Research Project Report

PIRE ExoGENI – ENVRI

Preparation for Big Data Science

Universiteit van Amsterdam

System and Network Engineering (MSc)

Authors

Stavros Konstantaras
(stavros.konstantaras@os3.nl)

IoannisGrafis
(ioannis.grafis@os3.nl)

Supervisors

Dr. Ana Oprescu
(a.m.oprescu@uva.nl)

Dr. Zhiming Zhao
(z.zhao@uva.nl)

February 10, 2014

Contents

List of figures

1. Introduction

Big Data is a new field in both scientific research and IT industry focusing on collections of data sets which are so huge and complex that create numerous difficulties not only in processing them but also in transferring and storing them [1]. The Big Data science tries to overcome problems or optimize performancebased on the “5V” concept: Volume, Variety, Velocity, Variability and Value [2]. A Big Data infrastructure integrates advanced IT technologies such as Cloud computing, databases, network and HPC, providing scientists with all the required functionality for performing high level research activities [3]. The EU project of ENVRI is an example of developing Big Data infrastructure for environmental scientists with a special focus on issues like architecture, metadata frameworks, data discovery etc. [4].

In Big Data infrastructures like ENVRI, aggregating huge amount of data from different sources, and transferring them between distribution locations are important processes in the many experiments [5]. Efficient data transfer is thus a key service required in the big data infrastructure.

At the same time, Software Defined Networking (SDN) is a new promising approach of networking. SDN decouples the control interface from network devices and allows high level applications to manipulate network behavior [6]. However, most of the existing high level data transfer protocols treat network as a black box, and do not include the control for network level functionality.

There is a scientific gap between Big Data science and Software Defined Networking and -until now- there is no work done combining these two technologies. This gap leads our research on this project.

1.1 Scope of the work

The scope of this project is to discover how Big Data science can benefit from the SDN technology and boost the transfer of huge amounts of data without losing reliability. By investigating related work, we discovered that we can bridge these two technologies and enhance the data transfer between two single points of the network. The project will focus on the most common data transfer protocols and the SDN technologies, and investigate the degree to which these SDN technologies may be used to optimize data transfer services. The project will not aim at solutions such as modifying SDN controller or customizing particular algorithms for data transfer protocols.

1.2 State of the art technologies

The current demand for transferring large volumes of data sets across the Internet led to the development of new technologies which promise to utilize the latest high speed links (>10Gbps) and minimize the transfer time. At the transport layer of the OSI stack, RDMA over Converged Ethernet (RoCE) provides better performance than TCP at speeds of 40Gbps and promises to overcome problems such as CPU limitation.

Above the transport layer, GridFTP from Globus is a well-known state-of-the-art tool which uses parallel TCP connections to enhance data transfer. Globus provides also an RDMA version of this tool which replaces TCP, but according tothe research of Brian Tierney et al. [5] many improvements need to be done for deploying it over 40Gbps networks.

On the other hand, in the field of Software Defined Networking, there is only one protocol implemented from Stanford University, named OpenFlow. By using a single entity called "Controller", OpenFlow enabled switches connect to the Controller and receive instructions for handling different types of network traffic. The rules that the Controller establishes at their flowtables is the way for building network paths between two end points. However, OpenFlow allows administrators to use the protocol for many different purposes like building Firewalls, creating Virtual LANs, separating traffic etc.

1.3 Problem statement

The performance of the data movement protocols depends on not only the implementation of the protocol but also the quality of services (QoS) provided by the network. We are motivated to investigate what are the network problems that can be found on transferring large volumes of data. Moreover, we study what different approaches can be used in order to solve the QoS problem and what are the parameters of every solution. Finally we are going to use SDN technology to control the network and enhance the data transfer rate according to an algorithm which provides the required intelligence.

The main research question is the following:

- To what degree can the performance of the data movement protocols be optimized by using Software Defined Networking technology?

The main research question includes the following sub-questions:

- What network level problems exist which limit the performance of the data movement protocols?
- How can SDN eliminate these problems?

1.4 Outline

In this report, we will first investigate the network problems that exist and limit the performance of the data movement applications. After that, we discover the existing technologies that can help us and we build a decision tree based on the QoS problem and the possible answers. According to that tree, we discover the possible solutions and we select the appropriate one according to our needs.

After completing the theory study, we built a prototype as a proof of concepts. Our component has the name HIDE (Hybrid Intelligent Data Enhancer) and we tested it using ExoGENI [7], a testing environment which allow us to create virtual topologies. The results of our research are presented and discussed in the fifth chapter.

Finally, we discuss our work along with the advantages and the disadvantages of our implementation and we conclude the report with our personal opinion and suggested future work.

2. Problem analysis on transferring big data

At the field of Big Data science we discovered three major applications which promise to enhance the transmission of huge amount of data. GridFTP from Globus [8], bbFTP from NASA Research and Engineering Network [9] and FDT from CERN [10] are based on the same idea to achieve their purpose: tuning the TCP protocol and initialize many parallel TCP connections in order to fill the capacity of the link and keep a stable high rate.

2.1 Data movement tools

At the user space, we compare the successful application GridFTP among its competitors, bbFTP and FDT. Despite the fact that all three applications are open source and use parallel TCP streams to transfer data, Globus enriched GridFTP with extra features such as data management, resource management, fault detection and security. The table below provides a comparison between GridFTP and the other two data movement applications together with their network limitations.

As a result of the proposed comparison, we can extract the information that all three applications suffer from the TCP limitations and they are not able to use efficiently high bandwidth links (over 20Gbps). Moreover, even the RDMA version of GridFTP is not able to exceed the speed of 13Gbps and this means that there is a lot of work to be done in order to improve the application’s performance and successfully utilize all the available capacity.

2.2 Network protocols for Big Data

At the network level, the old designed TCP protocol starts to reach its limit when it is used over high bandwidth links. Until the speed of 10Gbps TCP is able to perform well and utilize all the available capacity of the link, but fails to achieve the same behavior when it is deployed over 40Gbps or faster networks. The main issue is the CPU limitation which creates a bottleneck on the data transfer and according to our literature study, the maximum performance that an application can get from TCP over 40Gbps link is around 13Gbps.

Application	Positives	Negatives	Network limits
GridFTP	-Open source -High scalability -High reliability -Versions for TCP and RDMA -SSH option -Widely adopted -Option to resume transfers that are stopped because of failures	-Difficult to deploy -Network speed limit: i) 13 Gbps (TCP version) [5] ii) 13 Gbps (RDMA version) [5]	-Decrease window size for every loss packet and resend the packet -Application is not aware for the topology and the path that data flows -Most of times the speed of transferring data is limited due to network traffic
bbFTP	-Open source -High scalability -High reliability -Multi-stream TCP -Secure channel over SSH -On the fly compression -Easy to deploy -Resume file transfer session	-Transfer only files, not directories -Little industry adoption -Little documentation	
FDT	-Open source -Runs on all major platforms (Java application) -Multi-stream TCP -SSH option -Easy to deploy -Resume file transfer session	-Little industry adoption -Little documentation -Network speed limit (4.5 Gbps) [10]	

Table 1: Comparison between GridFTP, bbFTP and FDT

The CPU limitation addressed by the Remote Direct Access Memory (RDMA) [12] protocol which tries to minimize the CPU utilization by writing the data directly to the machine's Random Access Memory (RAM). This means that the network adapter bypasses the Operating System (OS) and it is totally responsible for handling the required memory operations.

By replacing TCP with RDMA it is possible to achieve better data transmission over high speed links but this requires also big data buffers and a loss-free network. Performance can be improved also by using UDT [11], an UDP-based protocol which promises to move data faster than TCP. The above table provides a comparison between RDMA and UDT but both protocols promise fast and reliable transfer of big volumes of data.

Another useful protocol which tries to overcome the congestion problem is the MultiPath TCP (MPTCP) [13]. The purpose of this new implementation is to achieve better link utilization, better load balancing than the network can do and also make usage of multiple available network paths. MPTCP works as a component which initiates multiple TCP streams and implements a congestion control mechanism across the subflows. Each subflow is treated as a separate TCP connection from the network and MPTCP can recognize which one of them faces congestion problems in order to apply load balancing. However, MPTCP cannot deal with the situation that network broken links which require creations of new paths.

Protocol	Positives	Negatives
RDMA	-Reduce latency -Reduce CPU overhead -Reduce memory overhead -OS-bypass protocols -High throughput -Bypass limitations of network speed due to CPU limitations	-Little industry adoption due to special hardware required
UDT	-UDP based -Reliability -Good performance on high-delay networks -Good performance when competes TCP based protocols	-Increased overhead -CPU limitation causes network speed limitation -Difficult practical deployment inhigh-speed networking applications -Difficult firewall reconfiguration

Table 2: Comparison between RDMA and UDT

2.3 Main problems

Most of the performance results reported by the related work presented in Section 2.2 were achieved in an experimental environment where there was no external traffic to interfere with the experimental TCP connections. In a real world scenario where the scientists transfer data across internet over many heterogeneous networks, these numbers would be lower from the ideal. Also since the Internet routers decide the destination of each packet, it is possible that the selected path is not always the best option and congestion problems can appear.

In addition, high speed networks require more resources from the hosts in order to succeed full utilization of the link. This problem can be minimized in the future by adopting the RDMA protocol and extending the applications to use it efficiently. Meanwhile, in classic Ethernet networks where the successful TCP protocol is being used, the main problem that these applications can face is the linkcongestion.

To conclude, overloaded links minimize the transfer rates and increase the transfer times and this leads to lower QoS than expected. In addition,huge transfer times are observed when a link is selected with less bandwidth than the required and this also leads to low QoS. Data movement applications do not have any control over the network but all the recommended solutions introduce the main idea for solving the congestion problem: redirecting the connection to a better performed path.

3. Enhancing Big Data transfer using SDN

From the analysis in chapter two, we know that the network problems of traffic congestion and bandwidth limitation can seriously affect the performance of data movement applications. The SDN technology allows us to control the network based on run time application information. In this chapter we will investigate mechanisms for diagnosing application performance, obtaining the network status and controlling behavior of the network. Then we use a decision tree to model the control intelligence for solving those application performance problems using the SDN technology. Finally, we will also discuss the required programming profiles for building a SDN based solution for improving data movement performance.

3.1 Performance diagnosis

The data movement application is not aware of the network topology or the path where the data flows inside the network. As a result of it, the application can face a network problem but it is no able to proceed to network changes in order to solve that. However, it is possible for us to detect that our application does not perform well due to a network problem, according to Figure 1.

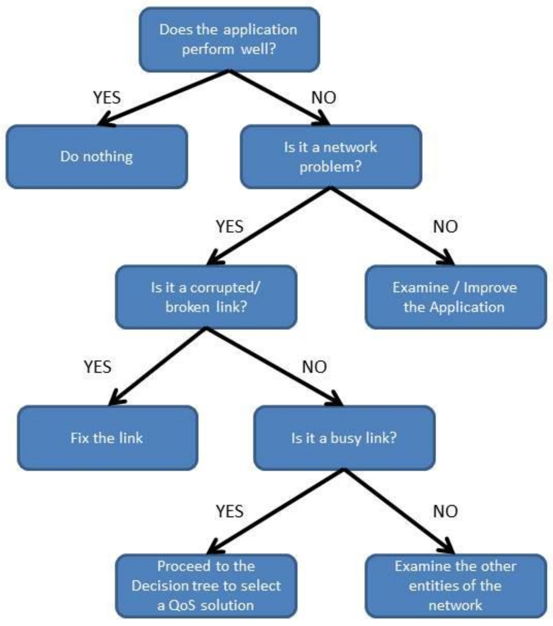

Does the application perform well?
YES
NO
Do nothing
Is it a network problem?
YES
NO
Is it a corrupted/ broken link?
Examine / Improve the Application
YES
NO
Fix the link
Is it a busy link?
YES
NO
Proceed to the Decision tree to select a QoS solution
Examine the other entities of the network

Figure 1: Problem detection tree

By following the problem detection tree, if the application does not perform as expected while the network provides all the available resources, we realize that the application needs to be examined or improved. On the other hand, if the application is facing performance problems without detecting congestion on the network paths, then we have to examine the intermediate entities of the network (for example routers), but this is out of scope for this report.

