Multi-domain network infrastructure based on P4 programmable devices for Digital Data Marketplaces

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

DOI
10.1007/s10586-021-03501-2

Publication date
2022

Document Version
Final published version

Published in
Cluster Computing

Cluster Computing (2022) 25:2953–2966
https://doi.org/10.1007/s10586-021-03501-2

Multi-domain network infrastructure based on P4 programmable devices for Digital Data Marketplaces

Sara Shakeri[1] · Lourens Veen[2] · Paola Grosso[1]

Received: 30 April 2021 / Revised: 21 September 2021 / Accepted: 24 November 2021 / Published online: 24 January 2022

Abstract
There are many organizations interested in sharing data with others, and they can do this only if a multi-domain secure platform is available. Such platforms, often referred to as Digital Data Marketplaces (DDMs), require that all the transactions follow the pre-defined policies that are established by the participating parties i.e, domains. However, building a multi-domain network infrastructure in which each domain can manage its own connectivity while at the same time all of the transactions follow the sharing agreements is still a challenge. In this paper, we introduce a multi-domain containerized DDM that is built upon a P4-based network. It can handle the communication of multiple domains and guarantee that the operation of transactions is based on the pre-defined policies. We also studied the setup performance by defining a model which we demonstrated follows the real measurements, and we can use for decision making. The results also show the low overhead of using P4 switch in network setup time. In addition, we conducted a security evaluation which showed that our P4-based network setup is secure against most types of attacks.

Keywords Digital Data Marketplaces (DDMs) · P4 program · Programmable networks · Data sharing · Container

1 Introduction

Data sharing is currently emerging as an essential element in fields like healthcare systems [1], supply chain logistics [2], and IoT devices in smart cities [3]. Digital Data Marketplaces (DDMs) [4] are emerging frameworks that provide secure data sharing mechanisms among participating parties. In DDMs, data providers and data consumers exchange access to data for monetary compensation in business settings or as part of open science practices in research environments. In both cases a DDM needs to provide a means of data exchange and/or processing, while implementing and enforcing sharing policies in its infrastructure. In some cases, legal restrictions keep data from being exchanged at all, for example when data are privacy sensitive. Still, even under these constraints, it is possible to combine data from different sources through techniques like distributed machine learning, secure multiparty computation, and homomorphic encryption. Initial implementations of such algorithms are becoming available in systems like Vantage6 [5], PySyft [6], MPyC [7] and IBM's Federated Learning Library [8].

Data exchange entails copying data (sub)sets from one system to another. These DDMs operations lead to very stringent and specific requirements for the underlying network, in terms of security, isolation, and multi-domain operations.

DDMs algorithms are communication intensive, and in order for a DDM to support them efficiently, processes running on different independent systems will need to be able to exchange data directly still maintaining the security level specific to the policies governing the application.

In data exchange, many transactions may be active simultaneously and need to be isolated from one another.

In addition, a DDM is a multi-domain environment. For preserving privacy and providing the required security all

✉ Sara Shakeri
s.shakeri@uva.nl

Lourens Veen
l.veen@esciencecenter.nl

Paola Grosso
p.grosso@uva.nl

[1] Multiscale Networked Systems (MNS) Research Group, University of Amsterdam, Amsterdam, The Netherlands

[2] Netherlands eScience Center, Amsterdam, The Netherlands

the connections related to each participating party, i.e, domain, has to be managed by its own domain administrator.

For building such a DDM we use containers. In fact, containers can provide the required isolation between sharing transactions. When a container is deployed on a host, each container's resources, such as its file system and network namespace, are placed in an isolated environment which no other container can access. In each sharing transaction, containers act on behalf of the participating party to exchange data. Containers belonging to different domains have to be able to connect to each other in a secure manner so data can be transferred from one party to the other.

In our previous work, we showed how we can build a single-domain data sharing platform by using containers [9]. We used both Kubernetes [10] and Docker Swarm [11] as container network orchestrators for creating network overlays between containers and setting up filtering rules between them in order to improve security and performance in the network. However, these methods are not applicable in a multi-domain environment: in all of these frameworks one node should be selected as a master node so it has access to all of the containers in the network and can manage all of the connections. In a multi-domain scenario, each participating party should be able to manage its own domain independently from other domains participating in the sharing platform. Building a container-based DDM network infrastructure that can integrate with the per-domain domain orchestrators is still an open challenge, which we address in this paper.

For building a multi-domain DDM we adopt P4 as the network data plane technology. Deploying P4 programmable switches satisfies our requirements, i.e., per domain orchestration and managing container connections. All the connections between containers will be programmed in such switches and can be handled by the administrator. In addition, the connections can be reconfigured when it is needed.

In Sect. 2, we will provide the motivation for adopting P4, and a background on P4 programs and integration of containers with P4 switches. Then, we will present the DDM architecture (Sect. 3); we will explain how two domains cooperate with each other to set up a connection between containers (Sect. 4). We will extensively discuss the setup time required to get to an operational network in Sect. 5. We will present our proof of concept implementation (Sect. 6) where we show how the network and the whole chain of connections between domain orchestrators and switches work. In Sect. 7 we show the measurements of setup time and discuss the overhead of using P4 switches. The security implications of using P4 as underlying network technology are discussed in Sect. 8. We conclude our article discussing our findings (Sect. 9) and relating our work to previous efforts (Sect. 10).

2 Containerized P4-based DDM

Motivation for P4 adoption For constructing a container-based DDM we follow three main goals:

- Constructing a *multi-domain* environment in which each domain can manage its own configuration because a DDM consists of different, independently managed, and configured domains, with data sharing to be done across the domains.
- Improving security through better connection *isolation* between containers by controlling their network connections and providing more advanced filtering possibilities.
- Providing *programmability* in the network. Because of the dynamic behavior of a container-based network, it is important to provide the ability to program and change the network configurations instantly. Especially in a DDM that sharing policies may change at any moment.

We selected the P4 switch for connecting containers to be able to handle all the mentioned requirements in one setup. In overlay technologies like Calico or Cilium [12, 13] filtering rules can be installed between containers, so they can provide required isolation. However, in these methods, all the hosts are under the control of one master node that makes a cluster. Therefore, these methods are single-domain and are not applicable in our case.

The simplest way of connecting containers in a network that can be used in a multi-domain environment is using the Docker bridge in each domain [14]. However, in this method, all the containers are connected to the same bridge. Therefore, are connected to each other and it is not possible to set the filtering rule between them in the host. Therefore, it cannot provide the required network isolation between containers.

The other method can be deployed in a multi-domain environment is using a user-defined network and defining a separate bridges for each group of containers, so that the containers in different network bridges are separated from each other [15]. It can provide better isolation compared to the single Docker bridge method, however, still, the containers that are connected to the same bridge are connected to each other and their connections cannot be filtered in the host so the containers are not fully isolated.

