Traffic Monitoring in Software Defined Networks

Felix John <ujerb@student.kit.edu>
Mo Shen <uzwvg@student.kit.edu>
Hermann Krumrey <uodxh@student.kit.edu>

Paper: PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks

1 Introduction

This section introduces the reader to the work titled *PayLess: A Low Cost Network Monitoring Framework for Software Defined Networks* by S. Chowdhury, F. Bari, R. Ahmed and R. Boutaba,which was published at the IEEE Network Operations and Management Symposium (NOMS) in 2014.

The paper addresses a common network monitoring problem: the trade-off between monitoring accuracy and network overhead. Increasing the frequency of polling the switches in the network and thereby increasing the monitoring accuracy comes at the cost of greater network overhead. To tackle this problem, the paper proposes a monitoring framework for Software Defined Networks (SDNs), called PayLess. The paper puts forward the following two main contributions: a RESTful API and an adaptive statistics collection algorithm. The API provides a high-level abstract view of the network, allowing to request monitoring statistics through a uniform channel. PayLess itself is comprised of a collection of pluggable components and can easily be extended or modified with custom ones, owing to the well-defined interfaces. Furthermore, the paper features an adaptive statistics collection scheduling algorithm that provides real-time network statistics information while inducing minimal overhead to the network. Finally, the performance of the framework is examined in detail in different simulations in *Mininet*.

1.1 Implementation

This section provides details of the implementation as described in the paper. PayLess resides on top of the OpenFlow controller's northbound API, featuring a high-level RESTful API. Network monitoring applications can request the required statistics through the API by registering a *MonitoringRequest* in the form of a JSON object. More specifically, the user can detail the required statistics by specifying the metric or level of aggregation.

The center piece of the paper is the adaptive statistics collection algorithm that aims at solving the trade-off between monitoring accuracy and network overhead. The pseudocode of this algorithm is depicted in figure 1. In contrast to FlowSense [2], which makes use of *PacketIn* and *FlowRemoved* messages only, this algorithm requires the controller to send additional

Algorithm 1 FlowStatisticsCollectionScheduling(*Event e*)

```
globals:  $active\_flows$ //Currently Active Flows
          $schedule\_table$ //Associative table of active flows
                            // indexed by poll frequency
          $U$ // Utilization Statistics. Output of this algorithm
if $e$ is Initialization event then
    $active\_flows \leftarrow \phi$, $schedule\_table \leftarrow \phi$, $U \leftarrow \phi$
end if
if $e$ is a PacketIn event then
    $f \leftarrow \langle e.switch, e.port, \mathcal{T}_{min}, 0 \rangle$
    $schedule\_table[\mathcal{T}_{min}] \leftarrow schedule\_table[\mathcal{T}_{min}] \cup f$
else if $e$ is timeout $\tau$ in $schedule\_table$ then
    for all flows $f \in schedule\_table[\tau]$ do
        send a FlowStatisticsRequest to $f.switch$
    end for
else if $e$ is a FlowStatisticsReply event for flow $f$
then
    $diff\_byte\_count \leftarrow e.byte\_count - f.byte\_count$
    $diff\_duration \leftarrow e.duration - f.duration$
    $checkpoint \leftarrow current\_time\_stamp$
    $U[f.port][f.switch][checkpoint] \leftarrow \langle diff\_byte\_count,$
                                         $diff\_duration \rangle$
    if $diff\_byte\_count < \Delta_1$ then
        $f.\tau \leftarrow \min(f.\tau\alpha, \mathcal{T}_{max})$
        Move $f$ to $schedule\_table[f.\tau]$
    else if $diff\_byte\_count > \Delta_2$ then
        $f.\tau \leftarrow \max(f.\tau/\beta, \mathcal{T}_{min})$
        Move $f$ to $schedule\_table[f.\tau]$
    end if
end if
```

Figure 1: Pseudocode of the flow statistics scheduling algorithm (from [1])

FlowStatisticsRequest messages to the switch. For every *PacketIn* message the algorithm adds a new flow to a hash table, including the corresponding switch, port and an initial statistics collection timeout value. This table is used to keep track of the active flows in the network, indexed by the flow's respective timeout value. Additionally, the flows that exhibit the same timeout value are grouped together in the same bucket inside the hash table. The following two scenarios are distinguished: Either the timeout value is surpassed or the flow expires within the specified timeout. If the flow expires, the controller will consequently receive the statistics in a *FlowRemoved* message. Otherwise, in response to the timeout of a flow, the controller sends a *FlowStatisticsRequest* message to the corresponding switch, requesting the statistics for the flow. Each bucket in the hash table is assigned a worker thread, which is responsible for periodically requesting the statistics in accordance with the timeout value of the bucket. When receiving the corresponding *FlowStatisticsReply* message, the algorithm calculates the difference between the previous and current byte count as well as the difference in duration. Afterwards, the timeout value of each flow is modified, depending on the magnitude of difference in byte count. This adjustment aims at maintaining a higher polling frequency for flows that currently make a significant contribution to the overall link utilization, while setting a lower polling frequency for flows which contribute less.

