A Distributed Energy Saving Approach for Ethernet Switches in Data Centers

Weisheng Si
School of Computing, Engineering, and Mathematics
University of Western Sydney
Sydney, Australia
w.si@uws.edu.au

Javid Taheri and Albert Zomaya
School of Information Technologies
University of Sydney
Sydney, Australia
firstname.lastname@sydney.edu.au

Abstract—With popularity of data centers, energy efficiency of Ethernet switches in them is becoming a critical issue. Most existing energy saving approaches use a centralized methodology that assumes global knowledge of data center networks. Though these approaches can achieve nearly optimal energy saving for static traffic patterns, they are not suitable when the traffic patterns can change rapidly or the data centers have a large size. To overcome these limitations, this paper proposes a novel distributed approach called eAware that dynamically idles a port or a switch to save energy by examining the queue lengths and utilizations at switch ports. Through extensive simulations in ns-2, we compare eAware with an existing energy oblivious approach, showing that eAware can save 30%-50% on the total energy consumption by switches in data centers, and only increases the average end-to-end delay of packets by 3%-20% and the packet loss ratio by 0%-0.9%.

Keywords – Energy efficiency; Ethernet switches; data centers

I. INTRODUCTION

Data centers (DCs) multiplied in recent years, and contribute a significant portion of global electricity consumption [1]. DCs mainly consist of servers and Ethernet switches (shortened as switches hereafter), with the former consuming about 60% and the latter consuming about 10%-20% of the total energy [2]. Consequently, the current energy-saving approaches for DCs can be classified into three categories according to the equipment considered: (1) energy saving for servers, (2) energy saving for switches, and (3) energy saving for servers and switches together. This paper falls in the second category.

Though the majority of energy is consumed by the servers in DCs, the energy efficiency of switches becomes a critical issue due to the following factors. First, the percentage of energy consumed by switches will grow considerably in the near future, since the demand on bandwidth in DCs is increasing rapidly. Second, to accommodate peak traffic load, the network topologies of DCs have to employ a rich connectivity of switches that can provide multiple paths. For instance, in the typical fat tree [3] topology (see Figure 1), a pair of servers from different pods have a large number ($k^2/4$) of identical-length shortest paths available. Note that here $k$ is the number of pods in the fat tree and this notation is followed in the rest of this paper. Third, the peak-traffic time in the DCs is much shorter than the non-peak time. For instance, the traffic load in DCs at night time is generally very low [2]. To sum up, there is great room to save energy for the switches in DCs.

Figure 1. The fat tree topology with $k=4$

A. Related Work

Several works [4-10] have already appeared to save energy for the switches in DCs or the switches/routers in the Internet. Note that the switches in DCs also have the functionality of routers. Though DCs and Internet are different environments, the approaches in [4-10] share similar methods, so we summarize them together here.

First of all, there are altogether three hardware features exploited by these approaches to save energy:

- If a switch/router is idle entirely (including the chassis, linecards, and all ports on it), it consumes much less energy. Note that the premise for the entire switch/router to idle is that all ports on it are idle.
- When a port on a switch/router is idle, the port also consumes much less energy than when it is active.
- A port working in a lower bit rate consumes less energy than a port working in a higher bit rate. For instance, it is shown that a 100-Mbps Ethernet port only consumes about one third of the energy consumed by a 1-Gbps port [11]. The technique of saving energy by changing the bit rate of a port is called Adaptive Link Rate (ALR) [7].

Note that the remaining 20%-30% energy is consumed by the cooling system, power distribution utilities in DCs, etc.
With these hardware features, different software (i.e., algorithmic) approaches are proposed in [4-10] to use some or all of these hardware features to save energy. Specifically, Gupta and Singh originated the research on energy efficiency of switches/routers by giving a position paper [4], in which they advocate putting a switch/router into sleep during low traffic time. Later, they presented a concrete approach based on the idea of coordinated sleeping for LAN switches in [5]. Nedevschi et al. [6] propose to aggregate the network traffic at edge routers into small bursts so that long intervals between packet transmissions are generated to facilitate the sleep of links. Gunaratne et al. pioneered the research of using ALR to save energy for an Ethernet link [7]. Specifically, they proposed energy saving policies based on solely queue length, or a combination of queue length and utilization, or a combination of queue length and time-out period. The work [7] inspired our work in referencing queue length and utilization to adjust port states. The differences between our work and [7] are that (1) [7] only proposed policies to save energy for a single Ethernet link, while our work proposes a distributed approach for an entire data center; (2) [7] only uses the hardware feature of ALR to save energy, while our work idles a port/switch to save energy.

