



## UvA-DARE (Digital Academic Repository)

### Tracking container network connections in a Digital Data Marketplace with P4

Shakeri, S.; Veen, L.; Grosso, P.

**DOI**

[10.1109/CITS55221.2022.9832915](https://doi.org/10.1109/CITS55221.2022.9832915)

**Publication date**

2022

**Document Version**

Final published version

**Published in**

IEEE CITS 2022

**License**

Article 25fa Dutch Copyright Act (<https://www.openaccess.nl/en/policies/open-access-in-dutch-copyright-law-taverne-amendment>)

[Link to publication](#)

**Citation for published version (APA):**

Shakeri, S., Veen, L., & Grosso, P. (2022). Tracking container network connections in a Digital Data Marketplace with P4. In M. S. Obaidat, G. A. Tsihrantzis, M. Virvou, K.-F. Hsiao, P. Nicopolitidis, & Y. Guo (Eds.), *IEEE CITS 2022: proceedings of the 2022 IEEE International Conference on Computer, Information, and Telecommunication Systems, CITS 2022 : July 13-15, 2022* (pp. 7-14). IEEE. <https://doi.org/10.1109/CITS55221.2022.9832915>

**General rights**

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

**Disclaimer/Complaints regulations**

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

# Tracking container network connections in a Digital Data Marketplace with P4

Sara Shakeri

Multiscale Networked Systems group  
University of Amsterdam  
Amsterdam, The Netherlands  
s.shakeri@uva.nl

Lourens Veen

Netherlands eScience Center  
Amsterdam, The Netherlands  
l.veen@esciencecenter.nl

Paola Grosso

Multiscale Networked Systems group  
University of Amsterdam  
Amsterdam, The Netherlands  
p.grosso@uva.nl

**Abstract**—There are multiple organizations interested in sharing their data, and they can only do this if a secure platform for data sharing is available which can execute sharing requests under specific agreements and policies. Digital Data Marketplaces (DDMs) aim to provide such an infrastructure. For building a DDM infrastructure, we use containers to provide the required isolation between different sharing requests. However, one important challenge in a containerized DDM infrastructure is providing the ability to monitor the behavior of containers that are involved in the sharing transactions. In addition, the monitoring information in the network layer should be reported in a way that can be interpreted by the upper layers of DDM for further analysis. In this paper, we design a containerized DDM using P4. In our design, the flow traffic between containers is associated with the shared data in a DDM and can be understood by the upper layers. We present different scenarios to demonstrate how our setup can assist in tracking the behavior of containers and providing better performance and security.

**Index Terms**—Digital Data Marketplaces (DDMs), Containers, P4 language, Flow tracking

## I. INTRODUCTION

There is an increasing demand for data sharing among organizations in both science and industry. Digital Data Marketplaces (DDMs) aim to meet these requirements and provide a framework that guarantees secure data exchange [17]. DDMs currently in development support downloading data from other parties, but also federated machine learning and other forms of distributed data processing. In these systems, data sets and software are exchanged, but the software is also executed, and data may be accessed remotely. A primary consideration of a DDM is to implement the agreements between organizations concerning who can access which asset. We call these agreements *DDM policies*. The *policy driven data exchange management system* of the DDM is in charge of checking the consistency of the user's sharing requests with the available policies and then planning the execution in the network infrastructure. The network infrastructure of a DDM has to provide secure connections between participating parties for data exchange and to support the secure execution of distributed algorithms. Each sharing request must be isolated from other requests to 1) protect the shared asset from policy-

violating requests and 2) be able to enforce the policies for each sharing request independently of each other.

In previous work, we have constructed an infrastructure that supports multi-domain data sharing through the use of containers and P4 switches [8]. To execute a *sharing request*, containers in domains controlled by different participating parties are connected to each other to share the assets. Containers provide isolation between requests, while the connection policies are enforced by programming the P4 switches. In addition, P4 programs can be run on Smart Network Interface Cards (SmartNICs) to offload the network processing tasks from server CPUs to hardware [5]. As a result, the total system throughput improves by freeing up the server CPU cycles of the server. We presented more details of the implementation in [14].

DDMs that support distributed applications such as federated machine learning need to run software made by third parties or even users of the system. This presents a security risk. Besides sandboxing and enforcing network connectivity policies, two other ways exist to mitigate this risk: code inspection and monitoring. Code inspections can detect bugs and malicious code, but they cannot do so perfectly; they are time-consuming, and sufficient expertise needs to be available. Furthermore, measures need to be taken to ensure that the code that runs matches exactly the code that was inspected, which entails repeatable builds and re-reviewing every new release. As a result, code inspections are expensive and cumbersome. Monitoring execution may provide an alternative or addition. Monitoring can be automated completely using AI, can deal with small software upgrades, and the potential for catching malicious behavior will deter would-be adversaries and increase trust in the system even if it is not guaranteed to catch all malicious behavior.