Furthermore, the algorithm is optimized by batching *FlowStatisticsRequest* messages of flows which show the same timeout value. The algorithm is implemented using the OpenFlow controller Floodlight.

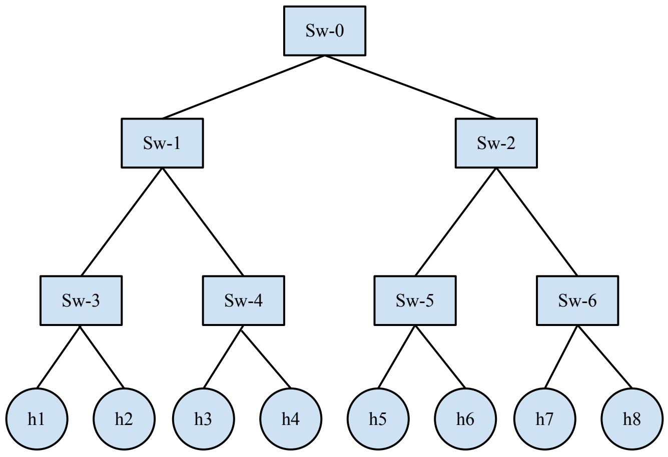


Figure 2: Topology of the network (from [1])

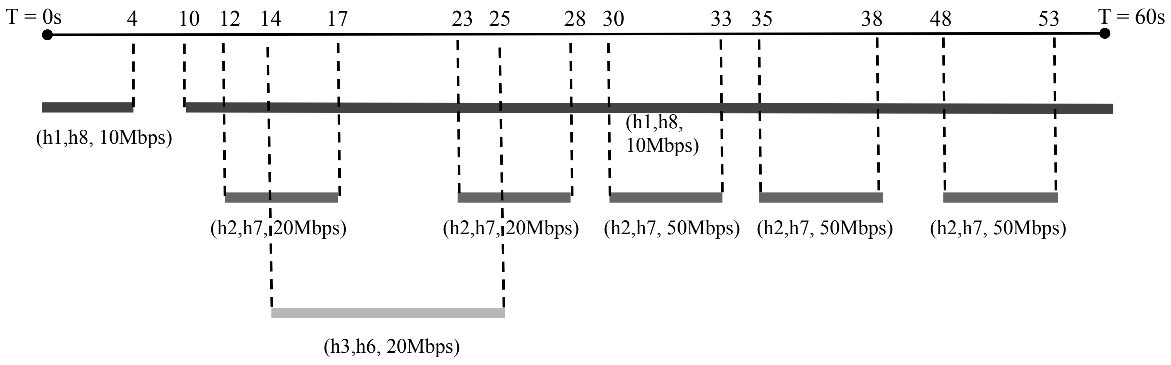


Figure 3: Timing diagram of the generated traffic (from [1])

1.2 Evaluation Results

This section described the evaluation results as reported in the paper. The authors conduct a number of different experiments to underscore the effectiveness of the algorithm using *Mininet* to simulate a network and *iperf* for traffic generation.

The topology of the network can be seen in figure 2. The network features 6 switches and 8 different hosts in the 3-level tree topology of the network. The details of the generated traffic are found in figure 2. UDP traffic flows through the network for a total time of 100 seconds. One experiment focuses on the link utilization between the switches in the network. The performance of the algorithm is compared to a baseline that periodically polls the statistics and the implementation of [2]. The results are depicted in figure 4.

According to the graphs, the periodic polling most accurately reflects the generated traffic, but inducing a lot of network overhead. Flowsense [2] on the other hand fails to capture the traffic spikes as it only keeps track of *FlowRemoved* and *PacketIn* messages. In contrast, PayLess closely resembles the measured utilization pattern of the periodic polling.

Another experiment computes the messaging overhead in terms of the number of OpenFlow *FlowStatisticsRequest* messages sent during the course of the experiment. The measured overhead of periodic polling and PayLess can be seen in figure 5. As Flowsense does not make

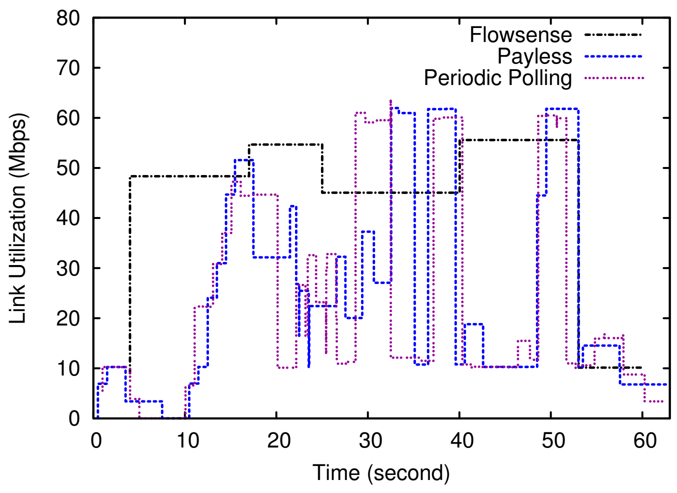


Figure 4: Utilization measurement (from [1])