Consolidating traffic carried by multiple paths/links into fewer paths/links such that redundant switches or ports can idle is considered in [8-10]. All the three approaches in [8-10] are centralized ones in that they assume there exists a central controller who knows global information including the entire network topology and the traffic matrix among the servers in DCs. The approaches in [8] and [9] are similar in that they both formulate a Integer Linear Program (ILP) to minimize the energy consumption of switches in DCs under the constraint that the traffic among different servers can still be accommodated. Since the formulated ILP problems are NP-hard, they both propose heuristic algorithms to solve the problem. The approach proposed in [10] is different from [8] and [9] in that it considers an additional feature: the virtual machines can migrate among the servers in DCs to better consolidating network traffic. This consideration represents a brand new perspective in energy saving. Again in [10], an NP-hard ILP problem is obtained and then heuristic algorithms are proposed. In short, all these centralized approaches have shown that their proposed heuristic algorithms can achieve close results to the optimal solutions. However, since the centralized approaches entail a central controller which collects the global information and then distributes the calculated results, the centralized approaches suffer from the limitations that (1) they cannot quickly respond to the traffic change in DCs and (2) they are not scalable.

B. Basic idea of our Approach

To overcome the limitations of centralized approaches, this paper proposes a novel distributed approach that dynamically idles a port/switch to save energy by examining the queue lengths and utilizations at switch ports. For convenience, we refer to our approach as eAware hereafter. The basic idea of eAware is as follows: each switch monitors the queue lengths and utilizations at its ports; if the queue length at a port exceeds certain threshold, some other ports will be activated to alleviate the long queue; if a port has a zero queue length and its utilization is below certain threshold, this port will be made idle to save energy; if all ports on a switch are idle, the entire switch will be made idle.

The reason of referencing queue length to idle/activate ports is that energy saving should not considerably impair the network performance of DCs. The most important metrics on network performance are packet end-to-end delay and packet loss ratio, both of which are closely related to queue length: (1) long queues lead to long end-to-end delays and (2) if queue length is close to buffer capacity, packet drops are prone to happen.

Moreover, only examining queue length is not enough. The utilization is also examined when putting ports into idle because of two reasons. First, a port can have a zero queue length but a high utilization at the same time (e.g., when a port transmits a network flow which has a constant bit rate slightly less than the port capacity). In this case, this port should not be made idle. Second, measuring utilization needs a period of time, which can prevent ports from oscillating between idle and active states too frequently. Especially, in the DC environment, the queue length at ports can change very quickly due to the bursty nature of network traffic.

The rest of this paper is structured as follows. Section II describes our system model. Section III details eAware. Section IV evaluates eAware and compares it with an energy oblivious approach by extensive ns-2 simulations. Finally, Section V concludes this paper.

II. SYSTEM MODEL

This section describes the power model for energy saving and the packet forwarding model at a switch.

A. Power model

In eAware, a port has two states: idle and active. Note that eAware does not exploit the hardware feature of ALR, because commercial switches supporting ALR seems not to appear in near future. On the contrary, the commercial switches supporting the idle/active states are already in market (e.g., the Cisco Catalyst 4500E switches [12]) now, since IEEE approved the Energy Efficient Ethernet as a standard (IEEE Std 802.3az) [13] in 2010. Specifically, in IEEE Std 802.3az, a new state for ports called Low Power Idle (LPI) is introduced and the mechanisms for entering and exiting LPI are defined.