Besides, monitoring can help to improve performance. As the size of a DDM increases, more data must be shared with more peers, and scalability of the system becomes important. In a containerized DDM, where both data sets (*data assets*) and software (*compute assets*) are containers, scalability can be achieved by adding more instances (horizontal scaling), if the network infrastructure is flexible enough to route incoming requests accordingly. To steer this process and select the method

of scaling, the container's transactions must be tracked.

In this paper, we monitor the network traffic between containers containing software or data. Since our infrastructure uses P4 switches, monitoring data can be gathered by P4 programs running on the switches. The sharing transactions can be logged in the switch and then sent to the controller or upper layers for further analysis. However, tracking the transactions related to the assets is not trivial. The traffic that is being logged should be related to each asset, i.e., what is understandable for the data exchange management system. Therefore, just the information about the traffic of different flows in the network is not helpful by itself. A mechanism should connect the traffic flows to the asset and their corresponding policies.

We design a P4-based containerized network that can handle these challenges. We specify a unique ID for each connection, and we use this ID in network connections for transferring the assets. By using this connection ID, we can track the transactions related to each asset in the DDM. We present sharing scenarios in a DDM to demonstrate how connection tracking can help in providing security and improving performance.

The main contributions of the paper are:

- Building a containerized DDM in which the flow traffic in the network infrastructure is associated with the shared assets in DDM
- Demonstrating how the sharing transactions can be tracked and monitored in that infrastructure
- Presenting examples of the tracking scenarios to show how logging information by P4 can provide better security and performance in DDM

## II. BACKGROUND

This section gives a brief introduction about data sharing workflows in a DDM to describe the mechanism of data sharing, i.e., how the sharing requests are executed. We then explain P4 programs and how they are helpful for application in a DDM network infrastructure.

### A. Data sharing workflows:

Different types of digital data marketplaces are currently in development, including centralized download sites, peer-to-peer data exchange systems, and designs for federated learning and other kinds of distributed data processing. These data exchange and processing patterns can all be expressed as workflows, as done for example in the Mahiru data exchange system [15]. Figure 1 shows an example of a workflow in a DDM that consists of multiple *sharing requests*. In each sharing request, an asset will be transferred from one domain to the other one over the network. The figure shows that a data asset from domain1 and a data asset from domain2 have to be transferred to domain3 for computation. The output of this computation is accessed in another sharing request as an input data asset.

In a DDM, each shared asset has a specific ID. All of the sharing policies are based on these IDs [13]. The data exchange management system is in charge of handling the policies,

allowing or rejecting a workflow, and planning the execution steps of the accepted workflows in different domains.

### B. P4 language:

The P4 language is used for processing packets by the data plane. It is a protocol independent language as it provides the possibility for defining new headers as desired and making new protocols.

The main P4 language components are headers, parsers, P4 tables, and actions [6]. Headers define the set of fields and their size. Parsers determine the sequence of the headers and identify header fields by extracting the packets. P4 tables contain match-action and keys. The search key is matched with the packet fields or metadata. For each matching key, there is an action. Actions determine common modifications to packets, such as changing header fields or adding or removing headers.

In P4, it is possible to gather telemetry metadata for each packet, such as ingress and egress timestamps, the latency of the packet, the link utilization, and the routing path of the packet [4]. Information can also be embedded into the packets and removed at the endpoints. P4 supports constants, variables, and also registers. Variables have local scope, and they do not keep any state. Registers keep the state between various network packets.

### C. P4-based DDM:

We use P4 to construct a *multi-domain* DDM, provide *programmability* in its network infrastructure, and improve its *security*. DDM is a multi-domain environment, and each domain has to be able to manage its configuration. In P4-based DDM, each domain administrator can control its container's connectivity by communicating with the switch controller.

In addition, any connection and filtering rule between containers can be programmed dynamically. For example, because of the failure of container service, the route may need to be changed from one container to the other. There may also be a need for an update in the policies of the DDM to remove access. This all can be done by updating the rules of the related connection. The other connections are not affected by these changes.

Moreover, in our setup, there is no connectivity between containers before setting up the rules in the switch. Every single container's connectivity has to be controlled by the switch. The rules are based on the connection ID, a unique identifier that is dedicated to each individual connection. Therefore, no other container can make an illegitimate connection.

## III. ARCHITECTURE

Figure 2 shows the architecture of the containerized data sharing system. It is constructed of four main building blocks: the data exchange management system block, the administration block, the containerization and networking block, and the monitoring block.

**Data exchange management system block:** This component is in charge of checking the policies and collaborating with other domains for scheduling a request or a chain of requests.

