

# FUSE: a Microservice Approach to Cross-domain Federation Using Docker Containers

Tom Goethals<sup>1</sup> <sup>a</sup>, Dwight Kerkhove<sup>1</sup>, Laurens Van Hoye<sup>1</sup> <sup>b</sup>, Merlijn Sebrechts<sup>1</sup> <sup>c</sup>, Filip De Turck<sup>1</sup> <sup>d</sup>, Bruno Volckaert<sup>1</sup> <sup>e</sup>

<sup>1</sup>Department of Information Technology, Ghent University - imec, IDLab, Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium  
[togoetha.goethals@UGent.be](mailto:togoetha.goethals@UGent.be)

Keywords: Collaboration, Unified Services, Microservices, Federation, Containers


Abstract: In crisis situations, it is important to be able to quickly gather information from various sources to form a complete and accurate picture of the situation. However, the different policies of participating companies often make it difficult to connect their information sources quickly, or to allow software to be deployed on their networks in a uniform way. The difficulty in deploying software is exacerbated by the fact that companies often use different software platforms in their existing networks. In this paper, Flexible federated Unified Service Environment (FUSE) is presented as a solution for joining multiple domains into a microservice based ad hoc federation, and for deploying and managing container-based software on the devices of a federation. The resource requirements for setting up a FUSE federation are examined, and a video streaming application is deployed to demonstrate the performance of software deployed on an example federation. The results show that FUSE can be deployed in 10 minutes or less, and that it can support multiple video streams under normal network conditions, making it a viable solution for the problem of quick and easy cross-domain federation.


## 1 INTRODUCTION


In crisis situations, a crisis center is usually formed which monitors the situation and takes suitable actions. It is important that crisis centers are able to quickly gather information from various sources, for example closed-circuit television (CCTV) or positional data, to get a complete and accurate overview of the situation. When the required information sources are owned by different companies, it is difficult for the crisis center to gain access to them or to interact with them uniformly. Fig. 1 illustrates the concept of joining companies and a crisis center into a single virtual network, from now on referred to as a federation. Company A joins the federation, making its IP cameras available for use by routing their streams, and company B allows the use of both its servers and sensory hardware. The crisis center gathers information from the exposed devices of both companies and


transforms it into a dashboard for its operators. To do this, a *federation service environment* is required which supports a wide range of operating systems and devices. Since containers are widely supported and easy to deploy, a container-based service environment is preferable.


In addition to varying devices and software platforms, companies participating in a federation often have different network and security policies, making it difficult to connect them quickly or to deploy software in their networks (domains) in a uniform way. Any solution to connect multiple domains should ensure that every company can choose exactly which resources it makes available to the federation, and that all communication between federated devices is secure. Only devices that are part of the federation must be visible from other domains, while devices from each domain that are not part of the federation should only be reachable from federated devices in the same domain. For example, in Fig. 1 the data storage of company B can be used by federation components deployed in its own domain, but it is unavailable to the rest of the federation. The same is true for the non-federated device at company A, which is invisible to the devices of company B and the crisis center.

<sup>a</sup>  <https://orcid.org/0000-0002-1332-2290>

<sup>b</sup>  <https://orcid.org/0000-0003-1192-4631>

<sup>c</sup>  <https://orcid.org/0000-0002-4093-7338>

<sup>d</sup>  <https://orcid.org/0000-0003-4824-1199>

<sup>e</sup>  <https://orcid.org/0000-0003-0575-5894>

Apart from being cross-domain and ensuring secure communications, a federation service environment focused on crisis situations must also be fast and easy to set up. Joining a federation should only take minutes, with minimal intervention from company administrators. Additionally, a company should be able to join or leave a federation at any time without destabilizing the federation, and after leaving a federation no trace of the federation service environment should be left on a company's devices. Furthermore, the components of a federation service environment should not interfere with other processes on a device, which means that the entire federation service environment and all its components should be isolated as much as possible.

The challenges for building a federation service environment using containers can thus be summarized as follows:

1. Enabling and securing fast cross-domain communication while restricting access to non-federated resources
2. Isolating the federation service environment from other software
3. Ensuring fast and easy deployment of the federation service environment on a large range of devices

This paper presents Flexible federated Unified Service Environment (FUSE) to tackle these challenges. FUSE provides a microservice-oriented, container-based service environment to deploy and manage software on federated domains. It is designed to quickly set up ad hoc federations with minimal intervention, ensures secure communication between domains and prevents non-federated devices from being visible from other domains.

Section II presents related work to the challenges presented in this introduction, and Section III describes how they are solved in FUSE. Section IV describes the test setup for a basic FUSE federation, while section V details the system requirements of FUSE and presents performance results for a typical use case. Section VI discusses the results and applications of FUSE, and suggests some topics for future work. Finally, section VII concludes the paper.

## 2 RELATED WORK

Previous federation service environment projects have resulted in frameworks such as Fed4Fire (Wauters et al., 2014), Beacon (Moreno-Vozmediano et al., 2016) and FedUp! (Bottoni et al., 2016).

