Load Balancing between Controllers

Dhinakaran Vinayagamurthy Jaiganesh Balasundaram

Department of Computer Science
University of Toronto

December 14, 2012

Contents

1 Introduction

Software Defined networking (SDN) is a network architecture that decouples control plane from data plane. This provides numerous benefits such as a centralized controller that can be used to manage the entire network via user written control applications and simpler switches which only need to forward packets based on the flow table entries. This abstracts the underlying network infrastructure to the controller applications. The functions of a controller include, but are not limited to, handling new flow requests by setting up the new flow entries in the flow tables of the corresponding switches, updating the routing table and maintaining

a network wide state. Tootoonchian et al. in [TGG+12] provided a study on the performance of a few publicly available SDN controllers (while handling data path requests), including a multi threaded version of NOX which they had implemented. This was done by calculating two important parameters namely the *controller throughput* (the number of data path requests it can handle per second) and *controller response time* (the average time the controller takes to set up data paths) in different settings. They pointed out that the two aforementioned parameters (especially the response time of the controller) are the primary factors to decide if additional controllers are needed in a network.

When the number of switches in a network grows, having a single controller might not be a smart idea, as this will lead to larger flow setup latencies for some flows due to high workload on the single controller. This led to the development of a distributed control plane, involving many controllers with each controller serving a number of switches in the network. There were also issues of scalability and reliability in a one controller framework. Onix [KCG+10] provided a solution to these problems by presenting a distributed control plane framework. Though the controllers were physically distributed, they had a state synchronisation mechanism to have a *centralized logical view*. This was achieved by using a data structure called the *Network Information Base* (NIB) to track the network state and distributing it between multiple running instances. Frameworks such as Hyperflow [TG10], Kandoo [HYG12] provided alternate approaches to this problem. Hyperflow provided a distributed, event based control plane. It uses a *publish/subscribe mechanism* to maintain a logically centralized view. Kandoo provided a *two level hierarchy* for controllers, thereby distinguishing local control application from non local control applications (which were run on a root controller that had a network-wide state). Each framework has its own advantages and limitations. But in all the cases, maintaining a consistent logically centralized view was critical to the performance of the system.

Though there is this notion of *logical centralization*, the network state must still be physically distributed in order to achieve reliability and scalability goals and this can affect the performance of the control applications that have a logically centralized view but are unaware of the underlying physical state distribution. [LWH+12] discusses this issue and identifies two important state distribution trade-offs namely *performance of control application vs. the state distribution overhead* and *robustness to inconsistency vs. application logic complexity*. They simulate these trade offs on an existing load balancing control application and conclude that the inconsistency in control state reduces the performance of the logically centralized control applications that are unaware of the underlying physical state distribution.

Thus, Software Defined Networking being a budding technology, poses a multitude of interesting questions and directions. Distributed control plane framework in particular is of enormous interest to us, since it gives a new variety of questions to the networking community. We try to address one such interesting question in a distributed control plane architecture.

1.1 Motivation

We believe that in a distributed control plane framework (with appropriate state synchronisation mechanism), not all controllers are busy at any given instant of time. This idea arises from the fact that if all controllers are busy at any given point of time, it is only logical that the network manager would have installed a new controller in the network. (Busyness of a controller can be the measure of how close the current average response time is to the target response time set by the network manager.) Also when the increase in workload to a controller is only temporary, it does not make sense to install a new controller only for that time period, if other controllers can share the increase in load. So we need a mechanism to transfer a part of load from a 'heavy' controller (the busy controller) to 'light' controller(s) (the not so busy ones), so that all the controllers work within their set target.

1.2 Problem statement

Assume a distributed control plane framework with each controller assigned a specific target response time by the network manager. Consider a scenario where the current average response time of a controller is going to overshoot its target response time due to reasons like the load of the controller becomes heavier than

normal or the target response time has been set to a new lower value by the network manager (Note that these changes are only temporary). The aim is to design a protocol for transferring load between controllers such that their target response times are achieved.

The rest of this report is organised as follows. We define certain terms that we use in our protocol in Section 2. We then explain the framework on which our protocol is based in Section 3 along with various assumptions made. Our protocol is explained in detail in Section 4 and its analysis is provided in Section 5. Some future work that needs to be done in this direction is given in Section 6. Some work regarding the old version of our protocol, which is significantly different from this version and which was done during till the intermediate report submission, is provided in Appendix A.