In eAware, a switch also has two states: idle and active. Only when all ports on a switch are idle, the switch can be idle (i.e., the chassis and linecards on this switch can be idle...
to save more energy). Considering the above states of both ports and switches, the power consumed by a switch (denoted by $P_{sw}$) is given in the following formula:

$$P_{sw} = \begin{cases} 
P_{base\_sw} + n_{idle} \cdot P_{idle\_pt} + n_{active} \cdot P_{active\_pt} & \text{if } n_{active} > 0 \\
P_{idle\_sw} & \text{if } n_{active} = 0 
\end{cases}$$

where $P_{base\_sw}$ is the power consumed by the fixed parts of the switch such as chassis, linecard, etc.; $n_{idle}$ is the number of idle ports in this switch; $n_{active}$ is the number of active ports; $P_{idle\_pt}$ is the power consumed by an idle port; $P_{active\_pt}$ is the power consumed by an active port; and $P_{idle\_sw}$ is the power consumed by an idle switch.

It is worth noting that the actual values of $P_{base\_sw}$, $P_{idle\_pt}$, $P_{active\_pt}$, and $P_{idle\_sw}$ will change over time with the advance of power-aware hardware technologies. For the time being, in this paper, we set $P_{base\_sw}$ to 100W, $P_{idle\_pt}$ to 1W, and $P_{active\_pt}$ to 3W roughly according to the measurement study in [11]. As for $P_{idle\_sw}$, no measurement study is available since no existing switches support such an idle state yet. But we envision that such support will be available soon and can achieve a low power consumption at this state, so we tentatively set $P_{idle\_sw}$ to 10W.

Also note that there is a penalty for the port state transition between idle and active: this transition takes a short period of time, during which no packets can be transmitted and the same amount of power is consumed as the active state. This penalty is indispensable, because, otherwise, a port can change states arbitrarily without degrading performance. In this paper, this short period of state transition time is set to 1 ms according to the IEEE Std 802.3az [13].

### B. Packet forwarding model

Due to the rich network connectivity in DCs, the switches need to adopt a multi-path routing protocol such as ECMP [14], which can return multiple ports for forwarding a packet. To make eAware easy to implement and also portable, eAware does not require changes to the multi-path routing protocols used by DCs. That is, the routing protocols will be unaware of the idle or active states of ports, and just establish the routing tables by their own criteria such as shortest path, link working or broken, etc. On the other hand, eAware will maintain the idle/active states of ports and make the final decision of packet forwarding based on the routing table and the port states. Since routing messages only occur periodically, we assume they can still be transmitted/received even when a switch or a port is idle. We make this assumption because if a switch/port is completely off, there is no way to activate it again when needed.

Figure 2 illustrates the model in which eAware handles packet forwarding. Basically, when a packet arrives, eAware will look up the routing table established by the multipath routing protocol. Assume that ports $i$, $j$, $k$ are returned for this packet. Then, eAware will take the returned ports and their associated buffer queue length and utilization information as input to decide which ports among the ports $i$, $j$, $k$ should be idle/active and to which port this packet should be sent.

![Figure 2. The model in which eAware handles packet forwarding](image)

### III. DESCRIPTION OF eAwAre

#### A. Some basic concepts and mechanisms

Before detailing eAware, we first introduce the following basic concepts and mechanisms used in eAware.

**switch level:** eAware requires the network topology has a hierarchical structure. Each switch in eAware has a variable indicating its level in the hierarchy.

**port order and switch order:** we stipulate an order on the ports of a switch and switches in the same level as their order of appearance from left to right in the network topology (e.g., as shown in Figure 1). Without loss of generality, we require the network topology satisfies that if port $i$ appears left to port $j$ in a switch, port $i$ will connect to a switch appearing left to the switch that port $j$ connects to. The idea here is trying to consolidate traffic to the ports and switches in the left, such that the ports and switches in the right can idle to save energy.

**port ID and switch ID:** to indicate the above order, if a port $p$ appears left to a port $q$ on a switch, port $p$ has a smaller ID than port $q$; similarly, if a switch $s$ appears left to a switch $t$ on the same level, switch $s$ has a smaller ID than switch $t$.

