Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

SpiderWin: Distributed Network Monitoring over Sliding Windows with Programmable Switches

Author: Menno Vermeulen (2612364)

1st supervisor: Lin Wang
2nd reader: Balakrishnan Chandrasekaran

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 8, 2022

Abstract

With DDoS attacks growing in magnitude, network monitoring is important to detect such attacks. Streaming-based network monitoring approaches are the de-facto solution, in which every packet is counted in a sketch. In order to have data in the sketch that represents the current network state, the sketch is periodically reset, discarding all prior values. Maintaining no history makes it impossible to perform certain monitoring tasks, such as change detection. Network monitoring systems that can be run in a distributed setting often make use of the control plane to synchronize state. However, using the control plane for this is slow and scales poorly.

This thesis introduces SpiderWin, an algorithm that can run entirely in the data plane in a distributed setting. By using a window-based network monitoring mechanism, SpiderWin maintains recent history that can be queried. Our results show that SpiderWin can run on a real-life network topology. A practical implementation of SpiderWin for a bmv2 software switch in P4 is available at
`https://github.com/gitMenv/SpiderWin`

1 Introduction

A large number of daily Distributed Denial of Service (DDoS) attacks cripple services on the Internet, causing financial damage [23, 28]. It is important to detect and mitigate such attacks before they cause problems to services, which requires active network monitoring [25]. In general, network monitoring consists of recording the number of packets that belong to flows that pass through the network. While different network monitoring approaches have been proposed, streaming-based approaches with approximate data structures have become the de-facto solution [11,12,18,19,24,27,30]. In these approaches, all packets that pass through the network are counted, contrasting sampling-based approaches [6,10,16,21].

In DDoS attacks over the Internet, attack traffic may enter a network at different entry points. A network monitoring approach that is deployed network-wide is desirable, as this may detect these attacks [17]. Monitoring the network from different vantage points in the network introduces synchronization complexity and communication overhead [11]. Some network monitoring frameworks provide a “One Big Switch” abstraction over all switches in the network [11, 18]. This abstraction allows a network operator to view a network of switches as a single switch, making it easier to reason about the current network state.

To find the overall network state, the state of the individual switches must be shared, but the switches themselves have limited memory and processing power. Therefore, the task of synchronization and calculations is often offloaded to the control plane [11,12,18,19,21,30]. In this configuration, one centralized controller in the control plane manages all switches in the network. The control plane is not limited by the hardware of a switch, but this also means that it can not process packets at line-rate. In addition to this, additional switches increases the load on the controller. Therefore, using a centralized controller in the control plane is slow, and it has poor scalability.

To perform network monitoring at line-rate, network monitoring can be performed entirely in the data plane [24]. Extending this to a network-wide setup is challenging, because the local state of the switches have to be synchronized. A framework that solves the challenge of synchronizing variables in the data plane, is SwiSh [29]. For network monitoring, local data structures must be eventually synchronized. SwiSh provides a solution for this in the form of Strong Delayed Write variables. Their solution ensures that the data is correctly synchronized across all nodes, but it does not provide a way to perform queries over synchronized values in the past, as no history is stored. By applying the sliding window model, multiple windows can be maintained in which recent history is preserved [1].

In this thesis, the following research questions are discussed: (1) How can window-based network monitoring be performed entirely in the data plane? (2) How can local state of switches be synchronized across all switches in the network? (3) What is a reasonable time interval for a window that is sufficiently large to allow for synchronization?

We propose SpiderWin[1], an algorithm to perform network-wide monitoring that can achieve fast and accurate monitoring entirely in the data plane. Our algorithm features the following designs: (a) a ring-based sketch data structure on switches for window-based monitoring, which allows for time-based queries over synchronized data in the past, (b) a configurable number of windows that are maintained by switches, (c) a decoupled window advancement and synchronization step to allow for windows of varying time.

We make the following contributions:

- A window-based network monitoring mechanism entirely in the data plane,
- A synchronization protocol across switches entirely in the data plane,
- A prototype implementation in P4, intended to be run on software switches,
- Experiments to demonstrate and explore the parameters of the system.

This paper is organized as follows. Section 2 describes important topics in network monitoring relevant for this thesis. Section 3 defines SpiderWin and addresses the problems it aims to solve. A practical analysis of SpiderWin, based on an implementation in the P4 programming language, is performed in Section 4. Section 5 describes related work. Finally, Section 6 concludes this paper.

2 Background and Motivation

This section discusses important topics that have been explored in the field of network monitoring and highlights some shortcomings.

2.1 Network Monitoring

The magnitude of DDoS attacks have grown over the years, resulting in more services being rendered useless on a daily basis [17, 28]. It is important to mitigate DDoS attacks before they can do damage, but before an attack can be mitigated, it must be detected. This requires active network monitoring, which can be deployed on network switches due to Software Defined Networking, as described in Section 2.2.

Network monitoring in a network switch is typically performed by counting the number of packets in a *flow*. A flow is defined as a unidirectional sequence of packets with some common properties that pass through a network device [6]. The common properties that

[1]Spiders monitor a web (network), just like our algorithm. Some spiders reset their web every day, while we wish to capture this data. The “Win” part addresses this, as the recent history of the network state is captured in windows.

are used are described by the *flow identifier*, which is often a 5-tuple of the following fields: (Source IP Address, Destination IP Address, Source Port, Destination Port, Protocol Number). For each packet that arrives at a switch, a counter associated with that flow identifier is incremented.

The number of possible flow identifiers greatly exceeds the amount of memory that is available in a network switch. Therefore, approximate data structures are used to store the data, known as *sketches*. A sketch consists of a table with r rows and c columns, where each row has an independent hash function associated to it. For each item that gets added to the sketch, the item is hashed with every hash function that is associated with every row in the sketch. The result is used as an index to select the correct column. Every cell in the table consists of a counter, which gets modified based on the sketch algorithm that is used. A sketch is memory-efficient, as clear memory bounds are defined, based on the number of rows and columns that are used [5, 8].

As hash algorithms are used to index flow identifiers, hash collisions may occur. Distinct flow identifiers may overlap for some counters in the sketch. This may cause inaccuracies when retrieving the true count from the sketch. Because of these inaccuracies, sketches are approximate data structures. By increasing the number of hash functions or columns, one can increase the accuracy of the sketch.