Finally, the overloaded links which have a serious impact to the performance of the data movement application is a topic which has multilevel parameters and requires careful examination. For this purpose we created a decision tree which is presented in the following chapters, in order to discover all the available solution and their requirements.

3.2 Available mechanisms

Before we proceed to the decision tree, we examine what network level mechanisms exist which are required for traffic monitoring and network controllability. As a traffic monitoring technique we found three basic options that could be adopted.

- Deep Packet Inspection (DPI) [18]: the technique of DPI examines the TCP packets in order to understand the type of the data traffic, the state of the communication, and the addresses of the source/destination nodes.
- Inspect client/server interfaces: Similar to the DPI technique, this solution uses classic Linux tools (for example tcpdump) at the client/server nodes, in order to get information about the transfer rate, the client/server IPs etc.
- Inspect flow counters: This is the simplest method as it is based on the counters of the flowtables. Each time a packet matches a flow and an action is applied, the flow counter increases its value. OpenFlow controllers can request these counters from the switches and by inspecting them in a stable time interval we can extract useful information such as the load of the switches and available bandwidth of the links.

The purpose of traffic monitoring is to extract useful information about the status of the network and the type of the packets but does not provide any solution to the problem. This means that in order to increase the QoS it is necessary to have some network controllability. In a SDN topology the entity which has the overall control of the network is the OpenFlow controller which implements the algorithms and the policies of the development team by inserting flows into the switches. As a result of the SDN concept, we recognize two options for controlling the network:

- Commands to the switches: A component which inject rules to the switches in order to manipulate the flows and change the path of a current connection. This solution requires less strict security configuration for all the OpenFlow switches but also requires for the implementation to be located inside the SDN topology.
- Commands to the controller: Whoever controls the Controller controls the SDN topology. If the Controller provides an API it is easy to send abstract commands to it which then will be transformed into rules at the OpenFlow tables.

After examining the SDN concept we reach to the conclusion that the key point in an OpenFlow topology is the flow entry inside the table of the switch. By manipulating the correct flows we change the behavior of the network and this is something useful for improving the QoS. The flow management is another important parameter that needs to be investigated for building our component. This can happen in different levels such as:

- Port level: Based on the port where the packet comes from, we apply the desired action. These generic flows are useful at the core switches of an SDN topology where it is possible to address the network's entire traffic by using a minimum number of flows.
- Socket level: Based on the destination IP and TCP port we decide where to forward the packet. This level could help at the edge switches where traffic shaping and load balancing is needed in order to avoid overloading of the paths.

All these options are not available for. According to our decision of how to solve the QoS problem we inherit automatically different choices in every parameter and this means that a decision tree will help us to understand better the conditions.

3.3 Intelligent treatment

Since we detected that our application is facing a performance problem as a result of overloaded links, we will try to select the appropriate mechanisms to solve the problem through a decision tree which is based on two different tracks.

The first one is an Application based approach which is going to interact with the data movement application in order to enhance the performance. The second one is a network based approach at two different levels: the high dependency level where the implementation is embedded inside the controller and the more abstract level where the implementation is interacting with the controller through an API. The following figure demonstrates our decision tree with the mentioned approaches as solution tracks to solve the problem.

The two-level concept which is defined at the network based approach is excluded automatically at the application approach. The reason is that the data movement applications already extend well known protocols (such as FTP) and use various techniques (such as Linux *zero_copy*) in order to increase the data transfer rates. This means that any solution that can be provided according to the application approach should be outside of the data movement tool in order to get a meaning.

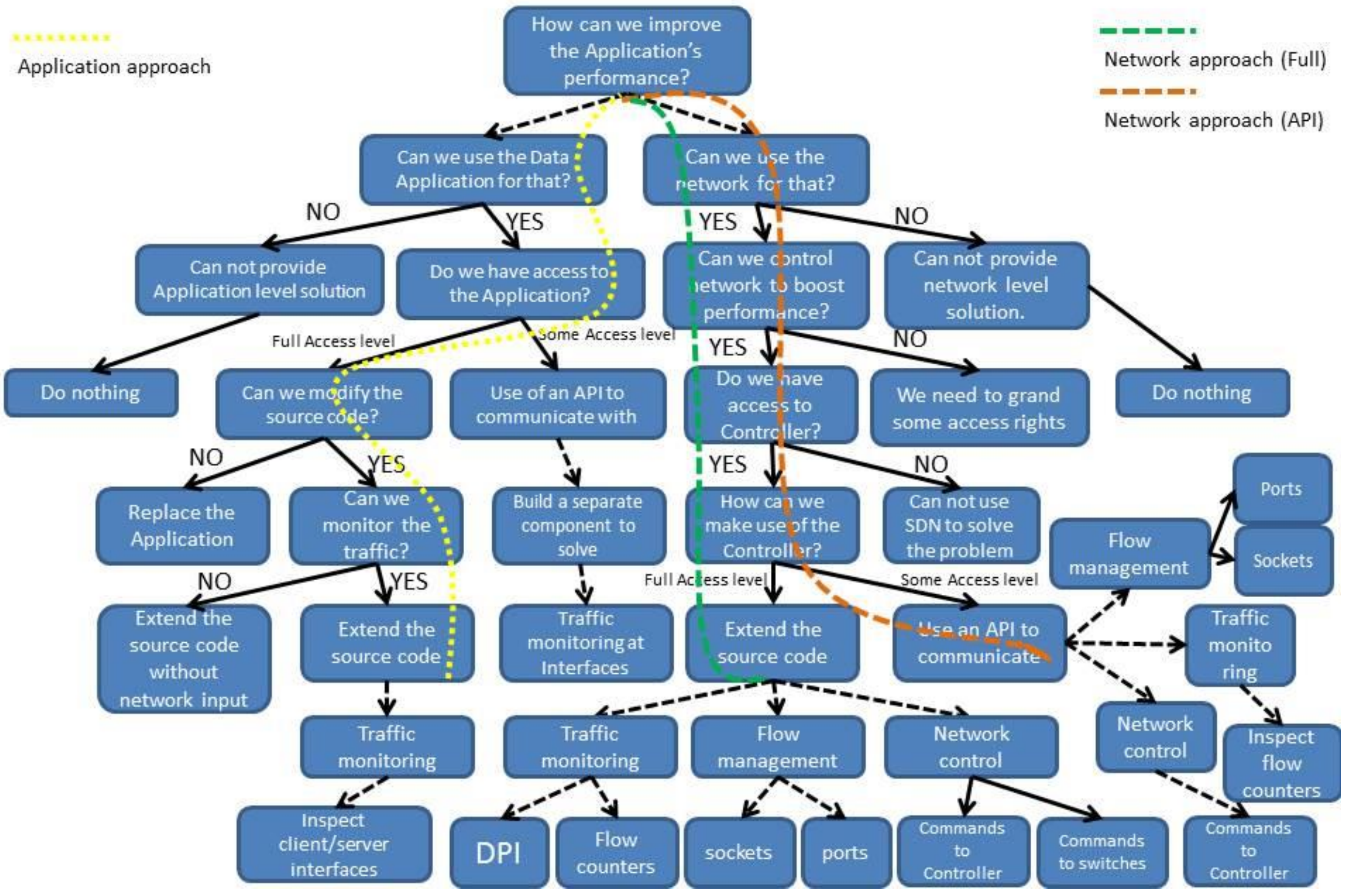

Application approach
Network approach (Full)
Network approach (API)
How can we improve the Application's performance?
Can we use the Data Application for that?
Can we use the network for that?
NO
YES
Can not provide Application level solution
Do we have access to the Application?
Can we control network to boost performance?
Can not provide network level solution.
Do nothing
Full Access level
Some Access level
Can we modify the source code?
Use of an API to communicate with
Do we have access to Controller?
We need to grand some access rights
Do nothing
Replace the Application
Can we monitor the traffic?
Build a separate component to solve
How can we make use of the Controller?
Can not use SDN to solve the problem
Flow management
Ports
Sockets
Extend the source code without network input
Extend the source code
Traffic monitoring at Interfaces
Extend the source code
Use an API to communicate
Traffic monitoring
Traffic monitoring
Traffic monitoring
Flow management
Network control
Network control
Inspect flow counters
Inspect client/server interfaces
DPI
Flow counters
sockets
ports
Commands to Controller
Commands to switches
Commands to Controller

Figure 2: Our decision tree with three discrete solution tracks

A “hybrid” approach would be an ideal solution for increasing the QoS because it combines both tracks at such a degree that provides also an adequate level of abstraction and portability. This is the preferable by us solution and leads us to the development of an autonomous component which has the ability to interact with the OpenFlow Controller and the data movement application at the same time. The next picture demonstrates this type of approach at the decision tree we described above.

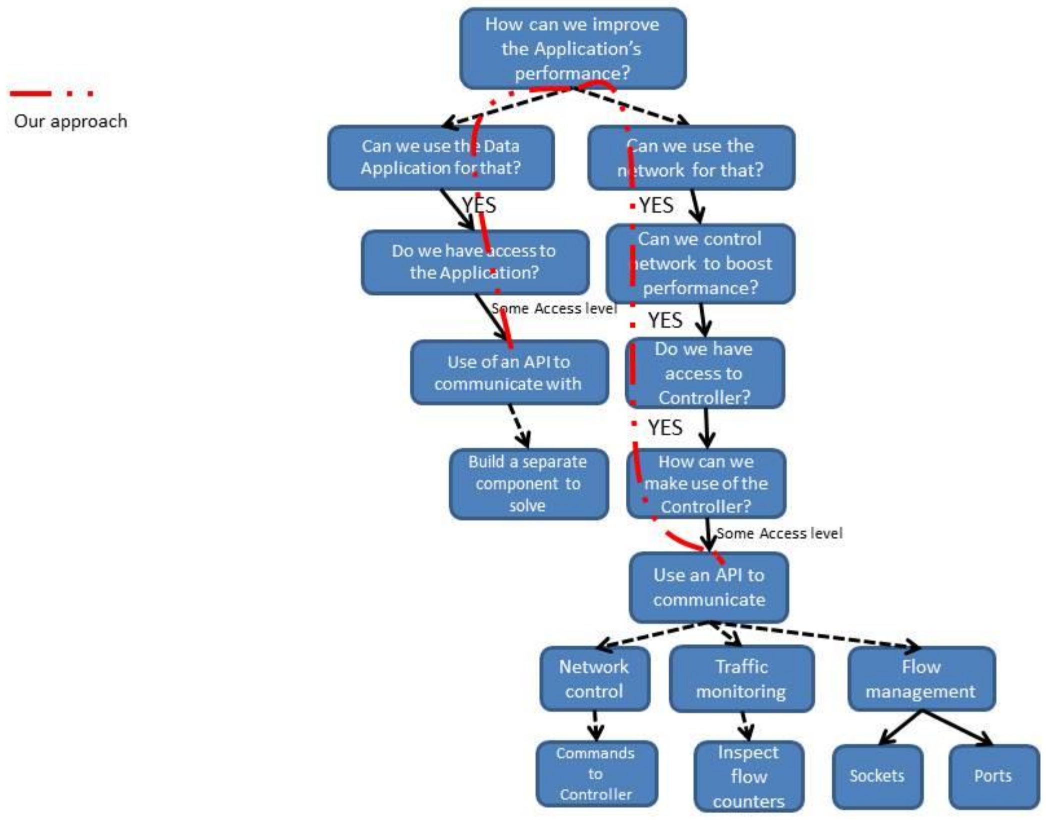

Our approach
How can we improve the Application's performance?
Can we use the Data Application for that?
Can we use the network for that?
YES
YES
Do we have access to the Application?
Can we control network to boost performance?
Some Access level
YES
Use of an API to communicate with
Do we have access to Controller?
YES
Build a separate component to solve
How can we make use of the Controller?
Some Access level
Use an API to communicate
Network control
Traffic monitoring
Flow management
Commands to Controller
Inspect flow counters
Sockets
Ports

Figure 3: Part of the decision tree with our approach

3.4 Development constraints

If we place the mentionedapproaches in a decision space which is defined by two important parameters, "Application dependency" and "Controller dependency", we can realize better what solution fits to our needs. High dependency level means that the preferable solution requires wide changes in order to be adjusted in another network topology.