In our setup, in each domain, the P4 switch is in charge of routing the traffic based on match-action rules independently from other domains so it can be deployed in a *multi-domain* environments. In addition, before setting up

the rules in the switch there is not any kind of connectivity between containers. Therefore, every single container's connectivity can be controlled by the switch. This provides better *isolation* between containers compared to available methods. Finally, any connection and filtering rules between containers can be programmed dynamically to provide *programmability*.

In addition to the aforementioned capabilities, P4 can provide more features in the DDM:

- As the packet's header can be parsed in the P4 switch (that is explained in the next section), every single field of the header at different layers can be checked against match rules. This provides high flexibility in deploying the filtering rules.
- As containers are individually connected to the P4 switch interfaces, different kinds of telemetry information, like the amount of traffic that is being transferred via a specific interface, or the time of entering of a packet to that interface can be tracked in P4 [16, 17]. This will help for managing the traffic preventing different kinds of attacks [18–20] to improve security.
- The P4 program can be run on SmartNICs [21, 22]. Therefore, the packet processing can be offloaded to the hardware, and the host's CPU is not involved anymore. This leads to better performance.
- New protocols can be defined in the P4 program to transfer extra information between multiple domains. For example, for our future work, we are adding extra auditing information to the packets that are being transferred between containers to improve the security of the network.

In the rest of this section, we give a background about P4 program and then an explanation about containers' network configuration is presented.

P4 language P4 is a high-level language for programming packet processing systems that is designed to program the data plane of packet forwarding devices. It is widely used by researchers, network owners, and developers as the new paradigm of network programmability that does not have the limitations of Openflow [23].

P4 is more flexible by offering the capability of defining new headers and making a protocol-independent system. P4 devices can be reconfigured in real-time. In addition, P4 is a target independent program and can be deployed on different forwarding hardware. A compiler is used to map those programs to target devices. The architecture of P4 compiler helps to make it target independent by separating language and the target model. This is an advantage in deploying P4 language in a multi-domain DDM, as the P4 program can be used in different domains infrastructure that the target hardware is not necessarily the same.

A typical packet processing pipeline in P4 includes parser, ingress match-action, egress match-action. The parser is a stage that parses the packet according to the header definition that is performed by user in program. Both ingress and egress match-action includes logical tables that match the packet against protocol field and determine the required actions to apply to packet. Actions determine common modifications to packets, such as changing header fields, adding/removing headers or packet cloning.

Containerized DDM Containers use Linux network namespaces to isolate containers from host network namespace. A Linux network namespace is a virtual network that isolates a process's network connectivity and resources (i.e. network interfaces, route tables, and rules) from other processes. For example, when Docker creates and runs a container, it creates a separate network namespace and puts the container into it. Then, Docker connects the new container to a Linux bridge *docker0* using a veth pair. This also enables the container to be connected to the host network and other containers in the same bridge. Using a Docker bridge or other overlay methods for connecting containers cannot provide isolation between containers on the same overlay network, however, complete isolation between containers is what is needed in a DDM platform. In this paper, the connection between containers is limited to switch configurations that can be completely programmed and changed in any time. By this method, maximum isolation can be provided in a network of containers.

3 Architecture

Figure 1 shows the architecture of the multi-domain containerized DDM with three distinctive blocks: the orchestration block in charge of the coordination of the operations in the DDM, the containerization block in charge of creating and management of containers, and the networking block in charge of setup the P4 network and the connection between containers and the outside world.

In each domain, all of the components are under the control of the domain administrator. The domain administrator can handle and manage all connections based on the policies and rules that have been established in the DDM. The role of each component in the architecture is as follows:

- Orchestration block:
  - *Domain administrator* Manages all components in its own domain and controls the sequence of steps for running the execution of a sharing request.

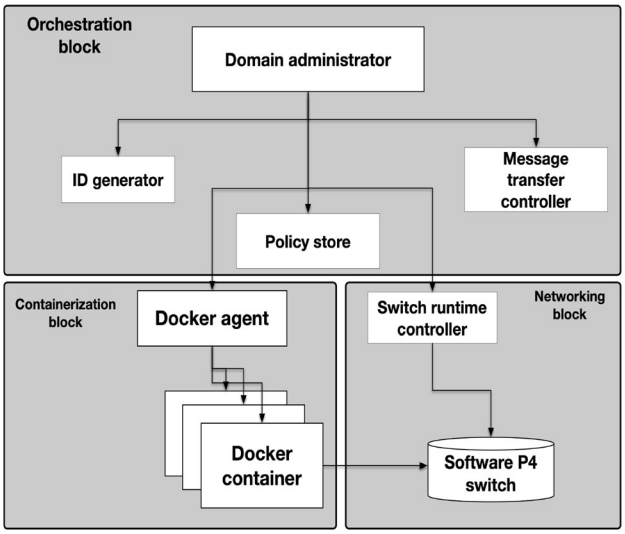


Fig. 1 A Multi-domain containerized DDM architecture. It includes three main blocks: orchestration block, containerization block and networking block

- *Message transfer controller* To make the connection between containers of multiple domains, they should be able to exchange necessary information. The message controller is in charge of sending/receiving the required information to/from the other domains involved in the sharing request.
- *ID generator* In each domain every single connection between two containers is identified by a unique number. We call this the *connection ID*. This number identifies the destination port of the client container, i.e, the initiator. The ID generator generates this unique ID in each domain.
- *Policy store* All of the policies and rules about permissions to access specific data in a domain are recorded in the policy store. Before running any request, the domain administrator checks in the policy store if the request is permitted or not. If it is not permitted the request will be rejected.

– Containerization block

  - *Docker agent* This component is in charge of setting up all of the required configurations for creating a container and its network interfaces.
  - *Docker container* The containers are created based on the incoming requests.

– Networking block

  - *Switch runtime management* This component sets the appropriate rules for handling the switch connections based on the information that is taken from the Docker agent and the generated connection ID. The rules allow traffic with a specific connection ID (port number) to a specific container.
  - *Software P4 switch* This is the core networking element that will switch the traffic appropriately.

4 Workflow scenario

The operation of our proposed architecture can be better understood by looking at a concrete scenario. We consider two domains, domain A and domain B, in a DDM and assume the whole architecture described in the previous section is running on two servers, one in each domain (see Fig. 2). Therefore, Domain A is a Linux server that is connected to other server in Domain B.

In the following, we will guide the reader step by step through each of the operations that occur to set up the connection between containers in the two domains for running a sharing request.

