UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

EPI framework: Approach for traffic redirection through containerised network functions

Kassem, J.A.; Valkering, O.; Belloum, A.; Grosso, P.

DOI
10.1109/eScience51609.2021.00018

Publication date
2021

Document Version
Final published version

Published in
Proceedings, IEEE 17th International Conference on eScience

License
Article 25fa Dutch Copyright Act

Link to publication

Citation for published version (APA):
Kassem, J. A., Valkering, O., Belloum, A., & Grosso, P. (2021). EPI framework: Approach for traffic redirection through containerised network functions. In *Proceedings, IEEE 17th International Conference on eScience: e-Science 2021 : 20-23 September 2021, online event* (pp. 80-89). IEEE Computer Society. https://doi.org/10.1109/eScience51609.2021.00018

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, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:10 Mar 2023

EPI Framework: Approach for Traffic Redirection Through Containerised Network Functions

Jamila Alsayed Kassem, Onno Valkering, Adam Belloum, Paola Grosso

2021 IEEE 17th International Conference on eScience (eScience) | 978-1-6654-0361-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/eScience51609.2021.00018

***Abstract*—Utilising programmable infrastructures is a promising approach to support secure data-sharing across healthcare domains of different capabilities in terms of network and security. The EPI[1] (Enabling Personalised Interventions) framework automates the setup of the underlying infrastructure while considering different requirements communicated by the EPI components, such as logic area generator and policy management system.**

In our approach to dynamically provide collaborative environments, we containerise network functions (VFs), that are shipped out and instantiated at the edge of the network. We use container-based Virtual Network Function (VNFs) to accomplish fast deployment, high reusability, and low-performance overhead.

Traffic interception and redirection through the chain of containerised network functions is a core feature of our framework. In this paper, we focus on the implementation and design of this tool for packet interception and redirection. We evaluate the performance of two approaches: an NGINX-based reverse proxy method and a SOCKS protocol-based method. We benchmark them to determine the overhead compared to a direct data-sharing session with no proxy. We conclude that the reverse proxy performs better in terms of overhead. Nonetheless, the SOCKS-based method works on a lower network level support all traffic types and offer a higher processing rate. We compare the methods according to other performance parameters. Subsequently, the choice of method will depend on the application performance requirements.

***Index Terms*—containerised network functions, traffic redirection, programmable infrastructures, personalised medicine**

I. Introduction

SHARING data securely among healthcare providers is still a unresolved challenge. Programmable and virtualised infrastructure can provide the mechanisms to support this.

In fact, the current generation of network and ICT infrastructures already heavily rely on virtualisation, given the successful evolution of virtualisation over the past decades [1]. The ETSI (European Telecommunications Standards Institute) standardised the NFV architecture, which can be extended to define the next generation of network infrastructures [2]. The architecture addresses the management and coordination of network resources for cloud-based applications and the network services lifecycle. Moreover, the NFV paradigm allows on-demand implementing and instantiating of NF's such as firewalls, segmentation, Deep Packet Inspection (DPI), etc. This is fundamental when dealing with heterogeneous collaborative domains.

SDI's (Software-defined Infrastructures) are emerging as programmable infrastructures that are managed by centralised control plane components. SDI is an approach for integrating control and management of heterogeneous computing and networking resources in software. The proposed technology offers adequate programmability that can be further exploited for the development of infrastructure-independent NFV frameworks to facilitate cross-domain innovation through the use of open interfaces. These complementary technologies can be utilised to build a dynamic network infrastructure to adapt to different requirements/ healthcare application request.

Our work is part of the EPI (Enabling Personalised Interventions) project that ultimately aims to empower patients through self/joint management of their personalised treatment and recovery cycle. In our work, we aim to promote secure and reliable health data-sharing across heterogeneous domains.

On the other hand, NF's can be container-based, which is the design decision we're taking. In this case, NF's and all their dependencies are encapsulated in lightweight Docker containers to offer platform independence, fast instantiation time, low resource utilisation, and high processing rate. After setting up the NF's, redirection tools should be employed through a specified route.

In this paper, we evaluate packets redirecting tools that would serve as a building block of the proposed EPI framework - EPIF [3]. We implement several methods to effectively control traffic routing and redirection. That will be used to force traffic through Bridging Functions (BF's) containers (network and security services).

At a lower level, we need appropriate technologies to implement efficient traffic redirection functionality. We expect that the infrastructure's dynamicity comes at a price of network performance and overhead. We aim to experiment and investigate the impact of introducing different redirection proxies in the middle of a data-sharing session.

We evaluate and benchmark the performance of reverse proxy and SOCKS based implementations according to the overhead, processing rate, and defined parameters. We contribute to the existing work in the same field as follows:

- We introduce a novel data-sharing framework that supports healthcare applications
- We implement different proxy tools to accomplish traffic manipulation through containerised NF's
- We benchmark redirection tools and evaluate according to performance parameters
- We compare and discuss the results which aren't specific to EPIF and can be reused in a different scope.

This paper is organised as follows: In Sec. II, we introduce the EPIF in the context of personalised medicine. In Sec. III, we elaborate on the concept of bridging functions and

Authors are with Multiscale Networked Systems (MNS), IvI, University of Amsterdam, Amsterdam, 1098 XH, e-mail: j.alsayedkassem@uva.nl

[1]https://enablingpersonalizedinterventions.nl/

978-1-6654-0361-0/21/$31.00 ©2021 IEEE
DOI 10.1109/eScience51609.2021.00018

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

the area logic model employed in the EPIF. In Sec. IV, we touch on the implementation designs of the proxy component utilising a traffic redirection functionality. In Sec. V, we illustrate the protocol traffic of each proxy in order to estimate the connection setup time of each. In Sec. VI we detail the evaluation of the implementations with some insights. In Sec. VII, we compare the proxies according to a number of parameters. Lastly, we showcase related work in Sec. VIII and conclude our work and introduce potential future work in Sec. IX.

II. Towards personalised medicine

Personalised medicine is a novel approach to improve clinical care by using an individualised or stratified approach to diagnosis and treatment rather than a group treatment approach [4].

On the road towards personalised medicine, it is necessary to enable collaboration between healthcare providers. In fact, effective medical data and electronic health records (EHR) sharing is a key enabler in health research and achieving personalised medicine. Due to the inherently sensitive nature of this information, it is vital to securely transmit, store, and process the shared data sets. With that in mind, the main challenge is to dynamically set up a collaborative environment/data sharing application request to ensure maximal security across different healthcare domains. A data-sharing framework is needed to bring together cooperative efforts of research centres, health institutions, and patients groups. The framework should consider policies, different parties' infrastructural capabilities (network and security functions supported), and the methods to dynamically match requirements to setup actions.