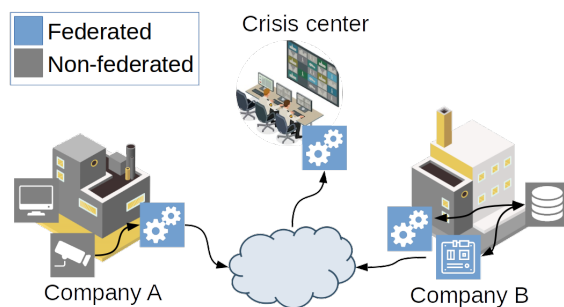


Figure 1: Example federation.

Fed4Fire has a different use case from FUSE and requires the implementation of an API to integrate devices into a federation, which makes it inadequate for the rapid ad hoc use cases of FUSE. BEACON is focused on cloud federation and security as a function of cloud federation, but the use case of FUSE requires it to work in company networks and around existing and unchangeable security policies. FedUp! is a cloud federation framework focused on improving the setup time for heterogeneous cloud federations. Unlike previously mentioned frameworks, FUSE operates on company networks rather than cloud infrastructure and aims to cut down set-up time to minutes or less with minimal intervention from system administrators, independent of target devices and operating systems. Many tools have been created for the different aspects of federation, for example jFed (Vermeulen et al., 2014) (general lifecycle management), OML<sup>1</sup> (measurement and monitoring) and OpenID (Recordon and Reed, 2006) (trust, user authentication). Each of these tools only solves part of the problem FUSE faces and often requires the implementation of specific APIs to work with. For example jFed, which was developed for the Fed4Fire project, provides high-level control over federated resources, but it only works with known hardware types and pools of pre-configured resources.

Container engines such as Docker<sup>2</sup> and rkt<sup>3</sup> have been studied and evaluated extensively in literature (Cailliau et al., 2016), for example in performance reviews (Felter et al., 2015) and in overviews of virtualization technology (Bernstein, 2014). Similarly, the capabilities of container orchestration tools such as Kubernetes<sup>4</sup> and its precursor Borg are explored in various studies (Verma et al., 2015; Casalicchio, 2017). Microservice architectures using containers have seen a lot of attention, specifically for their use

<sup>1</sup><https://wiki.confine-project.eu/oml:start>

<sup>2</sup><https://www.docker.com/>

<sup>3</sup><https://coreos.com/rkt/>

<sup>4</sup><https://kubernetes.io/>

in rapid and easy deployment of (cloud) applications (Kratzke, 2017; Amaral et al., 2015). However, Kubernetes by itself is insufficient to build a federation service environment when the devices in the federation are hidden from public view by firewalls or by other means, which is often the case in company networks.

Docker security has been thoroughly studied (Bui, 2015), and there are security best practices for Kubernetes (Kubernetes, 2016). However, no work is found on securing network traffic specifically, which is required when sending valuable data between domains over the internet. Kubernetes is capable of forming federations of multiple Kubernetes clusters (Kubernetes, 2018), but to the best of our knowledge, no work has been done on a single Kubernetes cluster spanning multiple physical domains.

Certain studies investigate the usefulness of edge computing and edge offloading (Shi and Dustdar, 2016; Samie et al., 2016), two concepts whereby computing workload is moved from cloud hardware to edge devices (or vice versa) based on hardware load and service demand. Of particular interest are studies where virtualization is employed for edge offloading purposes (Morabito et al., 2018). This work is closely related to how and why federations could include edge devices to serve as information sources or processing hardware.

Various aspects of cloud resource management have been studied (Jennings and Stadler, 2015), for example the scalability of certain topologies. Studied topologies include a centralized controller (Atrey et al., 2016), management hierarchies (Whaiduzzaman et al., 2014; Wang and Su, 2015) and fully distributed approaches (Miraftabzadeh et al., 2017).

### 3 COMPONENTS AND ARCHITECTURE

Within a single domain, a federation can easily be formed using Docker containers and Kubernetes. The use of Docker containers ensures that software can be deployed to a wide variety of target devices, as long as they support Docker. Kubernetes is used to join and manage all the devices in the federation, and to deploy software on those devices. The only downside to this approach is that software needs to be containerized in order to deploy it.

Kubernetes identifies the roles of specific devices in a cluster by making them either a master or a worker node. FUSE, being built around Kubernetes, adopts these two roles while giving them additional responsibilities. Master nodes are the equivalent

of Kubernetes masters and consist of a Kubernetes control plane and other services required for FUSE. Worker nodes perform the function of Kubernetes workers and generally only contain a kube-proxy and deployed containers.

The rest of this section details how FUSE solves the problems posed in the introduction and describes how they fit into the basic Docker and Kubernetes setup described here.

#### 3.1 Cross-domain Federation and Security

Kubernetes deploys containers in groups called pods, which have their own virtual network for communication between all devices that constitute a cluster. This inter-pod communication is done with the aid of a Container Network Interface (CNI) driver<sup>5</sup>, such as Flannel<sup>6</sup>, which assigns an IP address from a configurable range to every pod running on the Kubernetes nodes under its control. However, Flannel runs into problems when a Kubernetes node is unreachable, for example when it is hidden behind a firewall or when no route to the target machine exists. Kubernetes, being primarily built for cloud environments, has no facilities to work around this problem.