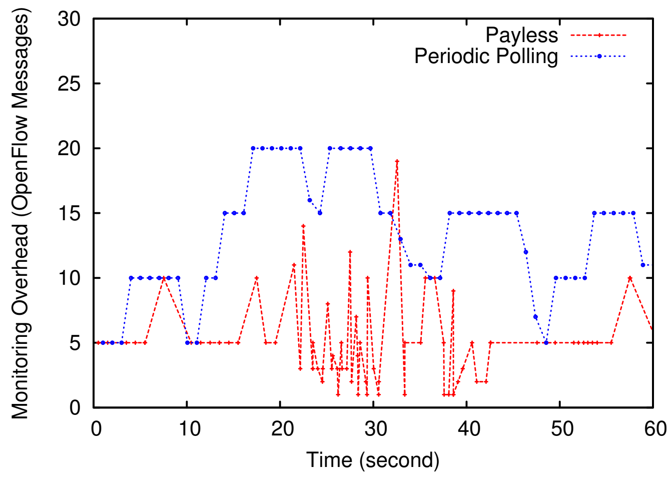


Figure 5: Messaging overhead (from [1])

Component	Paper	Our Implementation
Controller	Floodlight Controller	Ryu Controller
Routing Algorithm	Not specified	Dijkstra's shortest path algorithm
RESTful API	Yes	Yes
Output Format	JSON on demand (via API)	JSON
Visualization	Not specified	matplotlib
Traffic generation	iPerf	Python script

Table 1: Differences between our implementation and the one in the paper

use of *FlowStatisticsRequest* messages, the algorithm is left out in this comparison. PayLess clearly reduces the amount of overhead in the network when compared to the periodic polling. In fact, the proposed algorithm significantly reduces the overhead spikes by adjusting the timeout value and spreading the messages over time. In summary, the evaluation demonstrates the effectiveness of Payless and its capability of maintaining a high monitoring accuracy, while reducing the induced network overhead.

2 Report on Reproducibility

2.1 Implementation

This section explores the implementation details of our recreation of the PayLess framework. The major differences and similarities between our implementation and the one described in the paper are listed in table 1. Instead of using the Floodlight controller, we opted to use the Ryu controller due to prior experience with Ryu. The routing algorithm used to determine the flow rules to be installed during experiments was not specified in the paper. For this purpose, we implemented Dijkstra's shortest path algorithm, enabling our implementation to calculate paths for any topology. Our implementation of the monitoring framework generates the statistics output data in JSON format and saves the data to a local file, whereas the implementation in the paper aims to offer access to the monitoring data via the RESTful API. Our project also includes a rudimentary RESTful API that is based on the RESTful API described in the paper. To visualize the raw data, we made use of the *Matplotlib* plotting library. While the original paper relied on *iPerf* to generate experiment traffic, we chose to write our own Python script to set up the traffic for greater flexibility.

2.1.1 Framework

We developed a framework that supports monitoring of arbitrary network topologies using interchangeable monitoring algorithms. The framework is implemented in the form of a Python library. Figure 6 illustrates the various classes and their relationships with each other as well as important attributes and methods by means of an UML class diagram.

The framework features classes that inherit from the RyuApp class and add methods as well as fields to simplify the integration of a monitoring algorithm. This is done in the *RyuWrapper* and *MonitoringFramework* classes.

Inheriting directly from the *RyuApp* class, the *RyuWrapper* class implements some convenience methods to simplify the tasks of installing new flow rules, forwarding packets and sending flow statistics requests to a switch. The class also keeps track of connected switches and installed