2 Definitions

1. Average response time t: The average time taken by the controller to serve one request (for example, a flow setup request). Note that this includes the average waiting time of that request in the queue.

2. Target response time t_t: This is set by the network manager for each controller. This is like a guarantee provided by the network manager that the controller would serve the requests with an average response time $\leq t_t$

3. Busyness of a controller: For a controller i,
$Busyness_i \triangleq$ Target response time$_i$ - Current average response time$_i$

4. Distribution D: This represents a *frequency* distribution of number of requests of different types in a unit time interval. The requests are categorized by their service times. The total service time of a distribution D_i is defined as the total time taken to serve the requests in the distribution and is denoted by T_{D_i}. Note that we use T to denote the service-only time i.e time taken by the controller to serve the request after it reaches the controller (excluding the waiting time in the queue).

5. *APP*: This is the application in the controller which runs our protocol.

3 Framework

3.1 A distributed control plane framework

We assume a distributed control plane framework like hyperflow as in Figure 1. Each domain has a controller and a set of switches in it. We assume that each switch talks to exactly one controller at any instant of time. The set of switches talking to the same controller along with the controller they talk to, constitutes a domain. For every domain i, C_i represents the controller in that domain (note that each domain has exactly one controller according to our assumption) and S_{ij} represents the j^{th} switch of the i^{th} domain. State synchronisation is based on a *publish/subscribe* mechanism. An event logger application running on each controller captures the events that affect the control state, serializes it and publishes the events of general interest on the data channel. The controllers advertise themselves on the control channel. Each controller has a separate channel to which it subscribes. Each controller can write in every other controller's channel. For example, if Controller C_i wants to send a message to controller C_j, C_i can write it in $Ctrl_{C_j}$. Since C_j is subscribed to $Ctrl_{C_j}$, it can read the messages from it.

3.1.1 Three Regions

The current average response time (t) of the controller falls into one of the following three regions based on its proximity to the target response time.

- **Green** When t is in this region, it is highly unlikely that it will move to the red region in the near future. We can call it as the safe region.

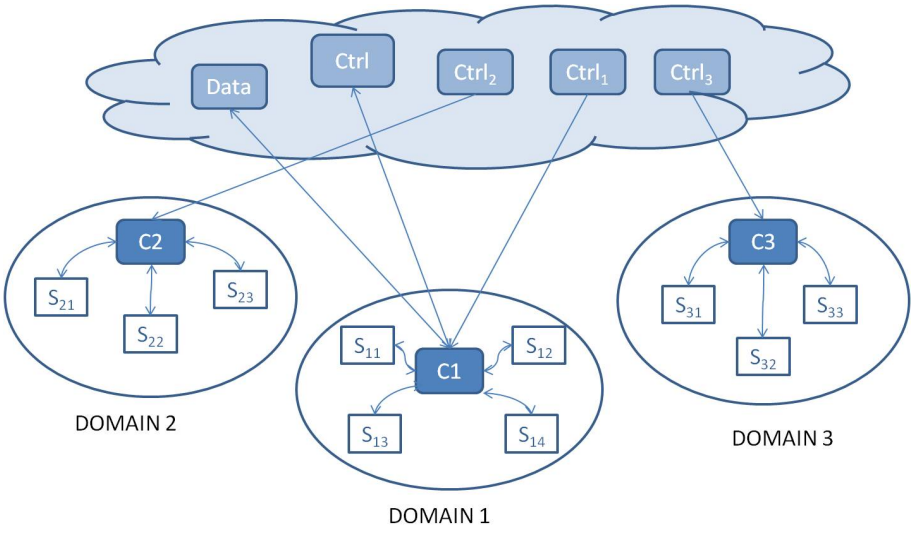

Figure 1: A distributed control plane architecture simliar to hyperflow

- **Yellow** In a certain time interval δ_p, with high probability, t would move to the red region. δ_p can be viewed as the time taken for our whole protocol to complete. t_y represents the average response time just entering the yellow region.
- **Red** The average response time in this region is greater than or equal to the target response time.

These regions are defined based on the statistics collected by the applications in the controller.