**baseline spanning tree:** a spanning tree of the network topology in which the switches and ports are always active. This is to ensure that any pair of servers are connected by at least one path at any time, such that the network always responds quickly to traffic arrivals. This baseline spanning tree is established by using the leftmost ports and switches in the network as much as possible. For instance, the baseline spanning tree for the network illustrated in Figure 1 is given in Figure 3, where the shaded switches and the thick lines depict the tree. In eAware, a switch determines whether this
switch and some of its ports are on this spanning tree by looking at the routing table, its switch level, its switch ID, and its port IDs.

![Diagram](image-url)

**Figure 3.** The baseline spanning tree for the topology given in Figure 1.

**primary ports** and **secondary ports**: If a port is on the baseline spanning tree, it is a primary port; otherwise, a secondary port.

**primary switches** and **secondary switches**: If a switch is on the baseline spanning tree, it is a primary switch; otherwise, a secondary switch.

**port utilization**: the average traffic transmission rate at a port during certain period divided by the capacity of this port. Its value ranges between 0 and 1.

**utilization period**: the period for measuring the port utilization. During a period, each port accumulates the amount of traffic transmitted by this port, and calculates the utilization at the end.

**high_length**: the threshold of the queue length to trigger the increase of active ports.

**low_util**: the threshold of the port utilization to trigger the decrease of active ports.

**flow**: a flow is a sequence of packets that have identical selected packet header fields. In eAware, a flow is identified by the following five fields: source/destination IP addresses, source/destination ports, and type of protocol.

**Hash-Threshold method**: if packets in a flow use different paths to reach the destination, the packets can arrive out of order, which degrades TCP performance [14]. In eAware, the Hash-Threshold method in RFC 2992 [15] is used to prevent a flow from splitting. Given a packet with \( m \) active ports available for forwarding, this method works as follows: (1) dividing the hash function's output space evenly into \( m \) subset, with each subset corresponding to an active port; (2) calculating the hash value for this packet using its flow-identifying header fields; (3) examining which subset the hash value falls in, then forwarding the packet to the corresponding port.

**B. Details of eAware**

EAware consists of two algorithms running in parallel at a switch: **increasing ports algorithm** and **decreasing ports algorithm**.

The **increasing ports algorithm** is invoked after the routing table lookup for every packet, dealing with the increase of active ports. This algorithm is invoked for every packet because (1) only after the routing table is looked up for a packet, we know which ports can be used to forward this packet and thus which ports should be made active to alleviate long queues if they are present; (2) the queuing delay of a packet should be reduced whenever possible.

The **decreasing ports algorithm** is invoked every utilization period, dealing with the decrease of active ports. This algorithm is triggered by a periodical timer instead of packet arrivals, because a port should be able to become idle even when no packet arrives.

The initialization procedure for these two algorithms is as follows. When a switch joins the network, it will run a multipath routing protocol to build the routing table and also determine whether it is a primary switch, and if so, which ports are primary ports. Then, a primary switch will initialize its state to **active** and also initialize the states of its ports: if a port is a primary port, its state is initialized to active (otherwise, **idle**). A secondary switch will initialize its state to **idle** and the states of all its ports to **idle**.

In the following algorithm description, given port \( u \), we use \( v \) to denote the other port that shares a link with port \( u \). The following constraints are maintained by the two algorithms: (1) if port \( u \) becomes active, port \( v \) must also become active; (2) port \( u \) can idle only when port \( v \) can also idle. Note that the signaling between ports \( u \) and \( v \) to maintain these two constraints is supported by the link layer mechanisms in IEEE Std 802.3az.

The increasing ports algorithm is detailed as follows.

**Input**: (1) a packet and (2) the \( n \) ports returned for forwarding this packet from the routing table.

**Output**: (1) the port for forwarding this packet and (2) a port to activate if there should be one.

1. Assume \( m \) of these \( n \) (\( m \leq n \)) ports are currently in **active** state. The Hash-Threshold method is called to select one of these \( m \) active ports as the port (denoted by port \( f \) hereafter) for forwarding this packet.
2. After sending this packet to the buffer of port \( f \), if queue length of port \( f > high\_length \), go to step 3. Otherwise, the algorithm ends here.
3. If all \( n \) ports are currently active, the algorithm ends. Otherwise, determine the leftmost **idle** port (denoted by port \( u \) hereafter) among those \( n \) ports.
4. Set port \( u \) to **active** and also signal port \( v \) to become **active**.