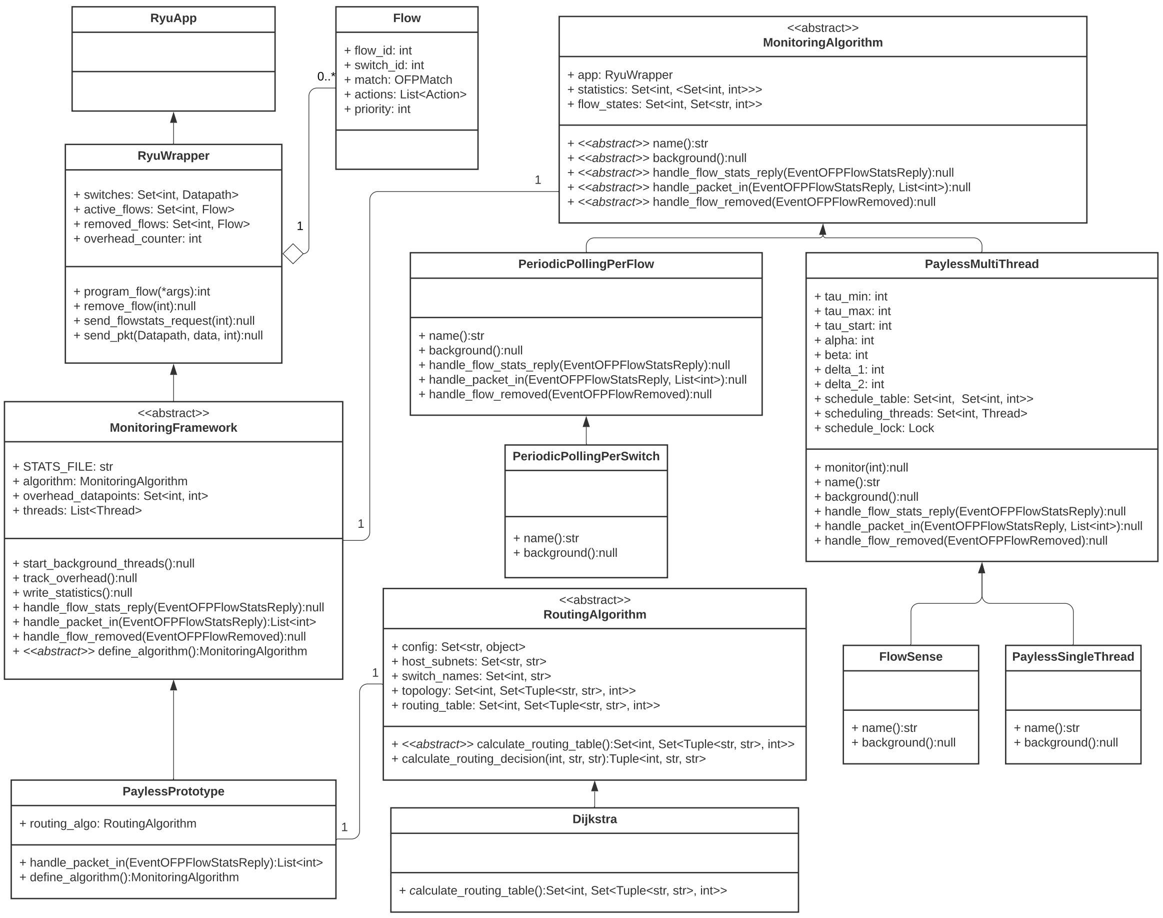

RyuApp
Flow
+ flow_id: int
+ switch_id: int
+ match: OFPMatch
+ actions: List<Action>
+ priority: int
0..*
1
<<abstract>>
MonitoringAlgorithm
+ app: RyuWrapper
+ statistics: Set<int, <Set<int, int>>>
+ flow_states: Set<int, Set<str, int>>
+ <<abstract>> name():str
+ <<abstract>> background():null
+ <<abstract>> handle_flow_stats_reply(EventOFPFlowStatsReply):null
+ <<abstract>> handle_packet_in(EventOFPFlowStatsReply, List<int>):null
+ <<abstract>> handle_flow_removed(EventOFPFlowRemoved):null
RyuWrapper
+ switches: Set<int, Datapath>
+ active_flows: Set<int, Flow>
+ removed_flows: Set<int, Flow>
+ overhead_counter: int
+ program_flow(*args):int
+ remove_flow(int):null
+ send_flowstats_request(int):null
+ send_pkt(Datapath, data, int):null
1
PeriodicPollingPerFlow
+ name():str
+ background():null
+ handle_flow_stats_reply(EventOFPFlowStatsReply):null
+ handle_packet_in(EventOFPFlowStatsReply, List<int>):null
+ handle_flow_removed(EventOFPFlowRemoved):null
PaylessMultiThread
+ tau_min: int
+ tau_max: int
+ tau_start: int
+ alpha: int
+ beta: int
+ delta_1: int
+ delta_2: int
+ schedule_table: Set<int, Set<int, int>>
+ scheduling_threads: Set<int, Thread>
+ schedule_lock: Lock
+ monitor(int):null
+ name():str
+ background():null
+ handle_flow_stats_reply(EventOFPFlowStatsReply):null
+ handle_packet_in(EventOFPFlowStatsReply, List<int>):null
+ handle_flow_removed(EventOFPFlowRemoved):null
<<abstract>>
MonitoringFramework
+ STATS_FILE: str
+ algorithm: MonitoringAlgorithm
+ overhead_datapoints: Set<int, int>
+ threads: List<Thread>
+ start_background_threads():null
+ track_overhead():null
+ write_statistics():null
+ handle_flow_stats_reply(EventOFPFlowStatsReply):null
+ handle_packet_in(EventOFPFlowStatsReply):List<int>
+ handle_flow_removed(EventOFPFlowRemoved):null
+ <<abstract>> define_algorithm():MonitoringAlgorithm
1
PeriodicPollingPerSwitch
+ name():str
+ background():null
<<abstract>>
RoutingAlgorithm
+ config: Set<str, object>
+ host_subnets: Set<str, str>
+ switch_names: Set<int, str>
+ topology: Set<int, Set<Tuple<str, str>, int>>
+ routing_table: Set<int, Set<Tuple<str, str>, int>>
+ <<abstract>> calculate_routing_table():Set<int, Set<Tuple<str, str>, int>>
+ calculate_routing_decision(int, str, str):Tuple<int, str, str>
1
FlowSense
+ name():str
+ background():null
PaylessSingleThread
+ name():str
+ background():null
PaylessPrototype
+ routing_algo: RoutingAlgorithm
+ handle_packet_in(EventOFPFlowStatsReply):List<int>
+ define_algorithm():MonitoringAlgorithm
1
Dijkstra
+ calculate_routing_table():Set<int, Set<Tuple<str, str>, int>>

Figure 6: UML class diagram of the framework.

flow rules. All flows are assigned a unique ID. The flow's information is stored in a so-called Flow object, facilitating the access of information about the flow. Additionally, the *RyuWrapper* class keeps track of the total number of flow statistics requests sent by the controller.

