Empowering Applications with RFC 6897 to Manage Elephant Flows in Datacenter Networks

Alan C. Silva, Regis Martins and Fábio L. Verdi
LERIS – Federal University of São Carlos – UFSCar
Sorocaba – SP – Brazil

***Abstract*—The Multipath TCP (MPTCP) is able to explore network resources with multi-connected devices. Altogether with MPTCP, the RFC 6897 defines an API that adds the freedom for applications to manage the MPTCP subflows by giving them the power to control important aspects of the MPTCP implementation's behavior (e.g. to open and close subflows as they may wish). This paper presents an initial implementation of the RFC 6897 so that MPTCP subflows can be added by the application. To demonstrate the benefits of such mechanism, we build an HTTP application that detects elephant flows and breaks these into mice flows using the API. The tests were done using a cubic network topology and showed that the power given to the application to break elephant flows into mice flow decreased the Flow Completion Time (FCT) due to the spreading of subflows in the network.**

***Keywords*—Multipath Routing, MPTCP, RFC 6897, Application Awareness, Elephant Flows Management.**

I. Introduction

Today, end-host devices usually have multiple network interfaces mainly in datacenter networks, allowing the simultaneous utilization of such interfaces through applications and protocols, which may efficiently exploit these resources. This kind of mechanism can assist the traffic management, enabling the bandwidth control usage and more efficient flows distribution. The applications that are capable of using these alternatives to better distribute the flows can be referred to as aware applications [1], [2], given its scientific characteristic concerning what is happening inside the network.

Typically, an aware application is based on some mechanisms that work with rules to define the operating mode of an application. In some cases, we may have socket modifications at the user level to allow the implementation of these rules [2].

Utilizing the very same concept, this article presents the usage of an Application Programming Interface (API) that was developed based on the *RFC 6897* [3] aiming at enabling aware interaction between applications and network, taking the advantage of the resources offered by the *Multipath TCP (MPTCP)* protocol [4], [5]. By using this *TCP* protocol's extension, we have the possibility to generate rules in the application so that it can open various flows over one or several network interfaces [6].

In this paper, we are interested in empowering the application for management of elephant traffic. Typically, the flows in the network are classified in two types:

- *Mice Flows*: Concise flows, which have small amount of data;
- *Elephant Flows*: Large and lasting flows, which have high amount of data.

The importance of detecting elephant flows is due to the fact that they promptly and continuously fill network buffers, introducing non-trivial traffic congestion, triggering a significant delay and harming other applications and services.

Some studies performed upon datacenter traffic characteristics [7] show that about 90% of flows are classified as mice flows. These flows, as previously mentioned, are quickly distributed and do not impact in the network's performance. However, the remaining 10% are considered to be elephant flows carrying the biggest part of existent data, requiring fast and effective management and processing to avoid delays, which may compromise applications. The same studies show that 90% of the transfered data in a datacenter are within this 10% of the flows [8].

The treatment of elephant flows is associated to their detection, which is not an easy task, regarding their variable size which requires the utilization of an index factor. This index factor appoints if the flow is elephant or if should not be parametrized through the network traffic study.

Nowadays, there are some solutions that deal with the elephant flows problem. They are based on the idea of detection and treatment in the core of the network using controllers that schedule flows through different paths, as can be seen in [9]. Other solutions change the behavior of switches according to the reports [9], [10], change the behavior of protocols, as in [11], [12] or use an hybrid form such as in [13].

The *RFC 6897* [3] suggests the creation of some extensions in the basic socket interface (API) of the TCP protocol enabling applications to add or remove subflows in an MPTCP connection. The MPTCP is an extension of the TCP protocol defined by *IETF* [6], allowing a TCP connection, which is by default single path, to work as a multiple path connection.

Based on these works' reviews and identifying the main difficulties which still permeate the management of elephant flows, this paper presents an initial implementation of the RFC 6897 and tests it through an application capable of managing such flows without any modifications in the TCP/IP architecture. The application was implemented being aware of the RFC 6897 API so that it could add new MPTCP subflows as required. By doing this, the application could improve its

978-1-5090-4026-117$31.00

final throughput which was evaluated using a typical data center network topology.