3.1.2 An application maintaining distributions

We assume that in each controller, there exists an application that maintains the

- Arriving distributions of requests
  - The arrival distribution from each switch (overall average & average of recent few time intervals)
  - The aggregate arrival distribution from all switches (again, overall & recent)
- Service distribution D_S

Our protocol has a strong assumption on the arrival distribution that the arrival rate for the controller does not increase after the average response time t reaches a certain 'high' value (i.e after it reaches the $Yellow$ region). Also, all controllers categorise the requests in a related (if not in the exactly same) manner i.e if one controller categorises the requests into a number of types according to time taken to serve that request and sends the frequency distribution, all other controllers should be able to parse the input distribution and find a frequency distribution which suits them. This may be ignored by assuming all the controllers work in a similar manner.

3.1.3 Simulator TSIM

We assume that every controller in the network has a black box called TSIM. A simulator TSIM as in Figure 2, which on inputting a set of distributions, outputs the average response time for servicing the aggregate of its input and the current workload of the controller.

TSIM provides a mechanism for determining the average response time for a certain new input workload without actually servicing it. In our protocol, it assists the 'light' controller in checking if it can accept the load from a 'heavy' controller without its average response time going too close to its target response time.

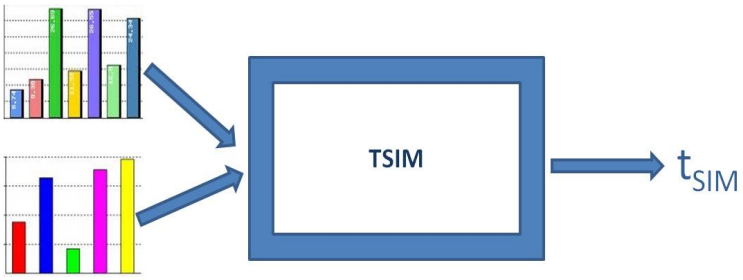

Figure 2: The Black-Box TSIM

4 Protocol

4.1 NH-Find

We identify a set N (which represents the 'nearby' or 'neighbourhood' controllers) for each domain i. The set of 'Nearby' controllers (for domain 1) is given by

$$N = \{C_i \,|\, \exists s_{1j}, [h(C_i, s_{1j}) = min(\{h(C_k, s_{1j}) | \forall C_k \in T - \{C_1\}\})]\}$$

- $h(a, b)$ is the number of hops from a to b
- T is the set of all controllers in the network
- S_{1j} is some switch in Domain 1

By the term 'nearby' controller for a switch, say S_{ij} of domain i, we mean the 'next closest' controller to it (Note that 'next closest' does not always mean the second closest, rather it means closest among all controllers in the network except C_i, the controller to which S_{ij} currently talks to. After the completion of the first run of our protocol, many switches will have these two to be different). Every switch belonging to domain i has one such controller and the set N consists of all such controllers. (Note that this set for each domain can be calculated efficiently from the network graph using a shortest path algorithm).

For example, consider the architecture in Figure 3. Let C1 of domain 1 be the 'heavy' controller which wishes to transfer a part of its load (by transferring a subset the switches in its domain) to meet its target response time. The 'nearby' controller of each switch of domain 1 is represented by a matching color in the Figure 4. For example, controller C2 of domain 2, is the closest to switch S_{11} among all the controllers in the network except the controller in the same domain as S_{11} (which is $C1$). So the 'nearby' controller set for domain 1 is $N = \{C2, C3, C4, C5\}$.

4.2 NC-S List

We maintain a list L_i for every $N_i \in N$. L_i has $\{s^{i1}, \ldots, s^{ip_i}\}$ sorted in the ascending order of $h(s^{ij}, N_i)$ where p_i initially is the total number of switches in the domain i. Hence, the switches close to N_i are higher in the list L_i.

If the network manager wants to avoid switches from being migrated to 'far' controllers (the controllers which belong to N but, in practice, very far from a subset of switches in the domain), he can obtain an upper bound on the distance between the switch and N_i and remove any switch from L_i whose $h(s^{ij}, N_i)$ exceeds this bound. That is, he ensures that $\forall j, h(s^{ij}, N_i) \leq g \cdot h(s^{i1}, N_i)$, for some constant g determined by him.