The first parameter, Application dependency, measures at which level our solution is going to be bounded to the data movement application. A solution which is developed for one specific application can probably perform well but does not provide portability or an adequate abstraction level for all the data movement applications.

At the same time, "Controller dependency" describes how dependent to an OpenFlow controller the solution can be. There is a high probability to solve the QoS problem by embedding the implementation inside the controller (for example extending the controller's source code) but this approach automatically leads to different versions of the same solution, each one dedicated to different controller.

The figure below tries to visualize the mentioned approaches in our solution space. The white area is the most obtainable for our short time research while the grey areas are more abstract but require uncountable hours of research and development. In addition, it is also quite difficult to develop a solution into the yellow area as it requires a common way of communication between all the data movement applications.

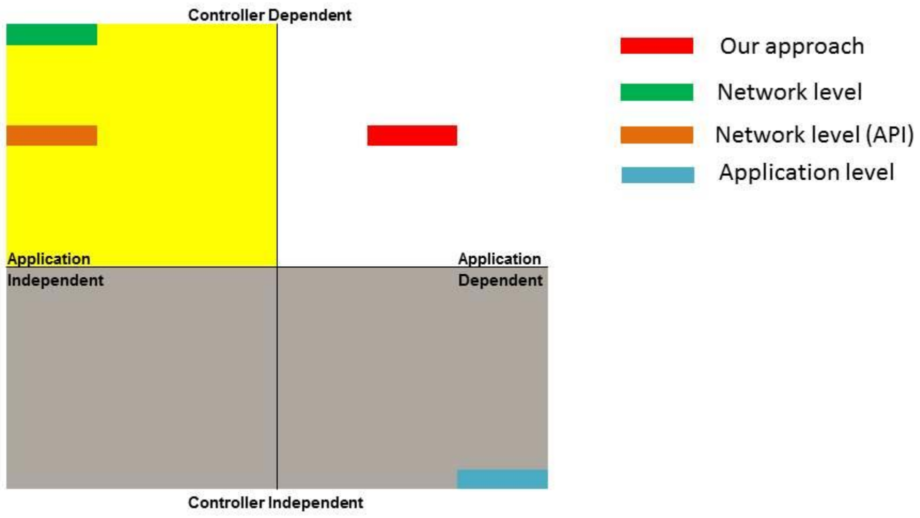


Figure 4: Visualization of our decision space

Moreover, the table below defines four different programming profiles for developing a solution and also clarifies which requirements are needed to be fulfilled before selecting the appropriate one.

Based on the decision tree, Table 3 clarifies under which conditions a solution should be selected. We focus on the Hybrid programming solution which is our selected approach according to the decision space. The selected profile combines the application level profile at such a degree where we can monitor the performance of the TCP connection. At the same time, we combine also the network profile at such a degree where we use an API to send commands to the controller and request network knowledge. As a result of having an adequate level of controller access, we can use the benefits of having network controllability in a SDN based solution.

To finalize our theoretic approach, in order to solve the congestion problem we created a decision space in which we examine what is the best solution that can be achieved in the remaining time of this project. Two high level prospective solutions defined, an application level and a network based one. But according to our research both of them require implementations on a deep level which oppose to our requirements for abstraction and portability. Following our decision tree a Hybrid programming solution appeared to fulfill our needs based on two principles: connection monitoring and network controllability. As a result, the component which is going to be created as proof of concepts should be located at the user space of the OS and also make use of the SDN technology.

Requirements	**Application level Programmer**	**Network Programmer (API)**	**Network Programmer (full)**	**Hybrid Programming**
Develop at Application level	***YES***	*NO*	*NO*	***YES***
Develop at Network level	*NO*	***YES***	***YES***	*NO*
Make use of SDN Technology	*NO*	***YES***	***YES***	***YES***
Access to the Data Movement Appl.	***YES***	*NO*	*NO*	***SOME***
Access to the OpenFlowController	*NO*	***SOME***	***YES***	***SOME***
Network topology knowledge	*NO*	***YES***	***YES***	***YES***
Network status knowledge	***SOME***	***YES***	***YES***	***YES***
Traffic monitor using DPI	*NO*	*NO*	***YES***	*NO*
Traffic monitor on flow level	*NO*	***YES***	***YES***	***YES***
Traffic monitor at Interfaces	***YES***	*NO*	*NO*	*NO*
Flow management	*NO*	***YES***	***YES***	***YES***
Network controllability	*NO*	***SOME***	***YES***	***YES***

Table 3: Programming profiles for building the solution.

4. Hybrid Intelligent Data Enhancer (HIDE)

In this chapter, we use the technologies and the decision tree discussed in the previous chapter to demonstrate the feasibility of improving the performance of the data movement application using SDN. We prototyped a system called HIDE (Hybrid Intelligent Data Enhancer) and it is described below together with its algorithm and the behavior of it.

4.1 The basic idea

The goal of our prototype is to provide the required intelligence to an OpenFlow controller for improving performance on Data Movement applications in a busy SDN topology. This intelligence is based on the knowledge of the available paths and their status. By gathering statistics from the network, HIDE is able to calculate the available bandwidth in every path and compare the numbers in order to discover which of them has the fewer load. Based on that result, HIDE will decide if it is useful to change the path of the connection being examined. The following figure visualizes our algorithm which is going to be implemented and examined in a testing environment.

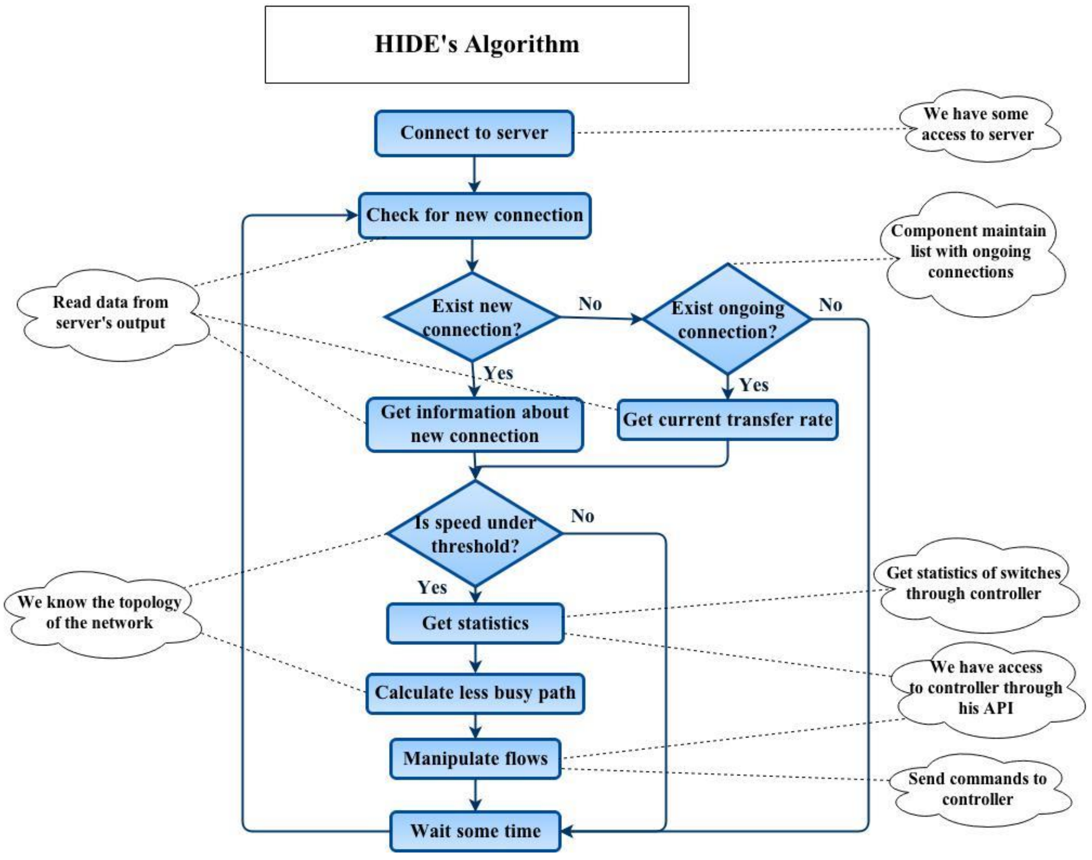

HIDE's Algorithm
Connect to server
We have some access to server
Check for new connection
Component maintain list with ongoing connections
Read data from server's output
Exist new connection?
No
Exist ongoing connection?
No
Yes
Yes
Get information about new connection
Get current transfer rate
Is speed under threshold?
No
Yes
We know the topology of the network
Get statistics of switches through controller
Get statistics
We have access to controller through his API
Calculate less busy path
Manipulate flows
Send commands to controller
Wait some time

Figure 5: The algorithm of HIDE

The decisions that HIDE has to take are extremely critical and highly depended from the thresholds provided. During the development of the prototype two important questions appeared which their answers guide the complete behavior of HIDE.

- Under which transfer rate we mark a connection as problematic?
- What is the minimum extra benefit that an alternative path should provide in order to change the path of a connection?

By inspecting the output of the data movement application we can get the current transfer rates of the connection between the server and the client and the first question can be answered by using the knowledge of the network topology. For example, a transfer rate of 95Mbps is extremely low for a network which was built by Gigabit links, but almost perfect for another one which uses 100Mbps links. Moreover, if the algorithm is triggered for a 2% divergenceof the maximum transfer rate (which is normal on a busy network) this will cause extra overhead to the machine which hosts HIDE, since it needs to gather statistics and proceeds to calculations for discovering a better path. On the other hand, if the algorithm needs a 60% divergence of the available capacity and the connection is established at the half of it, then the client node will never gain the benefits of a possible path change and the QoS will never increase.

The second question is also quite difficult to answer and it also requires the knowledge of the network. For example, if the current rate of an established connection is at 200Mbps in a link which has maximum bandwidth of 1Gbps, then probably this is a busy link and a possible redirection to an alternative path worth. When HIDE discovers an alternative path which is totally empty but offers maximum bandwidth of 100Mbps, then a redirection to this path will double the transfer time and worsen the QoS problem.