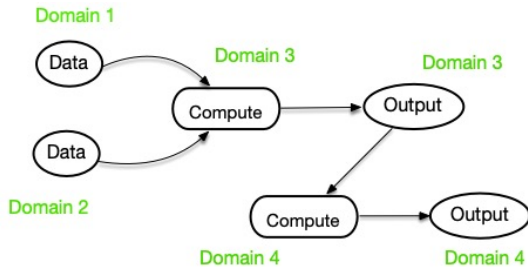


Fig. 1: An example of a sharing transaction in a DDM

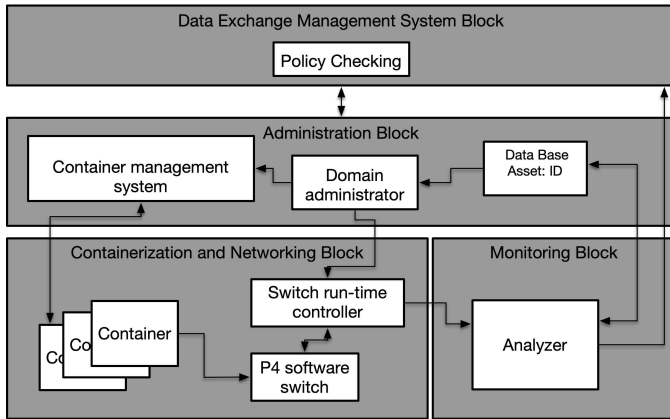


Fig. 2: Architecture of the containerized DDM using P4 for tracking capability

**Administration block:** This block contains three components. The container management system creates the container and attaches the related asset. Containers containing a data asset are called *data containers*, and containers containing a compute asset are called *compute containers*. The containers are then connected to the switch.

The unique connection ID related to each connection is generated and then saved in a database. A container can access an asset only if it has the connection ID. This connection ID will then be used for tracking the sharing transactions. Finally, there is a domain administrator. It takes the information about the created container from the container management system and the connection ID from the database. It then creates the appropriate rules that make the legal connections to this asset possible. These rules are passed to the controller.

**Containerization and networking block:** In this block, the P4 switch connects the containers. When the packets arrive with the specific connection ID, they are matched with the rules that are set by the controller and sent to the corresponding container. Importantly, the information of the flows going through the switch will be saved in the switch's registers. The controller is in charge of setting up the rules on the switch. In addition, it reads the information from the registers of the switch when it is needed and sends it to the analyzer.

**Monitoring block:** In this component, the monitoring information will be translated from the networking flow information to *asset* access information. Of course, it uses the connection

ID mapping database to relate the flows with specific connection IDs to assets. The information can be reported to the data exchange management system or domain administrator for further analysis.

#### IV. PROOF OF CONCEPT

For our implementation, we used Ubuntu 18.04 and Linux kernel 5.4.0 as the host OS, Docker Community Edition 20.10 for container management, and bmv2 P4 switch as the programmable software switch [2].

Figure 3 shows the steps for setting up the network for transferring a data asset from a data container in domain2 to a compute container in domain1. We call domain1 that is requesting the data the *client-side* and domain2 that is serving the data the *server-side*. Domains are distinguished by their IP addresses. The IP address is the IP of the server hosting the container.

In the following, we will guide the reader step by step through each of the operations that occur in a domain to set up the connection between containers:

**step1:** The request from the other domain will be checked against the available policies in the data exchange management system. If the request is allowed, the data exchange management system sends the request to the domain administrator to set up the network for the required connection.

**steps 2 and 3:** The domain administrator asks the ID generator to generate the ID for the connection for transferring the data asset from the data container to the compute container in step 2. The connection ID will be sent back to the administrator in step 3. The connection ID is a unique number for each Data asset ID, client-side IP address, and Compute asset ID:

$$\text{Connection ID} = \text{Unique number of}(\text{DataAssetID}, \text{Client-sideIPaddress}, \text{ComputeAssetID})$$

**step4:** The domain administrator asks the container management system to create a container for the Data Asset. The container management system then sends back the container specification to the domain administrator.

**step5:** The domain administrator connects the container to one of the ports of the switch.

**step6:** The domain administrator creates the appropriate rules and sends them to the controller of the switch for setup. In the connection from the client-side to the server-side, the connection ID will be used as the destination port of the connection.

In the following, we explain the rules of the server and client-side.

**server-side:** Listing 1 shows the rules and actions for incoming packets to the server-side in our setup. In this case, the packet's destination port number (the connection ID), and its source IP address - which is the client-side IP address - will be checked (line 4-5).