EPIF we proposed in [3] is the solution to this problem, as it manages and configures the underlying infrastructure to effectively run healthcare applications and support relevant collaboration data-sharing models (*archetypes*).

A. The EPI framework

EPIF is a dynamic health data sharing framework that accommodates different domains' infrastructural capabilities by shipping and managing containerised security and network services called *bridging functions*, which run at special *proxy nodes*. We will describe in detail these capabilities in Sec. III. An automated setup of the infrastructure is required to achieve:

- Reachability of the end-points nodes;
- Optimal security across collaborating domains;
- Reasonable network performance;
- Bridging services availability;
- Hardware selection and scalability (horizontal vs vertical scaling);
- At a higher level, abiding and enforcing policy management requirement.

As shown in Fig.1, EPIF has different components that are crucial to automate the data sharing processes between participating parties. The main components of the framework are the orchestrators (both at the application level and infrastructural level); the policy management system and the components required to be present at the participating institutions, namely the resource provisioner and the authorizers.

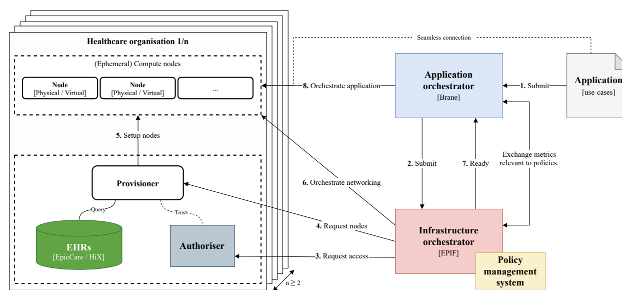


Fig. 1: A high level view of the different EPIF components and their interactions after an application request.

1) Orchestrators: EPIF employs a dual orchestrator design. There is an application-focused orchestrator and an infrastructure-focused orchestrator. The two orchestrators, in principle, operate independently but exchange runtime information so they can collaboratively enforce policies and give insight into the functioning of the combined deployment.

- **The application orchestrator** is the EPIF starting point for executing applications. After receiving an application submission (step 1 - Submit - in Fig.1), it first sends a request to the infrastructure orchestrator (step 2 - Submit) to streamline any discrepancies between the targeted infrastructure domains. This streamlining, described in detail in Sec. III, is done transparently to the application orchestrator. The final result of the communication with the infrastructural orchestrator is the receipt of confirmation of setup (step 7 - Ready). At this point, the application orchestrator considers the targeted infrastructure domains as ready and that there is an inter-connected execution environment. The application orchestrator will handle the heterogeneity of the made-available resources, i.e. compute nodes by direct interactions (step 8 - Orchestrate Application).
- **The infrastructure orchestrator** arranges all aspects related to the multi-domain target environment. The orchestrator requests access to the domain resources (step 3 - Request Access); after being granted access it requests specific nodes (step 4 - Request Nodes); finally, it manages the required bridging functions (BF) that are needed basing on the area logic model (Sec. III-A) via step 6 - Orchestrate Networking.

Decoupling the two orchestration concerns into separate systems increases the applicability of the EPIF. That is apparent with infrastructure orchestrator's independent operation that makes it possible to supporting alternative application orchestrators, such as Vantage6 [5] that is privacy preserving federated learning infrastructure. On the other hand, implementers targeting the EPIF can deploy the application orchestrator independently for development and testing purposes to improve productivity.

Brane [6], a framework for programmable orchestration, is at the core of both orchestrators. Programmability is an essential aspect of enabling dynamic adaption to heterogeneity across infrastructure domains.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

2) Policy management system: The policy management system captures allowed and denied data flows. It considers different constraints such as GDPR, patients' consent, and the intended goal of the application scenario. The system can deduce extra setup requirements and communicate them to the infrastructure orchestrator via specific Domain-Specific Languages. The communicated policy is estimated to be dynamic, as it can change during an application request, and the EPIF is asked to adapt and comply with any policy change.

3) Domain components: At all the participating organizations a resources provisioner and authoriser components are required. They might be implemented directly by the domain, in which case they have to implement the proper APIs to interact with the EPIF, or they can be provided as pluggable components from the EPIF itself, in which case they are compatible with the infrastructure orchestrator.

In the specific case of one single domain, an EPI client can run an application on (physical/virtual) node(s) within one healthcare domain by only submitting a request via Brane. This is illustrated in Fig. 1 with steps 1 and 8. In all other cases, we need multi-domain communication and this is taken care of by the EPIF components in steps 2-7.

III. BRIDGING POLICIES AND INFRASTRUCTURAL RESOURCES

A core element of the EPIF operations is the need to match the policy constraints provided by the policy management system with the underlying capabilities of the infrastructure. To support this, we defined in [3] two core concepts: *security areas* and *bridging functions*.

The area logic model works on grouping end-points nodes partaking in a data-sharing application according to the associated network and security functions that are supported and applied by the node itself into security areas. After that, the logic model deduces which bridging functions need to be instantiated to move data between nodes in different areas.

A. Area logic model

A health domain party has a number of nodes which are the endpoints of physical/virtual resources. We represent all resources endpoints included in the infrastructure by the set N of nodes:

$$N = \{n_i \mid i = 1, ..., m\}, \tag{1}$$

where n_i represents a single node in the infrastructure and m is the total number of nodes.

To evaluate the security and network capabilities of a node with a domain, we consider the associated attributes set. Attributes refer to the node's supported and applied network and security functionality within a network, such as segmentation, data encryption and key management, firewalls, scalable network links, etc. Let A be the set of all possible attributes:

$$A = \{a_k \mid k = 1, ..., d\}, \tag{2}$$

where a_k represents a distinct attribute and d is the total number of attributes.

Once the application scenario is made clear, the infrastructure is initialised to support it. Let N_{App} be the set of nodes relevant to an application scenario, such that:

$$N_{App} \subseteq N, \tag{3}$$

where N is the silo of resources (physical/virtual nodes) of all parties and N_{App} specifies the nodes collaborating in a specific application scenario. Each node's capabilities are described by a set of attributes, which can be any subset of A. Subsequently, every node n_j in N_{App} is assigned $A_j \subseteq A$.

