in ZooKeeper, PANE needs to determine all the arguments needed to setup reservations. In our implementation, there is a reservation for each flow between map and reduce, and the reservation is made at the reduce side. The address of mappers involved in this transmission is known to the reduce. And the port for sending data at mapper is a constant configured before the system starts. But since a reducer may fetch data from different maps simultaneously, the fetcher thread of reducer will pick a random available port for receiving data.

As for the bandwidth and time of the reservation, by now we leave it as relying on client's configuration. A client can specify a deadline for all the flows in shuffle phase. As the fetcher thread knows the size of the data to be transmitted, it can compute the bandwidth needed by the flow to meet the deadline by simply dividing the size by the deadline. We have implemented having the fetcher thread compute the bandwidth based on a single deadline defined in configuration file.

Input As explained above, to read input data of the job, then server will first read it from HDFS, divide the data into splits, translate the data into key/value pairs and then assign it to mappers. Hadoop uses "RecordReader" to handle all the processes from reading raw data to passing the data as key/value pairs. And in our implementation, we also use RecordReaders for making reservations. For reservations involved in input, ports at both sides are configured when the system starts. Information of port , along with hosts need to be passed to RecordReaders to make the reservation. There is a RecordReader for each format in Hadoop, Which means, if the format of the job is implemented by client, the client will need to also implement the RecordReader with PANE support embedded in code. Specifying bandwidth and time of

reservations works in similar manner as in shuffle phase. Clients can provide a deadline for all mappers' input in configuration file. And the mapper itself will compute how much bandwidth it will need.

Output When a reducer is finishing its execution, it will request HDFS's central server called NameNode for a list of locations to store all the copies of the data. All the copies are written simultaneously in a pipeline. More specifically, the reducer will get a list of positions from NameNode, and it only writes the data to the first machine, the first machine would write it to its local storage and meanwhile write it to the second machine, and so forth. Therefore, one reservation is needed between each of the machines for passing the replica. It is all done at reducer's side when the list of machines is returned to the reducer. Ports needed by the reservation are pre-configured values and can be easily read from configuration. Bandwidth is computed by dividing the total size of the flow by the deadline set by the client, as well as what we did in shuffle phase and input phase.

One goal of our implementation is to have the reservations ready exactly before the traffic of the flow is being sent into the network. But one issue of it is that for very small flows, since the request for reservation is sent just a little bit ahead of the traffic, it is possible that before the reservation is actually setup on the switch, the traffic of flow has already finished. Then it ends up wasting time and space on creating an idle reservation. To address this, we simply do not create reservations for very small flows. The intention is that if it is very small and can already be finished in very short time, it would not gain much from reserving bandwidth.

By now, for all three types of reservations presented, clients can give a deadline for each of the types and all reservations of the same type will have the same deadline. But ultimately, we would like to enable Hadoop to allow clients to specify a priority for the entire job, and the job scheduler may accordingly compute a deadline for input, shuffle and output phase. And these deadlines are used furthermore to compute a deadline for each flow. The major challenge here is that there is no trivial approach to split the deadline of a phase into the accurate deadline of each flow in this phase. Take shuffle phase as example, each flow in shuffle phase starts at a different point of time, it all depends on when a mapper has finished its job. When the last flow starts, the first flow may have already been finished long before, meaning that when we are creating reservation for the first flow, there is no clue about what deadline would meet the total deadline because we can not predict how long the shuffle phase will last at this point. Similar issue also exists in input/output phases because mappers/reducers do not start their job at the same time. For mappers, it depends on whether there is available slot, and for reducers, it also depends on whether the intermediate data has been finished by mappers. It makes it challenging to predict how long a certain flow will last given only the total deadline. We leave all such issues as part of future work.

4.3 Evaluation

We installed our instrumented version of Hadoop onto our testbed to measure its performance. There are 22 machines in the testbed where 20 are slave nodes. All machines are connected by a Pronto 3290 switch. In our measurement, we focused on how well PANE's reservation might work under priorities. Existing Hadoop resource scheduler is able to allocate CPU and memory resources to different jobs by having client create different queues for jobs. Therefore jobs' priority can be represented as the amount of resources they are allocated. Our evaluation expands it with network bandwidth resource. In this experiment, bandwidth for all reservations are set in proportion to CPU resource of the job. And all reservations last for 8 seconds because of the link speed and the default block size of Hadoop. We submitted 3 jobs with different priorities. They were all sort jobs with 40GB of input. 2 of them were allocated with 25 percent of the resource and the other one got 50 percent. The experimental result shows that the run time of the job with more bandwidth

allocated was decreased by 19 percent in comparison with non-reservation execution. And run time of the other two were decreased by 9 percent, partly because the first one finished earlier and released resources.

5 Related Work

Guohui Wang et. al [6] aimed at using SDN technique to reconfigure the network topology at run-time to improve the performance of Hadoop. Their focus is to reconfiguring the routing of packets by creating spanning trees based on aggregation patterns. We did not completely disable certain flows, but instead we create queues to reserve bandwidth for flows.

The work of Sandhya et.al's [5, 7] indicates that OpenFlow has the potential of improving Hadoop's performance by protecting its traffic against other traffic in the same network. But in their work, flows were manually setup on the switch priori to the start of the job. Different from their work, our work tries to allow the applications to, based on the current status of its execution, reconfigure the network at run-time and by the application itself.

6 Future Work

We have shown that PANE can be applied to improve performance of ZooKeeper and Hadoop. And the idea of using SDN is not limited in just these two distributed systems, many other distributed systems may also be benefited from taking advantage of configuring lower level network. But it may not be practical and effective to do such application-specific instrumentation for all the distributed system as we did. We hope our work could provide insight for developing a general approaches of using SDN to improve performance. And this may involve effort in developing SDN controllers that can automatically detect traffic pattern of distributed applications at run-time.

7 Conclusions

One trend in recent years is that networks are designed and configured to meet the need of applications running on the network. PANE provides interfaces for applications to request abstracted network resource from high level. We aim at measuring how much real distributed systems may gain from having PANE do bandwidth reservations. And we showed clearly by experimental results that PANE successfully installed the client's requests into the network and improved the performance of real distributed systems.

References

[1] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi Hierarchical Policies for Software Defined Networks In *Proc. Workshop on Hot Topics in Software Defined Networks (Hot-SDN)* August 8, 2012

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed ZooKeeper:Wait-free coordination for Internet-scale systems In *Proc.USENIX ATC 10, Boston* MA, 2010

[3] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini Zab: High-performance broadcast for primary-backup systems In *Proc. of IEEE DSN (2011), pp. 245256*

[4] `http://hadoop.apache.org/`

[5] Sandhya Narayan, Stuart Bailey, Matthew Greenway, Robert Grossman and Anand Daga OpenFlow Enabled Hadoop Over Local and Wide Area Clusters In *IEEE/ACM Super Computing 12 (SC12) SCinet Research Sandbox Project at the International Conference for High Performance Computing, Networking, Storage & Analysis* Nov 2012

[6] Guohui Wang, T. S. Eugene Ng and Anees Shaikh Programming Your Network at Run-time for Big Data Applications In *Proc. Workshop on Hot Topics in Software Defined Networks (Hot-SDN)* August, 2012

[7] Sandhya Narayan, Stu Bailey and Anand Daga Hadoop Acceleration in an OpenFlow-based cluster In *IEEE/ACM SC12 International Workshop on Network-Aware Data Management* Nov, 2012