We first save the destination port number (connection ID) in a variable for further use (line 14). For sending the packet to the server-side container, we have to change the destination IP address to the server-side container's IP address (line 17) and the port number to the number that the server-side container

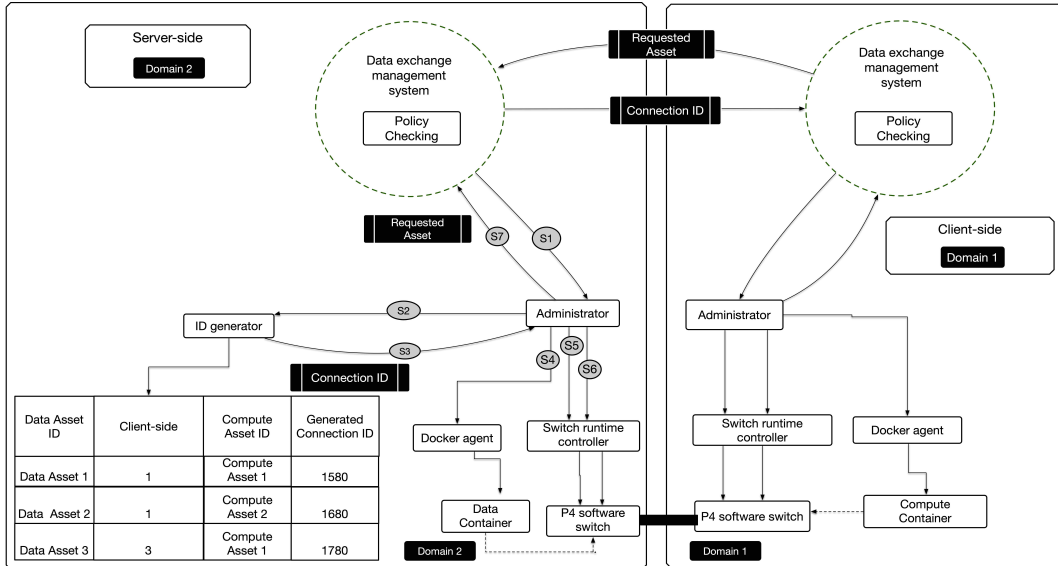


Fig. 3: Steps of setting up the network for a request from client-side to server-side

is listening to (line 18). At this point, the packet is ready to be sent.

When a packet comes back, we have to be able to set the port number and IP address back to the original one. Therefore, before sending the packet to the server-side container, we save the original source port in a register (line 15) and modify the source port number of the packet to the previously saved connection ID (line 16).

```

1
2 table server_receive_t {
3   key = {
4     hdr.tcp.dstPort :exact;
5     hdr.ipv4.srcAddr : exact;
6
7   }
8   actions = {server_receive; }
9 }
10
11
12 action server_receive(bit<32> dst_ip, bit<16>
13   dst_port, bit<9> port){
14
15   Connection_ID_s = hdr.tcp.dstPort;
16   reg_srcport.write(Connection_ID_s ,
17   hdr.tcp.srcPort);
18   hdr.tcp.srcPort = Connection_ID_s;
19   hdr.ipv4.dstAddr = dst_ip;
20   hdr.tcp.dstPort = dst_port;
21   stdmeta.egress_spec = port;
22 }

```

Listing 1: P4 table and actions related to the incoming packets in the server-side

Listing 2 shows the rules and actions for outgoing packets from the server-side to the client-side. In this case, the packet's destination port number (the connection ID) and its destination IP address, i.e., the client-side IP address, will be checked (line 4-5).

The destination port of the current packet is saved in a variable (line 12). When a packet is coming back from the server-side container, we have to change its source IP address

to the IP address of the server-side (line 13). We then have to change the source and destination port number. We change the source port number to the connection ID that we saved in the variable (line 14) and the destination port to what we have saved in the register (line 15). Finally, the packet will be sent out to the related egress port of the switch (line 16).

```

1
2 table server_send_t {
3   key = {
4     hdr.tcp.dstPort :exact;
5     hdr.ipv4.dstAddr :exact;
6   }
7   actions = {server_send; }
8 }
9
10 action server_send(bit<32> src_ip, bit<9> port){
11
12   Connection_ID_c = hdr.tcp.dstPort;
13   hdr.ipv4.srcAddr = src_ip;
14   hdr.tcp.srcPort = Connection_ID_c;
15   reg_srcport.read(hdr.tcp.dstPort, Connection_ID_c);
16   stdmeta.egress_spec = port;
17 }

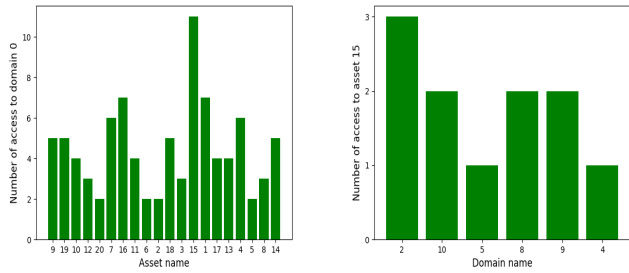
```

Listing 2: P4 table and actions related to the outgoing packets from the server-side

**Client-side:** On the client-side, the routing between containers is more straightforward. When a packet is sent out to the server-side, we change its source IP address to the host IP address. When a packet comes back from the server-side, it can be directed to the right container by looking at the connection ID (source port). However, as different domains may give the same connection ID for different requests related to the same host, the connection ID may be used by another server-side host. Therefore, what is unique is the combination of the source IP address and the connection ID, and that will be used for matching rules.