Beside the fact that the mentioned two questions are extremely critical and difficult to answer, we decided to build and test our component in order to prove that improving the performance of a data movement application by using SDN technology is still possible.

4.2 Description of the functions

According to the Hybrid programming profile, HIDE should be placed above the Transport layer of the OSI model and use SDN technology to control the network. In addition, it will have the required intelligence to decide if there is a QoS problem and what changes have to be made in order to improve the data transfer rate. The key point in this procedure is the time that the server provides the first performance information of a new connection. This parameter is not changeable but the component is able to read this transfer rate and if the imported number is below a threshold we define (threshold **S**), then the algorithm is triggered.

When a problematic connection is detected, HIDE requires the flow counters from the controller for the core OpenFlow switches two times in a row with defined by us interval between them (interval **I**). After that, it is able to calculate the available bandwidth in both paths and gain knowledge about the status of the network.

Traffic redirection to the alternative path will happen if the performance gain exceeds a second threshold (threshold **G**). In other words, if the alternative path offers X% more bandwidth than the current bandwidth that connections is consuming, then HIDE sends commands to the controller in order to manipulate the flows in the flowtables. This threshold was introduced in order to make HIDE behave more stable but also to reduce the overhead of the network, otherwise HIDE will change paths to the connections even for percentages smaller that 1%.

The following figure provides a zoom-in at the first seconds of HIDE's life in order to clarify the time required for our prototype to react in a possible path change. The symbols **X, Y** and **Z** represent time values in seconds. **X** is the time that is required for the server to provide its first output regarding the performance on a new connection where **Z** is the time that is required from HIDE to verify that the performance has been improved. **Y1** is the time that is required from HIDE to get the statistics from the network while **Y2** is the time needed to calculate which is the best alternative path and sends commands to the controller for redirecting the traffic.

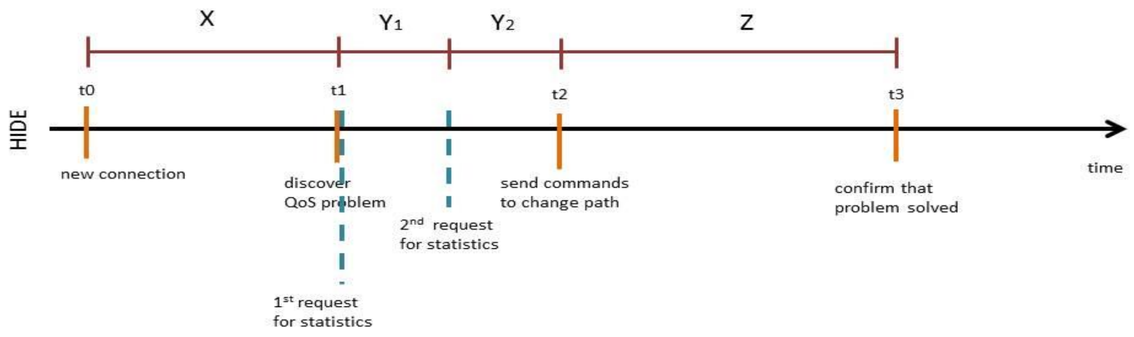

X
Y1
Y2
Z
t0
t1
t2
t3
HIDE
new connection
discover QoS problem
2nd request for statistics
send commands to change path
confirm that problem solved
time
1st request for statistics

Figure 6: Time diagram of HIDE

If we define as **Δ** time the required number of seconds for considering a connection as corrected from the moment it was established, then

- **Δt= Δ(t0t1) + Δ(t1t2) + Δ(t2t3) =X + Y1 + Y2 + Z**

As a result of our design, we define as best performance the minimum time that it is required for our prototype to change the path of a problematic connection. If we assume that HIDE and the data movement server are synchronized correctly, then the best performance that our prototype can provide in an ideal environment is **X + Y1 + Y2** seconds. This result occurs as a summary of the timethatthe server needsto provide us the first performance output plus the time that HIDE requires to collect the statistics of the network and calculate the best path.

In case that the server provides its output exactly after the last check of HIDE, then the worst performance behavior that we can get from the prototype is **X + Y1 + Y2+C** seconds where **C** is the interval for checking the server for a new connection. These numbers are seconds and the summary of them maybe exceeds the time that is needed to transfer a small sized file, but in Big Data science where the researchers are transferring huge amount of data which require even hours to be moved, this delay will not have a serious impact.

Finally, if we consider HIDE as a function ***f*** where its performance depends on a collection of various hidden and obvious parameters, then:

- Y = ***f***(*statistics_interval, check_output_interval, server_output_interval, change_flow_delay*)

From the mentioned parameters, two of them are uncontrollable by the component: the *server_output_interval* and the *change_flow_delay*. The first one is under the control of the data movement application and –at least for the application we used- is not changeable by any command line parameter or configuration file. The second one is the time required from the moment that the component sends the commands to the controller until the new flows have been inserted into the flowtables.

4.3 System prototype

The component was developed as a script written in Python language [see Appendix 1]. Immediately after its execution it connects to the data movement server by using SSH, in order to read the output of the data movement application. In order to achieve this, we need some access to the Data Movement application and also some access to the OpenFlow Controller. To have controllability over the network, Controller has to provide us an API in order to send commands which will be transformed into flows inside the switches. To send the correct commands and manipulate the flows which are going to change the network behavior, we need to insert the topology of the network inside HIDE. At the same time, the component needs to apply a monitor technique in order to have knowledge of the network status.

Since our strategy is to make use of the SDN technology in every aspect, the chosen monitoring technique to extract information about the network status is to inspect the flow counters at the OpenFlow switches. Moreover, flow management has to be carefully selected otherwise the manipulation of the wrong flows will affect all the traffic of the network and unwanted results will appear to our experiments.

By examining the Data Movement applications in both Networks and Transport layer of the OSI stack, we discovered that these tools create numerous parallel TCP connections under the same IP address. The option to create or manipulate flows at port level is incorrect because it will cause traffic redirection from other nodes also and it is quite possible to overload a link with unwanted traffic. On the other hand, if the manipulated flows are based on both IP addresses and TCP ports then for every TCP connection between one server and one client we need dedicated flows in every switch between them. This will cause not only extra overhead on the network but it will increase also the size of the flowtables at the switch close to the data server (if this node is going to serve multiple clients).

The solution is to manipulate flows at the Network level and redirect the traffic based on the IP addresses of the nodes. By changing only few flows all the TCP connections between the server and the client will follow the new path and benefit of the increased bandwidth.

To summarize, according to the theory we proposed above, if the component is able to change the correct flows at the OpenFlow switches and redirect the connection to a less busy path, the QoS will increase and the transfer time will be minimized. As a result of this redirection, the TCP connections will gain benefit from the increased bandwidth and this will have an immediate impact to the transfer rate.

5. Experimental results

For demonstrating the functionality of our system and investigating its runtime performance characteristics, we created a testing environment on the ExoGENI infrastructure. The following chapter is going to present the testing environment and its configuration, the scenarios under which the prototype has been tested but also the results that have been collected from these tests. Figure 7 shows the network topology and the links between the switches.

5.1 Configuration of the testing environment

For deploying the topology we used the ExoGENI [7] rack of the SNE research group. In fact all the entities of the testing environment are Virtual Machines (VM) with the OS of our choice. The links between the VMs are virtual links but the desired bandwidth is guaranteed from the software managing the rack. The OpenFlow switches are VMs with multiple interfaces which they run OpenVSwitch in order to behave like real OpenFlow switches. They are connected between them by using 100Mbps links but the links between the nodes and the switches are ten times faster (1Gbps) in order to avoid bottlenecks outside of the OpenFlow network.

FloodLight [14] was the OpenFlow Controller of our choice since it provides an API to send commands and due to the fact that it is written in Java, the deployment is a typical procedure. We selected FDT as the data movement application because it provides to the user an output of the connection’s current transfer rate, which is the one of the two inputs that HIDE needs to have.

For loading the links with useless traffic in order to create congestion, we used Iperf [15] between client1 and server2. The FDT server was located at server1 while the client2 was pulling data from it. In the topology we created there are two equal paths connecting server1/client1 with server2/client2:

- Path1 or upper path which connects switch1 (SW1) with switch4 (SW4) through switch2 (SW3).

- Path2 or down path which connects switch1 (SW1) with switch4 (SW4) through switch3 (SW3).

HIDE was located at server2 but we configured it to use the management network and the public IPs of the other VMs in order to build more realistic scenarios. Finally, all the flows inside the switches were static and the Avior GUI [16] was used in order to insert them correctly and easily.

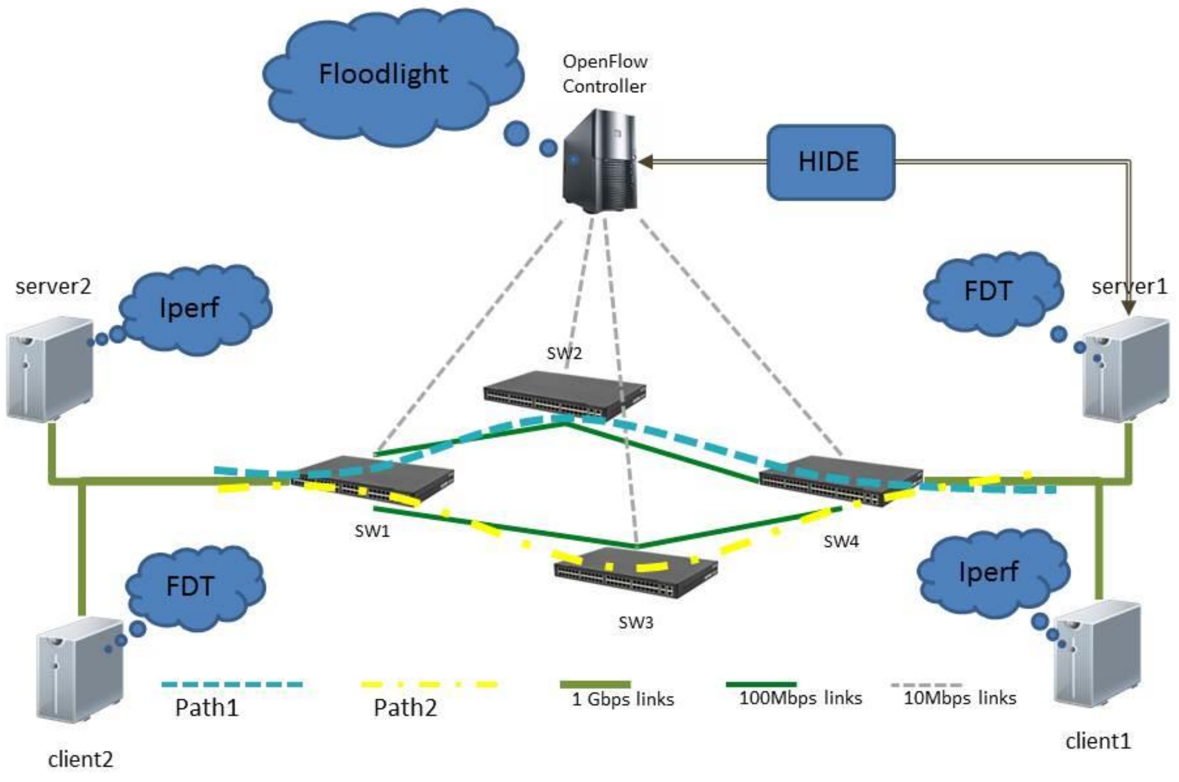


Figure 7: The topology of our testing environment

HIDE was configured to check the log file of the FDT server every one second, in order to discover a new connection between the server and a new client. It is possible for our prototype to improve and supervise the performance of many TCP connections which are running in parallel on the FDT server but due to limited available time we did not explore this performance limit of our component.