- *Step 1* When a request arrives at domain B that asks for access to an asset from domain A, the domain administrator of domain B sends a request to the administrator of domain A through the message controller. The request identifies the data that it wants to access. It also asks for the unique ID that is needed for starting the connection between containers.
- *Step 2* The domain administrator in domain A checks the request permission in the policy store and if the request is allowed it starts the required actions for making the connection.
- *Step 3* In this step the ID generator in domain A creates the connection ID.
- *Step 4* The domain administrator asks the Docker agent to create the necessary containers to satisfy the request. Containers are initially created without any interfaces. Then, the Docker agent configures the network

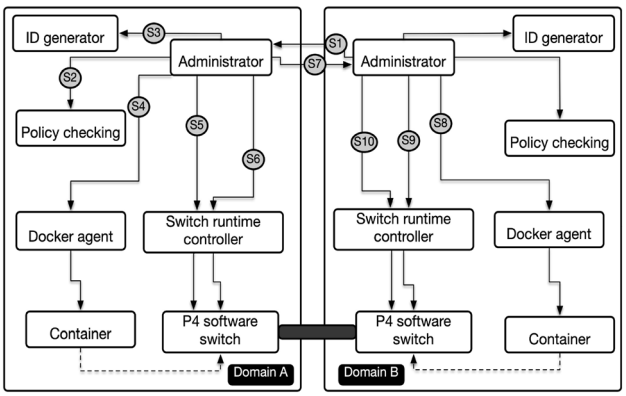


Fig. 2 Steps for making the connection between containers of two different domains using a unique connection ID

interface of the container with a specific IP address, and after that it peers the container interface with one virtual interface (*veth*) of the host server. It must be noted that at this stage the container is still not connected to the switch.

- *Step 5* After the container has been created and its IP address and interfaces are set, the domain administrator instructs the switch runtime controller to connect the container interface to one of the ports of the software P4 switch.
- *Step 6* Here the domain administrator generates the rules to allow establishing a connection between the two containers of the sharing request and to set the rules in the switch. Two rules are set on each side: one for sending packets from the container via the switch to the other domain, and one for sending packets coming in from the outside to the related container. In this step, the server-side is configured. The rule for incoming packets matches packets sent to a specific destination port (the connection ID), sets the destination IP address and port number to that of the container and the service running inside it, then sends it to the switch port to which the container is connected. The rule for outgoing packets matches packets from the container's switch port, sets the source IP address to the host's public IP address and the source port number to the connection ID.
- *Step 7* When all of the required configurations are set in Domain A, the domain administrator sends the unique ID out to Domain B. This also shows that Domain A is ready for a connection with that specific ID.
- *Step 8* Like domain A, domain B creates the container related to this request. The interface configuration is the same as domain A.
- *Step 9* In this step the domain administrator connects container to the switch port via the switch runtime.
- *Step 10* Finally, the domain administrator sets two rules that are needed for making the connection to the specific container based on the unique ID. On the client-side, the combination of destination IP address and destination port is unique. Therefore, packets sent from a local container to the switch are matched with these two specifications and then sent out to the server-side. The second client-side rule is for packets coming from the server-side. These are matched by the same unique combination (which here is the source IP address and source port number) and then sent to the specific container.

By performing all of these steps the DDM creates a connection between two containers in a multi-domain environment. The whole connection between domains is isolated and based on a unique number that is specific to that request and is not repeated by any other connection, hence guaranteeing isolation between concurrent requests.

5 Request setup time

In a containerized DDM, the setup time is the time from when a client issues a request until when the network is ready for starting the data transfer between containers. In many cases, it is important and critical for a client to know an approximate setup time of the request; for example, for federated machine learning the knowledge of resource availability is critical for running the model efficiently.

In a single domain DDM, as there is one centralized controller that is handling all of the resources, the network setup is simple and straightforward. However, in a multi-domain DDM, as domain administrators have to communicate with each other, and all of the setup processes in one domain are separated from the other one, setup is lengthier. In fact, the client-side should be sure that the destination is ready for establishing the connection.

There are two approaches we can take to measure the setup time: one is a *Global view* that considers the time from when a request comes in to when the network is ready for data transfer. It is measured by setting a timestamp from starting a request until it is ready to start transferring data.

The other one is a *Step view* which looks at the duration of each individual step. We measured the duration of each step by setting timestamps before and after running each step. For example, for measuring the duration of step 4, that is creating containers in domain A, we register the time stamp before running step 4: $C_A(start(S_4))$. It is a timestamp of the clock of domain A related to the start point of step 4. Likewise, we register the time stamp after finishing step 4 as $C_A(end(S_4))$. Therefore, the duration of step 4 is calculated based on Eq. 1.

$$Ts_4 = C_A(end(S_4)) - C_A(start(S_4)) \tag{1}$$

However, we must observe that there will be a challenge in measuring the duration of Ts_1 and Ts_7. That is because the starting and ending of these steps are not in the same domain and the time is dependant on the clock time of both domains. To solve this problem, we assume that Ts_1 and Ts_7 are equal and eliminate clock difference by adding timestamps of different domains according to Eq. 2.

$$\begin{aligned} Ts_1 = Ts_7 = (&C_A(end(S_1)) - C_B(start(S_1)) \\ &+ C_B(end(S_7)) - C_A(start(S_4)))/2 \end{aligned} \tag{2}$$

We investigated two different approaches to setting up the network: a sequential and a parallel mode.

5.1 Setup time in sequential mode

As explained in Sect. 4, 10 steps have to be executed for setting up the network. If they are executed sequentially then the total time is the sum of the duration of each step. If we define Ts_n as the time taken to run step n then the total time can be calculated as in Eq. 3.

$$\text{Sequential setup time} = \sum_{n=1}^{10} Ts_n \tag{3}$$

5.2 Setup time in parallel mode

To optimize the total setup time, it is possible to perform some of these steps in parallel. Looking back at Fig. 2, after the first three steps the other steps can be run simultaneously as they are in different domains and independent from each other. The parallel setup time is the sum of the duration of the three first steps and the maximum time of running the other steps in each domain. This is expressed in Eq. 4.

$$\text{Parallel setup time} = \sum_{n=1}^{3} Ts_n + \text{Max}\left(\sum_{n=4}^{6} Ts_n, \sum_{n=7}^{10} Ts_n\right) \tag{4}$$

5.3 Global view and step view comparison

Table 1 shows the theoretical model for calculating the setup time. The total time in Table 1 represents the global view and the calculated time represents the step view. *Communication delay*, *Creating container*, *Adding interfaces*, and *Adding rules* are individual steps that are considered in step view model. By comparing total time and calculated time (Model error in Table 1), we can determine the accuracy of the step view model. Concretely this would tell us if there is any overhead in running the steps that the model does not capture. If the overhead is negligible then we can conclude that the time to setup is predictable, which enables prediction of the setup time and other calculations for decision-making for making any further improvements.

T_A, T_B, T'_A, and T'_B in Table 1 are calculated based on Eqs. 5–8.

$$T_A = T_{s_1} + C_A(end(S_6)) - C_A(start(S_2)) \tag{5}$$

$$T_B = T_{s_1} + T_{s_7} + C_A(end(S_{10})) - C_A(start(S_8)) \tag{6}$$

$$T'_A = \sum_{n=1}^{3} Ts_n + \sum_{n=4}^{6} Ts_n \tag{7}$$

$$T'_B = \sum_{n=1}^{3} Ts_n + \sum_{n=7}^{10} Ts_n \tag{8}$$

6 Proof of concept

To test the operations of our architecture (see Sec. 3) we built a prototype DDM software suite. We implemented the connections between all the building blocks, starting from the domain administrator at the higher level all the way down to managing the network configuration in the P4 switch. We then instantiated two DDM domains and we connected them to validate the scenario in Fig. 2.