## V. TRACKING SCENARIOS

We evaluated the use of P4 switches in Digital Data Marketplaces in two different cases corresponding to the two concerns



(a) Number of access to the data assets (b) Number of access to asset15 from different domains

Fig. 4: Asset access tracking in a DDM using connection ID

raised in the introduction: scalability and security. For the scalability case, we consider a DDM in which containerized data is accessed via the network. A sharing request in this case, could be a dataset download or a database query. If a DDM participant has a popular dataset, or the DDM contains another participant with a large number of users that access many datasets simultaneously, then the system of the data provider must scale to meet demand, which must be measured, and the network reconfigured accordingly. For the security case, we focus on distributed data processing, in particular federated learning. In this use case, data transfer needs to be repeated multiple times. Therefore, the compute container keeps the connection open and asks for data whenever it is needed within the same session. In this type of connection, there will be an *active\_time* where the packets are being transferred and an *idle\_time* where no packet is being transferred, but the connection is open.

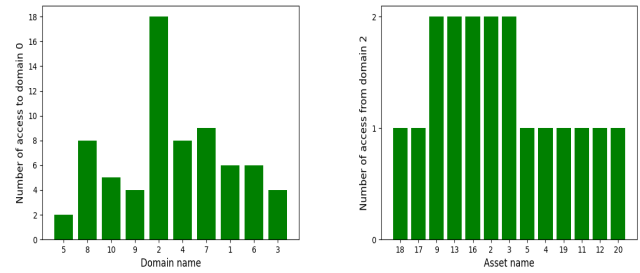
#### A. Access tracking

This case allows us to track the access behavior of the domains in a DDM to a specific data asset. This information can help in enhancing the performance by scaling the number of containers. When the load on the system is high, new containers are generated, and the load is distributed between these containers. In such a case, the containers that have the most effect on the load have to be selected to be duplicated in another server. After duplication, part of the load should be rerouted to these new containers.

We look at two values to select which containers should be duplicated and run on another server. The first is the number of access to each asset, and the second is the number of access from each domain. The calculations are based on the fact that by looking at the connection ID we can identify the client-side's domain, the client-side's compute container and the server-side's data container.

Accordingly, there are two scenarios for tracking the access. 1) asset access; 2) domain access; which are explained in the following. In addition, we explain a method for detecting the high load on the system at the end of this section.

- 1) **Tracking based on asset access:** In this case, we count the number of access to different data assets on the



(a) Number of access from the other domains to domain0 (b) Number of access to data assets from domain2 to domain0

Fig. 5: Domain access tracking in a DDM using connection\_ID

server-side to determine which asset or assets have the most access. To count the number of access to each data asset, we save the number of connections to data containers in the switch indexing by the connection ID. We then calculate the number of connections to the data containers from different domains. If the number of connections of a data container is more than a certain threshold, we select that data container for duplication. To illustrate with an example how this would work, we simulated a scenario of running sharing requests by considering 10 different client-side domains accessing 20 data assets in a server-side domain (Figure 4). We set the average access to assets as the threshold. We counted the number of access to each asset by relating the connection ID to its asset (Figure 4a). According to Figure 4a, the average number of access from different domains to all data assets is 7. The number of access to asset15 is more than the average. Therefore, it will be selected for duplication. We then have to select which domains requests have to be rerouted to the new container. This can be done by looking at the number of access of each domain to the selected asset as shown in Figure 4b. We divide the load related to asset15 between two containers by sorting the domains based on their number of access and then assigning each domain to one of the two containers from the beginning of the sorted list.

- 2) **Tracking based on domain access:** Another reason for a high load on the system can be due to lots of connections from client-side domain to server-side domain but not necessarily to one individual data container. A client-side domain may make connections to different data containers of the server-side domain so that the number of connections to a specific container is not high; however, the total number of connections related to this client-side domain is high, leading to increased load on the system.

In this case, we simulated a scenario of running the sharing requests from 10 different client-side domains to server-side domain0 (Figure 5). In Figure 5a, the

most number of access is from domain2. For distributing the load, by detecting the most referring assets of this domain (Figure 5b), we can duplicate the containers and reroute the connections of this domain to the new containers.

**Detecting high load:** To detect the high load on the system, we can consider the total time of a data transfer and the number of running connections as metrics. This method works when the connections that are made by the application are always active. In other words, the packets are being transferred continuously between SYN and FIN time. The thresholds in this method are application-specific and should be set accordingly.

The total time of a sharing request can be measured in a P4 program by calculating the time difference between the SYN and the FIN flags of a connection. In P4, whenever a SYN packet of a connection is seen, its ingress time will be saved in a register with the index of its connection ID. The total time is then saved in another register when the FIN flag of that connection arrives. When the total time is calculated, it will be compared with the expected total time of the connection that is set by the controller. If it is more than what is expected, it will be counted as the connections that their total time is more than the threshold. When the number of requests with the total time of more than a threshold crosses a specific number (that is determined by the controller), the switch sends a notification to the controller as a situation where the load on the system is high.