In order to enable Flannel traffic between domains, and to secure that traffic, OpenVPN tunnels<sup>7</sup> are created between FUSE worker nodes and the master node. Every FUSE master node runs an OpenVPN server, while all FUSE nodes (including the master) have an OpenVPN client that connects to the VPN server on the master node. Once a connection is established, a FUSE node gets an IP address from a configurable range of VPN addresses. This address is added to routing tables in FUSE services, together with the Flannel address range of a node. After this initial setup, all of the ports on a node's OpenVPN interface are forwarded to Flannel. To optimize performance, Flannel is run using the host-gw<sup>8</sup> back-end instead of vxlan, which puts pod network packets directly on the OpenVPN interface instead of encapsulating them. The downside of using only one OpenVPN server, running on the master node, is that all traffic is routed via the master node, even if it is just between worker nodes.

Apart from enabling cross-domain federation and securing communications, FUSE also needs to en-

<sup>5</sup><https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/>

<sup>6</sup><https://github.com/coreos/flannel>

<sup>7</sup><https://openvpn.net/>

<sup>8</sup><https://github.com/coreos/flannel/blob/master/Documentation/backends.md>

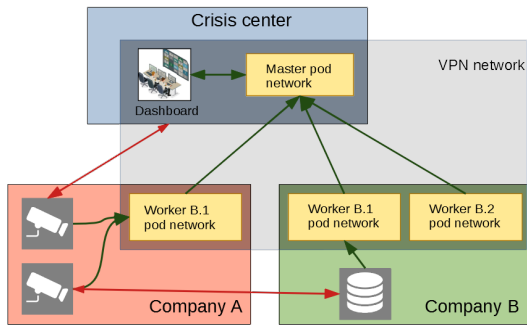


Figure 2: High-level network overview of an example FUSE federation.

sure that non-federated devices can not be reached from a domain other than the one they are in. This is achieved by generating specific routing rules for each node in a federation. The concept is shown in Fig. 2, where a red, green or blue box represents a company network (domain), yellow boxes represent the distributed parts of the Kubernetes pod network and the gray translucent box represents the VPN network which connects all FUSE nodes. Green arrows indicate which devices can interact with each other, while red ones show which ones can not. In company A, worker B.1 can receive camera streams from non-federated devices in company A through its company-assigned IP address, and it is also able to forward the stream to the crisis center over its FUSE VPN-assigned IP address. However, non-federated devices in company A can not be reached by any devices from either the crisis center or company B. This approach solves the first challenge posed in the introduction.

### 3.2 Encapsulation

To tackle the second challenge discussed in the introduction, a solution is needed that isolates FUSE components and minimizes the required software to deploy FUSE. All services deployed on a FUSE worker node are containerized, but the components of FUSE should be isolated as well. Docker-in-Docker (DinD<sup>9</sup>) enables the nesting of Docker environments by deploying a containerized Docker environment within another Docker environment. Using a DinD approach, FUSE components and services are deployed in the outer docker environment, while the inner environment runs containers deployed on the node by Kubernetes. Thus, FUSE processes remain isolated from other processes and a Docker installation is the only requirement to start a FUSE node. Ad-

<sup>9</sup><https://github.com/kubernetes-sigs/kubeadm-dind-cluster>

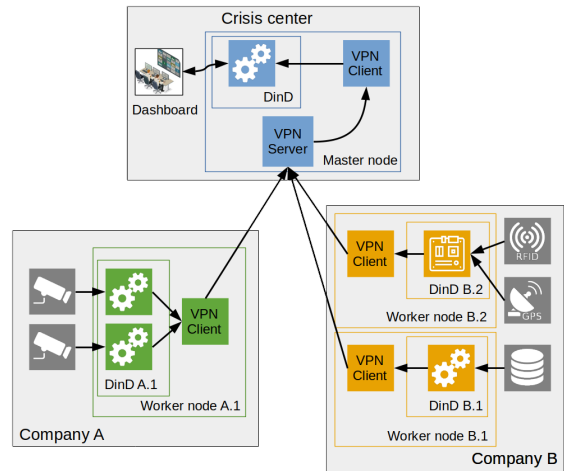


Figure 3: Nodes and information flow of an example FUSE federation.

ditionally, FUSE can ensure that none of its components remain on a device after it leaves the federation. However, both FUSE components and client software must be containerized in order for this approach to work. For OpenVPN, this means using an OpenVPN server container on the master node and OpenVPN client sidecars (Burns, 2018) on the worker nodes.

Using DinD creates an additional network layer between the host OS and the Kubernetes node. As previously mentioned, Flannel takes care of network traffic for the pods in the inner Docker environment, but the outer Docker environment still needs an addressing scheme. For this layer, a static address was chosen for each FUSE service, no matter which node it runs on, so that a FUSE node always knows where to reach a certain service running on it. For example, a VPN server container IP address always ends in .2, while a VPN client container address always ends in .3.