The *MonitoringFramework* class inherits from the *RyuWrapper* class and integrates an interchangeable monitoring algorithm. The class forwards *PacketIn*, *FlowStatisticsReply* and *FlowRemoved* messages to the monitoring algorithm. A new thread is also started for any background tasks for the monitoring algorithm. This class also keeps track of the amount of flow statistics requests that were sent each second, which is important for statistics on monitoring overhead over time. Additionally, the class periodically writes the collected monitoring and overhead statistics to a JSON file.

The monitoring algorithms were made interchangeable by encapsulating the algorithm logic in a class that inherits from the abstract class *MonitoringAlgorithm*. Any child class of *MonitoringAlgorithm* must specify functions that handle the incoming Openflow messages. Additionally, the *MonitoringAlgorithm* class has to define a method to handles any background logic, running independent of incoming OpenFlow messages.

The library includes an implementation for a periodic polling monitoring algorithm, implemented in the *PeriodicPollingPerFlow* class. Being based on the description in the original paper, this periodic polling algorithm serves as a baseline to which the other monitoring algorithms are compared to. As the name implies, the algorithm periodically polls the switches of any active flows once every second. Beyond that, we provide an extension to periodic polling, namely the *PeriodicPollingPerSwitch* class, which polls each switch every second.

We replicated the adaptive monitoring algorithm presented in the original paper in two different ways. Although both variants implement the algorithm described in the paper, they differ in their approach to threads. The first variant, namely *PaylessMultiThread*, spawns a new thread for each bucket, while *PaylessSingleThread* makes use of a one single, main background thread. Both variants support modifying the Payless algorithm parameters using constructor parameters. Additionally, we included a simple implementation of FlowSense [2] in the *FlowSense* class. This algorithm only reacts to incoming *FlowRemoved* messages and does not actively send out flow statistics requests.

The *PaylessPrototype* class is not part of the library, but rather makes use of the library to create a fully functional SDN application with monitoring functionality. The class inherits from the *MonitoringAlgorithm* class and can be configured using environment variables to specify the following properties:

1. The monitoring algorithm to use
2. Payless algorithm parameters
3. Output file path
4. Topology file path

The *PaylessPrototype* makes use of Dijkstra's shortest path algorithm to calculate appropriate flow rules for incoming packets based on the topology file. Dijkstra's algorithm is encapsulated in the *DijkstraRouting* class, which is a subclass of the *RoutingAlgorithm* class. This construct should support and ease any modification or replacement of the routing algorithm, if required in the future. After the flow rules are calculated, they are installed reactively with an idle timeout of 5 seconds.

2.1.2 Visualization

To visualize the resulting monitoring data, we made use of the *Matplotlib* plotting library. For the sake of visual consistency, we attempted to keep the style and formatting of the generated graphs in line with the style and formatting in the paper.

The visualization of the link utilization statistics ignores any flows that do not match on IPv4 sources and destinations, which in essence means that the statistics on the default flow rules are not represented. As the data collected by the monitoring framework are represented as individual events of a specified duration and byte count, the visualization must first prepare the data for display in a continuous graph, as well as normalize the data in regards to the first point of the graph on the time axis. This is required to effectively compare the graphs of different monitoring algorithms. Additionally, the unit of time is reduced to seconds, as opposed to milliseconds, which are used in the raw data generated by the monitoring framework. The link utilization statistics can be bundled in three different ways:

1. per flow
2. per IPv4 match
3. total link utilization of the network

The visualization of overhead statistics also requires normalized data, however, the plotting is simpler here, as the statistics are already in a format that easily translates to a continuous graph. We included two variants for visualizing overhead statistics. The cumulative variant shows the total overhead generated by an experiment over time, while the relative variant shows the overhead generated in one particular second. Examples for generated graphs can be found in section 2.3.

2.1.3 RESTful API

Next to the implementation of the statistics collection algorithm, we set up a basic RESTful API similar to what is described in the paper. The API is built using the web framework *Flask* and the database toolkit *SQLAlchemy*. Any client can interact with the API by making use of the API functions by the URIs specified in the paper. More specifically, every network client needs to create a *MonitoringRequest* object and register it with the API. The *MonitoringRequest* contains the details of the desired monitoring statistics and is specified using JSON. The implemented URIs provide the basic CRUD functionality for the *MonitoringRequest* object. In contrast to the paper that can retrieve the data on demand, this API can only fetch the data once it has been generated by the statistics collection algorithm. The architecture of the RESTful API can be seen in figure 7. Each *MonitoringRequest* is separately stored in an *SQLite* database. First, the client has to register his monitoring request, receiving an *AccessID* in return. Afterwards, the client can use this ID to retrieve the desired data or make changes to his initial monitoring request.