Summable sketches are sketches that can be combined without conflicts [30]. Two sketches are summable if they are identical in the following ways:

- The same sketch algorithm is used to modify the sketch.
- The sketches have the same dimensions.
- The independent hashing algorithms used are the same, using the same polynomials.

The entries of sketches that have the same three properties have the same meaning, and can therefore be summed without conflicts [5, 30].

2.2 Software-Defined Networking

The growing size of the Internet creates a high entry barrier to try out new ideas, as failing experiments may crash parts of critical infrastructure. Innovation had stalled because of this risk, creating the need for programmable network devices that allow researchers to reprogram switches to behave as they wish [20]. This ultimately led to the creation of the OpenFlow standard [20].

A network switch implementing the OpenFlow standard can take commands from a controller. The controller can configure the routing of packets, according to a flow identifier. This allows network researchers to direct packet flows throughout the network, but it does not provide full control over how the packets are actually forwarded.

The forwarding of packets is separated by the routing of packets, which has been referred to as the *data plane* and the *control plane* respectively in Software-Defined Networking (SDN) [4]. Everything that happens within a switch, is referred to as the data plane, whereas everything happening outside the switch, is the control plane. With OpenFlow, an operator can control the control plane, but they can not modify the data plane [20].

The P4 Programming language allows one to modify the data plane within a network switch [4]. The language provides an abstraction in the form of the Abstract Forwarding

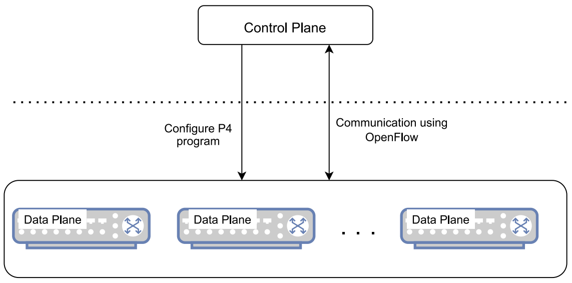


Figure 1: SDN Architecture with OpenFlow and P4 [4]

Model, consisting of a programmable parser followed by multiple stages of match+action, which could be arranged in series or in parallel [4]. For switches, the match+action stages can be configured by defining the behavior in P4. The associated match+action tables can then be populated by the controller in the control plane. Every packet that passes through a switch runs through this pipeline of the parser and the match+action stages. The match+action stages are divided between the *ingress* pipeline, where the packet enters the switch, and the *egress* pipeline, where the packet leaves the switch.

The P4 programming language can be used in the data plane, and it is compatible with control plane solutions such as OpenFlow [4,20]. Together, they provide flexibility in both the control plane and the data plane, which gives a network operator full control over the network. The architecture of SDN with OpenFlow and P4 is shown in Figure 1. In this figure, the controller in the control plane can be used to configure multiple switches in the data plane to run a P4 program. Using OpenFlow, the controller can communicate with the switches. With SDN, it is possible to perform large-scale experiments on the Internet, with full control.

2.3 Distributed State Sharing

Some proposed network monitoring frameworks can be deployed network-wide on multiple switches, which requires state sharing among the switches. In most network monitoring frameworks, this is commonly done by communicating with a centralized controller, as network switches have limited processing power [6, 11, 19, 21, 30]. The controller collects data from the individual switches, processes it and uses the results by instructing switches to perform a task. However, a controller is not a switch and it can not handle packets at the same line-rate as the switches. In Jaqen, a recent network monitoring framework, the controller collects data from its detecting switches every 5 seconds [19]. The more switches that send data to the controller, the larger the interval must be. Therefore, communicating with a controller is slow, and it may be poorly scalable.

By viewing the network of switches as a distributed system, the centralized controller can be omitted. However, the switches in a network now need to synchronize without a controller, which introduces challenges. One challenge is that only a few simple operations can be executed for each packet that arrives at a switch, while more may be needed. Switches must forward Internet packets as quickly as possible and additional operations may cause delays. Therefore, the operations used in the data plane for synchronization are to be kept to a minimum. Another challenge is sharing the state of the individual switches. In network switches, there is no support for a reliable data transmission protocol, yet we still require guarantees about this transmission. At the same time, the data that is shared between the switches should also not impact the performance of the switch. The communication overhead must be kept as low as possible, while providing some reliability guarantees.

SwiSh provides a solution for sharing state between multiple switches entirely in the data plane [29]. The authors provide three abstractions for different applications for memory usage that they identified. Of the three abstractions, the Strong Delayed Writes (SDW) abstraction is most suitable for network monitoring. This is because the SDW abstraction guarantees that the state is correct at all nodes at certain times.

SDW consists of two different windows that contain a pair of 32-bit register arrays. In the synchronization window, this pair consists of a source-register, which stores the local updates, and the merge-register, which stores the data that is merged from the different switches. In the other window, the pair consists of an update-register and a query-register, where the update-register is used to perform local updates and the query-register is used to perform queries. When synchronization is initiated, the source-register is sent to all other switches, after which the switch monitors for incoming acknowledgement messages. If all updates have been sent and received, the windows swap. The merge-register contains the complete data and is swapped with the query-register, whereas the source-register is swapped with the update-register.

While the authors provide an optimization for full sketches on their algorithm, messages still have to be acknowledged. This mimics a rudimentary TCP protocol, which may not be efficient if transmission rarely fails. As all switches already expect packets from other switches, it may be more efficient to request retransmission instead. The SDW algorithm that the authors describe allows for querying on a synchronized variable. However, once a new window is started, the prior value can not be queried anymore and the information is lost. SpiderWin takes inspiration from SwiSh by having a single window that is synchronizing its state among all switches [29].

Menno: I can add a section on maintaining the state at the different switches, which will mainly be on CRDTs. I do have 4 or 5 papers on CRDTs that are quite interesting and I took some inspiration from them. However, this inspiration is also covered by the SwiSh paper; they also refer to CRDTs. I can include it in the background section here, but I am not sure how to shape a subsection like this, as I can explain the algorithm and rationale without it.

3 Design

In Section 3.1, we introduce SpiderWin by describing its key features. SpiderWin is intended to be implemented on a network switch, and run in a network of multiple switches. Section 3.2 describes the window-based monitoring mechanism of SpiderWin. Section 3.3 describes the synchronization of windows among switches in a network. Section 3.4 discusses the limitations of the P4 programming language in the context of hardware and software. Lastly, we show how a practical implementation of SpiderWin can be realized in P4 on a software switch in Section 3.5.