When the controller receives a notification from the switch, it also reads the register that has the number of running connections. The switch keeps the number of running connections in the system by counting the number of connections that their SYN flag is seen, but their FIN is not. The combination of the high number of running connections and long total time is a sign that can be considered for when the system is under high load. The P4 code of this implementation is available on GitHub [1].

### B. Pattern tracking

The last illustrative scenario is about detecting a specific pattern in the activities related to the connection from a compute asset to a data asset. There are lots of algorithms whose behavior in making the connections and transferring the data has a specific pattern. By detecting this pattern, the malicious behaviors can be promptly identified to reduce the risk of data leakage. In addition, the state of the system, like the number of incoming requests and their corresponding load, can be predicted. Our goal is to detect the pattern, and we perform it by programming the switch and using the connection ID.

As an example, we consider a federated machine learning algorithms' behavior. When the data has to stay in its location and cannot be moved, federated machine learning algorithms are deployed for performing the computation on data. Therefore, each part of data is in a location that is different from the other, and the model will be trained in a location where

data resides [3], [12]. After training the model, the model parameters need to be transferred to where all the information is gathered, and the general model can be built [11]. This process is repeated multiple times so the training process is complete. In this case, the compute container on client-side keeps the connection open for better optimization. Therefore, sometimes the connection is active and the data is transferred, and sometimes it is idle. If this process is run in our setup, what we observe on the server-side is a pattern of connections from the client-side to a specific data container. In these connections, the same amount of data is transferred every time, and they repeat with a specific time interval.

For detecting the pattern, we have to be able to detect the `idle_time`, the `active_time`, and the amount of data that is transferred between containers in `active_time`. We use a list of registers in the P4 program for finding these numbers. We define the registers and index them with the connection ID. The registers are:

- 1) `Last_packet_ingress_time < 48bit >`: This register always keeps the arrival time of the last packet.
- 2) `Start_idle_time < 48bit >`: This register keeps the start time of an `idle_time`.
- 3) `End_idle_time < 48bit >`: This register keeps the end time of an `idle_time`.
- 4) `Total_data < 48bit >`: This register keeps the total amount of data that is being transferred from the first packet to the last packet of the connection.
- 5) `Data_before_max_idle_time < 48bit >`: This register keeps the amount of data that is being transferred before the maximum idle time.
- 6) `Time_interval < 48bit >`: This register keeps the maximum duration of the `idle_time`.

The algorithm we implemented in the switch to detect the `idle_time` is shown in Algorithm 1. Whenever a packet is coming to the switch, the difference between the arrival time of the packet and the arrival time of the last packet will be saved in a local variable in the P4 program. If the current difference is more than the previous difference that has been saved in `time_interval` register, then the `time_interval` will be updated. The transferred data will be updated accordingly.

The registers are read by the controller repeatedly. However, in this method, only the maximum `idle_time` is saved. To avoid that, every time the controller reads the registers, it sets the `time_interval` to zero. As a result, different duration of `idle_time` can be detected.

We performed an experiment to illustrate the pattern tracking method. In our experiment, a compute container was running on the client-side in a physical machine. A data container was running on the server-side in another physical machine. The physical machines were connected via the internet. We generated traffic that follows a specific pattern. The same amount of data was transferred through the network every 5 seconds within an open connection.

Table I shows the content of registers every time the controller reads the information from the switch. The polling time is when the controller reads the registers that is in every

TABLE I: P4 register's content for finding the pattern of the connection from a compute container to a data container

Polling_number	Polling_time	Start_idle_time	End_idle_time	Time_interval	Last_packet_ingress_time	Data_before_max_idle_time
1	174,59	173,64	173,65	0,01	174,59	60
2	177,94	173,64	173,65	0,00	174,59	60
3	181,25	174,59	179,33	4,74	180,24	1086475
4	184,59	174,59	179,33	0,00	180,24	1086475
5	187,91	180,24	184,88	4,64	185,75	2172751
6	191,22	185,75	190,33	4,57	191,22	3259027

**Algorithm 1** Finding the maximum idle\_time of a TCP connection in P4

**Input:** Connection ID, Total\_data

**Output:** Time\_interval, Start\_idle\_time, End\_idle\_time, Data\_before\_max\_idle\_time

**Define** Last\_time, Current\_time, Difference

Last\_time = Last\_packet\_ingress\_time[Connection ID]

Current\_time = packet.metadata.ingress\_time

Difference = Current\_time - Last\_time

**if** Difference > Time\_interval[Connection ID] **then**

Update Time\_interval[Connection ID] with Difference

Update Start\_idle\_time[Connection ID] with Last\_time

Update End\_idle\_time[Connection ID] with Current\_time

Update Data\_before\_max\_idle\_time[Connection ID] with

Total\_data[Connection ID]