The article is organized as follows: Section II presents some basic concepts. The implementation and the evaluation of the use case are detailed in Section III. Finally, the article's conclusion is presented in Section IV.

II. Fundamental Concepts

A. Application Awareness

The system's information awareness concept is firstly introduced in [1] as a mechanism to understand activities of a group which, eventually, presents a context to accomplish its own activity. This awareness helps to manage the collaborative working process among different systems, for example between a given application and the network.

When the concept addresses information, this science is started through a basic identification of the applications existing in the network. When the network understands the application's existence, the next step is to understand the communication state and flow established by these applications. Through this science, the network can understand how the application protocols work, thus enabling better network utilization. Therefore, it is possible to inject intelligence inside the user space applications throughout rules which will allow a better network management by these applications.

In our case, we used the concept of application awareness through a rule which provides the elephant flows identification and breaks them into mice flows achieving better flow rate and better usage of the resources available in the network.

B. Elephant Flows

In the literature, the definition of elephant flows differs according to the focus of the study. Therefore, we identified three definitions that normally are utilized to characterize elephant flows.

Firstly, an specific flow may be considered elephant when its duration exceeds a certain time range which may significant delay other simultaneous flows in the same network, as evidenced in [12]. Secondly, an specific flow may also be defined as elephant when its rate (in Mbps) exceeds a certain number [9]. Finally, an specific flow may be considered elephant when its quantity of bytes transfered exceeds a defined threshold, typically 100MB [8], [13]. This last definition is the one most used and then adopted in this work.

Diverse proposals suggest the creation of mechanisms to control bandwidth and flow delay, ensuring the quality and punctuality of the responses to the applications' users in a datacenter.

In *Casado et. al.* [14], the authors describe several items which are essential in the creation of a solution that may solve the problem of elephant flows, such as:

a) Utilize distinct queues to divide small flows from the elephant flows;
b) Utilize distinct paths to route elephant flows;
c) Forward elephant flows to an exclusive network;
d) Transform elephant flows in small flows and scatter these flows through diverse paths available in the network.

Nowadays, the existent solutions aim to identify and detach elephant flows from mice flows, however, they normally withstand some limitations such as performance decrease when there are network bottlenecks or collisions. This fact guided us to transform elephant flows into mice flows and scattering them through the open subflows, thus adopting the solution d) listed above.

C. Multipath TCP

The *Multipath TCP (MPTCP)* is a TCP protocol extension defined by the *IETF* in [6], allowing a TCP connection which is single path by default to work with a connection of multiple paths, supporting more than one data path of a connection. The MPTCP breaks the traditional TCP connection splitting it in various TCP connections referred to as subflows. These subflows can be spread in the network by using typical routing mechanisms such as ECMP or more intelligent solutions such as a Software Defined Networking controller [15].

One of the main advantages of the MPTCP compared to other solutions such as the SCTP, is the usage of the same TCP protocol structure to route information. This approach makes the MPTCP transparent to TCP applications since it seizes the same socket API of the TCP protocol, as shown in Figure 1.

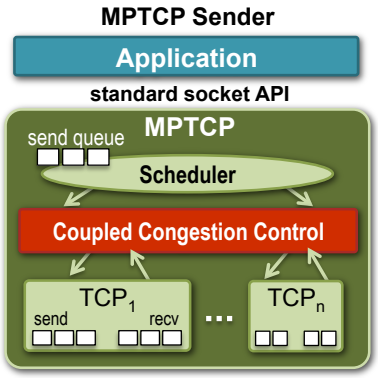

Fig. 1. TCP stack with MPTCP support.