3.1 Key Features

SpiderWin draws inspiration from the SDW abstraction of SwiSh [29]. SwiSh maintains two different windows, one for local updates and one for synchronization. Once the synchronization window is synchronized, the two windows swap and the older values are discarded. Losing this information means that no conclusions can be drawn from changes in network traffic, which is a use case in network monitoring [18]. We also note that window advancement in SwiSh depends on the time it takes for a window to synchronize, making it hard to reason about the actual timing of the data.

Multiple Windows SpiderWin maintains multiple windows of summable sketches, where the number of windows is configurable. This allows for access to prior synchronized windows, rather than only the most recently synchronized window. Each window consists of a sketch, in which all updates are stored and aggregated. The more windows a switch maintains, the more history it stores.

A switch that runs SpiderWin records its local data in the most recent window. When the algorithm advances a window, the switch simply starts recording its local data in the next window, where all sketch entries are reset. Since the windows wrap around, the switch always has access to the most recent history in all windows.

Decoupled Synchronization In SpiderWin, the window advancement protocol is decoupled from the synchronization protocol. This makes it possible to define the time interval for each window. Once a window is advanced, the window that used to be the most recent window will not receive any more local updates. This window will now be available for synchronization. SpiderWin keeps synchronizing all windows that are available for synchronization. Once there are no more such windows, all windows but the most recent one are fully synchronized.

The synchronization time depends on network conditions, and it can take a variable amount of time. Decoupling the window advancement from the synchronization ensures that each window can have a known time interval, independent of network conditions. If synchronization takes longer than the duration of a window, the number of windows that are available for synchronization grows. It can happen that all windows except the current window is available for synchronization, where the protocol is synchronizing the oldest window. In this case, window advancement must stop, as otherwise the least recent data gets overwritten without being fully synchronized.

Bookkeeping When a switch starts synchronizing a window W, the local data gets copied to a temporary sketch T_W. The data stored in T_W is sent to all other switches in the network. Likewise, all other switches send their local data of that window. Upon receiving data from other switches, the switch merges it into window W, which is possible as the sketches are summable. It is important to keep track of what data has been aggregated inside this window so far.

In addition to the sketches, each window also maintains a bitmap. For each switch, there is a bitmap entry. When receiving data from a different switch, the entry in the bitmap of window W, corresponding to that switch is set. When all bits in the bitmap of window W are set, it means that the data of all other switches is aggregated in the sketch of window W, hence the window is synchronized. Upon advancing a window, all bitmaps of that window must be reset as well.

Packet Retransmission For this thesis, we assume perfect networks in which no packets are dropped. However, we do propose a mechanism here to retransmit such lost packets, so that the switches can recover.

All switches in a network advance their synchronization window at approximately the same time. Once the synchronization has started, a timer starts in which the synchronization must have been completed. If this timer runs out and not all bits in the bitmaps are set, the switch sends a *request for retransmission* packet for a specific window and column to a switch. The timer is then also reset.

When a switch receives an *request for retransmission* packet, there are two cases; (1) the switch has already advanced this synchronization window, (2) the switch also misses packets in this synchronization window. In case (1), the switch sends the requested column of the synchronized window back to the requester. The requester then overwrites that column in its sketch with the received data. In case (2), the switch sends the requested column of the temporal sketch back to the requester. The requester merges the received data in its own sketch like a regular synchronization packet.

3.2 Window-based Monitoring

For the following discussion, we consider an undirected connected network of N switches on which SpiderWin runs. We assume that the network configuration is constant, switches do not break and that packets can not be lost. Another important assumption that we make is that the switches in the network are trusted; no switch uses an invalid ID, or an ID that is not their own.

At the basis of the SpiderWin algorithm lies a summable sketch algorithm, such as CountMin Sketch [8]. This sketching algorithm is responsible for counting packets according to a flow identifier, and can be configured with the number of columns and the number of rows. The number of rows correspond to the number of hash algorithms that are used on each flow identifier.

A switch that implements the SpiderWin algorithm maintains multiple windows of sketches and bitmaps in memory, and a few variables. An overview of the memory structure that is used by SpiderWin is shown in Figure 2. A switch in a network of N switches is assigned an ID, and it defines k windows, where each window consists of a summable sketch and N bitmaps. The number of bits in each bitmap correspond to the number of columns in the sketch. A "temporal sketch" is also defined, which has the same dimensions of a sketch in a window. Furthermore, a counter for the current window, and a counter for the synchronization window is defined. The counters only increment and they are used to derive the current window, and the current synchronization window, by using the modulo of the number of windows k.

3.2.1 Window Advancement

A switch always updates its sketch in the most recent window. To this extent, a variable win_{count} is maintained to point to the correct window in memory. Periodically, the switch advances to the next window. The sketch and the bitmaps in the new window are reset. In this new window, the bitmap with the ID of the switch should be all set to 1, whereas all other bitmaps should be set to 0. This indicates that the sketch in the window contains only local updates. After the new window is reset, the win_{count} is incremented to point to the next window. The window advancement protocol is listed in Algorithm 1.

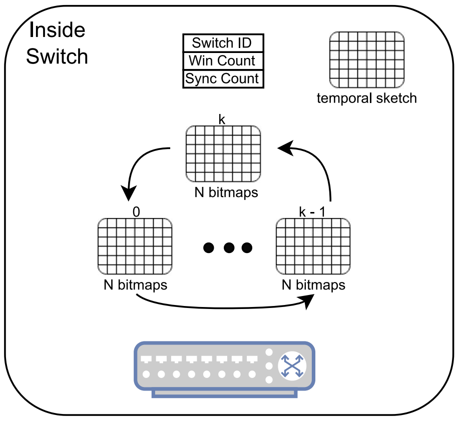


Figure 2: Memory footprint of SpiderWin on a switch

Algorithm 1 Advancing to Next Window

```
1: $win_n \leftarrow (win_{count} + 1) \pmod{k}$
2: $reset\_sketch(win_n)$
3: $reset\_bitmaps(win_n)$
4: $win_{count} \leftarrow win_{count} + 1$        ▷ Effectively advances window
```

In the algorithm, it can be seen that the win_{count} variable is only incremented, and a modulo operation is used to select the correct window. This ensures that the switch loops through its windows, always overwriting the oldest data in the sketch. If a switch maintains more than one window, the switch can still perform queries on its most recent history. Once a switch has advanced from its current window to a new window, the current window becomes available for synchronization.