After we define N_{App} and its associated attributes, we can evaluate these nodes' security and reachability across domains. The area abstraction is used to identify heterogeneity between nodes and deduce which data movements are supported and feasible: we call such supported communication between nodes in different areas *channels*. The logic dictates that data movement is supported when Eq. (4) is satisfied, where $n_g \Rightarrow n_h$ represents a one data movement support between nodes n_g and n_h. That means that a directional movement from n_g to n_h is supported when the capabilities of n_h (A_h) is the same or a superset of that of n_g (A_g)

$$n_g \Rightarrow n_h, iff A_g \subseteq A_h. \tag{4}$$

B. Bridge attribute gaps

Application requests, policy rules, and available channels might not necessarily be compatible, and this can be an issue to run an application successfully. Continuing with our formalism, we can see that condition $A_g \subseteq A_h$ supports data movement from node n_g to n_h. Otherwise, A_g is not a subset of A_h and communication is in principle not possible.

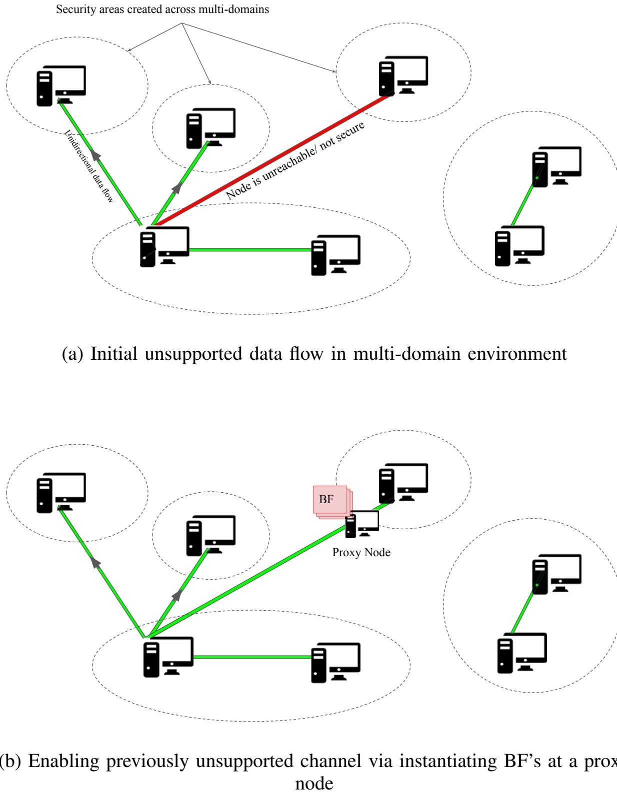
As shown in Fig. 2, BF's are introduced to bridge attributes gaps, and subsequently enable communication that wasn't previously supported across different areas and domains. Concretely, a bridging function is introduced to apply the missing attributes ϵ_{gh} and by that enabling the previously missing communication, if possible. In Eq. (5), ϵ_{gh} indicates the missing attributes

$$\epsilon_{gh} = A_h \cap \overline{A_g}. \tag{5}$$

There is a known set of bridgeable attributes A_δ. A_δ is set a priori. A_δ is bridged by an associated BF's that are containerised and shipped to be pooled on the proxy nodes. BF's are independent of the capabilities of the infrastructure. The bridging function applies the missing attribute if $\epsilon_{gh} \subseteq A_\delta$. This supports the data movement and ensures reachability and security within collaborating nodes across different domains.

The BF's images are provisioned and maintained in proxy nodes. As illustrated in Fig. 3, there exists a proxy per network, which allows the proxy to act as a single mini-orchestrator. The proxy intercepts, controls, and manipulates traffic. The EPIF employs redirection tools to enforce traffic through the proxy and consequently through the BF. Note that it is sometimes needed to have a number of cascading bridging functions, ie *function chains*, to put two areas in communication.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.



(a) Initial unsupported data flow in multi-domain environment

(b) Enabling previously unsupported channel via instantiating BF's at a proxy node

Fig. 2: An illustration of an example scenario of different domains nodes belonging to different security areas

The architecture in Fig. 3 shows how the proxy components fit with previously defined components in Sec. II-A.

Three main concerns arise when dealing with bridging functions:

- *unfeasible bridges:* There exists an unfeasibility ratio that needs to be considered in case $\epsilon_{ij}\, not \subset A_\delta$; an unsupported data movement is unbridgeable.
- *costs:* There is a cost associated with the bridging function in terms of time and complexity to set it up;
- *redirection tools:* There is the need to identify and test the appropriate tools to effectively intercept and redirect traffic through the instantiated BF's running on the proxy.

The EPIF manufactures by software network and security container-based functionalities, and host it on the proxy node. The newly introduced functionalities are flexibly and dynamically traded and provisioned by the orchestrators. As a result, the EPIF adapts the underlying infrastructure to support health applications (e.g. medical data streaming, EHR and backup, machine learning model training) with different archetypes.

In the next sections, we address the last concern and introduce the different proxy implementation approaches and the performance of the redirection tools.

IV. Proxy implementations

We evaluate two proxy implementation candidates for the EPIF. The first candidate that we consider is a reverse proxy. The second candidate is a SOCKS compatible proxy. Nodes route outgoing network traffic through a separate proxy node. On the proxy node, the appropriate BF chain processes the network traffic before the proxy implementation sends the network traffic to the destination.

A. Reverse proxy

The reverse proxy approach implements the NGINX reverse proxy tool [7]. This proxy tool is widely used to balance load through multiple servers, display content from multiple websites in a seamless manner, or transfer requests for handling to application servers using protocols other than HTTP. The reverse proxy works by proxying a request, redirects it to the specified server, retrieves an answer, and sends back the reply to the client. The proxy acts as the middleman handling requests, redirecting them, and forwards back the reply. The proxy can handle requests to non-HTTP servers (for example, PHP or Python) using a specified protocol. The implementation of the reverse proxy is illustrated in Fig. 4. The redirection table is customised and initialised at the proxy VM, and each port is coupled to identify a different destination server. This approach is very simple and effective, but it is vital to know the exact port to reach beforehand. Moreover, several reverse proxies can be to handle BF chaining, but the route must be static or reconfiguration is needed.