In our setup, we did not implement the policy checking part of the architecture and we assumed that all of the requests that come in are according to the agreed upon rules.

For our implementation, we used Ubuntu 18.04 and Linux kernel 4.15.0 as the host OS, Docker Community Edition 18.09 for container management, and bmv2 P4 switch as the programmable software switch in each server [24].

The scenario in Fig. 2 is written in a *bash* script, which initiates each step by calling the programs to implement the functionalities of each block of architecture.

We have two servers representing the two different domains, each running the full software suite. Each server

Table 1 Theoretical request setup time table based on time of each single step, T_{s_n} is the time taken for running step n

	Domain A	Domain B	Parallel time
Total time	T_A	T_B	$Max(T_A, T_B)$
Calculated time	T'_A	T'_B	$Max(T'_A, T'_B)$
Model error	$(T_A - T'_A/T_A) \times 100$	$(T_B - T'_B/T_B) \times 100$	$(Max(T_A, T_B) - Max(T'_A, T'_B)/Max(T_A, T_B)) \times 100$
Communication delay	Ts_1	$Ts_1 + Ts_7$	
Creating container	Ts_4	Ts_8	
Adding interfaces	Ts_5	Ts_9	
Adding rules	Ts_6	Ts_{10}	

can be the requester server (*client-side*) or the one that is requested to share data (*server-side*). The servers are connected together through a switch in the local area network, which serves as the physical underlay for our connectivity.

For message transfer between domain administrators, the script calls a message transferring program using RabbitMQ [25]. The receiver side of each server is always running and waiting for a new message. When a sharing request comes in from a client for access to data of the other domain, this program sends the access request information to the other domain. Likewise, the receiver side communicates back through the message bus.

After sending or receiving the required messages, the domain administrator starts to create containers and sets the interface configurations. This is the *containerization block* of the architecture.

Listing 1 shows the procedure to create and connect containers to the network.

```
1  sudo docker run
2      -it -d --net=none "container_image"
3
4  container_pid =
5      $(sudo docker inspect --format '{{
       .State.Pid }}' "container_name")
6
7  sudo ip link add
8      veth1 type veth peer name veth2
9
10 sudo ip link set
11     veth2 netns container_pid
12
13 sudo docker exec
14    "container_name" ifconfig veth2
15  "container_IP_address"
```

Listing 1 Container interface configurationprocedure

At first (lines 1–2) containers are created without any interface. Next (lines 7–8) we create the virtual interfaces and these are then moved to the container's network namespace (lines 10–11). Finally (lines 13–15) we configure the container interface.

After the container configuration, the switch with the compiled P4 program can start running.

P4 program Listing 2 shows the P4 program that was used for managing the connection between containers based on a unique ID.

```
1     table client_send_t {
2         key = {
3         hdr.tcp.dstPort :exact;
4         hdr.ipv4.dstAddr :exact;
5         }
6         actions = {client_send; }
7     }
8     table server_receive_t {
9         key = {
10        hdr.tcp.dstPort :exact;
11
12        }
13        actions = {server_receive; }
14    }
15    table server_send_t {
16        key = {
17    hdr.ipv4.srcAddr :exact;
18        }
19        actions = {server_send; }
20     }
21
22
23    table client_receive_t {
24        key = {
25    hdr.tcp.srcPort :exact;
26    hdr.ipv4.srcAddr: exact;
27    }
28    actions = {client_receive;}
29    }
```

Listing 2 List of tables used in the P4 program running in the switch defining the expected operations for packets sent or received by the server and client sides of the DDM

As each server can act as both server or client there are four tables defined in the P4 program:

- client_send_t (lines 1–7): when a packet is sent from client-side to server-side;
- server_receive_t (lines 8–14): when a packet is received on the server-side.
- server_send_t (lines 15–20): when a packet is sent from server-side to client-side;
- client_receive_t (lines 23–31): when a packet is received on the client-side;

A packet entering the switch is matched against the fields shown in Listing 3. If a packet matches any of the fields in one of the tables the specified action will be taken, else it will be dropped.

The actions for each table are shown in Listing 3. The action taken will depend on whether the packet is coming from the outside or from an internally connected container, and whether it is on the server-side or client-side.

```
1   action server_receive(bit<32> dst_ip,
    bit<9> port){
2       hdr.ipv4.dstAddr=dst_ip;
3       stdmeta.egress_spec = port;
4    }
5
6    action client_receive( bit<32> dst_ip,
    bit<9> port){
7       hdr.ipv4.dstAddr=dst_ip;
8       stdmeta.egress_spec = port;
9    }
10
11   action server_send(bit<32> src_ip,
    bit<9> port){
12      hdr.ipv4.srcAddr=src_ip;
13      stdmeta.egress_spec = port;
14   }
15
16   action client_send(bit<32> src_ip,
    bit<9> port){
17      hdr.ipv4.srcAddr=src_ip;
18      stdmeta.egress_spec = port;
19   }
```

Listing 3 P4 actions associated with the P4program tables

When a packet comes into the P4 switch to be sent to a local container, the destination IP address is changed to the correct local (container) IP address, and the packet is sent to the destination: these are *server_receive* (lines 1–5) or *client_receive* (lines 6–10) actions.

When a packet leaves the P4 switch toward the other domain the source IP address of the packet is changed to the public address of the local server: these are the *server_send* (lines 11–15) and *client_send* (lines 16–19) actions.

The last call is related to adding new rules associated with containers connection. Adding these rules allows connection between two containers created in two domains; that is the only permitted connection between these two containers. Domain administrator uses the switch command line to insert required rules for making connection possible. For example, listing 4 shows these rules related to client-side. The first rule is when the packet is outgoing from container to server-side. The second rule is when the response from server-side is arrived.

```
1 table_add client_receive_t client_receive
      "Source_port" "Source_ip_address" =>
      "Destination_ip" "Egress_port"
2
3 table_add client_send_t client_send
      "Destination_port" => "Source_ip_address"
      "Egress_port"
```

Listing 4 List of rules in the P4 program

Because filtering rules are a combination of IP address of source or destination and of the connection ID, the connection ID needs only to be unique in each host but not between domains. Therefore, there are 64K values for every host to assign as connection ID and that is enough in practice.

7 Measured request setup time

The next step for us was to measure the setup time in our experimental environment and try to numerically identify the possible overhead.