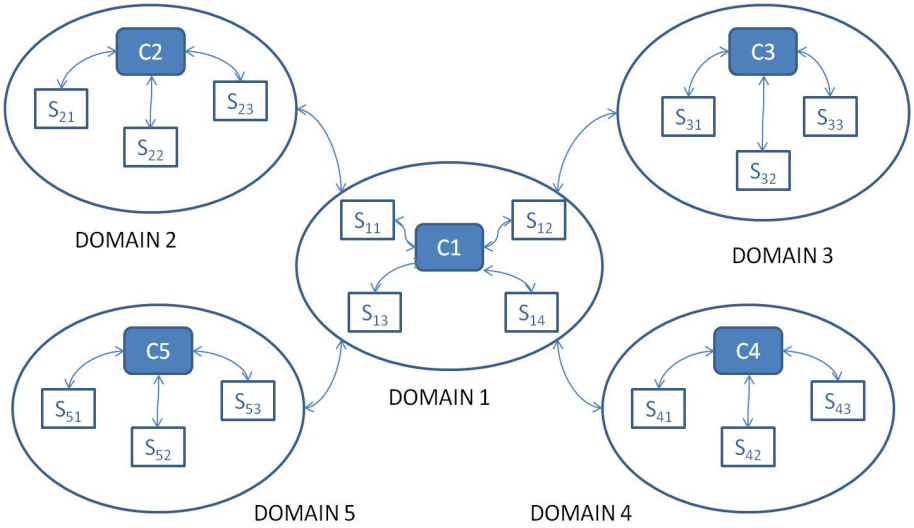

Figure 3: An example of a distributed control plane with a 'heavy' controller C_1

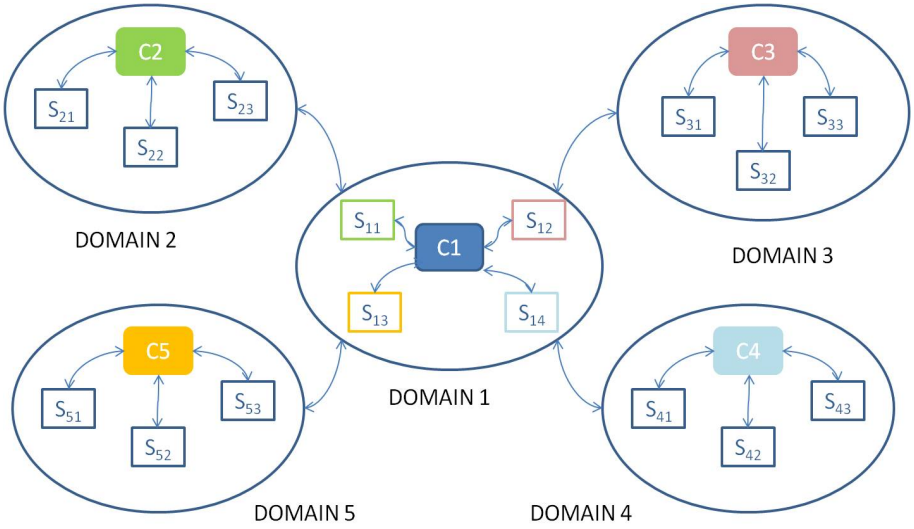

Figure 4: Nearby controllers for domain 1

4.3 Polling phase

This is the phase where the actual protocol begins. Assume C_1 to be the 'heavy' controller. This phase starts when t_1 enters the $Yellow$ region (i.e when t_1 becomes t_{y_1}) where, within the time interval δ_p, t_1 will reach the *Red* region (i.e $\approx t_{t_1}$). C_1 tries to find the appropriate controllers for offloading a part of its load, so that t_1 remains well within the target response time t_{t_1}. The application APP in C_1 follows Algorithm 1 to start this offloading process.

Algorithm 1 Polling phase - Part I

$T_d \leftarrow T_{D_A} - T_{D_S}$, where T_{D_S} = Unit time interval
for all L_i **do**
  Find $M = \{x | T_x < T_d\}$, where T_x is the total service time for aggregation of the arrival distributions from the first x switches in L_i.
  $m_i = 1 + \max(M)$
  Send $[\{D_{S^{i1}}, D_{S^{i2}}, \ldots, D_{S^{im_i}}\}, t_{t_1}]$ to the controller N_i
end for

T_d can be viewed as the time taken by the controller to serve a part of its load, which is more than the maximum that it could serve in a unit time interval. If we could remove the load which can be serviced in T_d, then the arrival rate becomes less than the service rate and hence the average response time would not increase further. And since we could complete this whole transfer process within time interval δ_p, we avoid t from reaching *Red* region. Hence, at this stage, we find the set of 'closest' switches to N_i, for each N_i, such that the aggregation of the load from those switches can be serviced in T_d. We send such a subset of switches to each N_i.