B. SOCKS compatible proxy

SOCKS is a standardised proxy protocol for TCP and UDP connections. Our SOCKS compatible proxy implements the latest version of the protocol 5 [8] and the latest draft of its next iteration 6 [9], currently under development. A major latency bottleneck of the latest version is the number of required round-trip times (RTTs) during connection setup (*handshake*). In total, this may be up to 5 RTTs. The next iteration reduces the number of required RTTs, introduces minor tweaks to the protocol, and specifies how SOCKS implementations can, optionally, utilise Multipath TCP [10] and TCP Fast Open [11].

Our SOCKS compatible proxy implementation, illustrated in Figure 5, relies on a redirector component running on all nodes that will be used by the EPIF in the various domains (Section II-A). Using `iptables`, a packet filtering utility for the Linux kernel[2], the redirector component intercepts all outgoing network connections and redirects them to the proxy. As specified by the SOCKS protocol, the redirector will also communicate the intended destination of connections with the proxy. Based on the source, intended destination, and the area logic model (Section III-A), the proxy can correctly process and forward the connections.

For the proxy, processing connections includes applying the appropriate BFs (Sec. III). The SOCKS protocol can also function as a template for the described BF chain mechanism (Sec. III-B). Specifically, SOCKS proxies are chainable and the intended destination of connections can easily be preserved. BFs can share metadata during SOCKS connection

[2]https://www.netfilter.org

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

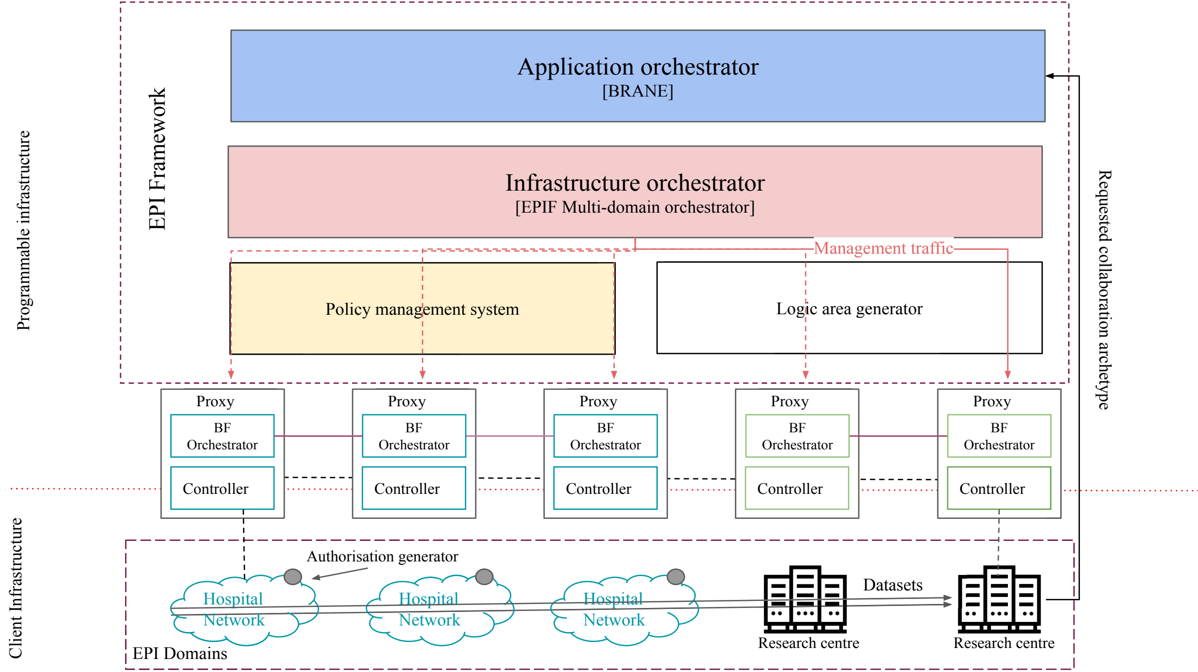


Fig. 3: A figure showing the EPIF architecture with different EPIF components running (including the proxy node).

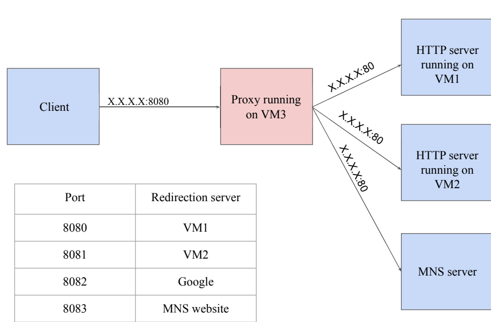


Fig. 4: Reverse proxy implementation, showing the redirection mechanism for data flow redirection through the proxy.

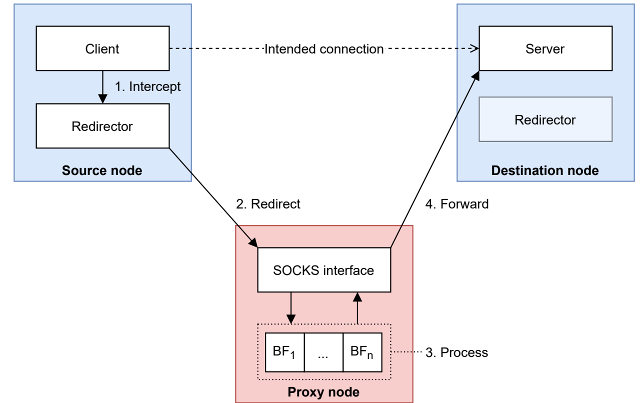


Fig. 5: The SOCKS compatible proxy receives redirected connections that the redirector on the client node intercepts.

setup(s), i.e., chain establishment, to enable inter-BF coordination. Furthermore, BFs with a uniform interface make dynamic and arbitrary chain compositions straightforward. The accumulative latency bottleneck of SOCKS is reduced using the latest iteration of the protocol and becomes negligible when the entire BF chain runs on the same proxy node.