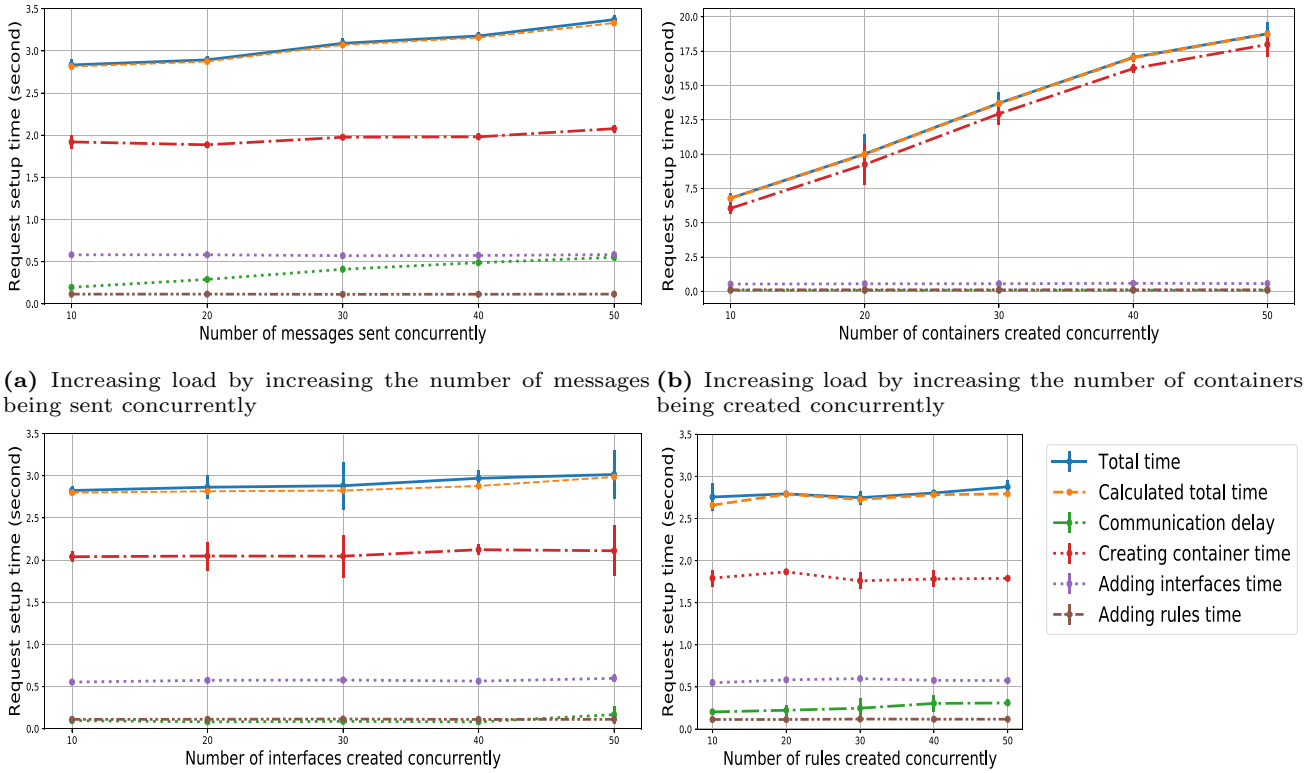
Setup time of one request According to Table 1 and based on what is explained in Sect. 5 we measured both the total setup time based on Eq. 4 as well as the duration of every single step (see Sect. 5.3). Table 2 shows the average value across 3 experiments. The results show that the model error is less than 3%. As the difference is negligible we can conclude that the stepwise model is reliable for estimating the setup time and that it can be used for further decision making and possible optimizations. The *creating container* step is the longest step for setting up the network ($\sim$ 1.8 s). The other steps related to the P4 switch are adding interfaces ($\sim$ 0.50 s) and adding rules ($\sim$ 0.110 s); they take much less time than creating the containers and this shows the low overhead of using a P4 switch in the setup process.

Setup time as a function of increasing load For further investigation and to observe the individual impact of each step on the setup time, we explicitly overloaded our system with concurrent operations, and observed the change of setup time as a function of the increasing load. In each experiment, we measured each step's duration and also calculated the total time based on the stepwise model. We performed four different experiments (Fig. 3):

– Message transfer experiment: Fig. 3a shows the setup time of one sharing request, when the number of concurrent messages that are being sent from client-side to server-side is increasing. To create this additional load we sent a number of messages unrelated to the request from the client-side to the server-side via the message bus. we varied the number of concurrent messages from 10 to 50. The plot shows that the delay for message transferring between two domains is positively correlated with the number of concurrent

Table 2 Request setup time table in seconds, numbers are according to Table 1

	Domain A	Domain B	Parallel time
Total time (s)	2.561	2.721	2.721
Calculated time (s)	2.499	2.700	2.700
Model error(s)	0.025	0.007	0.007
Communication delay (s)	0.082	0.163	
Creating container (s)	1.765	1.850	
Adding interfaces (s)	0.540	0.571	
Adding rules (s)	0.112	0.116	



(a) Increasing load by increasing the number of messages being sent concurrently

(b) Increasing load by increasing the number of containers being created concurrently

(c) Increasing load by increasing the number of interfaces being created concurrently

(d) Increasing load by increasing the number of switch's rules being set in P4 switch concurrently

Fig. 3 The impact of adding load on the system by increasing the number of operations related to each step separately

messages on the bus. We observe a maximum increase of $\sim$ 50%. On the other hand, as this step takes much less time than creating containers, its dependency on the number of concurrent messages does not have a substantial effect on the overall setup time.

– Creating container experiment: To produce additional load we created a specific number of containers not related to the specific request we are measuring. We issued requests to the Docker engine for 10 to 50 containers in step 10. As shown in Fig. 3b, by increasing the number of containers that are being created at the same time, the creating container time of a single request also increases. The additional amount of time is substantial compared to the other steps. Additionally, this step is always the longest (see Table 2), and its dependence on the load will have the greatest impact on the variability of the total time.
– Adding interface experiment: Fig. 3c shows the effect of running concurrent adding interface operations on the setup time. We created containers' interfaces (varying the number from 10 to 50) and added them to the P4 switch at the same time as the request being satisfied. The figure shows that the time is lower than container setup time and in addition, it does not change with an increasing number of concurrent operations.
– Adding rule experiment: In this experiment we created additional load by adding a varying number of rules, in the range 10 to 50, into the P4 switch. Figure 3d shows the same trend as for the adding interfaces experiment; in fact, the time for adding the rules to the switch does not increase with an increase in the number of rules.

As the results show, the creating container step is the step that is most affected by increasing load on the system, and more precisely when the system has to create many containers simultaneously. The steps that are related to the P4 switch, i.e, adding interfaces and adding rules do not change with increasing numbers of interfaces and rules. So we can conclude that the P4 switch is scalable enough for running multiple sharing requests. Also, in all experiments the calculated setup time is close to the total time and this proves the accuracy of the stepwise model.

8 Security

In a data sharing environment, one of the main concerns is providing security for protecting data against unauthorized access. Therefore, in a containerized DDM the security of the connection of containers should be guaranteed. We looked at different possible attacks that can happen in containerized network infrastructure and study how the architecture proposed in this paper is secure against these kinds of attacks.

Figure 4 shows the threat model of the architecture. In all attack scenarios, we assume that the attack originates in a malicious container. Three types of attacks can happen in a containerized DDM:

1. container to container of the same request (*atc* in the figure);
2. container to container of different request (also called *atc* in the figure);
3. container to its host's service (called *ath*); in this case a container attacks a service that is running in the host listening on a specific port number.