At the receiving controller N_i

When a controller receives a set of distributions and target response time of C_1, APP in N_i first checks if t_i is in yellow region. If so, then N_i is undergoing a similar 'offloading' process on its own. Hence, APP just returns $\perp$.

Otherwise, APP should find the number of switches that it can accept from C_1 considering its current load. In other words, APP should determine the number of switches y_i, such that when the control of those y_i switches are transferred to N_i, its average response time t_i does not reach the yellow region t_{y_i}. APP follows Algorithm 2.

Algorithm 2 Polling phase - Part II

$t_{agg_i} \leftarrow TSIM(D_{S^{i1}}, D_{S^{i2}}, \ldots, D_{S^{im_i}})$
$q_i \leftarrow m_i + 1$
$t_{y_i} \leftarrow$ value of t_i, when t_i enters the yellow region.
repeat
  $q_i \leftarrow q_i - 1$
  $t_{agg_i} \leftarrow TSIM(D_{S^{i1}}, \ldots, D_{S^{iq_i}})$
until $t_{agg_i} < t_{y_i}$ ▷ This loop can be done more efficiently by a modified binary search algorithm [Algorithm 3] for large values of m_i.
Send $[t_{agg_i}, q_i]$ to C_1.

Issues

In this process, we do not make sure that the switches moved to N_i get their service within time t_{t_1}, because if $t_{y_i} > t_{t_1}$ then there is a possibility of those switches not getting their service within (average response time

Algorithm 3 Modified Binary Search algorithm

Create a list L_s such that the element at position i in the list is the aggregation of the first i distributions received by N_i

$L_s \leftarrow 0||L_s||\infty$ ▷ L_s obtained eventually, has $m(+2)$ distributions (aggregations of distributions) in a non-decreasing order, since the distributions $D_{S_{ij}}, \forall j \in \{1, m_i\}$ takes non-negative values (i.e the total service time of the distributions keep on increasing as we go from the first to last element in the list)

function BINARY SEARCH(L_s,t_{y_i},$min = 0$,$max = m + 1$)

  while $max \geq min$ **do**

    $mid \leftarrow mid(max, min)$

    $T_{mid} = TSIM(L_s[mid])$ ▷ $TSIM$ values calculated can be stored in an array, to avoid recalculations.

    if $T_{mid} \leq t_{y_i}$ **then**

      if $T_{mid+1} > t_{y_i}$ **then**

        return $[T_{mid}, mid]$

      end if

      $min \leftarrow mid + 1$

    else if $T_{mid} > t_{y_i}$ **then** $max \leftarrow mid - 1$

    end if

  end while

end function

of) t_{t_1}. We can solve this to an extent by checking if $t_{agg_i} < min(t_{t_1}, t_{y_i})$. This check will ensure assured service for the 'offloaded' switches, for the time being, as long as the arrival from the already existing switches in the domain of N_i does not increase much. But, if the load due to them increases, then t_i can go higher upto t_{t_i}. So, to avoid this, one can assign higher priority for the requests from the newly arrived switches in N_i, so that they are serviced within t_{t_1}.

4.4 Setup phase

This phase takes place back at C_1. APP in C_1 ignores the N_is which have sent $\perp$. From each other neighbourhood controller N_i, C_1 gets (approximately) the maximum amount of load that N_i can take from C_1, along with the average response time for processing that load. Now, APP in C_1 tries to find the set of controllers for offloading switches so that its average response t_1 does not increase further.

APP first checks if any N_i can accept all the switches that it has been requested. If there exists any such N_i, APP finds the one with minimum t_{agg}. Else, the total load to be 'offloaded' has to be distributed among different N_is, according to the q_is received. APP follows Algorithm 4 in doing these steps. It is more meaningful for T_d to be calculated again, during this algorithm, rather than using the value that is calculated during the Algorithm 1. Here, we have assumed that the synchronisation of states between N_i and C_1, especially for processes which are interrupted due to the migration of the switches, occurs within the time interval δ_s. We include δ_s in δ_p which is the total time taken by our protocol to complete.