3.3 Window Synchronization Protocol

The switches in the network all have local state about the packets they have seen. However, to draw conclusions from the complete network data, the local state must be shared. This sharing is done through the synchronization protocol, which synchronizes on a per-window basis. The assumption is made that all switches in the network have synchronized times and that they advance their window at the same time.

Initially, the synchronization is halted, as there is no window available for synchronization yet. Once a switch has advanced its window, no more local updates will be recorded for the old window, hence it will be available for synchronization. This is a trigger for the synchronization protocol, and can be captured by modifying Algorithm 1 as shown in Algorithm 2.

Similar to how the switch maintains a counter for the current window, a syn_{count} variable is maintained to point to the window that is currently being synchronized. The synchronization of a window starts by copying the full sketch data into the temporal sketch. This is important, as the local data must be stored without updates from other switches, which allows for possible retransmissions. Once this is done, the switch sends the local switch data from the temporal sketch to all other switches in the network. The packet that

Algorithm 2 Window Advancement with Trigger for Synchronization

```
1: $win_n \leftarrow (win_{count} + 1) \pmod{k}$
2: $reset\_sketch(win_n)$
3: $reset\_bitmaps(win_n)$
4: if $win_{count} == syn_{count}$ & $isHalted$ then
5:     $isHalted \leftarrow False$
6:     $startSynchronization(syn_{count} \pmod{k})$
7: end if
8: $win_{count} \leftarrow win_{count} + 1$                ▷ Effectively advances window
```

contains this sketch data also contains the switch ID of the switch that sent it, and the current synchronization window. For our discussion, we define such packet to be a *state share packet.*

When a switch receives a state share packet, it first checks if the switch has already received this update from this switch, using the bitmap of that specific sketch in the synchronization window. If it was not included yet, it merges the local data from another switch in the state share packet into the synchronization window and it also sets the bit in the bitmap of the window that corresponds to the switch.

Using the bitmaps in the synchronization window, we know exactly what data has already been merged into the sketch of the current synchronization window. Once all bits of all bitmaps are set in the synchronization window, the window has been fully synchronized and the synchronization window can be advanced, by incrementing its syn_{count}. If the synchronization took shorter than the window advancement, there will not be an available window that can be synchronized. In that case, the synchronization halts. Otherwise it does another iteration of the advancing synchronization window protocol. An overview of this algorithm can be found in Algorithm 3.

Algorithm 3 Advancing Synchronization Window

```
1: $win_{syn} \leftarrow syn_{count} \pmod{k}$
2: $copy\_to\_temporal(win_{syn})$
3: $send\_state\_share\_packets()$
4: while not all bits in bitmaps are set do
5:     $ssp \leftarrow packet\_data$                ▷ simulating a received packet
6:     if $is\_set\_bitmap(spp_{switchID}, win_{syn})$ then
7:         $merge\_data(ssp_{data}, win_{syn})$
8:         $set\_bitmap(spp_{switchID}, win_{syn})$
9:     end if
10: end while                ▷ Ends when all bits are set
11: if $syn_{count} + 1 == win_{count}$ then
12:     $isHalted \leftarrow True$
13: end if
14: $syn_{count} \leftarrow syn_{count} + 1$
```

If synchronization takes much longer than window advancement, the next window advancement may overwrite the current synchronization window. When this is the case, a hard stop must be enforced to window advancement to prevent loss of data. Expanding SpiderWin to make the window advancement interval adaptable, this situation may be prevented, but this is out of scope for this thesis.

3.4 Software and Hardware Implications

As explained in Section 2, the P4 programming language makes it possible to program the different stages in the packet processing pipeline on a network switch. Software that runs on a switch must adhere to constraints that arise from either the hardware switch or the "Abstract forwarding model" [4]. In our case, we wish to provide a practical implementation on a software switch, so we include constraints introduced by a software switch as well.

3.4.1 P4 and the Abstract Forwarding Model

In the abstract forwarding model on which P4 is built, packets must go through a pipeline of different stages, as explained in Section 2.2. A switch that runs P4 does not naturally maintain any state between packets. Extern objects can be used that can maintain state, but the availability of these extern objects depends on the target that will run the program. A limitation of extern objects is that they are bound to one stage in the pipeline, so if an extern object is used in the ingress stage, it can not be accessed from the egress stage. If the egress stage requires this data, it must be passed along with the packet.

A reliable transport protocol is not present on network switches. Therefore, packets that are sent can be dropped without the switch knowing that it was delivered. For our practical implementation, we assume that no packets can be lost. Therefore, the retransmission mechanism as explained in Section 3.1 is not implemented.

In P4, it is possible to perform modular arithmetic, but only if the modulus value is a power of 2. This makes modular operations efficient, but it does limit inner workings of the algorithm. Note that in line 1 of Algorithm 1, a modulo k operation is performed on the window counter. As k represents the number of windows, this P4 limitation means that the number of windows must be a power of 2. The same holds for the number of columns in the sketches, as the result of the hashing algorithm indexes the columns, for which modular arithmetic must be used.

Some operations may not be possible to perform in just one pass through the P4 pipeline of the abstract forwarding model. It is possible to recirculate packets from the egress pipeline to the ingress pipeline [4]. Meta-information that is attached to the packet may be preserved, so that the state is kept. This makes it possible to perform complex operations to a packet.

Some devices that can run P4 have support for multicasting packets and cloning of packets. Both Intel's Tofino hardware switch and the bmv2 software switch support this [3, 13]. These operations are performed by the traffic manager [13]. Multicast operations can only be performed in the ingress pipeline, after which multiple copies of the same packet arrive at the egress pipeline. Cloning can be performed at both the ingress and the egress pipeline. In both cases, the original packet continues the regular steps, but the cloned packet will arrive at the egress pipeline [3, 13].

3.4.2 Hardware Switch Implications

As a target for the hardware switch, we use Intel's Tofino switch [13]. This switch can be programmed in $P4_{16}$, according to the specification that was released by Intel [13]. This specification also defines what and how extern objects can be used, which shows a few limitations and features.

A hardware switch is able to generate packets, based on timed triggers. A periodic timer can be used to advance the window. The generated packets can be used to transmit the state share data to other switches in the network. In case of packet loss, these timers may be used to request a re-transmission of state share data. These timers can only be configured in the control plane, which means that changes to periodic timers can not be made without intervention of the control plane.

Tofino switches can be configured to synchronize time among them. For advancing windows at the same time, it is beneficial to have a common notion of time. Advancing windows at the exact same time means that switches can share their sketch data simultaneously. This decreases the time it takes to synchronize.