When a connection is marked as problematic, HIDE sends a request to the controller requiring the flow counters for the core switches (SW2 and SW3). Threshold **S** which marks a connection as problematic is configured to 20% and the same value has also the threshold **G.** In other words, if the connection which is being examined by HIDE is consuming less than 80% of the maximum bandwidth of the current path, the prototype examines if the alternative path is able to offer 20%

more bandwidth than the currentconsumed. If this happens, then HIDE proceeds to the procedure of redirecting the problematic connection.

The key point in this procedure is that the server provides the first performance information of a new established connection after six seconds. This parameter is not changeable by any means (except manipulation of the source code)but through our experiments we realized that the output is quite accurate. The following time diagram visualizes in parallel the behavior of both the FDT server and HIDE as they are configured with the mentioned values. In Timestamp zero (t0) a new connection is established and it is assumed that HIDE is fully synchronized with the output of the server.

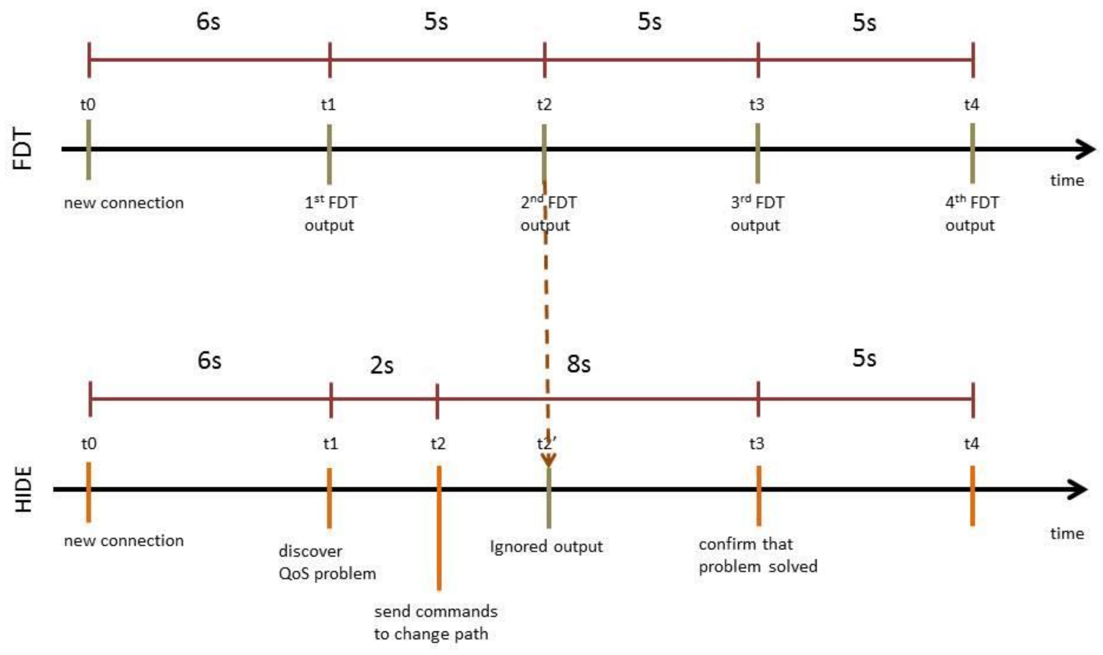


Figure 8: Time diagram of FDT server and HIDE

According to our configuration and the design of HIDE, our **Δ** time is 16 seconds. This happens because HIDE ignores the second output of the FDT server. This second performance information appears no more than three seconds after sending the commands to the controller for redirecting the traffic, which means that the specific output is unreliable for confirming about the correction of the QoS.

5.2 Performance characteristics

In order to test the performance of our component, we created three different scenarios which were based on transferring different files where each one had different size. The smallest file was 1 MB and the biggest was 9 GB. In total we had 25 different files, 9 files between 1MB and 100MB, 8 files between 125MB and 1000MB and also 8 files between 1250MB and 9GB. We could not perform tests on bigger files than 9 GB, due to disk size limitation on VMs. To be more specific, a VM had 10 GB total disk space from which 1 GB was occupied by operating system, other programs and libraries. In all tests the files were moved from the FDT server to client by initiating one TCP connection. We used FDT client for measuring the total transfer time of a file but we observed also a two-second difference at the given output between the client and the server. After investigation we discovered that this behavior happens due to the slow starting of the java application. The results from scenario one and scenario two are presented in Figure 9 where the scale of the X-axis is 12.5MB between 1MB and 100MB, 125MB between 100MB and 1000MB and also 1250MB between 1GB and 9GB.

5.2.1 First scenario

At the first scenario, we performed transfers of the files through Path1. The network had no other traffic interfering, so the transfer could reach easily the limit of the link. Because of that, we can assume that this is an ideal transfer of a file and we performed that test in order to extract the ideal transfer times and compare them with the results of the following tests. If we consider as stable parameters the TCP behavior, the total bandwidth of the links and the size of the files, it is impossible to have better performance than the measured one.

From the received results we observed that the smallest time which FDT requires to complete the transfer of a very small file (<12.5Mb) is about eight seconds. Some of the reasons for explaining this behavior are that FDT is a java based application so it takes some time to initialize the Java Virtual Machine (JVM) and exchange some handshake messages that are required to initiate a data transfer. In contrary, for big files these eight seconds can be considered as a small percent of the total transfer time. The results of this test have been marked by blue dotted line in Figure 8.

5.2.2 Second scenario

During the second scenario, we created one TCP connection on Path1using Iperf, in order to create fake traffic on the link. Through the whole duration of each file movement between server and client, Path 1 was occupied by the mentioned TCP connection while Path 2 was completely empty. As a result of this action, the maximum transfer rate that could be achieved was approximately 45Mbps - 50Mbps for each of them. The 50% availability of the bandwidth had an obvious impact at the total transfer time of each file which was approximately doubled in comparison with the ideal one.

At the second repetition of this test, HIDE was enabled and the Iperf was injecting again fake traffic on Path1. Each new file movement was using also Path 1 by default, as a result of having static flows on the network. Our prototype was able to recognize that every connection was facing the QoS problem and -according to the algorithm- HIDE was sending commands to the controller to switch path. For files having size smaller than 100 MB either there were no differences in transfer times or the difference were really small. Figure 9 demonstrates the results of this test along with the results of the other tests.

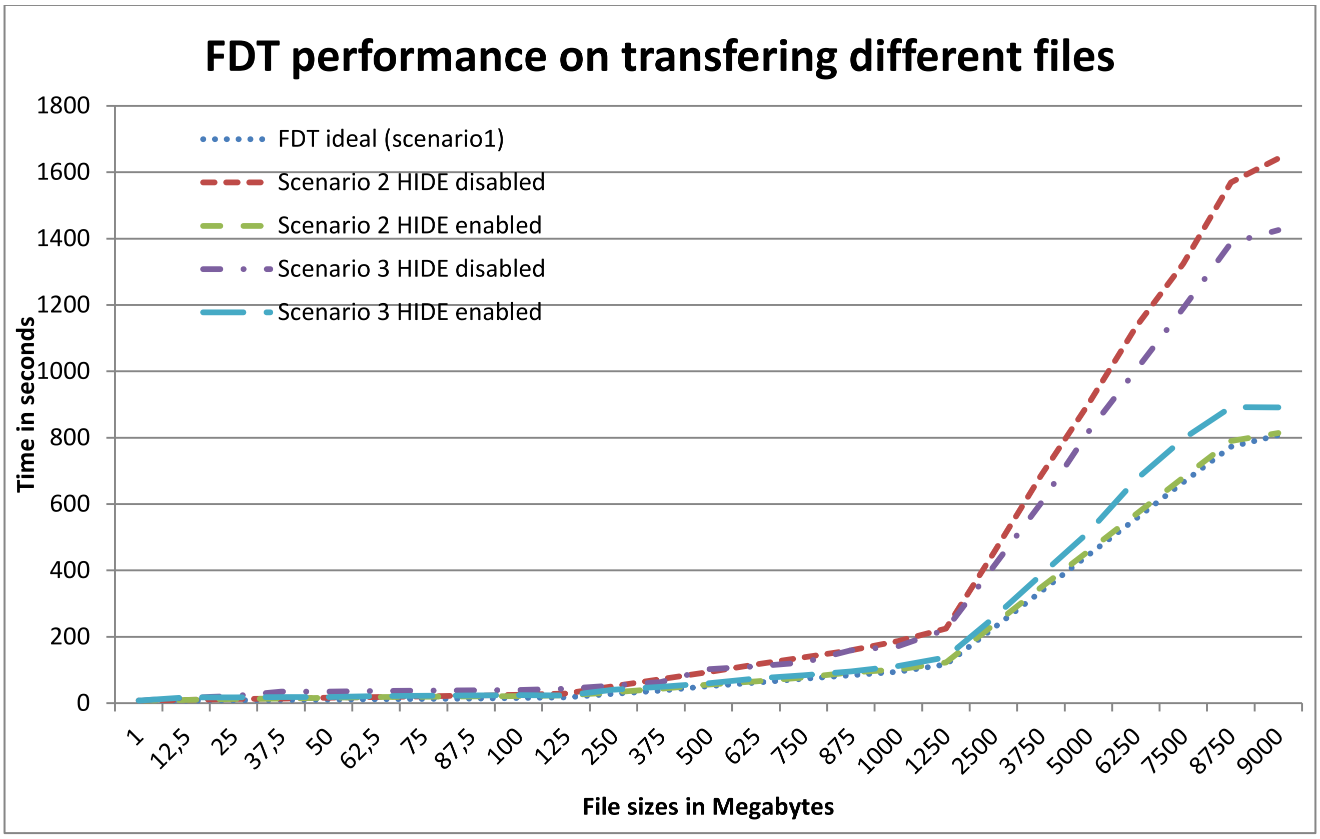


Figure 9: Results of all scenarios based on different file sizes

5.2.3 Third scenario

For our third scenario, we created a Python script [see Appendix 2] that it uses Iperf to create ten parallel TCP connections in order to create congestion on both paths. Because we wanted to create a more realistic scenario than the previous one, the script was injecting fake traffic with thirty seconds difference, first to Path1 and then to Path2. HIDE was enabled only during the second repetition and for both times, the FDT connections were initiated over the Path 1. As a result of the fake traffic, the transfer rates were quite low (around 10 Mbps) for the first thirty seconds of the data movement. When the script was switching path to the fake traffic, the FDT was able to utilize the maximum available bandwidth of the link for half a minute, before the noise returns back to the previous path.

When we repeated the scenario with our prototype enabled the results were better than the first test of this scenario. Because of HIDE, the file transfer was facing congestion for only eight seconds instead of thirty seconds that had before. Of course the total transfer time of each file movement was a little higher than the ideal (approximately 15% more), but there is no doubt that HIDE was trying to switch paths every time that an FDT connection was facing congestion problems. Figure 9 presents the results of this scenario along with the results of the other ones.