V. Communication with Proxy

To understand the effect of simulating varying network topologies distances, we take a closer look at the sequence of communication per each proxy implementation approach. Fig. 6 illustrates typical sequence communication steps between a client C and a server S in a no-proxy scenario. Throughout this section, we assume that this is an HTTP request over TCP: SYN/ACK+SYN packets are from TCP and the Request/Response represent HTTP messages. We formalise Eq. 6 to estimate the communication setup time in a no-proxy scenario, such that:

$$T_{No-proxy} \approx 2RTT_{CS} + t_{server}, \tag{6}$$

where RTT_{CS} is the time of a single round-trip time from C to S, and t_{server} is the server request processing time. We use the same server container in all scenarios so t_{server} does not change throughout this paper. $T_{No-proxy}$ is dependant on the CS distance, and it is used as a base reference to calculate Δt in other scenarios involving a proxy.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

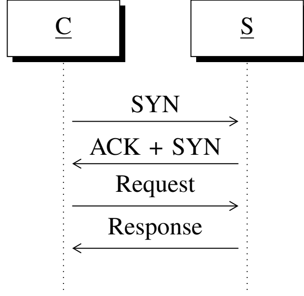


Fig. 6: A sequence diagram showing the communication steps in a no-proxy scenario.

Fig. 7 focuses on the NGINX communication sequence between the client C, proxy P, and the server S. In this diagram we have node P running between C and S, which introduces extra SYN and ACK traffic. Subsequently, we estimate the

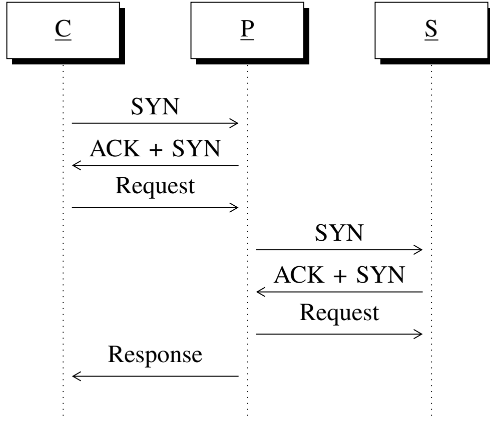


Fig. 7: A sequence diagram showing the communication steps in a scenario including NGINX-based proxy.

setup time of communication passing through an NGINX-based proxy with:

$$T_{NGINX} \approx 2RTT_{CP} + 2RTT_{PS} + t_{server} + t_{proxy}, \quad (7)$$

where RTT_{CP} and RTT_{PS} are the round-trip time between C-P, and P-S, respectively, and t_{proxy} is the processing time of an NGINX proxy.

Moreover, we illustrate a similar communication sequence for SOCKS5-based proxies in Fig. 8, where we introduce redirector element. The redirector always runs on the same host (as showed in Sec. IV-B), meaning that the round-trip time from C to redirector is negligible.

There are extra steps of authentication in this proxy implementation, which implies higher overhead, and an overall higher setup time due to additional round-trips.

The notation of the setup time for a request passing through a SOCK5-based proxy is T_{SOCKS5}, and we estimate it with:

$$T_{SOCKS5} \approx 5RTT_{CP}+2RTT_{PS}+t_{redirector}+t_{proxy}+t_{server}, \quad (8)$$

such that, RTT_{CP} is the round-trip time from C to P (passing through the redirector), and RTT_{PS} from P to S. t_{proxy} is

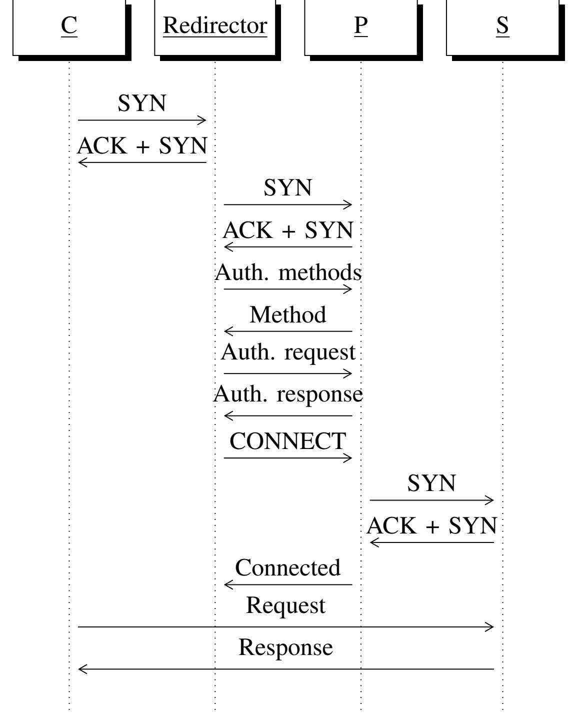


Fig. 8: A sequence diagram showing the communication steps in a scenario including SOCKS5-based proxy.

the SOCKS5 proxy processing time, and $t_{redirector}$ is the redirector processing time.

Similarly, Fig. 9 shows the communication steps with SOCKS6-based proxy. This proxy implementation optimises the required RTTs, which is apparent compared to the SOCKS5 proxy. The notation of the setup time for a request passing through SOCKS6 proxy is T_{SOCKS6}, and we approximate it with:

$$T_{SOCKS6} \approx 3RTT_{CP}+2RTT_{PS}+t_{redirector}+t_{proxy}+t_{server}, \quad (9)$$

where t_{proxy} is the time for the SOCKS6 proxy to process an upcoming forwarding request.

Implementing different proxies implies a different overhead compared to a no-proxy scenario. The evaluation of the two implementations will be detailed in the next section.

VI. Evaluation

To determine which implementation should be adopted, and under which conditions, we benchmark the two approaches (Sec. IV) to evaluate their performance in terms of time overhead and the rate of processed transactions. In our experiments, we fully containerise and automate the benchmark setup in Docker containers for reproducible results. Our benchmark implementation is publicly available online [3].

[3]https://github.com/epi-project/proxy-bench

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

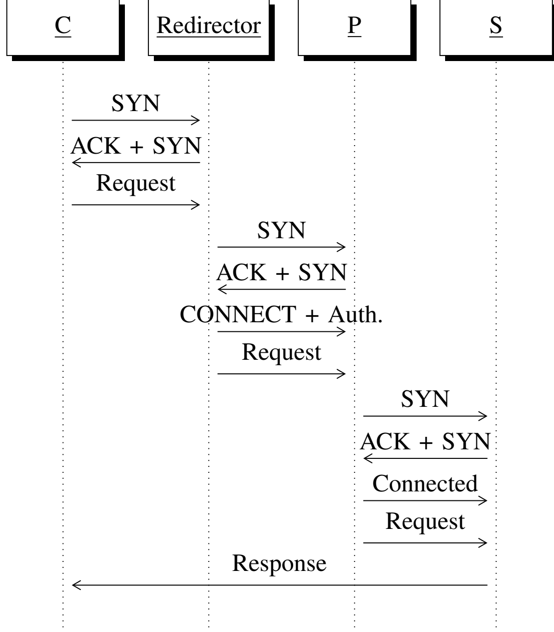


Fig. 9: A sequence diagram showing the communication steps in a scenario including SOCKS6-based proxy.