3.4.3 Software Switch Implications

As a target for the software switch, we use the simple_switch_grpc of bmv2 [3]. This switch has extern objects that are similar to a hardware switch, but it does have limitations. This has major implications for an implementation in P4.

Using this software switch as a target, branch instructions are not supported inside actions. This means that all of the logic must be implemented inside the "apply"-block of the control blocks.

The software switch does not support the generation of packets, or timed triggers. Since packets can not be generated, existing packets must be cloned or multicast, to be modified without losing the original packet. As there are no timed triggers, the switches must wait for a regular packet to arrive before it can decide to use that packet as a trigger. To solve these limitations, we can send a packet with a custom protocol header that acts as a trigger. This trigger packet can also be used for cloning, multicasting and for modifying.

Another limitation of the software switch is that the extern object of the hash algorithm can not be configured to use a given polynomial. This effectively means that there is only one hash algorithm available. However, the result that the algorithm provides is a 32-bit integer. Provided that we use no more than 256 columns in the sketching algorithm, it is possible to split this integer into four 8-bit numbers. This way, the sketch in a software switch can accommodate up to four rows. Using different flow identifiers for different rows of the sketch allows for an arbitrary number of rows, but this makes it almost impossible to reason about the sketch data.

The software switch is not meant to be a production-grade switch. In a hardware switch, packets will be processed almost instantly due to hardware support. For the software switch, this is all simulated in software, which may introduce large delays.

A feature of the software switch is that it allows for multiple read- and write-accesses to a certain register array. This is in contrast to a hardware switch, as one can only perform one read and one write operation to each register array for one packet.

3.5 Practical Implementation in P4

For this thesis, a practical implementation of SpiderWin was made for a bmv2 software switch in P4. The code for this can be found on GitHub[2]. This includes a description on how this code can be used and run. The remainder of this section describes how challenges were solved.

For the window advancement protocol, it is necessary to trigger an event at given time intervals so that the window can be advanced. However, it is not possible to generate new packets at given intervals in the bmv2 software switch [3]. To overcome this obstacle, external triggers are required, for which a new packet header is defined specifically for this trigger. This custom-made packet can be sent from a Mininet host to a network switch, so that the event can be triggered at the switch.

On a network of hardware switches, the switches would be synchronized and this event would happen at each switch simultaneously. To simulate this, we introduce an extra host to a Mininet network that is connected to every switch in the network. This makes it possible to send a trigger packet to every switch in quick succession.

On line 3 of Algorithm 3, the switch sends state share packets, and they must arrive at every other switch. As a first step, the switch multicasts the packets to all of its links that are connected to other switches, meaning that all neighbouring switches get that state share packet. However, in a network that is not fully connected, there are still switches that must receive the packets. For every destination switch, a set of shortest paths S is created from each source that received the packets. Paths are removed from S until each destination occurs only once, so if these paths are followed, every switch receives a copy of the state share packets. Multicast rules are constructed so that the packets are forwarded to one or more switches according to the path.

State share packets are sent per column, but only one trigger message is sent. The data of one column is copied from the ingress register to a state share packet header. To multicast packets for every column, we add one more copy to the multicast group that gets recirculated from the egress pipeline to the ingress pipeline. Meta-information gets attached indicating that the next column must be sent. This process is illustrated in Figure 3.

The Tofino hardware switch prohibits multiple reads or multiple writes to one register array, which is not a requirement in the software switch. In our practical implementation, we aimed to meet the requirements of the hardware switch, but for some operations we deviated from this. For resetting the sketch in a window and the bitmaps, we perform multiple writes to one register array. A solution to this problem is proposed by Zeno et al. in their paper on SwiSh [29]. It is possible to piggyback initialization on the first write by maintaining a single bit per register to determine whether the register was initialized. This method can be used to reset the registers in a register array, while meeting the single read and single write requirement of a hardware switch.

Each software switch must know its ID that can be used to identify itself. However, since each software switch is given the exact same P4 code using bmv2 and Mininet, it is not possible to define this as a variable [3, 7]. It is possible to set match+action table rules that differ for each switch. To this extent, we create a separate match+action table for initialization, where we pass the ID value using the individual table rules. Upon

[2]`https://github.com/gitMenv/SpiderWin`

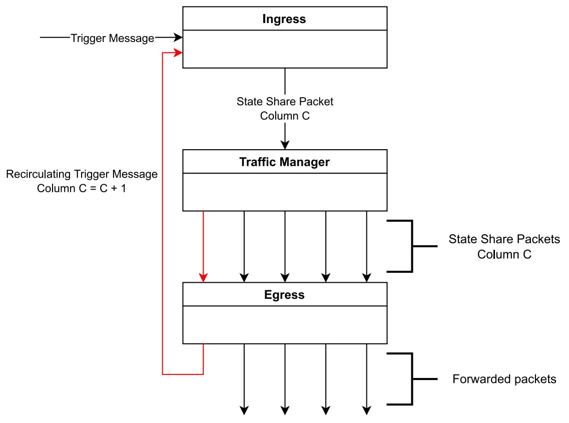


Figure 3: Sending all columns using Multicast

receiving the very first packet, a switch will initialize itself by using the ID provided by the match+action table rule.

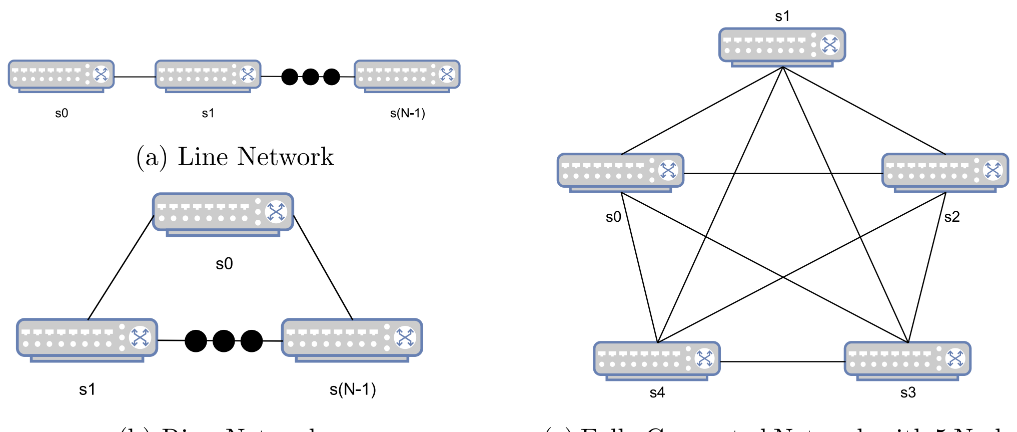
4 Evaluation

In this section, we evaluate the P4 implementation on the bmv2 software switch [3]. The experiments are run as a software simulation in Mininet [7]. Since this software switch is not meant to be production-grade software, we can not accurately assess the performance in terms of efficiency. However, it is possible to assess how the network topology influences the performance on a software switch. The software is run in a Virtual Machine of Ubuntu 20.04, with 15 CPUs allocated to it.