Figure 10 is a representative case of a file transfer (1.25 GB file size) we performed on scenario three. The transfer rates were taken from the output of FDT server at a stable interval of five seconds. With HIDE enabled, we can see that the speed had smaller reductions and stayed more at high level due to the fact that HIDE was redirecting the connection to a less busy path. This had also a serious impact to the time required for completing this data movement.

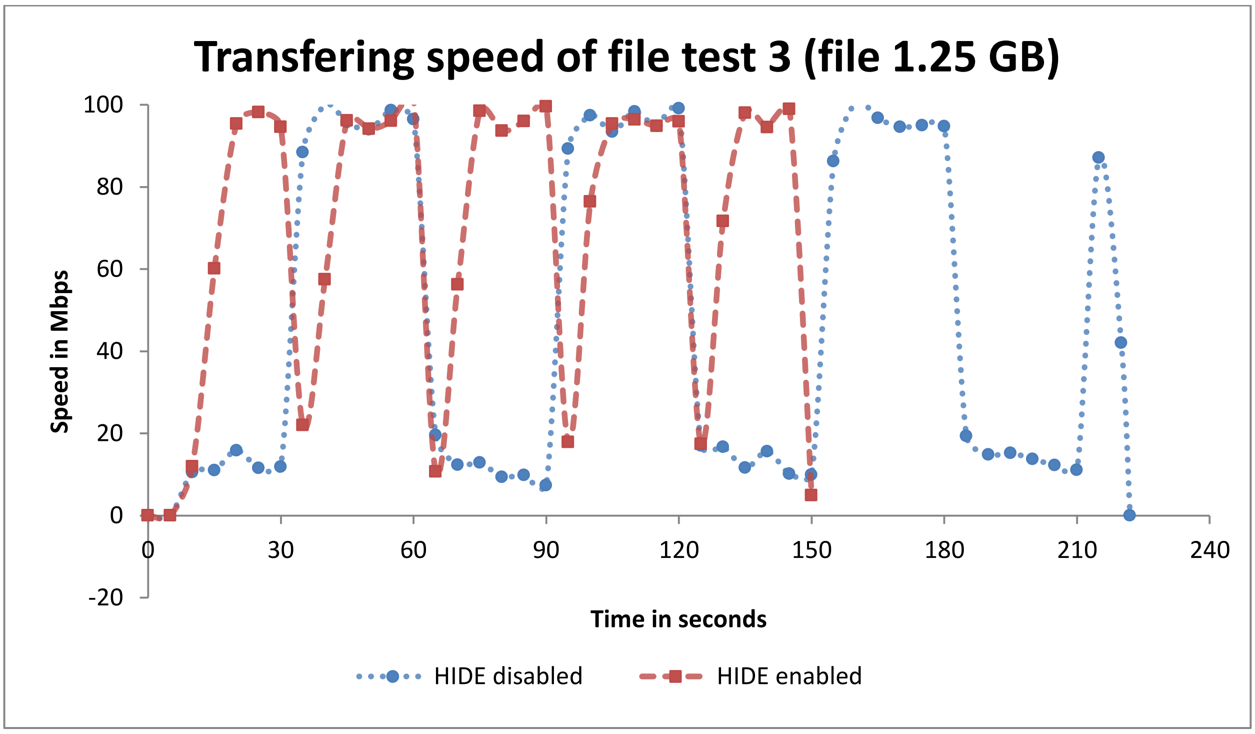


Figure 10: Visualization of the results for scenario 3 (file 1.25 GB)

We observed the same behavior also when we transferred bigger files which require more time to be transferred. Figure 11 visualizes the transfer of a file which is 8.75 GB and was completed in 45% less time when HIDE was applied.

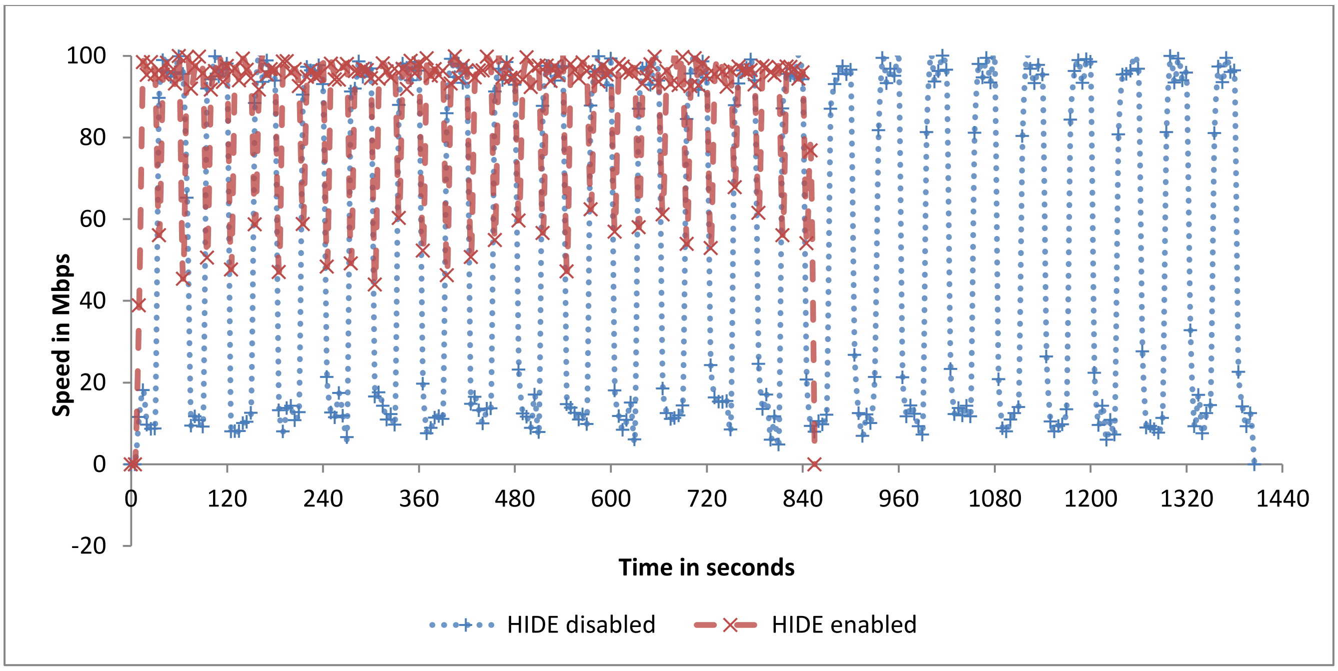
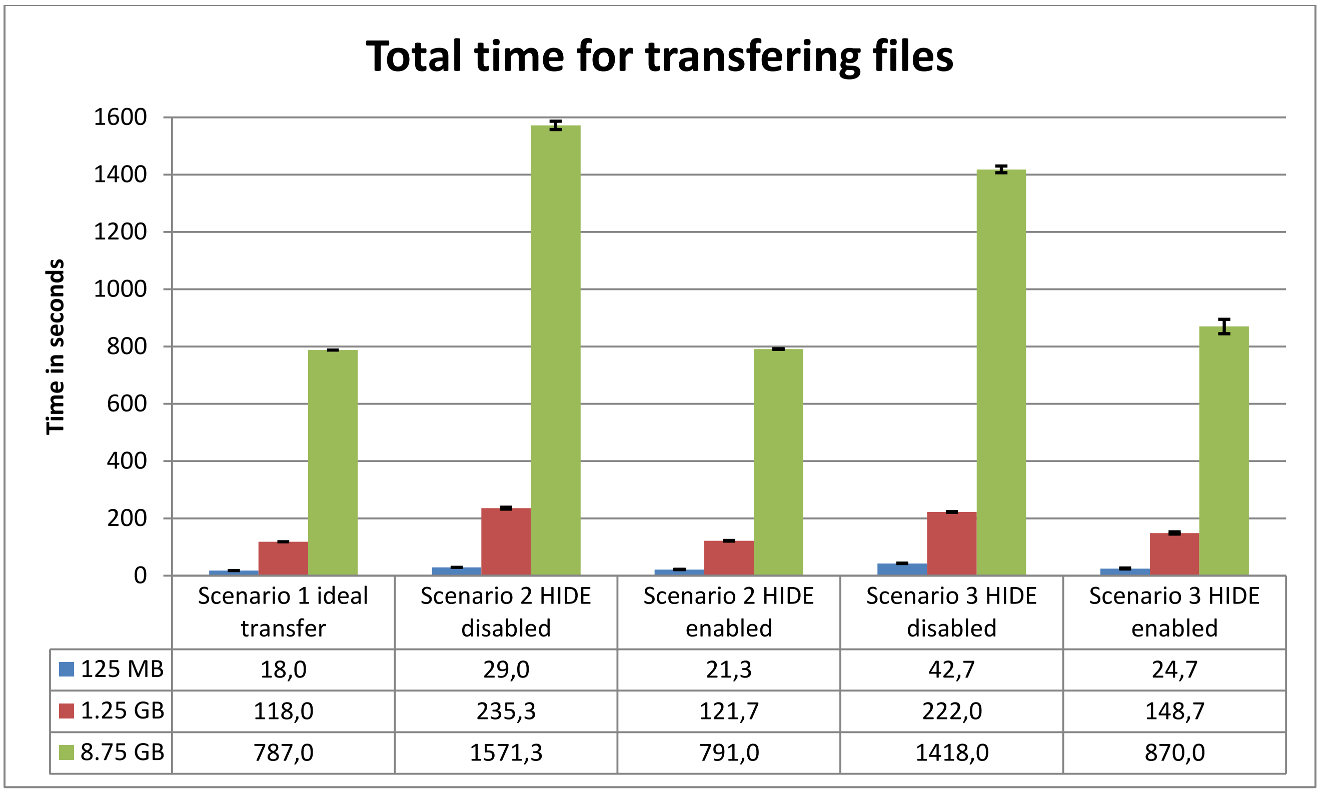


Figure 11: Visualization of the results for scenario 3 (file size 8.75GB)

Moreover, Figure 10 presents the results of three different files in all three scenarios. The benefits of applying HIDE are not so obvious at file sizes smaller than 500MB, due to the eight seconds time required for detecting and correcting the QoS problem. But as the as the file size increases and more transfer time is required, the link congestion creates a significant delay on and the FDT connection gains important benefits from path change. By studying these results we reach to the conclusion that this prototype can be adopted by Big Data science where huge volumes of data are being transferred.



	Scenario 1 ideal transfer	Scenario 2 HIDE disabled	Scenario 2 HIDE enabled	Scenario 3 HIDE disabled	Scenario 3 HIDE enabled
🟦 125 MB	18,0	29,0	21,3	42,7	24,7
🟥 1.25 GB	118,0	235,3	121,7	222,0	148,7
🟩 8.75 GB	787,0	1571,3	791,0	1418,0	870,0

Figure 12: Total time for transferring three different files

Finally, the interval for gathering the statistics from switches is a parameter that has been investigated also. By default it is configured at one second but this changeable value is a tradeoff between speed and quality. For interval which was equal to half a second, we observed significant higher speeds such as 130 Mbps or above for links that were 100 Mbps .When the interval was configured to even lower values, we observed transfer rates equal to zero for an ongoing connection that was at maximum speed.

On the other hand, we tried also higher values for the interval in comparison to one second, such as two or three. The results were extremely close to those that we observed with the one second interval. But the extra second delay in our small testing environment is an overhead which provides very few extra accuracy and significant delay when we transfer small or medium size files.