Fig. 3 shows the federation illustrated in Fig. 1, with the concepts discussed in this section. Fig. 3 shows that every node, even the master node, has a VPN client which is connected to the master node's VPN server. Company A has a single worker node, which is running multiple containers to forward video streams to the crisis center through a single VPN client. Company B, on the other hand, has two worker nodes, each with their own VPN clients. Node B.1 has a single container which gives the crisis center access to data from company B, and node B.2 gathers positional data from various sources and makes it available to the crisis center.

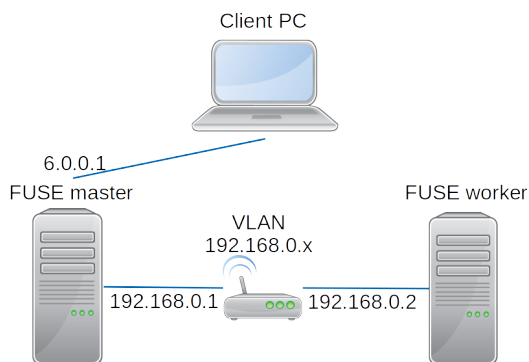


Figure 4: Test setup overview.

### 3.3 Fast and Easy Deployment and Teardown

The DinD solution from the previous subsection enables easy deployment and teardown of FUSE. Since all FUSE components are running in containers, only a single startup script is needed to deploy FUSE on a node or to remove it from a node, with any required containers being pulled from a remote or local Docker registry. Thus, DinD also satisfies the third requirement from the introduction, but tests are needed to confirm that this solution can run on a wide range of devices.

## 4 TEST SETUP

Considering the use cases of FUSE, it is important that it can be deployed on a large range of devices and that both master and worker nodes can quickly join the federation when needed. These requirements are also part of the challenges stated in the introduction, of which the first one demands that FUSE has good network performance, and the third one demands that FUSE is fast to set up and tear down on a wide range of devices.

To confirm that FUSE meets these requirements, measurements were performed to determine memory consumption, required hard disk space, deploy times for nodes and network performance of the federation. Being able to run FUSE on a Raspberry Pi 3 is set as a concrete goal for low-end devices. All tests were performed on the imec/IDLab Virtual Wall environment<sup>10</sup> using bare metal servers. Special care was taken to ensure that the hardware of all servers was

<sup>10</sup><http://doc.ilabt.iminds.be/ilabt-documentation/virtualwallfacility.html>

identical for every test run. The hardware configuration used for every device consists of two Intel Xeon E5620 processors clocked at 2.4 GHz, 12GiB DDR3 memory and a 16GiB partition on a 160GiB WDC WD1600AAJS-0 hard drive.

Fig. 4 shows the test setup. The fuse master and worker nodes are connected by a VLAN which allows setting specific amounts of packet loss and delay. Meanwhile, a client PC is connected to the FUSE VPN and interacts with Kubernetes services via the VPN IP address assigned to the master node. By default, the 6.0.0.x address range is used, but it is configurable. While the client PC is not a part of the virtual wall environment, it is directly connected over LAN, with a ping time of only 0.24 +- 0.01ms.

The required hard disk space for a node was measured by summing the disk usage of all the required containers and FUSE scripts. Memory consumption was determined by comparing available memory as reported by the free command<sup>11</sup>, before and after deploying FUSE. Network throughput was determined with iperf3<sup>12</sup> in the DinD container on both FUSE nodes and running it in both TCP and UDP mode. The resulting physical traffic is thus TCP-on-TCP and UDP-on-TCP, respectively.

To determine how quickly FUSE can set up a federation and how quickly worker nodes can join or leave the federation, the time command was used to measure the duration of the relevant FUSE commands. The final results were calculated from ten successful runs of each test.

As a test of FUSE performance in a typical use case, a container is deployed to the worker node which simulates a camera stream by looping a 720p video file, recorded from a security camera, to a web-socket using ffmpeg<sup>13</sup>. The video has mostly static images with little movement and is encoded at 2 Mb/s. The frame rate of the recording is 25 frames per second (FPS), but the output frame rate is not limited. Because the frame rate is not limited and the video stream consists of mostly static scenery, this is a good test of FUSE network performance. However, in real crisis situations, there would likely be a lot of activity on camera streams, possibly influencing the frame rate from one moment to the next. Since this would be harder to quantify, the FPS test is only meant as an indication of FUSE bandwidth. A dashboard application container, consisting of a web page with a Node.js<sup>14</sup> back-end, is deployed on the master node. The dashboard back-end receives the

<sup>11</sup><http://www.lininfo.org/free.html>

<sup>12</sup><https://iperf.fr/iperf-download.php>

<sup>13</sup><https://www.ffmpeg.org/>

<sup>14</sup><https://nodejs.org/en/>

worker node’s stream via a websocket and sets up a proxy websocket, which in turn sends the data to the web page opened by the client PC. The proxy websocket acts as an aggregator for multiple streams, but for the tests below only one stream was used. The web page itself plays the video using `jsmpeg`<sup>15</sup>. The results, measured in fps, were calculated by hooking into `jsmpeg`’s render loop and calculating the average and standard deviation over 70 seconds of streamed video.