To this extent, different types of topologies are built and simulated through Mininet on the bmv2 switch. The topologies that are chosen, are a line topology, a ring topology and a fully connected topology. An overview of these three generic topologies is shown in Figure 4. These topologies are scaled to a certain number of nodes to see how they perform on the software switch, and how they compare to each other. The data that is collected for the experiments conducted in the context of this thesis is obtained by logging messages in the right locations. To keep the logging overhead to a minimum, only the log messages relevant for that experiment are included. Section 4.1 discusses the logging overhead problem.

In this experiment, we let all switches advance a window and synchronize that window accordingly. To make it more resistant to chance, we perform this synchronization 50 times for all switches and average the result. We assume that the time it takes depends on the size of the network, and on the network topology. As discussed in Section 3.5, each switch sends state share packets to all its neighbours when synchronizing. Therefore, we hypothesize that the fully connected network topology performs fastest, as packets only need one hop, and that the line network topology performs slowest, as all packets must be forwarded multiple times. The ring network topology is similar to the line network topology, but the path length is divided by two, making it seemingly faster than the line.

The results of the experiment are presented in Table 1. This table shows the time it took for one window to be synchronized among all switches, averaged over 50 windows.



(a) Line Network

(b) Ring Network

(c) Fully Connected Network with 5 Nodes

Figure 4: Generic Network Topologies Overview

The switch that takes longest to fully synchronize the window decides the synchronization time of that window. The table also shows the total number of merge operations in the network per window, which is based on a sketch that has 8 columns. This can be calculated using the formula $N * ((N-1) * 8)$. The window synchronization times are also summarized in Figure 5. This shows that the synchronization time grows with the number of switches that are in the network. However, the synchronization time between the three different networks does not show much difference. While these results do not meet our expectations, they can be used as a resource for choosing a suitable window advancement time interval. If we choose a time interval smaller than the reported times for these topologies, the synchronization window would not keep up, requiring a hard stop on window advancement. For a fully connected network topology with 16 switches, a suitable time interval of 0.5 seconds may be chosen, as this is longer than the average synchronization time.

In Figure 5, it can be seen that the fully connected network with 16 switches takes much longer to synchronize one window than the line and the ring network topology. For all other network sizes, they show similar results. The Virtual Machine that is used to run the experiments was allocated 15 CPUs, meaning that context switching of CPUs is required for the larger networks. This can explain the difference in performance for the networks of size 16. There are several possible explanations as to why there is little difference between the network topologies of the smaller sizes. For each of these possible explanations, we ran an experiment.

- The number of links maintained by Mininet and bmv2 cause overhead.

The number of links in a fully connected network, is $n(n-1)/2$, the number of links in a line network is only $n-1$, and the number of links in a ring network is n. If simply having these links in a simulated Mininet network cause overhead, it may be the case that the algorithm performs better on the fully connected network, but this does not show in the results due to the link overhead. To test this, we add links to a line network of 16 switches until the network is fully connected, while keeping the same forwarding rules as in the line topology. This way, the extra links are not used, but they are present. We did the same

Table 1: Window Synchronization Times (in ms) for Line, Ring and Connected Networks

Nodes	**Total Merge Operations**	**Line**	**Ring**	**Connected**
2	16	11.3	-	-
3	48	23.3	15.5	-
4	96	32.1	21.3	20.9
5	160	40.1	27.7	26.7
6	240	47.5	36.7	33.4
7	336	58.2	44.7	40.6
8	448	61.3	54.7	47.8
9	576	71.0	64.9	57.8
10	720	84.8	77.5	69.8
11	880	109.5	96.5	85.5
12	1056	118.3	110.6	109.3
13	1248	140.2	129.4	136.9
14	1456	162.8	151.6	166.4
15	1680	191.7	185.1	214.3
16	1920	224.7	225.7	458.16

for the ring topology. Rerunning the experiment with these configurations shows that the number of links does not impact the time it took for windows to synchronize.

- The overhead of multicasting many packets is exponentially larger than multicasting few packets multiple times.

For the fully connected network, each switch multicasts its packets to every other switch in one stage, whereas in the ring network, every switch multicasts its packets to at most two switches. If multicasting packets to many destinations causes exponentially more overhead than multicasting packets to one or two destinations, it explains why the performance is similar. To test if this is the case, the experiment is run on a fully connected network with 16 switches. One switch is instructed multicast 100 times to a varying number of instances, up to and including 15, as this is the maximum number of multicast instances that can be sent in a network of 16 switches. The time interval between initiating a multicast and sending the last packet of that multicast is measured. Of the 100 multicast operations, the average time is reported.

The results are shown in Figure 6. The line that can be seen in the figure more closely resembles a linear line than an exponential curve. Therefore, we conclude that this is a linear relation, which means that the overhead of multicasting many packets is not exponentially larger than multicasting a few packets multiple times.

- The overhead of merging the state share packets are much larger than the sharing overhead.

The constant factor among all three topologies is the number of state share packets that are generated that all have to be merged at every switch. If the merge operation on the bmv2 software switch takes too much time so that most packets queue up at the switch,

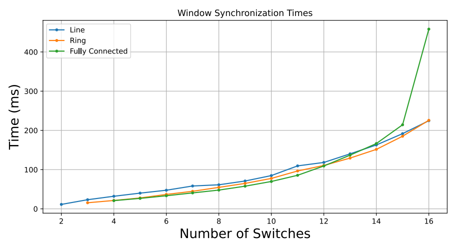


Figure 5: Synchronization times of the different generic topologies

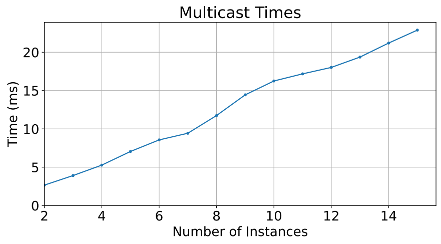


Figure 6: Time to send multicast messages

Table 2: Merge Operation Times per Switch

Switch ID	Average of Single Merge (ms)	Total Time of Merge Operations (ms)
0	0.17	10
1	0.29	16
2	0.36	20
3	0.29	16
4	0.46	26
5	0.26	15
6	0.30	17
7	0.34	19

it might make the network topology irrelevant. To test if this is the case, we measure how long it takes for all merge operations to complete on any switch. The experiment is run on a fully connected network of 8 switches, where one window is synchronized. As can be seen in Table 1, the total number of merge operations that are performed is 448.