The MPTCP uses the TCP three-way handshake process to perform the necessary transaction which enables its usage. Basically, the following 3 stages are done by the MPTCP:

- *Establish a new MPTCP connection:* In this stage, the sender host checks if the destination supports MPTCP communication *(MP_CAPABLE)* and generates the key exchange which will serve as the connection identifier;
- *Add subflows in an MPTCP connection:* In the event of needing to add a new subflow *(MP_JOIN)*, a new three-way handshake is performed utilizing the key generated as the connection identifier;
- *Transmit data throughout the MPTCP connection.*

D. RFC 6897

On most of operating systems, when an application needs to establish communication it requests the opening of a socket, which may be TCP or UDP, by using the universal and well-known socket API. Then, the operating system maps the socket to a specific network interface using the parameters that are shared through the API and utilizes this information to establish the communication. The *RFC 6897* [3] describes an API that aims to expand this communication through a set of functions present in the MPTCP protocol.

Table I displays a set of functions implemented in the socket API to enable MPTCP manipulation through the applications.

TABLE I
OPERATIONS OF THE MPTCP API

Name	Get	Set	Data Type
TCP_MULTIPATH_ENABLE	x	x	Boolean
TCP_MULTIPATH_ADD		x	Addresses List/Ports
TCP_MULTIPATH_REMOVE		x	Addresses List/Ports
TCP_MULTIPATH_SUBFLOWS	x		Addresses Pair List/Ports
TCP_MULTIPATH_CONNID	x		Integer

The options specified in Table I have the following functions:

- *TCP_MULTIPATH_ENABLE*: enables or disables MPTCP;
- *TCP_MULTIPATH_ADD*: Connects the MPTCP protocol to a set of defined local addresses or adds a new set of local addresses in an existent MPTCP connection. In practical terms, this option allows the inclusion of new subflows to an existing MPTCP connection;
- *TCP_MULTIPATH_REMOVE*: Removes the local address of an MPTCP connection. In practical terms, this option allows that existing subflows are excluded from an MPTCP connection;
- *TCP_MULTIPATH_SUBFLOWS*: Recovers addresses pairs currently utilized by MPTCP subflows;
- *TCP_MULTIPATH_CONNID*: Returns the local connection identifier of the current MPTCP connection.

III. IMPLEMENTATION AND EVALUATION

On the architecture's perspective, the MPTCP protocol may be seen as a set of TCP connections referred to as subflows and are grouped and managed through two points in an MPTCP connection. These subflows are not static and may be established or terminated during the extent of an MPTCP connection.

In the existent implementation, the two main structures for an MPTCP connection control are the subflow's metasockets and sockets. The metasocket is the structure which bridges the main TCP socket and the MPTCP subflow to accomplish control information exchange among the applications. It also has pointers required to access the MPTCP internal structures, including a linked list of the established subflows.

The subflow socket is the internal structure that controls the inclusion and exclusion of subflows. Consequently, we need to create an API that accesses this control structure to allow a complete management of the internal functions. It should be also possible to dynamically control inclusion and exclusion of subflows and other functionalities that were shown in Table I. Then, as a consequence, the subflow socket of this structure needs to be exposed through the developed API.

Currently, the set of operations available in our API implementation that expand the socket API inside the kernel are: `TCP_MULTIPATH_ENABLE` (to enable/disable the protocol by the application), `TCP_MULTIPATH_CONNID` (to obtain the TCP connection ID) and `TCP_MULTIPATH_ADD` (to add a new subflow).

In this regard, we added the interface calls to the MPTCP protocol inside the socket interface in the Linux kernel, represented by the functions *do_tcp_setsockopt*, concerning data configuring functions, and *do_tcp_getsockopt*, concerning data extracting functions. Such functions define the socket interface entry point and are implemented in the file *tcp.c* located at *net/ipv4* inside the kernel source code. A short excerpt of the functions is shown in Listing 1.

```
static int do_tcp_getsockopt(struct sock *sk, int level,
                int optima, char __user *optval,
        int __user *optlen)
{
        case TCP_MULTIPATH_ENABLED:
        [...]
        case TCP_MULTIPATH_CONNID:
        [...]
}

static int do_tcp_setsockopt(struct sock *sk, int level,
                int optima, char __user *optval,
        int __user *optlen)
{
        case TCP_MULTIPATH_ENABLED:
        [...]
        case TCP_MULTIPATH_ADD:
                mptcp_add_subflow(struct sock *sk,
                                  char *num_subflows);
        [...]
}
```

Listing 1. Implemented functions for the MPTCP API extension.