After observing carefully the results, we conclude that file transferring using a data movement application can gain benefits from our prototype. When it was necessary, HIDE was able to redirect the connection to a path which was less loaded and this had an immediate impact to the transfer time. According to the results, when HIDE was enabled, the transfer time was approximately 15% more than the ideal one and this number demonstrate that the algorithm which we used is effective.

One parameter that can be changed in order to decrease the reaction time of the component is the interval between gathering the statistics from the network. We tried different values in order to discover a balance between minimum reaction time and accuracy but the tests which we presented at this chapter had this interval configured at one second. A lower value of this interval raise automatically concerns about the quality of the statistics and how reliable can be.

5.3 Lessons learned

In this section we will mention the problems that occurred during our research. We were able to overcome all of them but we lost significant time and we hope that our findings will help the people involved at these projects to correct the related parts.

5.3.1 ExoGENI related problems

For deploying the virtual topology at the ExoGENI rack we used Orca Flukes [17], a Java based tool to design and configure the topology of a virtual network by using a user friendly environment. Linux users have to install Oracle's official version of Java in order to run the tool. The negative side of this convenient application is that does not allow to the user to make changes on the topology after deploying it into the rack.

On the other hand, ExoGENI was facing problems on deploying the topology which was designed by Orca FLUKES. To be more specific, ExoGENI was not able to activate some high speed links (1Gbps) despite the fact that the physical servers of the rack are connected with 10Gbps Ethernet links. Moreover, ExoGENI was not able to attach the Virtual Disk image files with the desired OS to the VMs. The "Ubuntu 12.04" images could never be loaded to the VMs even if we tried several times to several VMs and topologies.We faced also the same behavior with the "Debian 7" image. Only the "OVS" (Debian 6 with OpenVSwitch installed) was able to be loaded successfully anytime to any VM.

5.3.2 Problems at the data movement applications

Our first decision was to deploy GridFTP at the edge nodes of our virtual topology, but the time which is required to achieve this is unpredictable. After installing and configuring the Debian 6 version of GridFTP by following the instructions at the official web site of Globus, the machines were not able to connect because GridFTP is incompatible with the security library of Debian 6 and as a result of it, the SSLv3 handshake was unsuccessful. After upgrading the OS to Debian 7 and reinstalling GridFTP with the related version, connection could not be established even after disabling all the authentication configurations. We managed to run GridFTP locally (moving one file from one folder to another on the same machine) but not over the network. At this point we decided to replace GridFTP with FDTand recover the lost time.

6. Discussion

The topic of our research essentially relies on the usefulness of the SDN technology in Big Data science in order to improve the QoS. The data movement applications have been developed on top of well-known protocols by extending their capabilities in order to increase the transfer rates and decrease the required time. We presented three of them, GridFTP from Globus, bbFTP from NASA Research & Engineering Network and FDT from CERN.

In a virtual topology we built, our Python based component was able to provide us some interesting results. Based on our algorithm and the knowledge of the network topology, the main purpose of our prototype was to redirect the traffic of a problematic connection to a less busy path.

The results were promising, since we were able to complete data transfers very close to the ideal required time. But this does not mean that our solution is the perfect one, like all the others it has its positives and negatives.

Firstly, HIDE is using SDN technology to enhance the data movement. This new promising networking concept provides flexibility and high networking controllability and our component inherits these benefits. Secondly, our prototype provides an adequate level of abstraction and portability. The first one means that HIDE is located in the user space of the OS and can be adapted to the needs of many SDN networks. By changing parts of the source code HIDE can be used with any controller or data movement application and this is a result of our theoretic approach. Lastly, the intelligence of HIDE is based on real time input, which means that it always tries to select the best path for every TCP connection that needs to be redirected.

The negative part of our solution is that the lower bound of the reaction time is depended on the FDT server. HIDE is not able to react faster since it needs the server’s output for making calculations and taking decisions. The current topology knowledge is something that does not provide more flexibility to HIDE, because it has been hardcoded inside the source code. To be more specific, the script contains hash tables with the available network paths, maximum bandwidth of each link and a list of all the OpenFlow switches. This means that it is not able to perform its operations in another topology than the one we used.

7. Conclusion

This report is a preliminary study on a research gap we discovered between Big Data science and Software Defined Networking. Beside the fact that data movement applications enhance well-known protocols in order to increase the transfer rate, the classic network problem of link congestion does not leave them unaffected.

By using SDN technology we demonstrated that the performance of the data transfer applications can be optimized. Based on network controllability and connection monitoring we were able to redirect the traffic whenever it was necessary and this action had an immediate impact to the behavior of the application. This result is very promising for the ENVRI project where many petabytes of experimental data are expected to be moved across the internet.

8. Future work

The prototype we developed follows the Hybrid programming approach we analyzed in chapter three, which means that it has some controller dependency and also some application level dependency. In our opinion the ideal prototype is completely independent from any controller and any data movement application, but this approach is difficult to be implemented because neither the available OpenFlow Controllers nor the data movement applications provide a common communication interface. It would be interesting for a solution to be investigated which contains one of these two parameters completely independent.

Another interesting research field of our work is the intelligence area of the algorithm. In order to prove our theory in short time the component contains network knowledge and hardcoded thresholds. This implementation decrease the abstraction level but since the network knowledge is part of the algorithm the component should require it from the controller and build the available paths automatically.

Finally, the main target of this project and its purpose is to use the results and the extracted knowledge in the ENVRI project where a more advanced component will cover the needs of transferring Big Data between experimental environments.

9. References

[1] Big Data. http://en.wikipedia.org/wiki/Big_data

[2] Dave Beulke. Big Data Impacts Data Management: The 5 Vs of Big Data. http://davebeulke.com/big-data-impacts-data-management-the-five-vs-of-big-data/

[3] JackuesBughin, Michael Chui and James Manyika. Clouds, big data and smart assets: Ten tech-enabled business trends to watch. August 2010

[4] ENVRI project. http://www.envri.eu/

[5] Brian Tierney, Ezra Kissel, Martin Swany, Eric Pouyoul. Efficient Data Transfer Protocols for Big Data. *In Proceedings of the 8th IEEE International Conference on eScience*, October 2012

[6] Open Networking Foundation. Software Defined Networking (SDN) Definition. https://www.opennetworking.org/sdn-resources/sdn-definition

[7] The SNE ExoGENI rack. http://sne.science.uva.nl/openlab/

[8] Globus. GridFTP. http://toolkit.globus.org/toolkit/docs/latest-stable/gridftp/

[9] bbFTP, http://www.nren.nasa.gov/bbftp.html

[10] Fast Data Transfer – FDT, http://monalisa.cern.ch/FDT/

[11] Yunhong Gu and Robert L. Grossman. UDT: UDP-based Data Transferfor High-Speed Wide Area Networks. May 2007

[12] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. *RFC 5040 (Proposed Standard)*, October 2007.

[13] A. Ford, C. Raiciu, M. Handley, S. Barre and J. Iyengar. Architectural Guidelines for Multipath TCP Development. *RFC 6182*, March 2011

[14] Project FloodLight. http://www.projectfloodlight.org/floodlight/

[15] The Iperf measuring tool. http://iperf.fr/

[16] Avior, Open Source Floodlight GUI Released. http://www.projectfloodlight.org/blog/2012/07/25/avior-open-source-floodlight-gui-released/

[17] Orca Flukes. https://geni-orca.renci.org/trac/wiki/flukes

[18] Michael Jarschel, Florian Wamser, Thomas Hoen, Thomas Zinner and Phuo Tran-Gia. SDN-Based Application-Aware Networking on the Example of Youtube Video Streaming. *In Proceedings of the 2th IEEE European Workshop on Software Defined Networks (EWSDN),* October 2013

[19] CosminDumitru, Ralph Koning and Cees De Laat. 40 Gigabit Ethernet: Prototyping Transparent End-to-EndConnectivity. March 2011