## 5 RESULTS

### 5.1 Hardware Requirements

Worker nodes require 529 MiB of disk space and are small enough that they can be deployed on a Raspberry Pi 3 or similar devices. However, in order to have enough room to deploy software, several gigabytes of free space would be recommended. Master nodes require 1576 MiB of disk space, which is about three times as much as a worker node. This makes sense, since they need to deploy all FUSE components, a VPN server and a Kubernetes master. Considering their role as communications and management hubs for the federation, master nodes will usually be deployed on hardware with orders of magnitude more free space than the required amount, so this should not be a problem.

Concerning memory consumption, a worker node could easily be deployed on a Raspberry Pi 3, since it needs only about 228 MiB free memory. Master nodes need around 851 MiB free memory, which is almost four times more than a worker node. Again, this is due to having to run Kubernetes and all FUSE services. It would be very hard to deploy a master node on hardware with 1 GiB RAM, even with an extremely slimmed down host OS.

### 5.2 Federation Setup and Teardown

Since master nodes can be started up front and kept ready-to-go, their start times are less important than those of worker nodes. They have no interaction with any other devices while deploying, so no special cases need to be examined.

Table 1 shows the minimum, median and maximum observed times it takes to set up a FUSE master node or tear it down. A master node takes only about 6 to 6.5 minutes to set up from scratch, while it can be removed from a device in about 8 to 9 seconds.

	Create	Leave
Min time (s)	356	7.87
Median time (s)	374	7.90
Max time (s)	381	8.90

Table 1: FUSE master node create and leave times.

For worker nodes, the quality of the network connection to the master node is important for federation setup and tear down. To examine the impact of the connection quality, a range of combinations of communication delay and packet loss were simulated. Delay ranges from 0ms to 400ms in 100ms steps, while packet loss ranges from 0% to 20% in 5% steps.

Considering all the network layers in the FUSE architecture, it is hard to model performance using existing research. FUSE traffic consists of TCP or UDP packets from containers wrapped in TCP packets by OpenVPN. A worker attempting to join a federation performs a number of requests. The execution time of a single request over TCP is

$$t_{op} = (w_s + \frac{w_v(d_v + d_s)}{1-l}) \quad (1)$$

where  $l$  is packet loss,  $d_v$  is network delay,  $d_s$  is delay resulting from handling network operations in software,  $w_s$  is work not influenced by network activity (for example, parsing JSON response data), and  $w_v$  represents work that depends on network performance.

However, this only works for a single request. A federation operation, for example joining a federation, requires several calls to web services. This introduces another factor caused by service call timeouts, which in turn can cause a retry of the entire operation:

$$t_{total} = \frac{t_{op}}{(1 - \max(\min(\frac{d_v - d_l}{d_c - d_l}, 1), 0))(1-l)} \quad (2)$$

Where  $d_l$  is the delay threshold below which no operation should time out and  $d_c$  is the critical delay threshold above which every operation results in a timeout. For this equation, delay is in the denominator because its effect is no longer linear. The constants in Eq. 1 and Eq. 2 have to be determined empirically and are different for every hardware setup, but with the results in Fig. 5 the most important effects on the test setup can be identified.

Fig. 5 shows the time it takes to join a federation for several combinations of delay and packet loss. In case of smooth network performance, meaning less than 100ms delay or less than 10% packet loss, joining a federation only takes as much as 1 to 4 minutes. As the model predicts, packet loss has a strong hyperbolic effect on the time it takes to join a federation, but

<sup>15</sup><https://github.com/phoboslab/jsmpeg>



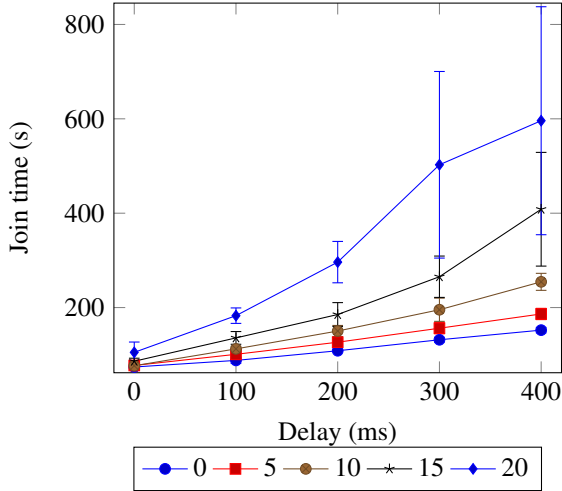


Figure 5: Federation join time for increasing delay at several percentages of packet loss.

it is still doable even with high rates of packet loss and high delay. Delay has a mostly linear effect, as shown by Eq. 1. Only for combinations of high delay and a lot of packet loss does it turn to a slight hyperbolic effect, explained by its term in Eq. 2. Further modeling is unreliable because of the large error margins on these data points. Eventually, around 20% packet loss and 400ms delay joining becomes so slow and erratic that it is unlikely to still be practical. The error margin on this data point shows that join attempts may take anywhere from 6 to 14 minutes.