Hence, the initial function call that adds subflows (`TCP_MULTIPATH_ADD`) is performed in this entry point, shown in Figure 2.

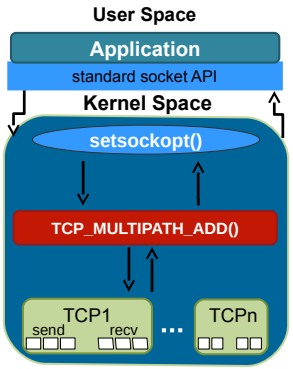

Fig. 2. Creation of new subflows using the API.

We wrote a function named *mptcp_add_subflow* that works as a wrapper to instantiate the new MPTCP socket when a *TCP_MULTIPATH_ADD* is called by the application. Note in Listing 1 that this function is called when the `TCP_MULTIPATH_ADD` flag is present.

The application must use *getsockopt* and *setsockopt* with the correct parameters for calling the MPTCP API functions. The application should inform the IP addresses, ports and the type of path manager. Below, in Listing 2, we show an example of a small source code used to add a new subflow by an application. Firstly, it is necessary to get the ID of the MPTCP connection which is obtained by calling the *getsockopt* function with the *TCP_MULTIPATH_CONNID* flag enabled. Note that the structure *sockfd* is fulfilled with several information, including the connection ID. This structure is then passed as a parameter to the *setsockopt* function with the *TCP_MULTIPATH_ADD* flag enabled.

```
error =  getsockopt(sockfd, IPPROTO_TCP, TCP_MULTIPATH_CONNID,
                &sub_sso, optlen);

error =  setsockopt(sockfd, IPPROTO_TCP, TCP_MULTIPATH_ADD,
                &sub_sso, optlen);
}
```

Listing 2. Example of adding a new subflow by an userspace application.

To validate the API utilization, we developed an *HTTP* application in C language which detects the presence of MPTCP support and establishes a connection with a web server requesting a considerably large file, with the intent of generating a flow which might be monitored. For this work, a flow is considered elephant if it reaches 100MB of data transfered. Whilst this flow is being monitored by the application, a threshold is verified and if the amount of data transfered reaches 100MB, the application establishes the creation of a new subflow so that this stream may be partitioned.

A. Use Case of RFC 6897: Elephant Flows Management

As a use case, we developed an aware application utilizing the implemented API which possesses the capacity of verifying if a certain flow is elephant. When an elephant flow is detected, the application requests the inclusion of subflows according to the need. The algorithm validation is determined through minimum and maximum values during its operation. These values are used as threshold parameters to detect an elephant flow defined from 100MB to 300MB, opening a new subflow for every 50MB until it reaches the maximum number of 4 subflows.

The developed application consists of downloading big archives from a web server by using a pre-established MPTCP connection to originally control and monitor the flow until the established threshold value is reached. When the defined value is achieved, the application can open new subflows guaranteeing that the original flow is shattered into mice flows. These mice flows are distributed along the subflows created in multiple existent paths in the network, as shown in Figure 3.

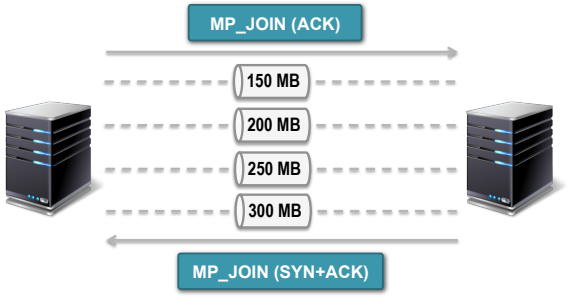

Fig. 3. Subflows Creation.