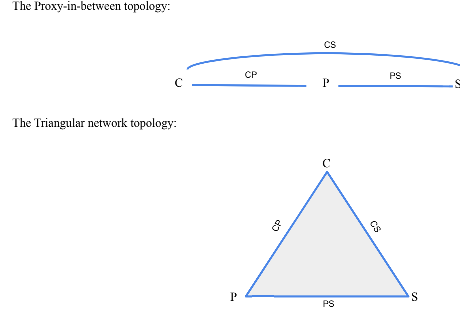


Fig. 10: The different network distance/latency typologies with C representing client node, P the proxy node, and S the server node.

Topology	Name	CP (ms)	PS (ms)	CS (ms)
Proxy-in-between	DOCKER 1	5	5	10
	DOCKER 2	5	10	15
	DOCKER 3	10	5	15
Triangular	DOCKER 4	1	1	1
	DOCKER 5	5	5	5
	DOCKER 6	10	10	10

TABLE I: The six network configurations used in our experiments and the respective latencies; three topologies (1-3) are related to *proxy-in-between* setup and three topologies (4-6) are related to the *triangular* setup

A. *Experiment topologies*

We run three different applications containers: client, web server and proxy. We generate requests traffic from the client to the server passing through the proxy, and use the no-proxy scenario as the baseline for this benchmark, using two network tools: *httping* [12] and *wrk* [13]. We are interested in the additional time that packets take to go through the proxy and give it the Δt notation.

In an attempt to have a controlled environment with reproducible results, we dockerise everything and run it all the application components on a single VM machine. We configure different network configuration combinations; we accomplish this by varying the distance/latency between containers using `tc`. By doing this, we mimic a real network with changing distances between nodes.

We run several network typologies scenarios on a basic Debian version 10 VMs, with 2 cores, 2 GB of RAM, and 20 GB of storage. In one topology, the proxy is placed between the client and the server; we call this the *proxy-in-between* topology. In a second topology, the proxy is placed at a location that is equidistant from client and server; we call this the *triangular* topology. The two types of setup are illustrated in Fig. 10. Namely, CP represents the client-proxy distance, PS the proxy-server distance, and CS the client-server distance (as defined in Sec. V).

The *tc* tool, a network control utility for the Linux kernel, is used to simulate different distances between the nodes to evaluate the effect of varying delay on the time overhead. Table I shows the six different latency configurations we have adopted.

B. *Proxy-in-between topology experiments*

In the first experiment, we place the proxy in between and on the direct path of C and S, and simulate distance differences between the nodes according to the values in Table I: DOCKER1, DOCKER2, DOCKER3. We instantiate the client, proxy, and server containers on a VM and we generate HTTP traffic via the *httping* tool. Then, we measure the average round-trip time of 120 consecutive requests.

The results of the measured round-trip time (*ms*) and the associated overhead Δt are plotted in Fig. 11: 11a and 11b, respectively. As the total distance between client and server increases, so does the total round trip time (Fig. 11a).

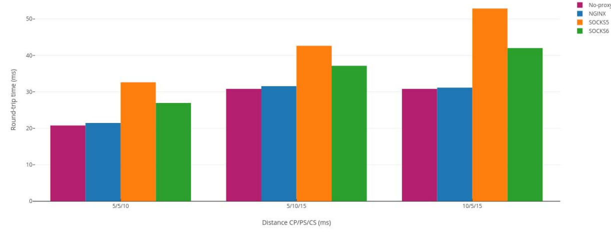
In Fig. 11b the observed Δt of NGINX is $< 1ms$ when it is deployed in between the client and server. If the SOCKS6 proxy is halfway of the distance, then the Δt is $\simeq 6ms$. As for SOCKS5 proxy, Δt is $\simeq 12ms$ and it increases rapidly with the CP distance.

C. *Triangular topology experiments*

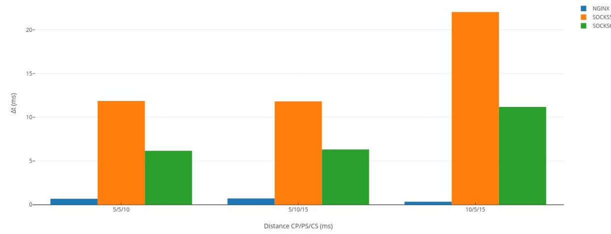
In this experiment, we simulate equidistant triangular topologies with CP/PS/CS values show in Table I: DOCKER 4, DOCKER5, and DOCKER6.

The plots in Fig. 12 shows the resulting round-trip time and Δt with 12a and 12b, respectively. We observe that the resulting overhead of the proxies implementations increases with distance. The overhead values are higher than the values recorded with the proxy-in-between scenarios, ranging from 2 ms to 21 ms for NGINX, from 5 ms to 42 ms for SOCKS5, and 4 ms to 32 ms for SOCKS6 (as shown in 12b).

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

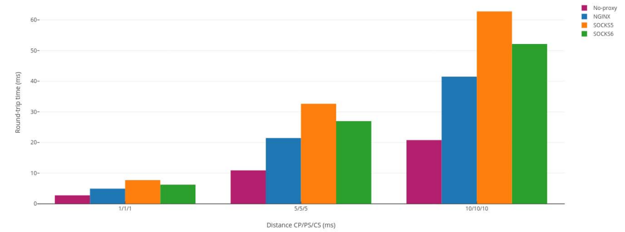


(a) The average of client to server 120 consecutive requests round-trip time (ms) with changing configured distances between nodes.

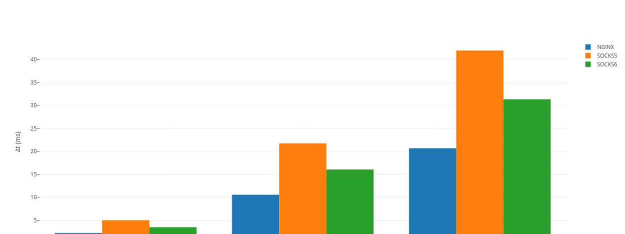


(b) The overhead of Δt (ms) of different proxy implementations compared to no proxy with changing configured distances.