### 5.3 FUSE Network Performance

Fig. 6 shows the communication speed between a FUSE master node and a worker node with packet loss and delay set to zero. While TCP performance is only 109 Mbits/s, the test setup has no support for hardware encryption using the AES-NI instruction set, which would give much better results (OpenVPN, 2018). UDP performance is very low with only 11.7 Mbits/s, meaning any FUSE traffic should be kept to TCP as much as possible. During the tests, OpenVPN used between 60% and 100% of a single CPU core. Since OpenVPN runs on a single thread, these results are about as good as they can get without optimizations. This means that communication alone takes a large part of the total processing power of a node.

For the video streaming test, the delay between the FUSE nodes was set from 0 to 100ms in steps of 25ms, while the effect of packet loss was examined for 0%, 0.2%, 1% and 2%. Because the software involved in this test does not use buffering, the results are a good reflection of network performance.

Since a TCP video stream is one-way traffic that

requires no response, a different model is used to predict performance than for joining a federation. Only packets that do not make it on the first send incur a penalty on the rendering process. Uniform delay has no effect on the quality of video, merely delaying the rendering of each frame by the same amount of time, resulting in Eq. 3 for the time to transmit and render a frame. In this case  $n$  is the number of packets that need to be sent for a frame,  $d$  is the network delay, and  $l$  is the packet loss.  $d_0$  is an intrinsic delay that occurs from endpoint handling of network traffic and the speed limit of electronic communication.

$$t_f = d_0 n + \sum_{i=1}^{\infty} n(d + d_0)l^i = d_0 n - \frac{(d + d_0)ln}{l - 1} \quad (3)$$

For a  $d_0$  that is sufficiently small compared to the delay caused by network problems and bad connections, Eq. 3 can be reduced to Eq. 4 to estimate FPS for video throughput. Since  $n$  can not be reasonably estimated for any frame, it is replaced by  $t_0$ , which represents the time it takes to transmit an average frame under ideal circumstances (no packet loss, only  $d_0$  delay).

$$f = \frac{1}{t_f} \approx \frac{1}{t_0(1 - \frac{dl}{l-1})}, d_0 \ll d \quad (4)$$

Fig. 7 shows the results of the video throughput test, for which 24 FPS is set as the minimum acceptable framerate. Despite the static bit rate of the source video, the standard deviation at every data point is nearly always over 20% of the average at that point, showing that there is a large fluctuation in performance. During testing, it was verified that this is not a side effect of the rendering process of jsmpeg, but because jsmpeg actually receives variable amounts of data each second. This variation is directly related to FPS variation.

From the simplified model, performance for 0% packet loss would be expected to remain level instead of slowly declining, but this is the result of a tiny amount of packet loss intrinsic in all systems. Even for as little as 0.05% packet loss, Eq. 4 shows a significant decline, which seems locally linear rather than hyperbolic for the examined range of delay. It was verified with tcpdump<sup>16</sup> and Wireshark<sup>17</sup> that a minute amount of packet loss was indeed present.

Similarly, the data points for 0ms delay for the different levels of packet loss do not all have the same value because of the intrinsic delay  $d_0$ . Since in this case the conditions for Eq. 4 are violated, it is better to go with Eq. 3.

<sup>16</sup><http://www.tcpdump.org/>

<sup>17</sup><https://www.wireshark.org/>

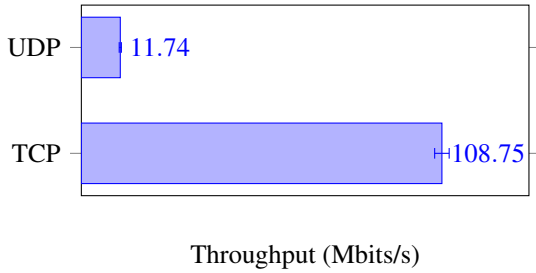


Figure 6: FUSE network throughput for UDP and TCP network traffic.

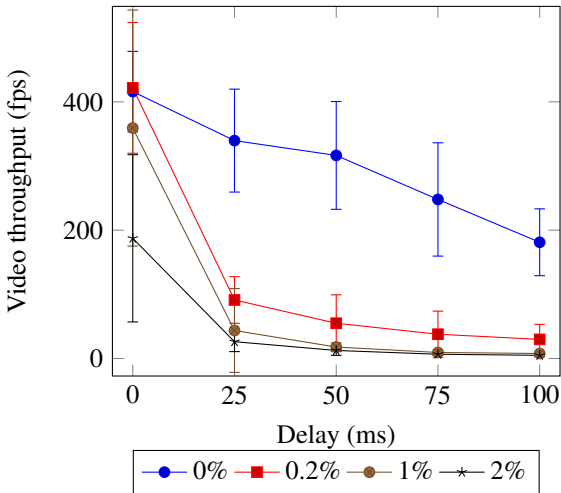


Figure 7: 720p video streaming performance between FUSE nodes for increasing delay at several levels of packet loss.