A window is fully synchronized only if all switches have performed all merge operations. Each switch performs $(8 - 1) * 8 = 56$ merge operations. The switch that takes most time to complete all merge operations decides the time it takes for a window to be fully synchronized.

After running this experiment, we found that a lot of merge operations are logged as 0, which means that they took less than 1 millisecond. This is not realistic, as a merge operation always takes some time. Table 2 shows the results of all eight switches. The table reports the average time of one merge operation and the sum of the time for all merge operations. The switch that takes longest to synchronize a window dictates the synchronization speed. Therefore, we look at the sum of times for the merge operations. This shows that the switch with ID 4 took the most time to perform all merge operations, namely 26 milliseconds. Note that this is the minimum, due to the millisecond granularity.

In Table 1, it can be seen that the time it takes for a fully connected network with 8 switches to synchronize a window is around 47.81 milliseconds. 26 milliseconds is already more than half of this synchronization time. This is quite substantial, especially because this is a minimum for only the merge operations.

While merge operations demand quite some processing power on this software switch,

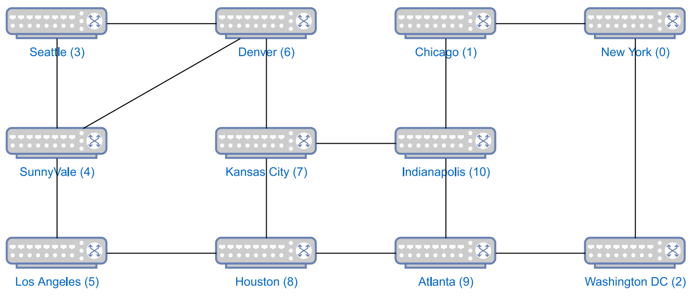


Figure 7: Abilene Network Topology

we believe that this would not be the case on a hardware switch. The merge operation checks the bitmaps of that window to decide whether it can be added to the sketch. If this is the case, it reads the specified column of each row register array, and writes back the updated result. It also sets the corresponding bit to prevent duplicate data. Since each bitmap of columns is a separate register array and the sketch rows are separate register arrays, both the read operations and the write operations can run in parallel, supported by hardware. This makes it much faster than a simulation in software. As the merge operations take more time than it would take for a state share packet to be forwarded, we believe that this is the reason for the performance to be so similar for each network topology.

The generic network topologies do not represent a real-world network. To this extent, we repeat the same experiment on the Abilene network, that is defined on the Topology Zoo [15]. This network has 11 switches and the switches are tied to real locations. This makes it possible to configure the delay of the links, based on the light speed and the distance between two switches. The Abilene network topology is shown in Figure 7. The experiment on this network generates 880 state share packets per window synchronization and took on average 5733.43 milliseconds. This shows that SpiderWin can also be used on a real-life network topology.

4.1 Logging Overhead

The SpiderWin algorithm was evaluated with multiple experiments. These experiments are carried out by logging messages in relevant places in the P4 code. The bmv2 software switch provides an extern object that allows for logging from P4 code [3]. Each log message is timestamped by bmv2 and these timestamps are used for time measurements.

Since logging is prevalent in all experiments, and the results of the experiments may depend on the overhead caused by logging, we also assess the impact of logging statements. Logging statements in P4 can print any number of variables. However, in the experiments only one variable is used in logging messages. Therefore, a small experiment is run to assess the overhead of logging, with a logging statement that includes one variable. This is done on a fully connected network of 16 switches, as this is the most demanding network that we assess.

Two logging message statements are inserted that log whenever a switch receives a trigger message. These logging messages are statements in P4, include a variable, and they provide timestamps between which the time can be measured. We repeated this experiment with an additional logging message statement in the middle. This way, the overhead of a single logging statement can be assessed. For this experiment, 100 trigger messages are sent to all 16 switches, giving 800 data points.

We found that both in the case of an additional logging message and in the case without a logging message, most measurements reported 0 milliseconds, with very few that reported 1 millisecond. It is not true that it takes no time to log a message, but when it takes less than 1 millisecond, it shows as 0. The logging granularity provided by bmv2 is in milliseconds, but to measure a single statement, a nanosecond granularity is required. Therefore, we can not accurately measure the time it takes for one logging message, making it hard to quantify the logging overhead.

For the experiment that assessed the merge operations, we also found that the timestamp granularity is too large, as many data points reported 0 millisecond time intervals. This is a limitation of the software switch that we can not circumvent. The bmv2 software switch is not meant to be a production-grade software switch. However, the results that are derived from the experiments do give a general impression about how the SpiderWin algorithm behaves on the software switch.

4.2 Duplicate Measurements

Deploying SpiderWin on multiple switches in the same network, it will result in duplicate measurements. Packets may be forwarded between switches and each switch on the path will count this packet. When the data is aggregated after synchronization, the packets that were forwarded between switches is counted multiple times.

A simple solution to this problem is to identify which ports are connected to other switches. All packets that are received through such ports are not counted. However, in case of unbalanced network traffic, this might lead to one switch counting all the packets whereas the other switches remain idle. The problem of preventing duplicate measurements has been explored by multiple network monitoring frameworks that can be run in a distributed setting [11, 18, 21].

5 Related Work

In network monitoring, sketching is often used to aggregate a lot of data over time [2]. A sketch is an approximate data structure that updates counters for each new data point that is observed. What the update and read function looks like, depends on the sketching algorithm that is used [5, 8].

In Software-Defined Networking (SDN), the network is divided between the control plane and the data plane. The control plane dictates how the Internet packets should be routed and this can be defined in software using OpenFlow [20]. The data plane forwards the Internet packets according to the rules that the control plane specifies. The P4 programming language allows network operators to modify how the packets are processed in the data plane within a network device [4]. It provides the abstraction of a pipeline of multiple stages within the switch that every packet has to go through.