The application requires a Linux kernel with MPTCP support. It is essential to remind that for the MPTCP module to work it is mandatory to control the connections and creation of subflows through application controlled routines, justifying the need for implementing the API that we developed.

B. Tests and evaluation

We built the whole environment in a *Dell PowerEdge R420* server with 2 *Intel Xeon E5-2430* processors running with *2.2GHz* clock and 48GB of RAM, using the *KVM* in a *Ubuntu Linux Server v.14.04*. Two virtual machines were instantiated, one for running the client application and one for running the server, both with *Ubuntu Linux Server v. 14.04* and MPTCP support. To construct the network topology, we used *Mikrotik* routers virtualized with *RouterOS v. 6.34.4* with ECMP support. The ECMP was chosen because of its flow distribution strategy, widely utilized in datacenters' networks to perform load balancing among different routes.

An advantage obtained by using the MPTCP is the possibility of doing traffic forwarding through diverse different routes, concurrently. So, the more paths the network topology has, the better is the usage of these paths by the MPTCP subflows. Figure 4 presents the network topology used in our tests.

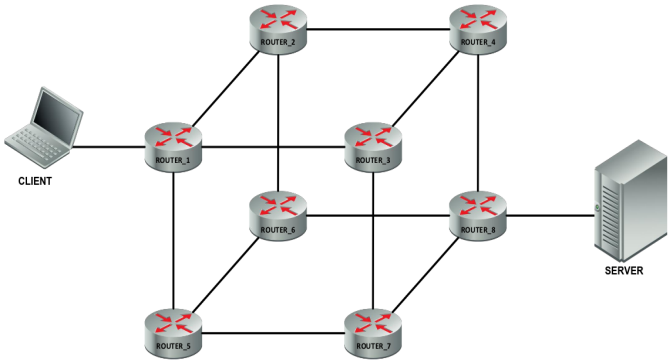

Fig. 4. Cubic network topology used in the evaluation.

We adopted the cube topology utilized in datacenters' networks [16] which enables a variety of paths among end hosts pairs. The adopted topology, although small, enables the establishment of three disjoined paths in the access routers primary links. Assuming that for each of these paths there

will be two other possible paths through the secondary links, we will have a total of six routing paths with the same cost among the hosts connected to the routers in opposite corners. This feature allows the creation of a satisfactory environment for ECMP routing.

The routers that have links with the end hosts are named access routers: *ROUTER_1* and *ROUTER_8*. The other routers are named core routers which are: *ROUTER_2*, *ROUTER_3*, *ROUTER_4*, *ROUTER_5*, *ROUTER_6* and *ROUTER_7*. The links between the access routers and core routers were entitled as *primary links* and the links which connect the core routers among themselves are named *secondary links*.

To evaluate the impact of opening new MPTCP subflows, we used the *HTTP* application which was developed utilizing the API implemented and we calculated the average Flow Completion Time (FCT) of 10 executions. Due to a limitation of the MPTCP kernel, the initial number of subflows is 3. So, the HTTP client application starts running with 3 subflows opened. Then, as already mentioned, a new subflow is created for every 50MB after reaching 100MB of data transfered. A total of 7 subflows is created. In each scenario we performed three file transfers of different size: 300MB, 600MB and 1GB, generating the following results.

In Figure 5, we can observe the FCT for the files of 300MB, 600MB and 1GB. When observing the 300MB file, we note that the *FCT* decreases until the limit of 5 subflows. However, when 6 subflows are opened we can observe the beginning of *FCT* increase. In the second scenario, with the 600MB file, we observe a benefit until opening 5 subflows and an stabilization of the FCT with 6 and 7 subflows. The same happens with the file of 1GB. Therefore, the overhead created by the MPTCP having more than 5 subflows negatively impacts the FCT in these scenarios.