We have three notes for this algorithm. First, in step 1, \( m \) is guaranteed to be at least one because of the existence of the baseline spanning tree. Second, in step 4, both ports \( u \) and \( v \) experience the short time of state transition. In our simulation,
a third port state transit is introduced to represent this short period. In practice, the state transitions at ports u and v are coordinated by the link layer mechanisms described in the IEEE Std 802.3az [13]. Third, this algorithm has the complexity of O(n), where n is the number of ports returned from the routing table. Since n is generally a small number (e.g., in a fat tree topology, n is equal to k/2), it is affordable to run this algorithm upon every packet arrivals.

The decreasing ports algorithm is detailed as follows.

**Input:** the queue length and utilization of each port on this switch.

**Output:** the list of ports to idle.

1. For each secondary port u that is active in this switch, execute the following steps.
2. Calculate the utilization of port u.
3. If (queue length of port u = 0) and (utilization of port u < low_util):
   a) Signal port v to see whether port v can idle.
   b) If (queue length of port v = 0) and (utilization of port v < low_util), both ports u and v transit to idle state. Otherwise, both ports stay in active state.

Note that this algorithm has the complexity of O(p), where p is the number of ports in a switch. Since the utilization period should be significantly longer than the transmission delay of a packet, this algorithm is invoked much less frequently than the increasing ports algorithm.

### IV. Evaluation

We implemented eAware using the network simulator ns-2 [16]. Our source codes are made available at [17], so others can provide feedback or utilize them. Without loss of generality, the fat tree topology [3] is used to evaluate eAware, since fat tree is a typical topology with rich connectivity adopted by data centers [8]. Figure 4 illustrates the fat tree topology with k=4 built under ns-2, showing a graph isomorphic with the graph in Figure 1. Specifically, in Figure 4, nodes 0-15 represent servers, nodes 16-23 represent bottom level switches, nodes 24-31 represent middle level switches, and nodes 32-35 represent top level switches.

Since the traffic load in data centers varies significantly over time, we use the Pareto traffic generator shipped in ns-2 to generate the network traffic used in our simulations. Specifically, this generator produces network traffic according to a Pareto On/Off distribution [18], and can be defined by the following parameters.

- **burst_time:** the average time for the ‘on’ periods
- **idle_time:** the average time for the ‘off’ periods
- **traffic_rate:** the sending rate during "on" periods
- **shape:** the "shape" parameter used by a Pareto distribution.

In our experiments, we fix burst_time to 500ms, idle_time to 300ms, and shape to 1.5, and vary traffic_rate to change the traffic load on the network. Since this traffic generator uses UDP as the underlying transport protocol, which has no flow control and congestion control, the measurement on the packet drop ratio is a direct reflection on the performance of eAware.

![Figure 4. The fat tree topology with k=4 shown by nam, the GUI of ns-2](image)

In each of our experiments, we run a Pareto traffic generator on each server in the network topology, and the recipient for this generator is randomly selected from other servers. We run 20 experiments with random recipient selections for a set of fixed experiment parameters, and the data plots presented later show the average results from these 20 experiments. The default values for the experiment parameters are summarized in TABLE I. Unless mentioned otherwise, we use the values in this table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_base_sw</td>
<td>100W</td>
</tr>
<tr>
<td>P_idle_pt</td>
<td>1W</td>
</tr>
<tr>
<td>P_active_pt</td>
<td>3W</td>
</tr>
<tr>
<td>P_idle_sw</td>
<td>10W</td>
</tr>
<tr>
<td>high_length</td>
<td>25 pkts</td>
</tr>
<tr>
<td>low_util</td>
<td>0.05</td>
</tr>
<tr>
<td>number of pods</td>
<td>8</td>
</tr>
<tr>
<td>link capacity</td>
<td>1Gbps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_base_sw</td>
<td>100W</td>
</tr>
<tr>
<td>P_idle_pt</td>
<td>1W</td>
</tr>
<tr>
<td>P_active_pt</td>
<td>3W</td>
</tr>
<tr>
<td>P_idle_sw</td>
<td>10W</td>
</tr>
<tr>
<td>high_length</td>
<td>25 pkts</td>
</tr>
<tr>
<td>low_util</td>
<td>0.05</td>
</tr>
<tr>
<td>number of pods</td>
<td>8</td>
</tr>
<tr>
<td>link capacity</td>
<td>1Gbps</td>
</tr>
</tbody>
</table>