The P4 programming language is designed with the restrictions of network switches in mind. Only the most basic data types are defined in P4. Loops are not allowed, as this would stall the stream of packets. All memory that will be allocated, must be known at compile time, so that the switch can perform as efficiently as possible. To support more sophisticated data structures, like registers, P4 allows manufacturers of network switches to define extern objects. These objects can be used by the programmer as well, but the availability of these depend on the hardware switch.

A Conflict-free Replicated Data Type (CRDT) is data structure that has useful properties for distributed systems [22]. A CRDT can be replicated among different nodes in the system, where they can be updated in isolation. In case of state-based CRDTs, the replicas are periodically shared among all nodes in the network, where they are merged. The merge and update operations should be commutative, so that given that replicas have received the same updates, they are in the same state (Strong Eventual Consistency) [22].

State-based CRDTs require sharing state periodically and in some CRDTs, the state grows in size. This causes a growing communication overhead. To reduce this overhead, Δ-CRDTs may be used that keep track of the updates in the current time window so that only the updates are shared [26]. The data that is shared is inherently dynamic in size. It is important that the updates are transmitted using a reliable transfer protocol, as a dropped update packet can not be recovered.

A recent paper has introduced a sliding window CRDT for sketches [1]. Their protocol relies on the Δ-state for performing updates. Therefore, the packet that are sent for updates are dynamic in size, which is undesirable for programming in the data plane. However, their solution does provide the possibility of performing point queries, which provides a higher granularity than the range queries that SpiderWin provides.

A recent paper that solves the challenges of synchronization in the data plane among multiple network switches, is SwiSh [29].

In network monitoring, the most recent data is more important than old data, which can be captured by using the sliding window model [9]. The windows of the sliding window protocol used by multiple switches must be synchronized at a precise time. Therefore, it is important to synchronize the time among all network switches in the data plane, for which the Data Plane Time-synchronization Protocol (DPTP) can be used [14]. In this protocol, one switch is designated as the master switch, and stores the reference time. Other switches send a DPTP request to query for the current time. Based on the response of the master switch, the switch can calculate the current time.

6 Conclusion

In this thesis, three research questions were stated; (1) How can window-based network monitoring be performed entirely in the data plane? (2) How can local state of switches be synchronized across all switches in the network? (3) What is a reasonable time interval for a window that is sufficiently large to allow for synchronization?

To answer research question (1), we proposed SpiderWin, an algorithm that monitors the network on a per-window basis, entirely in the data plane. This algorithm was designed within the constraints of P4 and the programmable bmv2 software switch. We prototype SpiderWin and show how window-based network monitoring can be performed entirely in the data plane.

To answer research question (2), we propose the synchronization protocol of SpiderWin, that synchronizes on a per-window basis. Once a window has been advanced, it becomes available for synchronization. Each switch sends its local data to all other switches, where the data is merged into the sketch in the window that is being synchronized. SpiderWin keeps track of what data has been included in the window, making it clear when it is fully synchronized.

Combining window-based monitoring with the per-window synchronization protocol, SpiderWin allows for network monitoring with access to recent history. This distinguishes SpiderWin from existing solutions.

By decoupling the window advancement protocol from the synchronization protocol, it is possible periodically advance the window. However, if the interval between time windows is too small, the synchronization will not keep up and if it is too large, the windows may not be useful. To find a reasonable time interval to answer research question (3), we ran experiments on the bmv2 software switch to find out how long it takes for one window to be synchronized. We found that for a fully connected network with 16 switches, the time interval to advance a window should be at least 0.5 seconds. This time does differ per network configuration, and an experiment should be conducted to find the optimal time interval for a specific network configuration.

We also found that the software switch spends around half the time of fully synchronizing a window on merge operations, which is only part of the protocol. The merge operations are designed such that only one read and one write is performed to a register array. A hardware switch should be optimized to do this in parallel. Therefore, we assume that the time it takes for a window to be synchronized must be significantly smaller. This also means that the minimum time interval of window advancement can be smaller.

References

[1] Dolev Adas and Roy Friedman. Sliding window crdt sketches. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 288–298, 2021.

[2] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

[3] Github P4 BMV2DevTeam. Behavioral model (bmv2). `https://github.com/p4lang/behavioral-model`, 2022.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[6] Benoit Claise, Ganesh Sadasivan, Vamsi Valluri, and Martin Djernaes. Cisco systems netflow services export version 9. 2004.

[7] Mininet Project Contributers. Mininet, an instant virtual network on your laptop (or other pc). `http://mininet.org/`. Accessed: 2022-06-03.

[8] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[9] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31:1794–1813, 09 2002.

[10] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, 2003.

[11] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 576–590, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Intel. P4 intel tofino native architecture - public version. `https://github.com/barefootnetworks/Open-Tofino/blob/`

`b9e57726204f44b10f11cb06c001a3a715c8ba45/PUBLIC_Tofino-Native-Arch.pdf`, 2021.

[14] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 8–20, New York, NY, USA, 2019. Association for Computing Machinery.

[15] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 – 1775, october 2011.

[16] Ramana Rao Kompella and Cristian Estan. The power of slicing in internet flow measurement. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 9–9, 2005.

[17] Daniel Kopp, Christoph Dietzel, and Oliver Hohlfeld. Ddos never dies? an IXP perspective on ddos amplification attacks. *CoRR*, abs/2103.04443, 2021.

[18] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 101–114, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security Symposium*, 2021.

[20] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[21] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. csamp: A system for network-wide flow monitoring. 2008.

[22] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] Kulvinder Singh and Ajit Singh. Memcached ddos exploits: Operations, vulnerabilities, preventions and mitigations. In *2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS)*, pages 171–179, 2018.

[24] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, page 164–176, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Anna Sperotto, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller. An overview of ip flow-based intrusion detection. *IEEE Communications Surveys Tutorials*, 12(3):343–356, 2010.

[26] Albert van der Linde, João Leitão, and Nuno Preguiça. δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 29–42, USA, 2013. USENIX Association.

[28] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.

[29] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. Swish: Distributed shared state abstractions for programmable switches. In *USENIX NSDI 2022*. USENIX, April 2022.

[30] Haiquan Zhao, Ashwin Lall, Mitsunori Ogihara, and Jun Xu. Global iceberg detection over distributed data streams. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 557–568, 2010.