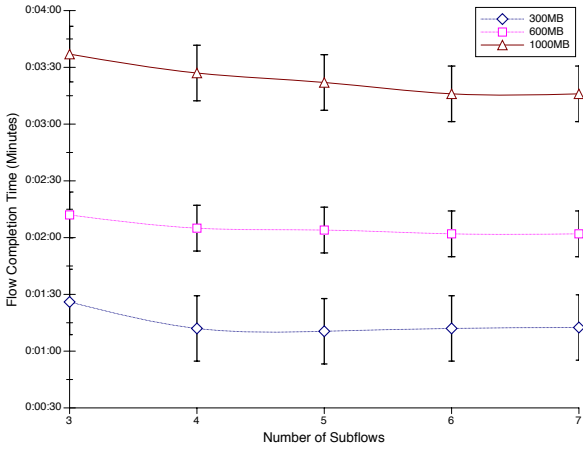

Fig. 5. Average FCT with different file sizes and number of subflows.

The *FCT* differs according to the number of open subflows as well as paths available (joined or disjoined) in the network topology. Clearly, the more available paths, preferably disjoined, the better will be the subflow distribution by the ECMP in the network. The conclusion with the results of Figure 5 is that the flow opening has to consider the network topology, more specifically, the number of available paths. Otherwise, the overhead caused by the MPTCP may harm the final *(FCT)* performance.

By doing this initial evaluation, we could verify that the most significant throughput occurs when we establish 3 to 4 subflows. Thus, the following tests open at the most 4 subflows.

Two scenarios in the same environment were proposed, varying the bandwidth occupancy inside the routes and also the use or non-use of the MPTCP. The CUBIC TCP implementation was used for the cases without MPTCP. The bandwidth occupancy (background traffic) was done by creating TCP traffic with *iperf*. The links' bandwidth among the routers was deliberately configured to *10Mbps* so that there would be no interference in the results, concerning processing bottlenecks of routers and hosts. The links among routers and hosts were configured with the same bandwidth *(10 Mbps)*. To each file transference, 10 executions were performed and the average values were collected.

In the first scenario, presented in Figure 6, no background traffic was inserted in the links. As we can observe in the figure, the FCT is smaller when the application controls the opening of MPTCP sublfows. As an example, the FCT for the 600MB file when the MPTCP-aware application is used is 3 minutes and 56 seconds. The FCT for the same file when the application does not use the MPTCP is 8 minutes and 56 seconds, a difference of 125%. This same analysis can be done for 100MB, 300MB and 1GB files.

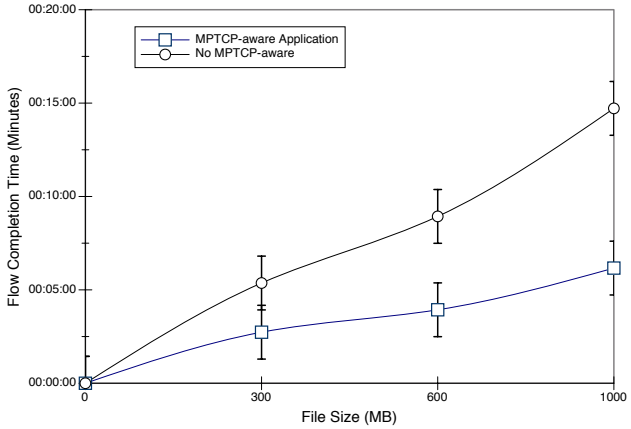

Fig. 6. MPTCP-aware application X no MPTCP-aware (no background traffic).

In the second scenario, with the intent of verifying the behavior of MPTCP traffic upon sharing bandwidth of a link with other flows, a TCP traffic was generated, enough to occupy 50% of the available bandwidth in each of the six paths used in the test. Furthermore, the same procedure described in the first scenario was utilized to transfer the files. The collected results can be verified through the graphic presented in Figure 7, exhibiting a significant aware application FCT benefit which transfered the 600MB file in 4 minutes and 44 seconds while the application without MPTCP required 16 minutes and 46 seconds. There is an even better benefit for the file of 1GB

which accounted 7 minutes and 57 seconds for the aware application against 24 minutes and 46 seconds for the one without MPTCP-aware support.