2.2 Environment Setup and Traffic Generation

Reconstructing the environment setup is as important as the implementation of the framework. Firstly, we want to reproduce and replicate the results presented in the paper. Moreover, we

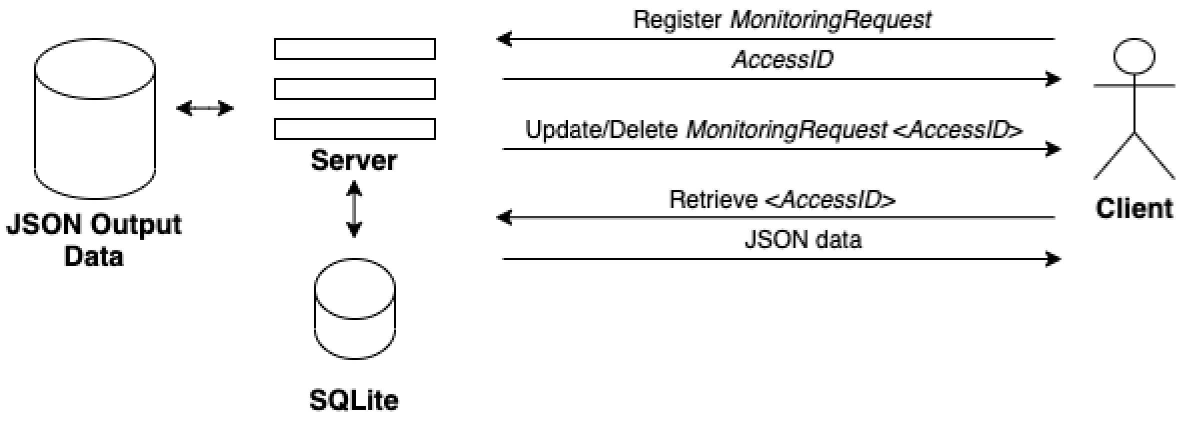


Figure 7: Architecture of RESTful API

aim to make a more in-depth performance comparison of the various monitoring algorithms by including more scenarios.

We used *Mininet* to set up the network topology and wrote the remote controller based on the *Ryu* framework. To ease the use of *Mininet* with different network topologies, we wrote a Python script that generates a *Mininet* topology based on a YAML configuration file.

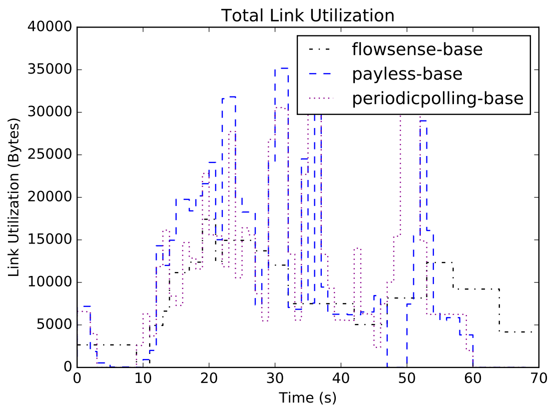
In order to perform the experiments, an appropriate traffic generator is required. A well-designed traffic generator should uniformly transfer data over time and be flexible enough to fit itself into various scenarios. There are several tools to choose from. First, we considered *trafgen* for traffic generation. It allows to define packets in plain text and the sending behaviour with command line arguments. However, because we faced some difficulties using it outside of *sdn-cockpit*, we abandoned *trafgen* at the very beginning. The second option is the high-level network bandwidth measurement tool, called *iperf*. It is able to generate and analyse the traffic at the same time. This option was also discarded as *iperf* hides the details of how the packets are sent, making the evaluation of the actual utilization of the network difficult. Looking into the implementation of *iperf* could clarify the behaviour, but will also add extra work to the project and it shifts from the topic of SDN too much. As a result, we implemented our own UDP traffic generator to ensure the most fine-grained control. Under the hood it utilizes raw sockets in the UNIX system. The script takes a YAML file as input to determine how the traffic should be generated. Additionally, the content of packets can be defined in Python code.

2.3 Evaluation Results

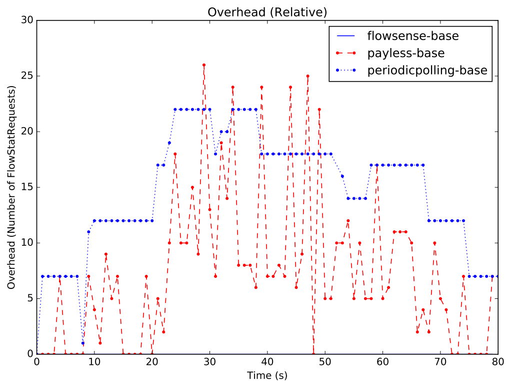
Our goal was to validate the claims made by the original paper as well as finding any potential scenarios in which the PayLess approach to monitoring can thrive. We also attempted to find any issues with using PayLess for monitoring SDN applications.

2.3.1 Experiment 1: Recreating Original Experiment

To confirm the claims made by the original paper, we recreated the network topology depicted in figure 2 as well as the traffic described in the timing diagram in figure 3 that were used in the experiment. Then, we ran the experiment using the periodic polling (PeriodicPollingPerFlow), PayLess (PaylessMultiThread) and FlowSense algorithms and visualized the resulting monitoring



(a) Total Link Utilization



(b) Relative Network Overhead

Figure 8: Results for the experiment based on the original paper (experiment 1).

data. The resulting link utilization and network overhead graphs can be seen in figure 8.