### TABLE I. DEFAULT VALUES FOR THE EXPERIMENT PARAMETERS
In our evaluations, we compare eAware with the approach described in [3], which presents the fat tree architecture to make the DCs scalable but considers no energy saving. Hereafter, we label the approach in [3] eOblivious. For each experiment with the same set of parameters, we run it on both eAware and eOblivious.

Specifically, we evaluate on the following three percentages on eAware in comparison with eOblivious: (1) percentage of energy saving, (2) percentage of packet end-to-end delay increase, and (3) percentage of packet loss increase. Within the evaluation on each percentage, we vary the following three parameters.

- **k**: we use four values of 4, 6, 8, 10. Note that a larger number of pods implies a greater redundancy of switches in a network.
- **traffic rate**: we use six values of 50M, 100M, 150M, 200M, 250M, 300M.
- **(high_length, low_util)**: we use three pairs of threshold values: (125 pkts, 0.25), (25 pkts, 0.05), and (5 pkts, 0.01). Note that the first pair of values favors the most on energy efficiency and the least on network performance, and the last pair, vice versa.

### A. Percentage of energy saving

In each experiment on eAware and eOblivious, we measure the total energy consumed by all switches in the network topology. For the measurement on eOblivious, we simply assume that each switch consumes constant power given by $P_{\text{base}} + n \cdot P_{\text{active}}$, where $n$ is the number of ports in a switch. To show how much energy can be saved by eAware over eOblivious during an experiment, we compute the percentage of energy saving (PES) as follows:

$$PES = \frac{\text{energy by eOblivious} - \text{energy by eAware}}{\text{energy by eOblivious}} \times 100\%$$

Figure 5 shows our experimental results for PES when $k$ and traffic rate change, and (high_length, low_util) is fixed to (25, 0.05). From this figure, we can see that (1) significant energy saving from 30% to 50% can be achieved in our experimental settings; (2) the PES decreases slowly with the growth of traffic rate, showing eAware is effective in putting redundant ports/switches to idle; (3) the PES increases with $k$, showing that more energy can be saved when the network topology has greater redundancy.

Figure 6 shows our experimental results for PES when (high_length, low_util) and traffic rate change, and $k$ is fixed to 8. From this figure, we can see that (1) PES increases when the setting of (high_length, low_util) moves from the pair (5, 0.01) that favors network performance to the pair (125, 0.25) that favors energy efficiency; (2) at each traffic rate, the PESs at (125, 0.25), (25, 0.05), and (5, 0.01) do not vary greatly, showing that the setting of (high_length, low_util) does not have a big impact on PES. However, as to be shown later, the setting of (high_length, low_util) will have a big impact on the packet end-to-end delay and packet drop ratio.

### B. Percentage of delay increase

During each experiment on eAware and eOblivious, we measure the average end-to-end delay of all packets that reach the destination. To show how much delay increase eAware will cause in comparison with eOblivious, we calculate the percentage of delay increase (PDI) as follows:

$$PDI = \frac{\text{avg delay eAware} - \text{avg delay eOblivious}}{\text{avg delay eOblivious}} \times 100\%$$

In Figure 7, we plot our experimental results for PDI when $k$ and traffic rate change, and (high_length, low_util) is fixed to (25, 0.05). This figure mainly shows that (1) the PDIs are below 30% at all data points, reflecting that eAware will not significantly increase the end-to-end delays of packets; (2) the PDI increases with $k$, because the number of servers grows cubically with $k$ and the number of switches only grows quadratically with $k$ in fat tree topologies.
Figure 7. Percentage of delay increase when \((\text{high\_length}, \text{low\_util}) = (25, 0.05)\).