Issues

- Consider the scenario where a switch, say S_r, occurs in two lists, say L_{r_1} and L_{r_2}, with L_{r_1} occurring higher in the sorted list. If S_r is reconfigured to the domain of N_{r_1} (the N_i corresponding to L_{r_1}) and T_{rem} does not become ≤ 0 before L_{r_2} is accessed.
  - S_r cannot be reconfigured to the domain of N_{r_2}, though it can accept a switch with an input load of (atmost) that of S_r.
  - One way of addressing this issue can be to reconfigure the next (non re-configured) switch or set of switches in the list L_{r_2} such that T for the aggregation of load from these switch(es) is less than or equal to that of S_r.

Algorithm 4 Setup phase

Require: Get $[t_{agg_i}, q_i]$ tuples from $N_i \in NS$, where $NS \subseteq N$ is the set of neighbourhood controllers which does not send $\perp$

```
Pick F = {N_i | q_i = m_i}
if F ≠ ∅ then
    t_min ← min({t_agg_j | N_j ∈ F})
    Synchronise states of C_1 and the N_i, corresponding to t_min
    Reconfigure the first q_i switches of L_i to communicate to N_i
else
    for all N_i ∈ NS do
        Find T_i ← total service time for aggregation of {D_{S^{i1}}, ..., D_{S^{iq_i}}}
    end for
    Sort NS in the decreasing order of T_i    ▷ Now, NS contains N_1, N_2, ... in the decreasing order of T_i
    T_rem ← T_d
    i ← 0
    repeat
        i ← i + 1
        Synchronise states of C_1 and N_i
        Reconfigure the first q_i switches of L_i to communicate to NS[i]
        T_rem ← T_rem − T_i
    until T_rem ≤ 0 or i ≟ |NS|
end if
```

5 Analysis

Our protocol assumes an arrival distribution such that arrival rate for the controller does not increase after the average response time t reaches the $Yellow$ region. In other words, once the arrival rate increases to a certain 'high' value, it should not increase thereafter. A value is rated as 'high' depending on the service rate of the controller.

But we do not use any other property of the arrival or service distribution in our protocol. This implies that if the protocol works well for a particular distribution, it should work well for any kind of distribution, satisfying the above condition.

Hence, we briefly instantiate our protocol with a very special type of arrival distribution, where each type of requests, from each of the switches follow poisson distribution with varying rates.

Claim

If the average response time increases beyond the target, then it contradicts the assumption that all the neighbourhood controllers together are free enough to accept the load to be offloaded by C_1.

Proof

Let N be the set of neighbourhood controllers as described in the protocol and let each $N_i \in N$ have a list L_i. Let λ_{kij} be the arrival rate for type k request sent by the jth switch in L_i (assuming that requests are categorised into types $1, 2, \ldots$). We consider all the assumptions made in our protocol to remain intact during this analysis. Note that if $\sum_{k,j} \lambda_{kij} > \mu$ throughout, then the system will be unstable. We consider the case where there is increase in some of λ_{kij}s for a significant period of time such that $\sum_{k,j} \lambda_{kij} > \mu$ during that period. The difference should be offloaded to some other controller(s).

Polling phase

According to Algorithm 1, APP finds $T_{D_A} = \sum_k Z_k st_k$, where Z_k is the total number of requests of type k that arrives in unit time interval, that is determined by taking a list, say L_i, which contains all the switches in the domain and calculating $Z_k = \sum_j \lambda_{kij}$. st_k is the service time for a type k request. APP then finds $T_d = T_{D_A} - T_{D_S}$, where T_{D_S} is the unit time interval.

Then, for every list L_i, APP finds the minimum m_i such that

$$\sum_{\forall k}(st_k \sum_{j=1}^{m_i} \lambda_{kij}) > T_d$$

Then, for each N_i, $\langle D_{S^{i1}}, \ldots D_{S^{im_i}} \rangle$ is sent.