The results of this experiment confirm that payless is able to achieve roughly the same degree of monitoring accuracy for this particular setup as the periodic polling approach while incurring less network overhead at almost every point in time. Our implementation seems to produce less stable results, with more jitter than the one depicted in the original paper, which can be observed by comparing figures 4 and 8a.

2.3.2 Experiment 2: Experiment with Less Traffic

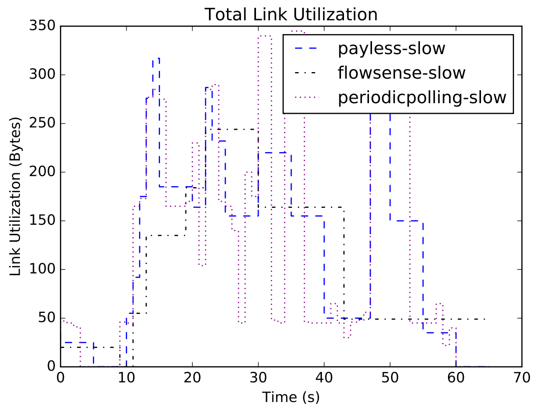
We also conducted further experiments. For example, using the very same scenario as before, we reduced the amount of packets sent by the hosts by a factor of around one hundred. The resulting link utilization and monitoring overhead graphs can be seen in figure 9.

As can be seen in figure 9b. the PayLess algorithm essentially acts the same way the periodic polling algorithm would if the polling interval were set to 5 seconds. This is due to the fact that the low amount of traffic causes all flows to eventually be assigned to the bucket with the lowest polling frequency. Although the PayLess approach loses some of its accuracy when dealing with less traffic, it produces considerably less network overhead than in the previous scenario. Figure 10 compares the network overhead generated by PayLess in both of these experiments.

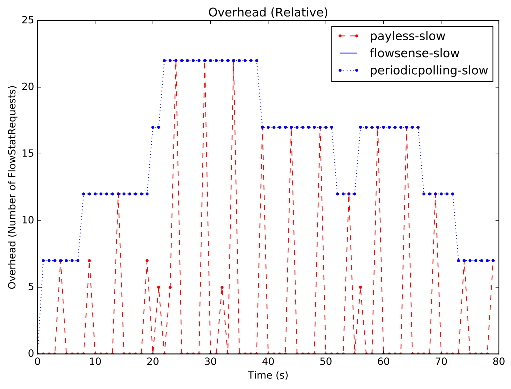
2.3.3 Experiment 3: Experiment with Temporary Pause in Traffic

Another experiment that we conducted is based on a new scenario that includes two 15 seconds long traffic spikes separated by a 20 second pause in which no new traffic is generated by the sending hosts. This experiment resulted in the link utilization and cumulative overhead graphs seen in figure 11.

This experiment demonstrates that PayLess generates noticeably less network overhead than the periodic polling approach during the phase without any new traffic. Additionally, PayLess can compete in terms of monitoring accuracy when compared to the periodic polling approach.



(a) Total Link Utilization



(b) Relative Network Overhead

Figure 9: Results for the experiment using less traffic (experiment 2).

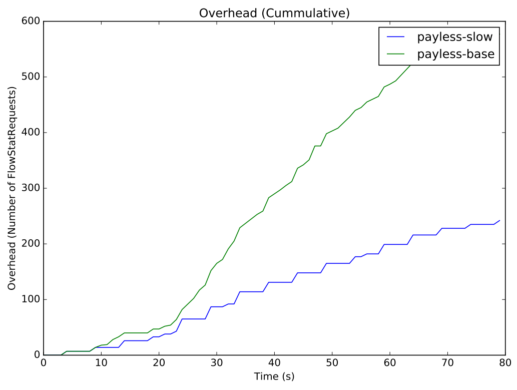
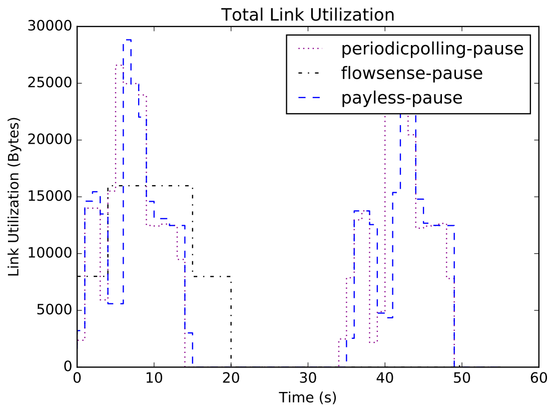
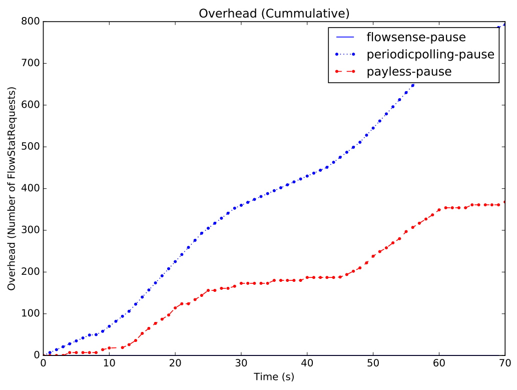


Figure 10: Comparison of cumulative overhead generated by payless during experiment 1 (base-payless) and experiment 2 (slow-payless).