In addition, in a multi-domain DDM, these types of attacks can be *local*, i.e. the attacking container is in the same domain as the victim, or *remote*, i.e. the attacker is in the other domain.

In particular, in this paper, we only consider networking attacks related to container connections. Based on the architecture and the possible connections that a malicious container can make, we discuss the feasibility of each attack.

Figure 5 shows all possible connections that a container can make to any other services of other containers or hosts in our multi-domain environment. These include the local connection that happens in the same host with the container of the same request (*c2c_local*) or the host service (*c2h_local*); or the remote connections (*c2c_remote*) to the other domain.

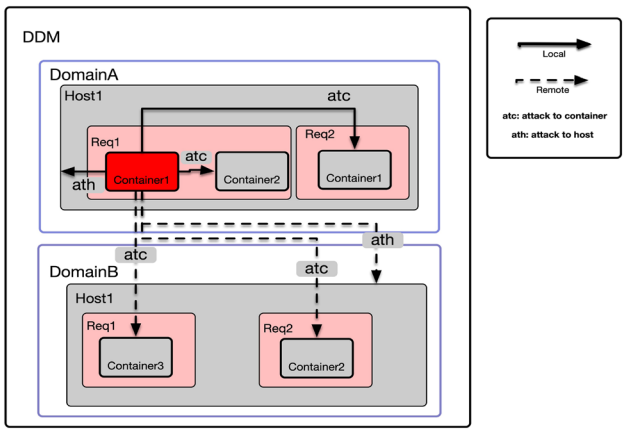


Fig. 4 Threat model in a multi domain DDM. Two types of attacks are possible: *local attacks*—Attacks from a container to host (*ath*) or to other container in the same domain. (*atc*); *remote attacks*—Attacks from a container to other container (*atc*) or host (*ath*) of the other domain

In our architecture any connection from a container to any other container is through the switch. However, for the connection between a container and the host, the packet does not need to go through the switch because of peering with the host virtual interfaces *veth* that are in the host network namespace.

Table 3 shows the possible attacks that can happen: ARP spoofing, MAC flooding, IP spoofing, Syn flooding, and HTTP flooding. In regard to the feasibility of each one of them there are three possible outcomes: 1) the attack is not possible; 2) the attack is possible but it can be mitigated; 3) the attack is possible and it cannot be mitigated.

The possibility of each kind of attack is explained in the following:

ARP spoofing and MAC flooding For these kinds of attacks to happen there must be a connection between containers at layer 2 [26]. Given that in our case each container is in a different network subnet the connections are at layer3 and ARP spoofing and MAC flooding are not possible in any of the possible scenarios.

IP spoofing In an IP spoofing attack scenario, an attacking container impersonates another container's IP address and sends packets with an incorrect source IP to another service that is running in the network. Therefore, the response of the packet will be sent back to the victim and not the actual source [27].

In our architecture, the feasibility of an IP spoofing attack depends on whether the attack is local or remote and on whether the attacked node is another container or a host.

More precisely we can observe that IP spoofing is not possible if the packet has to go through the switch, i.e. when the attacker wants to make a connection to other containers in the same host or to the outside world. As the routing is based on the unique ID number that is independent of container's IP address, the packet will be dropped as it does not match any rule. This means that *c2c_remote* and *c2h_local* are not possible.

IP spoofing is possible if the attacker tries to make a connection to its host service and perform the attack via host service, like in the *c2c_local* and *c2h_remote* types of attacks. In these cases packet does not go through the switch. However, its source IP address can be checked in host's IPtables. Therefore, when the source IP address is not correct, the packet will be dropped.

Syn and HTTP flooding In all cases that a container can make a connection, flooding attacks are possible. For cases in which the packet is going through the switch (*c2c_local* and *c2c_remote*) the attack can be mitigated by detection methods that can be implemented in the P4

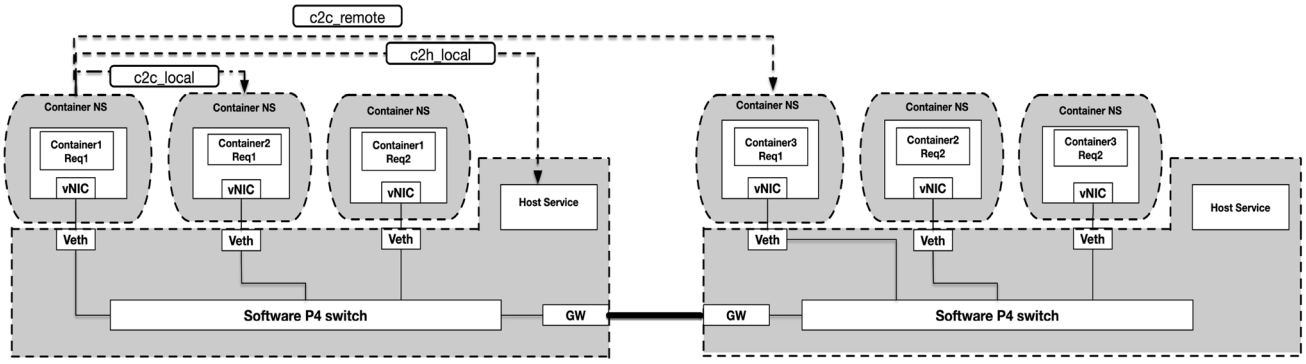


Fig. 5 Container connections possible in a multi-domain DDM: connection to containers of the same request in the local domain (*c2c_local*); connection to containers of the same request in the remote domain (*c2c_remote*); connection to the local host (*c2h_local*)

Table 3 Possibility of attacks in a multi-domain containerized DDM: ○ Attack is not possible ⊙ Attack is possible but it can be mitigated, ● Attack is possible and it cannot be mitigated

Type of attack	Domain	Arp spoofing	Mac flooding	IP spoofing	Syn flooding	HTTP flooding
Container to container of the same request	Single domain	○	○	⊙	⊙	⊙
	Multi-domain	○	○	○	⊙	⊙
Container to container of different request	Single domain	○	○	⊙	○	○
	Multi-domain	○	○	○	○	○
container to host	single domain	○	○	○	●	●
	Multi–domain	○	○	⊙	○	○

program [18–20]. However, for the *c2h_local* type of attack, the attack cannot be mitigated as the packet does not go through the switch and the connection is directly between containers and the host service.

To conclude, as it is shown in Table 3 most network attacks are either not possible in our proposed P4-based DDM or they can be mitigated by customizing the P4 program. We must observe that security depends also on the availability of the P4 switch. As all containers are connected to the P4 switch and the whole setup relies on the rules in the p4 switch, this can become a failure point in the network. However, as there can be multiple servers in a DDM when the switch fails in a domain, only the containers that are connected to that specific switch will be affected.

Confidentiality and integrity in transferring data can be supported by containers and happen in upper layers. Data encryption can also happen in the P4 switch. To this purpose, there are multiple available algorithms that encrypt the connection in a P4 switch and can also be deployed in this work [28–30].