**end if**

update Last\_packet\_ingress\_time[Connection ID] with packet.metadata.ingress\_time

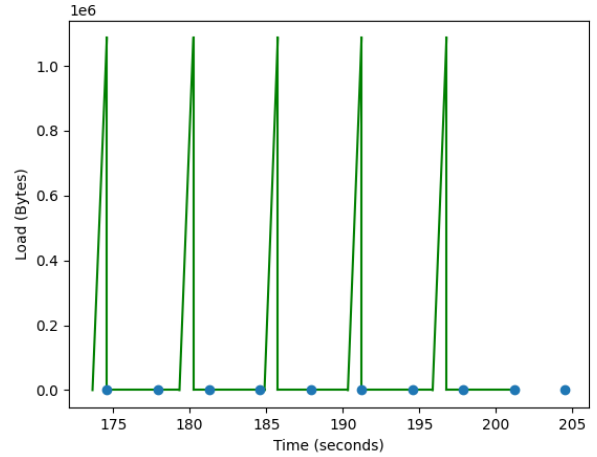


Fig. 6: Pattern Tracking: active and idle\_time of a connection from a compute container to a data container. The points show the time the information is read by the controller. The numbers of each reading time are shown in Table I.

3 seconds. Note that the time\_interval was set to zero after the controller read the registers. The times in the table are the switch's time; that is, the time from when it was started in seconds.

According to Table I, for finding the active\_time and idle\_time after reading the registers, if the Start\_idle\_time was equal to the previous one, we ignored that row of information. Among the remaining rows, we considered the time between Start\_idle\_time and End\_idle\_time of a row as idle\_time and the End\_idle\_time of a row and Start\_idle\_time of the next row as the active\_time. Figure 6 shows the pattern based on the numbers in the table. The blue points in the figure are the polling time.

In the second row of the table, the time\_interval is 0.0; that is because after the previous time that it was set to zero, no packet came, so it has not changed. This is the second point in Figure 6. In the figure, we show the load that is transferred each time based on Table I. The load that is shown in the table is the accumulative load that is read related to each connection ID.

The polling time should be less than the sum of active\_time and idle\_time. Therefore, it should be justified by the behavior of the compute asset. Polling time can be set with a small number at the start. After realizing the first active and idle\_time, we update the polling time. We then adjust the pulling time every time we read the information.

## VI. RELATED WORK

Multiple works have been done in the area of network flow monitoring in programmable data planes. For example, in [16] the authors design and implement a bandwidth-efficient in-band network telemetry system that can track the rules matched by the packets of a flow. They use a traffic reduction scheme in their INT system to reduce the rate of generated INT reports. They also store INT reports about the changes in the rule by using global unique IDs for every rule. Also, in [20] authors use hash tables to maintain the whole information about elephant flows, but summarize records for mice flows, by applying a novel collision resolution and record promotion strategy to hash tables. We use the connection ID for saving the information and, in addition, set it as the destination port number of the connection to reduce the overhead. In our methodology, we show how using the connection ID as the destination port number of the connections is possible.

[7] proposes FlowLens. It is a system for traffic classification to support security in network applications based on machine learning algorithms. The authors propose a novel memory-efficient representation for features of flows called flow marker. A profiler running in the control plane automatically generates an application specific flow marker that optimizes the trade-off between resource consumption and

classification accuracy according to a given criterion selected by the operator. The authors in [9] try to design a low overhead system for monitoring and gathering the information by deploying a P4 program. The monitoring phase in their setup includes a proactive phase that keeps the per-flow packet counter and a reactive phase that runs for large flows only and gathers metrics of the flow, e.g., packet counts and packet timestamps. Our proposed method is specifically designed for data sharing applications. It builds a multi-domain container-based DDM that provides the possibility of tracking the transactions related to the shared assets in the network layer.

In addition, there are some works that propose the methods of monitoring in a DDM. For example, in [18] the authors propose an architecture to distinguish programs running inside containers by monitoring system calls. [19] offers an intrusion detection system based on OC-SVM that monitors and analyzes system calls of containers. [10] designs an auditor node in containerized DDM that is a node responsible for authorization through signatures application transactions. These methods focus on the operation of the containers in system calls and on the application layer. Our method focuses on monitoring the data exchange behavior of containers in the network layer through P4 programming. In this way, extra information like the amount of transferred data or the number of connections that have been made during a specific time can be extracted to make them available to the application layer.