The rest of the chart follows the general shape predicted by the model. A roughly hyperbolic shape, with an increase in packet loss causing a faster degradation than an increase in delay. The general result is that performance quickly drops to a useless level, unless special care is taken to avoid noticeable packet loss. While 0.2% loss combined with 100ms delay still results in a useful stream at 29fps, as little as 1% loss combined with 50ms delay results in only 17fps. The only other good results were all obtained at 0ms delay (0.25ms counting  $d_0$ ), but those are unrealistic in practical federations, even if the nodes are very close to each other geographically.

## 6 DISCUSSION AND FUTURE WORK

A containerized approach makes FUSE easy to deploy on any device that supports Docker. Additionally, leveraging Kubernetes makes sure that containerized software can be deployed using familiar and reli-

able methods. However, FUSE can only be deployed on devices that support Docker, and existing software has to be containerized in order to be deployed in a FUSE federation.

The introduction puts forth three challenges in creating a federation service environment for crisis situations:

1. Enabling and securing fast cross-domain communication while restricting access to non-federated resources
2. Isolating the federation service environment from other software
3. Ensuring fast and easy deployment of the federation service environment on a large range of devices

It is shown that the FUSE architecture solves these challenges by using OpenVPN to enable and secure cross-domain traffic, and by using DinD to isolate FUSE from other software running on a device and simplifying deployment.

Tests are performed to empirically confirm the parts of the challenges related to FUSE performance and resource consumption.

The memory consumption and required disk space tests confirm that FUSE can be deployed on a wide range of devices, according to the third challenge. However, master nodes have to process a lot of OpenVPN traffic, which is CPU intensive, so devices with at least 2 available cores and hardware encryption such as AES-NI (Intel, 2018) are recommended.

Starting a FUSE federation is shown to be fast, taking only 5 to 6 minutes. The time to join a federation is dependent on network quality, but barring extremely hostile network conditions, a worker node should take about 1 to 4 minutes to join a federation. These numbers show that in most circumstances, it is possible to set up an entire federation in 10 minutes or less, which is enough to complete the third challenge. Because the resource requirements for master nodes are relatively low and every minute counts in crisis situations, it could be a good idea to keep a master node running at all times. This approach would cut response time to just 1 to 4 minutes when a situation arises.

The network performance of a FUSE federation is explored by both measuring pure throughput and by using a video streaming setup which mimics a client PC viewing a security camera stream. The results of the throughput test suggest that most of the performance limitations are due to OpenVPN. For TCP traffic the network speed is acceptable and saturates a 100 Mbit/s network. UDP is almost 10 times slower, so applications running on a FUSE federa-



tion should consider using TCP. Importantly, since OpenVPN performance is CPU bound, not connection bound, this bandwidth has to be shared by all worker nodes connected to the same master, which needs to be taken into account when setting up a federation.

The video streaming test shows the performance of streaming services under a variety of network conditions. If 24 FPS is taken as a minimum requirement for a smooth 720p video stream, performance is good enough to handle 17 video streams simultaneously under ideal circumstances, which is sufficient for the first challenge. However, performance drops quickly with increasing packet loss. Around 1% packet loss and 50ms delay, UDP is likely a better choice, since packet loss will only result in image corruption, but no frame rate decrease. A simple mathematical model has been worked out to predict FUSE TCP performance, which could help evaluate the choice for either TCP or UDP under a given set of circumstances.

In future work, OpenVPN performance could likely be improved (OpenVPN, 2018), especially since the test results show that FUSE throughput only reaches about 10% of the capacity of the gigabit line used for the tests. While improving OpenVPN performance may not help for scenarios with low network quality, it would at least increase throughput under optimal network conditions, reduce CPU load, or make larger worker pools practical. Alternatives to OpenVPN, such as Wireguard<sup>18</sup>, are also considered.

A high availability mechanism should also be implemented in the future. In the test setup, only one master node is used, but a failure of the master node would disrupt the entire federation. Some of the challenges are minimizing the number of hops for pod traffic between master nodes, handling the independent VPN networks started by each master, keeping configuration information up to date over the entire high availability network and making sure nodes get delegated to a new master when one of them fails.

The creation of multiple master nodes can also reduce OpenVPN CPU load compared to a single master, and optimize traffic flow. In a single master setup, there is not only a centralized controller for the federation, but since all traffic between worker nodes has to go via the master node's VPN server, it flows according to a star topology. In a high availability setup, each master node would have its own star topology formed by its set of worker nodes. Only when worker nodes with different masters need to communicate does any traffic flow between the master nodes, and this sort of traffic could be minimized by planning or reassigning worker nodes to a different master.

<sup>18</sup><https://www.wireguard.com/>

Kubernetes pods can be moved from one node to another to ensure the load is distributed across all nodes in the cluster. For stateless microservices this poses no problem but services such as databases or queues require backing storage that needs to persist across the same instances regardless where they run. This storage is typically configured in advance and a Kubernetes volume plugin is deployed to allow for dynamic volume provisioning. In the context of FUSE, further research can be done to automate this deployment and dynamically increase the storage through simple node labeling.