The orchestration services at the two sites communicate with each other via the public Internet, which means they must be publicly accessible, and they also control the network plane, which requires at least some administrator-level privileges. This is clearly a potentially risky combination. Several steps should be taken to mitigate this risk. First, the connection between the orchestration services needs to be authenticated and encrypted, for example by using an HTTPS connection with client- and server-side certificates for authentication.

Next, to mitigate the risk of the orchestration application software itself being compromised, the network administration functionality should be separated out into a separate program, which is given the minimum necessary privileges (e.g. CAP_NET_ADMIN on Linux) and implements the bare minimum functionality needed to support the functioning of the system. The orchestration service itself can then be run as an unprivileged user, so that if it is compromised the attacker will be limited to unprivileged operations plus a small number of very inflexible network administration functions.

Beyond these specific design features, all the usual general measures should be taken, including code quality assurance through testing, code reviews, and automated analysis, and in the deployed system firewalls can be used to block attacks, and intrusion detection systems may be used to monitor the system for abnormal behaviour.

9 Discussion

We must note that in the scenarios in Sect. 4 we assumed that all of the steps are run successfully. However, in reality, one or more of the steps may fail. This especially affects the parallel scenario (see Sect. 5), as the domain in which the error did not occur will continue to reserve resources that will never be used. In this case, running in sequential mode would be more efficient.

Our architecture is flexible and can easily scale up. What we have shown in this paper is an example of an implementation of a containerized DDM running in one server per domain, e.g. all the containers in one domain reside in one physical node. However, the number of required containers may be more than the capacity of one device. In the case where there is an increasing number of requests in DDM and therefore the need for more containers, other servers can be easily added to the domain. Each server has its own software P4 switch. All the P4 switches would then be connected with each other and with the physical network infrastructure in the domain.

As we explained all connections between switches are based on a connection ID. Even if a container can guess the connection ID, it cannot use it. As all the packets are going through the P4 software switch, the connection ID can be checked that it is from an eligible container.

The connection IDs are unique within the host. Therefore, the combination of connection ID and host's IP address is unique in the whole site and should be permitted in border firewall rules.

Defining a range of transport port numbers in border firewalls depends on the general filtering strategies of a site. What is important is allowing connection to a specific IP address and specific port number. One solution can be defining a range of permitted port numbers related to a host that is running a P4 software switch and allowing that range with host IP number in the border firewall.

10 Related work

DDM prototypes are currently in development in a number of scientific and industrial contexts, including the Internet of Things, supply chain logistics, health care, and exchange of personal information.

Datapace is a commercial blockchain-based DDM platform for trading IoT data streams [31]. It too has a blockchain-based trading infrastructure, in which URLs are traded at which data may be retrieved. Datapace sells a curated collection of streams, but also allows external sellers on its platform. Data is routed through its central infrastructure. The Ocean Protocol is similar but is inspired more by financial markets, with market makers and derivatives. Data exchange is done directly between buyers and sellers, and somewhat outside the scope of the platform [32]. Both of these systems list data processing as possible future extensions but do not currently support it.

International Data Spaces (IDS) is a DDM project addressing amongst others supply chain logistics. It defines data exchange protocols and provides central components including a data broker, clearing house, identity provider, app store, and vocabulary provider [33]. Data are requested from a data provider, optionally processed, and returned to the data consumer. An example use case is provided by the DL4LD project, which will apply this technology to enable sharing of potentially sensitive data regarding the transport of goods [4].

A science use case concerns personalized medicine: the EPI project will develop a secure and trustworthy platform to share patient data across medical institutions to help diagnosis and decision making for both patients and health providers; the sharing of information will still fully preserve patients' privacy. It also studies policies definitions and how to set up network infrastructure to enforce them [1].

None of these systems support processing of data, nor is the implementation of data exchange described at the technical level. Our paper shows, for the first time, how to realize the required network connectivity between DDM parties, e.g domains.

For building a containerized DDM within one domain, we had already proposed a number of solutions. In [34] we studied whether available container network overlay technologies are suitable for deployment in a DDM. We compared the performance of each technology and we concluded that scalability was the main attention point in choosing one technology over another. In [9] we remained focused on single domain DDMs but we shifted our focus onto providing isolation between containers and improving the security. We studied three methods of container overlay implementations with particular attention devoted the isolation between containers in order to satisfy the data sharing policies.

There is some work on the possible network solutions for multi-domain DDMs. Xin et al. proposed a multi-domain distributed architecture for policy-driven data sharing applications [35]. The architecture includes components to manage policy auditing as well as to implement network connections. To do this, they use Docker containers and connect containers of each domain via VPN connections. Although this approach can secure the data by encryption, the ease of connection management method and also the security aspects had not been studied.

In our current work, we are able to manage connections in a dynamic and straightforward programmable method.

As we mentioned we did not cover yet encryption, but this can also be done in P4 switches. The load of this operation could, if needed, be even offloaded to hardware by using P4 hardware switches.

11 Conclusion and future work

In this paper, we proposed a multi-domain data sharing architecture that is constructed with containers and software P4 switches. Our architecture supports the network connectivity between participating DDM domains. We explained each step of setting up the network for making the network ready for operation and showed the required configuration. We also studied the performance and security implications of adopting P4 programmable switches as underlying technologies. To support performance and planning, we introduced a model for measuring the setup time and then showed that the model reliably represents the real operations. We also determined that the overhead of using a P4 switch in the setup process is negligible, which makes an ideal technology to support the networking requirements of DDMs.

Besides, we studied the security aspects of the proposed architecture and by isolating every single connection via a unique ID number in DDM we showed how the architecture is secure against a number of typical network attacks.

Our future work will focus on improving the network's performance and security by enhancing the P4 program to be able to reconfigure the network dynamically based on real-time events and using monitoring information. We also want to look at the possibility of using containers as network virtual functions (NVFs) that can be used as part of the execution workflow scenario. Finally, we intend to complete the policy checking module of the architecture and integrate it with current work.

Acknowledgements This work is supported by the Netherlands eScience Center and NWO under the project SecConNet. We want to specifically thank Rena Bakhshi for the useful discussions and feedback.

Author contributions I confirm that all authors listed on the title page have contributed significantly to the work, have read the manuscript, attest to the validity and legitimacy of the data and its interpretation, and agree to its submission. Conceptualization: [SS], [LV], [PG]; data curation: [SS]; formal analysis: [SS], [LV], [PG]; funding acquisition: [PG]; investigation: [SS], [LV], [PG]; Methodology: [SS], [LV]; Project administration: [PG]; Visualization: [SS]; Software: [SS]; Supervision: [PG]; Validation: [SS], [LV], [PG]; Writing—original draft: [SS]; Writing—review and editing: : [SS], [LV], [PG].

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

Data availability Not applicable.

Code availability P4 software switch code has been used in this work that is publicly available on "https://github.com/p4lang/behavioral-model". The code that has been written for this project is available on "https://github.com/sarashakeri/P4-uniqueid".