## VII. CONCLUSION

In this paper, we improved the architecture of a containerized DDM using P4 by adding the capability of tracking the transactions related to the shared assets. We introduced a sharing request as a connection between two containers in two different domains for sharing a specific asset. We proposed using a unique connection ID for each sharing request that is related to the shared asset in that request. The connection ID is used as the destination port of the connection between containers. We used P4 to program the switch that connects the container and make the connections based on the connection ID. We explained each step of setting up the network and showed the required configuration. We then demonstrated how we can use the connection ID for tracking the sharing transactions between containers and, accordingly, the transactions related to each shared asset. We presented asset access, domain access, and pattern tracking scenarios to show what kinds of information we can extract from the switch based on our setup and how to use this information to provide better performance and security in a DDM. Our future work focuses on implementing more tracking scenarios, like bandwidth management between containers utilizing the logging information we can gather in P4 and the network programmability.

## ACKNOWLEDGMENT

This work is supported by the Netherlands eScience Center and NWO under the project SecConNet.

## REFERENCES

- [1] "Access tracking in P4," <https://github.com/sarashakeri/SecConNet-tracking/>, 2022, [Online; accessed April 2022].
- [2] "Behavioral Model," <https://github.com/p4lang/behavioral-model>, 2022, [Online; accessed April 2022].
- [3] "IBM Federated Learning," <https://github.com/IBM/federated-learning-lib>, 2022, [Online; accessed April 2022].
- [4] "Improving Network Monitoring and Management with Programmable Data Planes," <https://opennetworking.org/news-and-events/blog/improving-network-monitoring-and-management-with-programmable-data-planes/>, 2022, [Online; accessed April 2022].
- [5] "P4 Data Plane Programming," [https://www.netronome.com/media/documents/WP\\_P4\\_Data\\_Plane\\_Programming.pdf](https://www.netronome.com/media/documents/WP_P4_Data_Plane_Programming.pdf), 2022, [Online; accessed April 2022].
- [6] "P4: Open Source Programming Language," <https://p4.org/>, 2022, [Online; accessed April 2022].
- [7] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ml-based network security applications," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021, pp. –.
- [8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, jul 2014. [Online]. Available: <https://doi.org/10.1145/2656877.2656890>
- [9] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho, "Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.
- [10] R. Cushing, R. Koning, L. Zhang, C. d. Laat, and P. Grosso, "Auditable secure network overlays for multi-domain distributed applications," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 658–660.
- [11] A. Durrant, M. Markovic, D. Matthews, D. May, J. A. Enright, and G. Leontidis, "The role of cross-silo federated learning in facilitating data sharing in the agri-food sector," *CoRR*, vol. abs/2104.07468, 2021. [Online]. Available: <https://arxiv.org/abs/2104.07468>
- [12] K. V. Sarma, S. Harmon, T. Sanford, H. R. Roth, Z. Xu, J. Tetreault, D. Xu, M. G. Flores, A. G. Raman, R. Kulkarni, B. J. Wood, P. L. Choyke, A. M. Priester, L. S. Marks, S. S. Raman, D. Enzmann, B. Turkbey, W. Speier, and C. W. Arnold, "Federated learning improves site performance in multicenter deep learning without data sharing," *Journal of the American Medical Informatics Association*, vol. 28, no. 6, pp. 1259–1264, 02 2021. [Online]. Available: <https://doi.org/10.1093/jamia/ocaa341>
- [13] S. Shakeri, V. Maccatrozzo, L. Veen, R. Bakhshi, L. Gommans, C. de Laat, and P. Grosso, "Modeling and matching digital data marketplace policies," in *2019 15th International Conference on eScience (eScience)*, 2019, pp. 570–577.
- [14] S. Shakeri, L. Veen, and P. Grosso, "Multi-domain network infrastructure based on p4 programmable devices for digital data marketplaces," *Cluster Computing*, Jan 2022. [Online]. Available: <https://doi.org/10.1007/s10586-021-03501-2>
- [15] L. Veen, S. Shakeri, and P. Grosso, "Secure data sharing and distributed processing with Mahiru," 2022, Poster presented at ICT.Open 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.6497704>
- [16] S.-Y. Wang, Y.-R. Chen, J.-Y. Li, H.-W. Hu, J.-A. Tsai, and Y.-B. Lin, "A bandwidth-efficient int system for tracking the rules matched by the packets of a flow," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.
- [17] L. Zhang, R. Cushing, L. Gommans, C. De Laat, and P. Grosso, "Modeling of collaboration archetypes in digital market places," *IEEE Access*, vol. 7, pp. 102 689–102 700, 2019.
- [18] L. Zhang, R. Cushing, R. Koning, C. de Laat, and P. Grosso, "Profiling and discriminating of containerized ml applications in digital data marketplaces (ddm)," in *ICISSP*, 2021, pp. 508–515.
- [19] L. Zhang, R. Cushing, C. d. Laat, and P. Grosso, "A real-time intrusion detection system based on oc-svm for containerized applications," in *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*, 2021, pp. 138–145.
- [20] Z. Zhao, X. Shi, X. Yin, and Z. Wang, "Hashflow for better flow record collection," 2018. [Online]. Available: <https://arxiv.org/abs/1812.01846>