APPENDIX 1: HIDE (hide.py)

```
#!/usr/bin/python3

__author__ = 'skonstantaras, igrafis'

import os
import json
import time

#IP of the Floodlight Controller
controllerRestIp = "128.227.10.8"
#
#Datapath ID of switch1
sw1_dpdid = "00:00:42:40:40:22:70:40"
#
#Datapath ID of switch2
sw2_dpdid = "00:00:3e:37:65:59:8f:4e"
#
#Datapath ID of switch3
sw3_dpdid = "00:00:da:f6:f0:f8:1e:4c"
#
#Datapath ID of switch4
sw4_dpdid = "00:00:72:b3:60:41:0f:4b"
#
#IP of FDT server
fdt_srv = "128.227.10.17"
#
#local IP of FDT server
fdt_srv_lan = "192.168.2.2"
#
#Port connecting outside of network
outside = "out"
#
#Topology of network
topology = {sw1_dpdid: {1: outside, 2: outside, 3: sw2_dpdid, 4:
sw3_dpdid},
            sw2_dpdid: {1: sw1_dpdid, 2: sw4_dpdid},
            sw3_dpdid: {1: sw4_dpdid, 2: sw1_dpdid},
            sw4_dpdid: {1: sw2_dpdid, 2: sw3_dpdid, 3: outside, 4:
outside}}
#
#username of the server VM
user_name = "root"
#
#The commands that will run at the server VM
script = "tail -n10 /var/log/fdt_output.log"
#
#The ssh command
ssh = "ssh " + user_name + "@" + fdt_srv + ' "' + script + '"'
#
#Measurment between two changes of path
change_counter = 1
#
```

```
#Capacity of each path
path_capacity = {sw2_dpdid: (100 * 1024 * 1024), sw3_dpdid: (100 * 1024 *
1024)}
#
#Tolerance for check
tolerance_check = 0.8
#
#Tolerance for changes
tolerance_change = 1.2
#
#Time for taking statistics
statistics_time = 1
#
#Sleeping time when nothing happens
sleep_time = 1
#
#Table with FDT connections
fdt_connections = {}
#
#The server's output
last_output = ""
#
#Last command executed
last_command = ""
#
#FDT server word to specify that a connection started
start = "STARTED"
#
#FDT server word to specify that a connection started
finish = "FINISHED"


#check FDT server
def check_fdt_server():
    global last_output

    #get information from FDT server
    new_output = os.popen(ssh).read()
    #new_output = subprocess.check_output(ssh, shell=True,
universal_newlines=True)

    #check for new connections
    if new_output != last_output:
        last_output = new_output
        #returns 1 if something changed
        return 1
    else:
        #returns 0 if nothing changed
        return 0


#update the list with connections
def update_fdt_table():
    global fdt_connections, last_output, last_command

    #splits the output of FDT server line by line
    output_array = last_output.splitlines()

    #check for new lines at the output
    if last_command:
        for pos, com in enumerate(reversed(output_array)):
            if com == last_command:
                i = pos
                break
    else:
        i = len(output_array)

    last_command = output_array[-1]

    #update the list with connections
    for line in output_array[-i:]:
        output_line = line.split()

        #change the status of a connection from "start" to "finish"
        if output_line[0] == finish:
            if output_line[1] in fdt_connections:
                fdt_connections[output_line[1]]["status"] = finish
                print("Transfer finished\n")
        #add a new connection
        elif output_line[0] == start:
            fdt_connections[output_line[1]] = {}
            fdt_connections[output_line[1]]["IP"] = output_line[2]
            fdt_connections[output_line[1]]["port"] = output_line[3]
            fdt_connections[output_line[1]]["status"] = start
            fdt_connections[output_line[1]]["speed"] = -1
            fdt_connections[output_line[1]]["change_counter"] = 0
            print("Transfer started\n")
        #add speed to a connection (bps)
        else:
            for con_id in fdt_connections:
                if fdt_connections[con_id]["status"] == start:
                    if fdt_connections[con_id]["change_counter"] == 0:
                        scale = 1
                        if output_line[2] == "Kb":
                            scale = 1024
                        elif output_line[2] == "Mb":
                            scale = 1024 * 1024
                        elif output_line[2] == "Gb":
                            scale = 1024 * 1024 * 1024

                        fdt_connections[con_id]["speed"] =
int(float(output_line[1]) * scale)
                    else:
                        fdt_connections[con_id]["change_counter"] -= 1

#    for key, value in fdt_connections.items():
#        print(key, value)
#    print()


#check for ongoing connections
def check_ongoing_connection():
    new_cons = []

    #check the list with all connections to find ongoing connections
    for id_con in fdt_connections:
        if fdt_connections[id_con]["status"] == start:
            new_cons.append(id_con)

    #return a list with ids of ongoing connections
    return new_cons


#Get information for a list of switches per type "stat_type"
def get_information_switches(stat_type, *sw_ids):
    global controllerRestIp
    parsed_result = {}

    for sw_id in sw_ids:
        command = "curl -s http://%s:8080/wm/core/switch/%s/%s/json" %
(controllerRestIp, sw_id, stat_type)
        result = os.popen(command).read()
        temp = json.loads(result)
        parsed_result[sw_id] = temp[sw_id]

    return parsed_result


#Get rates for all flows from switch with id "sw_id"
def get_rate_from_switch(parsed_result, sw_id, var_statistics):
    stats = []

    for flow in parsed_result[sw_id]:
        #get information only for download
        download_input_port =
list(topology[sw_id].keys())[list(topology[sw_id].values()).index(sw4_dpd
id)]
        if flow["match"]["inputPort"] == download_input_port:
            stats.append(flow[var_statistics])

    return stats


#Get statistics for a list of switches
def get_statistics(var_statistics, *sw_ids):
    stats_all = []
    stats = {}

    #Get statistics two times with difference between the measurements
"statistics_time" seconds
    for i in range(2):
        stats_all.append([])

        #Get statistics for all switches in "sw_ids" list
        for sw_id in sw_ids:
            #Get statistics for switch with id "sw_id"
            parsed_result = get_information_switches("flow", sw_id)
            stat = get_rate_from_switch(parsed_result, sw_id,
var_statistics)
            stats_all[i].append(stat)

        if i == 0:
            time.sleep(statistics_time)

    #Calculate statistics
    for i, sw_id in enumerate(sw_ids):
        dif = 0
        for j, packet in enumerate(stats_all[0][i]):
            dif += (stats_all[1][i][j] - stats_all[0][i][j])

        stats[sw_id] = (dif / len(stats_all[0][i])) / statistics_time

    return stats


#Find the path that the flow is using
def find_current_path(parsed_result, con_id):
    information = []

    #check every switch
    for switch in parsed_result:
        #check every flow in a switch
        for flow in parsed_result[switch]:
            #check if this is a flow that the connection uses
            if (flow["match"]["networkDestination"] in (fdt_srv_lan,
fdt_connections[con_id]["IP"])) and (flow["match"]["networkSource"] in
(fdt_srv_lan, fdt_connections[con_id]["IP"])):
                port = flow["actions"][0]["port"]
                if topology[switch][port] != outside:
                    information.append([switch,
flow["actions"][0]["port"]])

    return information


#Check the speed of ongoing connection
def check_connection_speed(con_id):
    global path_capacity, tolerance_check
    status = 0

    speed = fdt_connections[con_id]["speed"]
    #check if component has connection's speed
    if speed > -1:
        best_speed = max(path_capacity.values())
        best_speed_tolerance = best_speed * tolerance_check

        #return 0 if the speed is good
        if speed >= best_speed_tolerance:
            print("speed is good", int(speed/1024/1024), "Mbps\n")
            status = 0
            fdt_connections[con_id]["speed"] = -1
        #return 1 if the speed is low
        else:
            print("speed is bad", int(speed/1024/1024), "Mbps")
            status = 1

    return status

#Check the statistics of network
def check_network(con_id, stats, parsed_result):
    global tolerance_change, change_counter, path_capacity
    best_p = []

    #find current path
    information = find_current_path(parsed_result, con_id)
    sw_id = information[0][0]
    port = information[0][1]
    current_path = topology[sw_id][port]

    #find alternative path
    for connections in topology[sw_id]:
        if topology[sw_id][connections] not in (current_path, "out"):
            alternative_path = topology[sw_id][connections]
            alternative_port = connections
            break

    #calculate used bandwidth of alternative path
    used_bandwidth = stats[alternative_path] * 8

    #calculate available bandwidth of alternative path
    available_bandwidth = path_capacity[alternative_path] -
used_bandwidth

    speed_tolerance = fdt_connections[con_id]["speed"] * tolerance_change
    print("Current path used bandwidth: ",
int(stats[current_path]*8/1024/1024), "Mbps")
    print("Other path used bandwidth: ",
int(stats[alternative_path]*8/1024/1024), "Mbps")

    #check if the alternative path has more free bandwidth than the
current
    if available_bandwidth > speed_tolerance:
        best_p.append([sw_id, port, alternative_port])
        sw_id = information[1][0]
        port = information[1][1]
        for connections in topology[sw_id]:
            if topology[sw_id][connections] not in (current_path,
outside):
                alternative_port = connections
                best_p.append([sw_id, port, alternative_port])
                fdt_connections[con_id]["change_counter"] =
change_counter
                break

    fdt_connections[con_id]["speed"] = -1

    return best_p
```

```
#Manipulate flow
def manipulate_flow(best_p, new_con_id):
    global fdt_connections

    #check if a flow need to change path
    if best_p:
        print("change path\n")

        #for all switches that need to change
        for i in range(len(best_p)):
            sw_id = best_p[i][0]
            #get all flows from switch with id "sw_id"
            command = "curl -s
http://%s:8080/wm/staticflowentrypusher/list/%s/json" %
(controllerRestIp, sw_id)
            result = os.popen(command).read()
            parsed_result = json.loads(result)

            port_old = best_p[i][1]
            port_new = best_p[i][2]

            #for every flow in a switch
            for flow_name in parsed_result[sw_id]:
                dst_ip =
parsed_result[sw_id][flow_name]["match"]["networkDestination"]
                src_ip =
parsed_result[sw_id][flow_name]["match"]["networkSource"]

                #check if the flow need to change
                if (dst_ip in (fdt_srv_lan,
fdt_connections[new_con_id]["IP"])) and (src_ip in (fdt_srv_lan,
fdt_connections[new_con_id]["IP"])):
                    port_out =
parsed_result[sw_id][flow_name]["actions"][0]["port"]
                    port_in =
parsed_result[sw_id][flow_name]["match"]["inputPort"]

                    #change the port that the connections will use
                    if port_out == port_old:
                        port_out = port_new
                    elif port_in == port_old:
                        port_in = port_new

                    #create the string for the changed flow and send it
to the switch
                    new_flow = """curl -s -d '{"switch": "%s",
"name":"%s", "ingress-port":"%s","active":"true", "actions":"output=%s",
"ether-type":"0x0800", "dst-ip":"%s", "src-ip":"%s"}'
http://%s:8080/wm/staticflowentrypusher/json""" % (sw_id, flow_name,
port_in, port_out, dst_ip, src_ip, controllerRestIp)
                    result2 = os.popen(new_flow)


#================Script starts here:
#endless loop
while True:
    #check FDT server
    output = check_fdt_server()
    #check if something change
    if output:
        #update the table with connections
        update_fdt_table()

    #check for ongoing connections
    ongoing_connections = check_ongoing_connection()
    if ongoing_connections:
        check_statistics = 0

        #for every ongoing connection
        for ongoing_connection_id in ongoing_connections:

            #check if the speed of the connection is good
            check_connection =
check_connection_speed(ongoing_connection_id)

            #if speed is not good
            if check_connection == 1:
                if check_statistics == 0:
                    #get rate for each path
                    statistics = get_statistics("byteCount", sw2_dpdid,
sw3_dpdid)

                    #get information for all switches
                    parsed_Result = get_information_switches("flow",
sw1_dpdid, sw2_dpdid, sw3_dpdid, sw4_dpdid)

                    check_statistics = 1

                #check connection's speed
                best_path = check_network(ongoing_connection_id,
statistics, parsed_Result)

                #manipulate flows
                manipulate_flow(best_path, ongoing_connection_id)

        if check_statistics == 0:
            time.sleep(sleep_time)
    else:
        time.sleep(sleep_time)
```

APPENDIX 2: Noise generator script (noise.py)

```
#!/usr/bin/python3

__author__ = 'skonstantaras, igrafis'

import os
import time

os.popen("iperf -c 192.168.2.3 -t3600 -P10")

x = 0
while True:
    if (x % 2) == 0:
        str = """curl -s -d '{"switch": "00:00:42:40:40:22:70:40",
"name":"s1-2.3-to-2.4", "ingress-port":"2","active":"true",
"actions":"output=3", "ether-type":"0x0800", "dst-ip":"192.168.2.4",
"src-ip":"192.168.2.3"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json; \
                 curl -s -d '{"switch": "00:00:42:40:40:22:70:40",
"name":"s1-2.4-to-2.3", "ingress-port":"3","active":"true",
"actions":"output=2", "ether-type":"0x0800", "dst-ip":"192.168.2.3",
"src-ip":"192.168.2.4"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json; \
                 curl -s -d '{"switch": "00:00:72:b3:60:41:0f:4b",
"name":"s4-2.3-to-2.4", "ingress-port":"1","active":"true",
"actions":"output=3", "ether-type":"0x0800", "dst-ip":"192.168.2.4",
"src-ip":"192.168.2.3"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json; \
                 curl -s -d '{"switch": "00:00:72:b3:60:41:0f:4b",
"name":"s4-2.4-to-2.3", "ingress-port":"3","active":"true",
"actions":"output=1", "ether-type":"0x0800", "dst-ip":"192.168.2.3",
"src-ip":"192.168.2.4"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json"""
        print("path 1")
    else:
        str = """curl -s -d '{"switch": "00:00:42:40:40:22:70:40",
"name":"s1-2.3-to-2.4", "ingress-port":"2","active":"true",
"actions":"output=4", "ether-type":"0x0800", "dst-ip":"192.168.2.4",
"src-ip":"192.168.2.3"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json; \
                 curl -s -d '{"switch": "00:00:42:40:40:22:70:40",
"name":"s1-2.4-to-2.3", "ingress-port":"4","active":"true",
"actions":"output=2", "ether-type":"0x0800", "dst-ip":"192.168.2.3",
"src-ip":"192.168.2.4"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json; \
                 curl -s -d '{"switch": "00:00:72:b3:60:41:0f:4b",
"name":"s4-2.3-to-2.4", "ingress-port":"2","active":"true",
"actions":"output=3", "ether-type":"0x0800", "dst-ip":"192.168.2.4",
"src-ip":"192.168.2.3"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json; \
                 curl -s -d '{"switch": "00:00:72:b3:60:41:0f:4b",
"name":"s4-2.4-to-2.3", "ingress-port":"3","active":"true",
"actions":"output=2", "ether-type":"0x0800", "dst-ip":"192.168.2.3",
"src-ip":"192.168.2.4"}'
http://128.227.10.8:8080/wm/staticflowentrypusher/json"""
        print("path 2")

    os.popen(str)

    time.sleep(30)

    x += 1
```