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Kassem, J.A., De Laat, C., Taal, A., Grosso, P.: The EPI framework: a dynamic data sharing framework for healthcare use cases. IEEE Access **8**, 179909–179920 (2020). https://doi.org/10.1109/ACCESS.2020.3028051
2. Bastiaansen, H., Nieuwenhuis, K., Zomer, G., Piest, J.P.S., van Sinderen, M., Dalmolen, S.: The logistics data sharing infrastructure. White Paper (2020)
3. AMdEX: THE DATA HYPERMARKET (2021). https://amsterdameconomicboard.com/en/news/research-organisations-and-commercial-parties-start-developing-the-new-amsterdam-data-exchange. Accessed Apr 2021
4. Zhang, L., Cushing, R., Gommans, L., De Laat, C., Grosso, P.: Modeling of collaboration archetypes in digital market places. IEEE Access **7**, 102689–102700 (2019). https://doi.org/10.1109/ACCESS.2019.2931762
5. priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange (2021). https://distributedlearning.ai/. Accessed Apr 2021
6. A library for computing on data you do not own and cannot see (2021). https://github.com/OpenMined/PySyft. Accessed Apr 2021
7. MPyC: Secure Multiparty Computation in Python (2021). https://www.win.tue.nl/~berry/mpyc/. Accessed Apr 2021
8. IBM Federated Learning (2021). https://github.com/IBM/federated-learning-lib. Accessed Apr 2021
9. Shakeri, S., Veen, L., Grosso, P.: Evaluation of container overlays for secure data sharing. In: 2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium), pp. 99–108 (2020). https://doi.org/10.1109/LCNSymposium50271.2020.9363266
10. Kubernetes (2021). https://kubernetes.io/docs/tutorials/kubernetes-basics/. Accessed Apr 2021
11. Use bridge network (2021). https://docs.docker.com/network/bridge/. Accessed Apr 2021
12. Calico (2021). https://www.tigera.io/project-calico/. Accessed Sept 2021
13. Cilium (2021). https://cilium.io/. Accessed Sept 2021

14. Default bridge network (2021). https://docs.docker.com/network/network-tutorial-standalone. Accessed Sept 2021
15. User-defined bridge networks (2021). https://docs.docker.com/network/network-tutorial-standalone. Accessed Sept 2021
16. Improving Network Monitoring and Management with Programmable Data Planes (2021). https://opennetworking.org/news-and-events/blog/improving-network-monitoring-and-management-with-programmable-data-planes/. Accessed Sept 2021
17. Manzanares-Lopez, P., Muñoz-Gea, J.P., Malgosa-Sanahuja, J.: Passive in-band network telemetry systems: the potential of programmable data plane on network-wide telemetry. IEEE Access **9**, 20391–20409 (2021). https://doi.org/10.1109/ACCESS.2021.3055462
18. Lapolli, A.C., Adilson Marques, J., Gaspary, L.P.: Offloading real-time ddos attack detection to programmable data planes. In: 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 19–27 (2019)
19. Febro, A., Xiao, H., Spring, J.: Distributed sip ddos defense with p4. In: 2019 IEEE Wireless Communications and Networking Conference (WCNC), pp. 1–8 (2019). https://doi.org/10.1109/WCNC.2019.8885926
20. Dimolianis, M., Pavlidis, A., Maglaris, V.: A multi-feature ddos detection schema on p4 network hardware. In: 2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), pp. 1–6 (2020). https://doi.org/10.1109/ICIN48450.2020.9059327
21. About Agilio SmartNICs (2021). https://www.netronome.com/products/smartnic/overview/. Accessed Sept 2021
22. P4SmartNics (2021). https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_nfp_architecture.pdf. Accessed Sept 2021
23. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. **44**(3), 87–95 (2014)
24. BEHAVIORAL MODEL (bmv2) (2021). https://github.com/p4lang/behavioral-model. Accessed Apr 2021
25. RabbitMQ (2021). https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html. Accessed Apr 2021
26. ARP spoofing. https://www.veracode.com/security/arp-spoofing (2021). [Online; accessed April-2021]
27. IP spoofing (2021). https://www.oreilly.com/library/view/ccna-security-210-260/9781787128873/78f2bb48-0c68-452b-8edc-eb1482f7dbfc.xhtml. Accessed Apr 2021
28. Hauser, F., Häberle, M., Schmidt, M., Menth, M.: P4-ipsec: site-to-site and host-to-site vpn with ipsec in p4-based sdn. IEEE Access **8**, 139567–139586 (2020). https://doi.org/10.1109/ACCESS.2020.3012738
29. Qin, Y., Quan, W., Song, F., Zhang, L., Liu, G., Liu, M., Yu, C.: Flexible encryption for reliable transmission based on the p4 programmable platform. In: 2020 Information Communication Technologies Conference (ICTC), pp. 147–152 (2020). https://doi.org/10.1109/ICTC49638.2020.9123251
30. Hauser, F., Schmidt, M., Häberle, M., Menth, M.: P4-macsec: Dynamic topology monitoring and data layer protection with macsec in p4-based sdn. IEEE Access **8**, 58845–58858 (2020). https://doi.org/10.1109/ACCESS.2020.2982859
31. Draskovic, D., Saleh, G.: Datapace (2017). https://datapace.io/datapace_whitepaper.pdf
32. Foundation, O.P., GmbH, B.: Ocean protocol: tools for the web3 data economy (2020). https://oceanprotocol.com/tech-whitepaper.pdf
33. International data spaces reference architecture model version 3.0 (2019). https://internationaldataspaces.org/download/16630/
34. Shakeri, S., van Noort, N., Grosso, P.: Scalability of container overlays for policy enforcement in digital marketplaces. In: 2019 IEEE 8th International Conference on Cloud Networking (CloudNet), pp. 1–4 (2019). https://doi.org/10.1109/CloudNet47604.2019.9064090
35. Zhou, X., Cushing, R., Koning, R., Belloum, A., Grosso, P., Klous, S., van Engers, T., de Laat, C.: Policy enforcement for secure and trustworthy data sharing in multi-domain infrastructures. In: 2020 IEEE 14th International Conference on Big Data Science and Engineering (BigDataSE), pp. 104–113 (2020). https://doi.org/10.1109/BigDataSE50710.2020.00022

Sara Shakeri received the B.Eng. degree in computer engineering from Isfahan University of Technology, Isfahan, in 2012, and the M.Sc. degree in computer architecture from Sharif University of Technology, Tehran, in 2015. She is currently pursuing the Ph.D. degree with the MultiScale Networked Systems, University of Amsterdam. Her research interests include novel network infrastructure, programmable infrastructure, secure container networks, and virtual network functions.

Lourens Veen is a Senior Research Software Engineer at the Netherlands eScience Center. As an engineer, his background is originally in databases and information system architecture, including geographical information systems, and more recently he has worked in High Performance Computing and networking. As a scientist, his skills are in modelling, multi-scale models, model coupling, parameter optimization and Uncertainty Quantification, with applications in biogeography, molecular simulation and computational biophysics.