Fig. 11: The variation of round-trip time (ms) and overhead Δt (ms) of proxied HTTPING requests with proxy-in-between network topologies.



(a) The average of client to server 120 consecutive requests round-trip time (ms) generated via HTTPING with changing configured distances between nodes.



(b) The overhead of Δt (ms) of different proxy implementations compared to no proxy with changing configured distances.

Fig. 12: The round-trip time (ms) and overhead Δt (ms) of proxied HTTPING requests with triangular network topologies.

D. Rate of processed transactions

To test the rate of processed transactions of each proxy implementation we set up a third experiment. In this case, we utilise 3 different VMs each running an application container (client—proxy—server). The reason that we don't containerise this setup is that with the *wrk* tool, applications will compete for resources and we'll end up with lower rate results.

Fig. 13 shows the rate of processed transactions of each proxy implementation measured with *wrk* as the number of concurrent HTTP connections increase (Fig. 13a) and the reduction of this rate compared to the no-proxy output (Fig. 13b). The network setup behind this plot simulates no varying distances.

Similar to the no-proxy plot, the rate of processed requests of all three implementations increases with the increasing number of connections, but it flattens it hits 8 concurrent connections. That might be explained due to the proxy reaching a bottleneck of resources consumption.

The bottleneck is further reflected in the second plot in Fig. 13b, the reduction of the rate compared to the no-proxy rate decreases initially to show that processing rate of the three proxy implementations is growing at a higher rate. When we reach 8 concurrent connections, the processing rate flattens out while the no-proxy rate is still increasing. Subsequently, The proxies' reduction rate starts to slightly increase afterwards.

We can concur from this plot that next to no-proxy, the SOCKS6 approach results with the highest processing rate with increasing connections. On the other hand, the SOCKS5 approach has the lowest processing rate and doesn't scale as well compared to SOCKS6 and NGINX.

E. Discussion

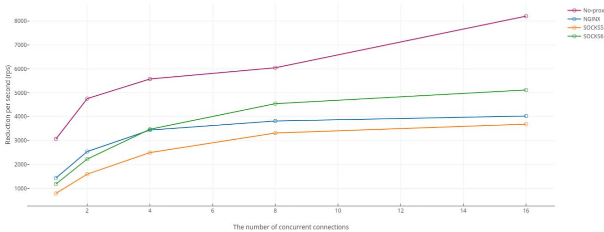
In Fig. 11b The proxies' overhead increases with the increasing distance between the proxy and the client, while it does not noticeably differ with increasing distance between proxy and server. Such a behaviour is logical and expected given the contributions to the total overhead of the various components as explained in Sec. V.

While in Fig. 12b we notice that the NGINX proxy performs better in terms of Δt in both setups. However, the SOCKS-based proxies imply more overhead, which is expected since it needs more authentication steps during connection setup (handshakes), while the NGINX proxy simply forwards requests.

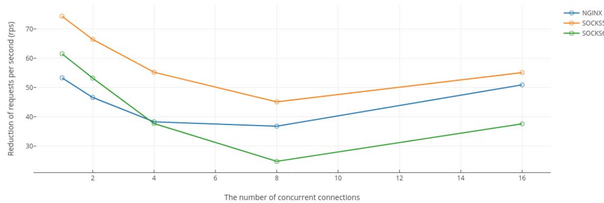
The overhead resulting from these experiments are varying according to $\Delta t = T_{NGINX|SOCKS5|SOCKS6} - T_{No-proxy}$. This explains the smaller overhead in the proxy-in-between topologies that implies higher RTT_{CS} time compared to triangular topologies.

Experiments in Sec. VI-B and VI-C show that the placement of the proxy node has a noticeable effect in terms of overhead. Moreover, it is proven to be a way of minimising the inevitable latency of introducing proxies.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.



(a) The rate of processed transactions resulting via *wrk* of different proxy implementations with increasing concurrent connections.



(b) The reduction of processed requests per second of different proxy implementations compared to no-proxy.

Fig. 13: The number of HTTP requests that are processed per second (rps) and the associated reduction with the increasing number of concurrent requests.

VII. Comparison

There are other evaluation performance parameters of each proxy that we want to consider. Here we look at the port scalability, optimisation, portability and reconfiguration, dynamicity, complexity and security.

Table II shows the comparison between different proxy implementations according to the previously defined parameters.

Parameters	NGINX	SOCKS5	SOCKS6
Δt	✓		
Processing rate			✓
Port scalability		✓	✓
Reconfiguration		✓	✓
Dynamicity		✓	✓
Security		✓	✓

TABLE II: The comparison between different proxy implementations according to six performance parameters; where the ✓represents an advantage over other proxies.

A. Port scalability

Due to their different mechanisms, different numbers of ports need to be open on a proxy server. As an example, we need one port for each proxy for $SOCKS(5|6)$, while one port per nodes pair of nodes for NGINX. This reflects on the hardware scalability of the proxy with higher requesting nodes.

B. Optimisation

NGINX is optimised for HTTP (more optimised in general as well), while SOCKS works on top of the TCP/UDP transport layer. This is reflected in the results of the experiments illustrated in Fig. 12 and 11.

C. Reconfiguration

NGINX is configured through configuration files, and subsequently, reconfiguration of the service might require a restart (downtime might be avoided through canary (re)deployment). On the other hand, SOCKS configuration is dynamic/programmable, and it is updateable without restart/downtime, *e.g.* database entry update.

D. Dynamicity

NGINX paths are static (based on the incoming port used), thus BF chains are static as well. Separate BF chains (x containers) needs to be deployed per node pair, thus scaling is also done per node pair BF chain. With SOCKS the destination is communicated as part of the connection setup, thus BF containers can remain stateless (assuming a BF implements a SOCKS interface). Meaning that, one BF container can be part of multiple BF chains *e.g.* through dynamic function composition $g(f(x))$ based on SOCKS chaining. Scaling can be done per BF across different nodes pair paths. (i.e. scale per number of connections to the proxy, instead of the number of connections per nodes pair.)

E. Security

From a security perspective, SOCKS authentication is part of the protocol. Whitelist, mutual TLS, and/or shared secret are supported. That is not part of SOCKS but possible, and handled by redirector component. On the other hand, with NGINX whitelist and mutual TLS are supported, but it is not transparent to client app.