Figure 8 shows our experimental results for PDI when \((\text{high\_length}, \text{low\_util})\) and traffic rate change, and \(k\) is fixed to 8. From this figure, we can see that (1) PDI increases when \((\text{high\_length}, \text{low\_util})\) moves from the pair \((5, 0.01)\) that favors network performance to the pair \((125, 0.25)\) that favors energy efficiency; (2) setting \((\text{high\_length}, \text{low\_util})\) to \((5, 0.01)\) can significantly reduce PDI, because this setting will cause \(e\text{Aware}\) to forward packets in a similar manner as \(e\text{Oblivious}\).

Figure 8. The percentages of delay increase when the number of pods is 8.

C. Percentage of packet loss increase

During each experiment on \(e\text{Aware}\) and \(e\text{Oblivious}\), we measure the packet loss ratios on both of them. To show how much packet loss ratio \(e\text{Aware}\) will increase in comparison with \(e\text{Oblivious}\), we calculate the percentage of loss increase (PLI) as follows:

\[
\text{PLI} = \frac{\text{loss ratio } e\text{Aware} - \text{loss ratio } e\text{Oblivious}}{\text{loss ratio } e\text{Oblivious}} \times 100\% 
\]

Figure 9 plots our experimental results for PLI when \((\text{high\_length}, \text{low\_util})\) and traffic rate change, and \(k\) is fixed to \(8\). From this figure, we can see that (1) PLI increases when \((\text{high\_length}, \text{low\_util})\) moves from the pair \((5, 0.01)\) favoring network performance to the pair \((125, 0.25)\) favoring energy efficiency; (2) setting \((\text{high\_length}, \text{low\_util})\) to \((5, 0.01)\) can dramatically reduce PDI, because this setting will make \(e\text{Aware}\) behave similarly as \(e\text{Oblivious}\).

Figure 9. Percentage of loss increase when \((\text{high\_length}, \text{low\_util}) = (25, 0.05)\).

Figure 10 shows our experimental results for PLI when \((\text{high\_length}, \text{low\_util})\) and traffic rate change, and \(k\) is fixed to 8. From this figure, we can see that (1) PLI increases when \((\text{high\_length}, \text{low\_util})\) moves from the pair \((5, 0.01)\) favoring network performance to the pair \((125, 0.25)\) favoring energy efficiency; (2) setting \((\text{high\_length}, \text{low\_util})\) to \((5, 0.01)\) can dramatically reduce PDI, because this setting will make \(e\text{Aware}\) behave similarly as \(e\text{Oblivious}\).

Figure 10. The percentages of loss increase when the number of pods is 8.

V. CONCLUSIONS

In this paper, we proposed \(e\text{Aware}\), a distributed approach based on queue length and utilization to idle/activate ports and switches to save energy for DCs. Due to its distributed and localized nature, \(e\text{Aware}\) can quickly adapt to the traffic changes in DCs and is also scalable. To date, we do not notice other distributed energy saving approaches for switches in data centers. To validate \(e\text{Aware}\), extensive simulations by ns-2 are conducted to compare it with an existing energy oblivious approach. The simulations mainly show that:
• E Aware can save 30%-50% energy compared with the energy oblivious approach.
• E Aware increases the average packet end-to-end delay by 3%-20%, and the packet loss ratio by 0%-0.9%, thus only impairing the network performance slightly.
• If the two thresholds high length and low util are set to very low values, eAware can achieve similar network performance to the energy oblivious approach but can still save energy in those long periods of low traffic load.

Note that eAware is independent of the multipath routing protocols adopted by DCs, which brings the advantages of simplicity and portability. A further reason that we do not combine eAware with the multipath routing protocol (e.g., making queue length and utilization part of the routing messages) is: the queue length and utilization at a switch port (especially the former) can change quickly in a DC environment, while the period of sending routing messages (usually 30s) is too long to propagate these two metrics.

REFERENCES