When N_i receives $\langle D_{S^{i1}}, \ldots D_{S^{im_i}} \rangle$, APP of N_i interacts with $TSIM$ as in Algorithm 2 and outputs $\langle q_i, t_{agg_i} \rangle$, where $q_i \leq m_i$ such that $t_{agg_i} = TSIM(D_{S^{i1}}, \ldots D_{S^{iq_i}}) < t_{y_i}$.

Setup phase

For all $N_i \in NS$, APP finds $T_i = \sum_{\forall k}(st_k \sum_{j=1}^{y_i} \lambda_{kij})$ and sorts NS in the descending order of T_i. Note that $NS \subseteq N$ is the set of controllers which do not send $\perp$. Then APP works as in Algorithm 4.

In this process, we can see that each neighbourhood controller receives set of switches, which when migrated away from C_1, stops the queue of C_1 from increasing further. Clearly speaking, after the migration, for each unit time interval, the queue size decreases by the number of requests which can be serviced in time $-T_{rem}$ (which is ≥ 0, since T_{rem} should become ≤ 0 at the end of the protocol if all the load that are needed to be offloaded, are indeed offloaded). Also, we have assumed the time interval δ_p, for t to increase from t_{y_1} to t_{t_1}, such that our protocol finishes within this δ_p.

Still, if at the end of the protocol, the average response time t_1 becomes greater than the target t_{t_1}, the only possibility (assuming all our other assumptions remain intact) is that t_1 has been increasing even after the protocol is over. This implies that $T_{rem} > 0$ at the end of Algorithm 4 and the loop in Algorithm 4 has stopped because of the condition $i \stackrel{?}{=} |NS|$ i.e the neighbourhood controllers together are not free enough to take the load that is needed to be offloaded from C_1, which contradicts our original assumption. Hence a new controller has to be installed in that region.

6 Future Work

- At the end of our protocol, the average response time t_1 of C_1 is well within the target response time t_{t_1}. But there is a very high probability for t_1 to remain in the $Yellow$ region, which would kick start the process again. To avoid this, APP can be programmed to start the process again only when t_i starts increasing again, since t_i would decrease momentarily due to the transfer of the switches.

  But then if t_1 starts increasing before reaching the $Green$ region, time may be insufficient for the protocol to complete before t_i reaches Red region. To avoid this, during the *Polling phase* C_1 may send distributions of $m_i + \epsilon_i$, for a very small ϵ_i. Or, continue transferring switches in the *Setup phase* till $T_{rem} \leq -\delta_T T_d$, for some fraction δ_T. Both these steps require that the neighbouring controllers are more 'free' than we assumed initially, which seems plausible, considering the fact that the network operators keep the 'utilisation' of a controller well below maximum for 'normal anticipated' workload.

  Here, we could not determine ϵ and δ since we could not model, what we claim to be the *three* regions for the average response time to exist. This follows from the problem of modelling the change in average response time of the controller with respect to the other controller parameters.

- We have just tried to propose an algorithm to achieve the target in all controllers and hence, our protocol does not find the best possible way to do this offloading. One better algorithm can determine the fraction of load to be transferred to a neighbourhood controller based on its percentage of 'freeness'.

7 Conclusion

Thus, we have proposed a protocol that avoids installation of a new controller for temporary increase in workload or temporary reduction in the target response time, which may cause the controller not to respond within its target, if the controllers in the neighbourhood are free enough to *share* the *extra* load of the *heavy* controller. All the controllers eventually work within their target response times.

We have presented an analysis of our protocol which follows the *contrapositive* approach in arguing that if the protocol does not terminate with success, that is only because the neighbourhood controllers are not free enough to share the load that is needed to be offloaded, provided all the other assumptions remain intact.

Though the primary aim of the protocol is achieved, there are several issues that it cannot address in certain situations. The main reason for this being the lack of a mathematical model which could determine the rate of increase in the average response time of a controller with regard to its various parameters.

Acknowledgements

We sincerely thank Amin Tootoonchian for very helpful discussions during various stages of the project. We also thank Prof. Yashar Ganjali, for his wonderful course and for spreading a part of his excitement on SDN on us.

References

[HYG12] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[KCG+10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[LWH+12] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[TG10] Amin Tootoonchian and Yashar Ganjali. Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[TGG+12] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

A Old Protocol

This protocol attempts to reduce the load of controller nearing its target response time, by transferring the packets in its queue to appropriate nearby controllers. Note that this differs significantly from our main protocol in the sense that our main protocol transfers a subset of switches to the nearby controllers, whereas this protocol only transfers the set of packets from its queue. A packet or a set of packets contain the request made by a switch.