VIII. Related work

Different redirection tools have been implemented in the past to fulfil a number of different purposes. In [14] they propose a dynamic hybrid honeypot system based on a transparent traffic redirection mechanism to address the identical fingerprint problem. They implement the Honeybrid gateway to capture data and control traffic. The Honeybrid gateway includes a Decision Engine and a Redirection Engine, which are in charge of orchestrating the filtering and the redirection between frontends and backends. The Decision Engine is used to select interesting traffic, and the Redirection Engine is used to transparently redirect the traffic. They employ a traditional TCP proxy that applies the TCP relay mechanism. In our case, redirection relies on the logic area generator, which has deep knowledge and integration with the policy engine; the Decision Engine in [14] would not be easy to adopt. Likewise, the Redirection Engine provides also orchestration function, which in the EPIF are already integral components of the application and infrastructure orchestrators.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.

For similar security reasons to ours, [15] employs a Subscriber Traffic Redirection software to redirect HTTPS requests via a redirection server. Similar to our proposed implementation, the server redirects traffic to a previously configured web server with a Secure Sockets Layer certificate (SSL). This tool is evaluated according to security parameters, such as security against DDoS attacks. In our case though, we need a proxy that is capable to redirect all kinds of application traffic, not only HTTPS.

Finally, [16] evaluates high availability of two proxy implementations: reverse proxy and SDN-based proxy, based on OpenFlow. The conclusion is that the SDN proxy proves to be robust against link failure due to its ability to monitor network interfaces. The idea of using an SDN-based proxy is certainly relavent in our effort to exploit programmable infrastructures. Still, OpenFlow is not any longer the main SDN technology, and it must be noted that most of the current EPI infrastructures would not have OpenFlow devices running in them. We foresee to evaluate data plane programmable solutions such as P4.

IX. Conclusion & Future work

Interception and redirection of traffic is a core feature of the EPIF. In this paper, we evaluated and benchmarked two different approaches; an NGINX-based reverse proxy method and a SOCKS-based method. We determined that the overhead of the EPIF proxies differs in the various implementations, and it relates directly to the positioning of the proxy within a network topology. Other than overhead, we also considered different performance parameters in evaluating the proxies implementations. We can conclude that the SOCKS6-based proxy offers the highest processing rate, and it supports all traffic type redirection. In addition to that, SOCKS proxies have advantages in terms of reconfiguration, dynamicity, security, and scales better in employing open ports. On the other hand, the NGINX-based approach processes forwarding requests with lower overhead.

Subsequently, to make a design decision on which proxy implementation should be used, we need to consider the type of application being served. Namely, the choice depends on the application requirements and the specific relevance of performance parameters. For example, if the application is time-critical and would require minimal overhead, then NGINX is the obvious choice. A data streaming application would instead benefit from a SOCKS6-based proxy because of the higher processing rate.

In our future work, we will be implementing more EPIF functionalities like BF chaining and offering uniform interfaces of bridging functions. This would enable programmability and add on the initial client infrastructure capabilities. Our focus will therefore be put on the extra plug-ins needed in the redirection tools needed for BF's chaining. Finally, we aim to deploy this with real test-beds and utilise this by introducing real health data and different archetypes.

Acknowledgement

The EPI project is funded by the Dutch Science Foundation in the Commit2Data program (grant no: 628.011.028).

References

[1] N. Omnes, M. Bouillon, G. Fromentoux, and O. L. Grand, "A programmable and virtualized network it infrastructure for the internet of things: How can nfv sdn help for facing the upcoming challenges," in *2015 18th International Conference on Intelligence in Next Generation Networks*, 2015, pp. 64–69.

[2] "Network functions virtualisation (nfv)." [Online]. Available: https://www.etsi.org/technologies/nfv

[3] J. A. Kassem, C. De Laat, A. Taal, and P. Grosso, "The epi framework: A dynamic data sharing framework for healthcare use cases," *IEEE Access*, vol. 8, pp. 179 909–179 920, 2020.

[4] M. I. Mccarthy, "Painting a new picture of personalised medicine for diabetes," *Diabetologia*, vol. 60, no. 5, p. 793–799, 2017. [Online]. Available: https://link.springer.com/article/10.1007/s00125-017-4210-x?&utm_medium=other&utm_source=other&utm_content=5022018&utm_campaign=10_dann_ctw2018_5_med_38

[5] A. Moncada-Torres, F. Martin, M. Sieswerda, J. van Soest, and G. Geleijnse, "Vantage6: an open source privacy preserving federated learning infrastructure for secure insight exchange," in *AMIA Annual Symposium Proceedings*, 2020, pp. 870–877.

[6] O. Valkering, R. Cushing, and A. Belloum. Brane: Programmable Orchestration of Applications and Networking. [Online]. Available: https://doi.org/10.5281/zenodo.3890928

[7] Nginx reverse proxy. [Online]. Available: https://docs.nginx.com/

[8] M. D. Leech, "SOCKS Protocol Version 5," RFC 1928, Mar. 1996. [Online]. Available: https://rfc-editor.org/rfc/rfc1928.txt

[9] V. Olteanu and D. Niculescu, "SOCKS Protocol Version 6," Internet Engineering Task Force, Internet-Draft draft-olteanu-intarea-socks-6-11, Nov. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-olteanu-intarea-socks-6-11

[10] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 8684, Mar. 2020. [Online]. Available: https://rfc-editor.org/rfc/rfc8684.txt

[11] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "TCP Fast Open," RFC 7413, Dec. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7413.txt

[12] httping(1) - linux man page. [Online]. Available: https://linux.die.net/man/1/httping

[13] wg/wrk. [Online]. Available: https://github.com/wg/wrk

[14] W. Fan, Z. Du, D. Fernández, and X. Hui, "Dynamic hybrid honeypot system based transparent traffic redirection mechanism," in *Information and Communications Security*, S. Qing, E. Okamoto, K. Kim, and D. Liu, Eds. Cham: Springer International Publishing, 2016, pp. 311–319.

[15] "Traffic redirection overview." [Online]. Available: https://www.juniper.net/documentation/en_US/src4.13/topics/concept/redirect-server-overview.html

[16] E. Konidis, P. Kokkinos, and E. Varvarigos, "Evaluating traffic redirection mechanisms for high availability servers," *2016 IEEE Globecom Workshops (GC Wkshps)*, 2016.

Authorized licensed use limited to: Universiteit van Amsterdam. Downloaded on May 10,2022 at 15:33:07 UTC from IEEE Xplore. Restrictions apply.