## 7 CONCLUSION

FUSE is introduced as a federation service environment to create and manage federations across domains. Its architecture is shown to fulfill the requirements stated in the introduction. Tests show that FUSE is fast and easy to deploy and that its hardware requirements allow for deployment on low-end hardware. Furthermore, FUSE is shown to be fast enough for high quality video streaming, an important practical use case. Note that the test is primarily aimed at measuring FUSE network performance and stability, while camera streams in real crisis situations may have more varying results. Potential future work is discussed to improve the speed and reliability of FUSE, which includes optimizing or replacing OpenVPN, adding support for a high availability setup and managing storage.

## 8 ACKNOWLEDGMENT

FUSE is a project realized in collaboration with imec. Project partners are Barco, Axians and e-BO Enterprises, with project support from VLAIO (Flanders Innovation & Entrepreneurship).

## REFERENCES

- Amaral, M., Polo, J., Carrera, D., Mohamed, I., Unuvar, M., and Steinder, M. (2015). Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34.
- Atrey, A., Moens, H., Seghbroeck, G. V., Volckaert, B., and Turck, F. D. (2016). Design and evaluation of automatic workflow scaling algorithms for multi-tenant SaaS. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science*.

- SCITEPRESS - Science and Technology Publications.
- Bernstein, D. (2014). Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.
- Bottoni, P., Gabrielli, E., Gualandi, G., Mancini, L. V., and Stolfi, F. (2016). FedUp! cloud federation as a service. In *Service-Oriented and Cloud Computing*, pages 168–182. Springer International Publishing.
- Bui, T. (2015). Analysis of docker security.
- Burns, B. (2018). Designing distributed systems (the side-car pattern).
- Cailliau, E., Aerts, N., Noterman, L., and De Groote, L. (2016). A comparative study on containers and related technologies.
- Casalicchio, E. (2017). Autonomic orchestration of containers: Problem definition and research challenges. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. ACM.
- Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE.
- Intel (2018). Intel data protection technology with aes-ni and secure key.
- Jennings, B. and Stadler, R. (2015). Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619.
- Kratzke, N. (2017). About microservices, containers and their underestimated impact on network performance.
- Kubernetes (2016). Security best practices for kubernetes deployment.
- Kubernetes (2018). Kubernetes federation.
- Miraftabzadeh, S. A., Rad, P., and Jamshidi, M. (2017). Distributed algorithm with inherent intelligence for multi-cloud resource provisioning. In *Intelligent Decision Support Systems for Sustainable Computing*, pages 77–99. Springer International Publishing.
- Morabito, R., Cozzolino, V., Ding, A. Y., Bejar, N., and Ott, J. (2018). Consolidate IoT edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111.
- Moreno-Vozmediano, R., Huedo, E., Llorente, I. M., Montero, R. S., Massonet, P., Villari, M., Merlino, G., Celesti, A., Levin, A., Schour, L., Vázquez, C., Melis, J., Spahr, S., and Whigham, D. (2016). Beacon: A cloud network federation framework. In Celesti, A. and Leitner, P., editors, *Advances in Service-Oriented and Cloud Computing*, pages 325–337, Cham. Springer International Publishing.
- OpenVPN (2018). Openvpn optimizing performance on gigabit networks.
- Recordon, D. and Reed, D. (2006). Openid 2.0: A platform for user-centric identity management. In *Proceedings of the Second ACM Workshop on Digital Identity Management, DIM '06*, pages 11–16, New York, NY, USA. ACM.
- Samie, F., Tsoutsouras, V., Bauer, L., Xydis, S., Soudris, D., and Henkel, J. (2016). Computation offloading and resource allocation for low-power IoT edge devices. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE.
- Shi, W. and Dustdar, S. (2016). The promise of edge computing. *Computer*, 49(5):78–81.
- Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at google with borg. *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015*.
- Vermeulen, B., Van de Meerssche, W., and Walcarus, T. (2014). jfed toolkit, fed4fire. In *Federation. In: GENI Engineering Conference*, volume 19.
- Wang, Z. and Su, X. (2015). Dynamically hierarchical resource-allocation algorithm in cloud computing environment. *J. Supercomput.*, 71(7):2748–2766.
- Wauters, T., Vermeulen, B., Vandenberghe, W., Demeester, P., Taylor, S., Baron, L., Smirnov, M., Al-Hazmi, Y., Willner, A., Sawyer, M., Margery, D., Rakotoarivelo, T., Lobillo Vilela, F., Stavropoulos, D., Papagianni, C., Francois, F., Bermudo, C., Gavras, A., Davies, D., Lanza, J., and Park, S.-Y. (2014). Federation of internet experimentation facilities: architecture and implementation. In *European Conference on Networks and Communications, Proceedings*, pages 1–5.
- Whaiduzzaman, M., Haque, M., Karim Chowdhury, R., and Gani, A. (2014). A study on strategic provisioning of cloud computing services. *The Scientific World Journal*, 2014.