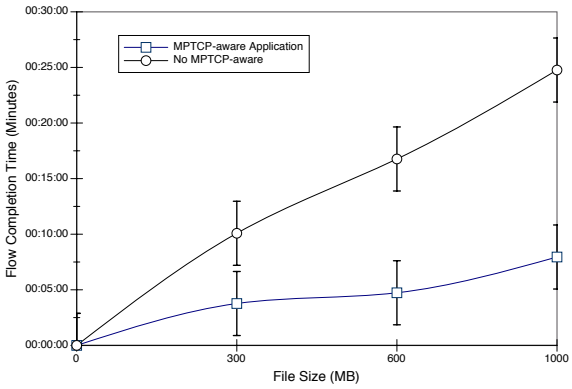

Fig. 7. MPTCP-aware application X no MPTCP-aware (50% bandwidth occupation).

By doing this evaluation, we verified the advantages of utilizing the *Multipath TCP (MPTCP)* protocol as an alternative for the creation of aware applications, having as a use case the treatment of elephant flows. The API implemented in this work can be largely used by other applications which desire to control the opening and closing of MPTCP subflows. Examples include load balancers, map-reduce applications, multipathing for flow resiliency and several others.

IV. Conclusion

In this article, we presented an implementation of the RFC 6897 as an alternative for empowering the applications to become aware. The idea behind this is to give to the applications the capability of managing the MPTCP functions such as opening and closing subflows based on the application need.

Therefore, it was fundamental to develop an implementation of the *RFC 6897* on *kernel* level so that applications use the *MPTCP* protocol, specifying when to add and remove flows, as well as turning the protocol on and off. With this implementation, the developers and *MPTCP* protocol users may find paths to extend their applications whereas using the existent resources in the protocol with the intent of obtaining the best results upon its utilization.

The next step of this work is to implement the option for closing the subflows (*TCP_MULTIPATH_REMOVE*) so that the application cannot only add new subflows but also close as necessary.

Acknowledgment

We thank CNPq and FAPESP for their financial support.

References

[1] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*. ACM, 1992, pp. 107–114.

[2] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, "Socket intents: Leveraging application awareness for multi-access connectivity," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 295–300.

[3] M. Scharf and A. Ford, "Multipath TCP (MPTCP) application interface considerations," RFC 6897, March, Tech. Rep., 2013.

[4] C. Paasch and O. Bonaventure, "Multipath TCP," *Communications of the ACM*, vol. 57, no. 4, pp. 51–57, 2014.

[5] C. Raiciu, M. Handley, and O. Bonaventure, "TCP extensions for multipath operation with multiple addresses," RFC 6824, Jan, Tech. Rep., 2013.

[6] A. Ford, C. Raiciu, M. Handley, O. Bonaventure *et al.*, "TCP extensions for multipath operation with multiple addresses," *IETF MPTCP proposal-http://tools. ietf. org/id/draft-ford-mptcp-multiaddressed-03. txt*, 2009.

[7] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–33.

[10] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi, "Elephanttrap: A low cost device for identifying large flows," in *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*. IEEE, 2007, pp. 99–108.

[11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[12] H. Xu and B. Li, "Tinyflow: Breaking elephants down into mice in data center networks."

[13] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 1629–1637.

[14] M. Casado, "Of mice and elephants," Nov. 2013. [Online]. Available: http://http://networkheresy.com/2013/11/01/of-mice-and-elephants/

[15] M. Sandri, A. Silva, L. Rocha, and F. Verdi, "On the Benefits of Using Multipath TCP and Openflow in Shared Bottlenecks," in *The 29th IEEE International Conference on Advanced Information Networking and Applications (AINA'15)*, March 2015.

[16] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.