(a) Total Link Utilization



(b) Cumulative Network Overhead

Figure 11: Results for the experiment that includes a 20 second pause in traffic (experiment 3).

2.4 How to Reproduce

This section describes how to reproduce the evaluation results. We use vagrant to build and maintain a portable development environment. A working installation of vagrant is required to successfully proceed with the following steps. Use Git to download all the configuration files and source code. Then setup the environment as shown in Listing 1.

Listing 1: Setup vagrant

```
git clone https://git.scc.kit.edu/tm-praktika/ppsdn-2020/g3.git
cd g3
vagrant up
vagrant ssh
./setup.sh
```

To interact with the framework via RESTful API, start the server with `python api/app.py` and send requests to `http://127.0.0.1:3000`. An example is given in Listing 2.

Listing 2: Start Server

```
import requests
request = {
    "MonitoringRequest": {
        "metric": "utilization",
        "aggregation_level": "switch",
        "monitor": "adaptive"
    }
}
url = 'http://127.0.0.1:3000/'
      'payless/object/monitor_request/register'
res = requests.post(url, json=request)
if res.ok:
    print(res.content)
```

Now it is possible to run experiments with the help of the python script `run_experiment.py`. It takes two YAML files, one for the topology and another for traffic generation. The third argument specifies the algorithm to use for monitoring. Available options are `payless`, `periodicpolling` and `flowsense`. The `-o` option can be used to specify the path to the file in which to store the collected statistics. Furthermore, one can append `-d` or `-v` for debug or verbose output. To reproduce the experiments conducted in the paper, execute the commands shown in listing 3.

Visualizing the statistics recorded in JSON format is also convenient with the help of the custom python scripts. They can be found under the `bin` folder. Listing 4 shows how to generate graphs for the total link utilization and relative overhead and export them to PNG files.

We also included `run_all.sh`, a script that automatically runs all experiments used to generate the graphs in this report. It saves the monitoring data in a directory named after the

Listing 3: Run experiments

```
sudo python run_experiment.py \
    scenario/topo-payless.yaml scenario/scene-payless.yaml \
    payless -o payless.json -v
sudo python run_experiment.py \
    scenario/topo-payless.yaml scenario/scene-payless.yaml \
    periodicpolling -o periodicpolling.json -v
sudo python run_experiment.py \
    scenario/topo-payless.yaml scenario/scene-payless.yaml \
    flowsense -o flowsense.json -v
```

Listing 4: Visualize statistics

```
g3-link-utilization total \
    payless.json periodicpolling.json flowsense.json \
    -o link_utilization.png
g3-overhead relative \
    payless.json periodicpolling.json flowsense.json \
    -o overhead.png
```

current date and also generates the graphs in both PDF and PNG format.

2.5 Conclusion

We successfully implemented and replicated the PayLess algorithm as described in the paper. Furthermore, we were able to reproduce the overhead and link utilization plots of the paper. When comparing the plots of the original paper to our very own plots, we can discover a lot of similarities. Regarding the total link utilization, the PayLess graph as well as the periodic polling graph resemble each other closely in both plots and show the same overall trend. The link utilization of Flowsense is lower in our own plot when compared to the original paper. Concerning the measured monitoring overhead, our reported are very close to the one in the original paper, confirming the effectiveness of PayLess and its capability of maintaining a high monitoring accuracy while reducing the induced network overhead.

We conducted additional experiments beyond the scope of the paper to further validate the effectiveness of PayLess. Using less traffic or pauses within the traffic generation highlighted the characteristics of PayLess. Although the monitoring accuracy may slightly diverge from the periodic polling, PayLess generates significantly less monitoring overhead. The paper proposes a promsing appoach to tackle this trade-off between monitoring accuracy and network overhead. However, PayLess is no all-in-one device that truly and completely solves the challenge of delivering high monitoring accuracy with minimal induced network overhead.

Furthermore, we implemented a basic version of the RESTful API as presented in the paper allowing a client to easily requests monitoring statistics from the server.

We had to face several difficulties when re-implementing the PayLess algorithm. The description in the paper was sometimes missing crucial implementation details. Also, we had some trouble setting up a single thread assigned to each bucket as opposed to having a single main thread. Moreover, the task of generating arbitrary traffic was very difficult as the most prominent tools either failed to work outside of *sdn-cockpit* or lacked important features. Hence, we resorted to create our very own UDP traffic generator.

References

[1] Shihabur Rahman Chowdhury et al. "Payless: A low cost network monitoring framework for software defined networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–9.

[2] Curtis Yu et al. "Flowsense: Monitoring network utilization with zero measurement cost". In: *International Conference on Passive and Active Network Measurement*. Springer. 2013, pp. 31–41.