Assumption 1.

There exists a channel between every pair of controllers in a distributed control plane.

There are three main steps involved in transferring the load from one controller to another.

A.1 Polling phase

The first step is the *Polling* process. This is the first step in finding an appropriate *light* controller. We assume that the controller has RTTs estimated for all the controllers connected to it.

When the average response time of a controller reaches a certain threshold (say the yellow region as in the main protocol), it sends the *distribution* of its transferable packets to all the controllers within its range. These may be the set of controllers with which the *heavy* controller shares its states, since these controllers bypass one of the major bottlenecks in load balancing i.e. sharing and synchronisation of the states between controllers.

At the receiving controller: When a controller receives a request from a *heavy* controller, it should decide whether it could meet its target response time, even after accepting the load from the *heavy* controller.

The average response time given by $TSIM$ is sent to the *heavy* controller, if the new average response time avg' is less than its target, else the request is ignored.

Assumption 2. We assume that a controller receives atmost one request or (arbitrarily) considers atmost one request, at an instant of time.

If a controller chooses to answer the best combination of $n > 1$ requests at an instant, there are several factors that could come into play like

- Time taken for selecting the best combination of n requests (2^n combinations), such that the target response time is satisfied.
- Even if an optimal solution is obtained, the solution is optimal only with respect to this particular controller. The requirements of other controllers are not taken into consideration, which may result in some controllers in the chosen combination reject the offer from this controller.

A.2 Setup phase

The *heavy* controller now has (RTT, average response time) pairs for all the controllers. It finds the best available option as follows.

- The delay due to transfer of packets to the *light* controller is calculated as follows.

$$t_r = \frac{n}{k\beta}\frac{RTT}{2}$$

  where k is the fraction of the channel bandwidth β between the two controllers that is free at that instant. The factor 2 appears since there is no need for any reply to be sent for the completion of the protocol. n is the total number of packets that are to be transferred.

- The total service time in the *light* controller for all the packets transferred is $t_s = n \cdot avg'$

- The controller which corresponds to $min(t_r + t_s)$ or $min(\frac{RTT}{2k\beta} + avg')$ (since n may not be predicted at this point in the protocol) is fixed as *light* controller for this iteration. If there is a tie between two or more controllers, then lower avg' can be used to break the tie (with the assumptions that $t\beta$ value remains almost same for all the channels and RTT is not high. This is reasonable since if RTT if high, then it would obviate this load transfer process).

A.3 Transfer phase

After an appropriate *light* controller is determined, the *heavy* controller has to find a start packet n_1 i.e the first packet in the queue such that the average response time for n_1 packets in the queue is approximately greater than the target response time.

$$avg_{n_1} > target$$

(By approximate, we intend to say that there is no need for an algorithm to find an exact start packet n_1 such that the condition does not hold good for $n_1 - 1$. Returning the largest approximate solution for the algorithm that is less than n_1 would be sufficient.)

From the previous assumptions that there exist atleast one controller that is operating well below its target response time,

$$avg' + \frac{RTT}{2k\beta} < avg_{n_1}$$

If there exists such n_1, then the weighted arithmetic mean of the response time of the packets transferred to the other controller and the response time of the packets processed in the *heavy* controller is less than or equal to the target response time. If n_2 is the end packet that is transferred to the *light* controller from the queue of the *heavy* controller, then

$$\forall n, \frac{avg' \cdot (n_2 - n_1) + avg \cdot (n - (n_2 - n_1))}{n} < target$$

where n is the number of packets in the queue at any instant.

A.4 Issues

We have a number of issues when balancing the load of a controller using this protocol.

- It is not very straightforward to determine when to stop transferring the packets to the light controller.
- This protocol does not guarantee in order processing of packets and this might be a cause of concern in many applications

We can avoid these problems, if we migrate a subset of switches to the light controllers instead. In this case, we do not need to worry about when to stop transferring the packets because the switches now go under the control of the light controller and will remain under its control (until maybe the light controller now becomes heavy and decides to migrate this switch at the end of its protocol). Also the packets would be processed in order since only switches are migrated, not packets.