Received: 1 February 2017 | Revised: 10 May 2017 | Accepted: 6 June 2017
DOI: 10.1002/nem.1988

SPECIAL ISSUE PAPER

NFV-FD: Implementation of a failure detector using network virtualization technology

Rogerio C. Turchetti[1,2] | Elias P. Duarte Jr.[1]

[1]Department of Informatics, Federal University of Parana (UFPR), Curitiba, Brazil

[2]CTISM, Federal University of Santa Maria (UFSM), Santa Maria, Brazil

Correspondence
Elias P. Duarte Jr, Department of Informatics, Federal University of Parana (UFPR), Curitiba, Brazil.
Email: elias@inf.ufpr.br

Funding information
Brazilian Research Agency (CNPq), Grant/Award Number: 309143/2012-8 and 141714/2014-0

Summary

A failure detector (FD) is a classic distributed systems service that can be used to monitor processes of any network application. Failure detectors provide process state information: A process is reported to be either correct or suspected to have crashed. In this work, we propose the implementation of a process monitoring service using Network Function Virtualization (NFV) in an OpenFlow network. The NFV is a technology that uses software virtualization techniques to drastically reduce the cost of deploying and managing network functions that are usually available as middleboxes. We describe NFV-FD, a network service that provides information about the state of both processes and links. The NFV-FD relies on an OpenFlow controller from which monitoring information is obtained. We investigate different ways to implement NFV-FD. We evaluate the performance impact of monitoring by implementing the virtual function both within and on a separate host from the controller. Then we compare an implementation based on a traditional virtual machine with another implementation based on containers. Experimental results are reported not only in terms of performance, ie, the amount of resources required by NFV-FD, but also for the quality of service provided by the failure detection and notification functions as they are used to deploy a reliable broadcast service.

KEYWORDS

failure detector, network function virtualization, network monitoring

1 | INTRODUCTION

Failure detectors (FDs) are a basic distributed system abstraction that is used to monitor processes, ie, programs running on multiple hosts of a network.[1,2] Given a set of processes that can execute a distributed application, an FD is used to determined which of those processes are fault-free and which are suspected to have crashed.[3] Failure detectors are said to be inherently unreliable, as they were proposed for asynchronous systems[4] in which it is impossible to know the upper limit on the delay for transmitting a message or executing a task. It is thus impossible to be sure whether a process that is silent has crashed or is simply slow to reply.

Originally, FDs were proposed to investigate whether the availability of process state information would make consensus possible in asynchronous distributed systems. Consensus solves the agreement problem that appears, for example, when it is necessary to keep consistent data across multiple hosts of a network. A very large number of applications use consensus and other equivalent distributed algorithms.[4,5] Nevertheless, FDs are a practical approach for monitoring any set of distributed processes.

Usually FDs are implemented by having each process periodically send messages (called "heartbeats") to all other processes to inform that they are alive and well. If N processes are monitored, this strategy generates $O(N^2)$ messages per monitoring interval.

In this work, we describe an effective way to implement FDs based on network function virtualization (NFV) technology.[6] We present NFV-FD (FD stands for failure detector) a cost-effective implementation that places the FD within the network fabric, so that it can effectively monitor the state not only of processes but also of network links, including link reachability, ie, determine which portions of the network are reachable or not.

The NFV technology allows the implementation of middleboxes[7] using software virtualization executed on commodity hardware.[8] Typically, a network function (eg, a firewall, network address translation device, and router) can be started on demand, just when it is needed. In comparison with their traditional equivalents that are implemented using proprietary hardware/software, virtualized network functions (VNFs) are easier to manage and operate and are simply destroyed when they are no longer needed. The NFVs also save energy and reduce the requirements on physical space.[6] The NFV is usually deployed with other closely related technologies: software-defined networking (SDN) and OpenFlow.[9,10] OpenFlow separates the network control plane from the underlying switches that forward the traffic (the data plane).[11]

The NFV-FD was actually implemented in multiple ways; we discuss and present results of those alternative implementations. The NFV-FD relies on an OpenFlow controller* to obtain monitoring information. Initially, we compare 2 alternatives: implementing the virtual function both within and on a separate host from the controller. In particular, although there are advantages to implement a VNF within a controller,[9] we argue that this strategy is not scalable.

Next, an implementation based on traditional virtual machines (VMs) is compared with another implementation based on containers. In one of their latest documents, the European Telecommunications Standards Institute (ETSI) describes the deployment of VNFs using containers.[12] Containers are a relatively new technology that allow the execution applications as *sandboxed* instances on the host machine.[13] Container-based virtualization is a potential alternative to hypervisors because they can share the same kernel of the host system. Containers are much simpler than hypervisors; a hypervisor partitions the resources of a single host and creates VMs isolated from each other. Each VM created by the hypervisor is assigned a virtualized vCPU, a vNIC, and a vStorage device. Containers do not need any of this; it is a virtualization strategy that presents significantly lower requirements on the system.[12] Our experimental results confirm that lightweight containers provide faster instantiation and deployment time as well as lower resource requirements for the implementation of NFV-FD.

A distributed application can access the system view provided by NFV-FD to make decisions. In this way, NFV-FD can be seen as a building block for the implementation of fault-tolerant distributed systems. As an example of this kind of application, we describe the deployment of a reliable broadcast service on top of NFV-FD. A reliable broadcast algorithm ensures that correct processes agree on the set of messages they deliver, even when the senders of these messages crash during the transmission.[4] Experimental results are reported for the quality of service provided by the failure detection and notification functions as they are used by the reliable broadcast service.

Note that an earlier paper[14] presenting a preliminary version of NFV-FD has been published before. In the current paper, we explore multiple strategies to implement NFV-FD and also present a specific application based on reliable broadcast running on top of NFV-FD.

The rest of the paper is organized as follows. Section 2 presents the system model, as well as the architecture of NFV-FD, describing how processes are monitored and how failures are detected. The implementations and experimental results are described in Section 3. Section 4 presents related work. Section 5 concludes the paper.

2 | NFV-FD: AN NFV-BASED IMPLEMENTATION OF A FAILURE DETECTOR

In this section, we describe the system model and architecture of NFV-FD, our failure detection service based on NFV technology. Instead of the usual implementation of FDs in which processes exchange a quadratic number of messages per testing interval, NFV-FD is implemented within an OpenFlow network, and besides monitoring processes takes advantage of the information available at the OpenFlow controller to determine the state not only of processes but also of network links.

Originally, unreliable FDs were proposed by Chandra and Toueg[1] as abstractions that, depending of their properties, can be used to solve consensus in asynchronous systems with crash faults. Consensus is a fundamental problem of fault-tolerant distributed computing.[5,15] An FD is defined as an "oracle" that can be accessed by a process to obtain information about the state of the other processes of the distributed system. Failure detectors are said to be *unreliable* as they can make mistakes, ie, report an incorrect state. For example, a monitored process can be incorrectly suspected to have crashed, but later the suspicion can be raised if the FD learns that the process is alive.

*http://www.openflow.org

A common approach to implement FDs is to monitor the message exchange pattern. Two basic approaches can be used for monitoring: push and pull.[16] Using a push FD, the monitored process sends heartbeat messages at periodic time intervals to the monitor. In a pull FD, the monitor periodically sends liveness requests to monitored processes. Based on the observed message arrival pattern, the FD computes a timeout interval. If a message is not received by the monitor within this timeout interval, the monitored process is suspected to have crashed. The strategy used by NFV-FD to monitor processes can be classified as a pull-based.

2.1 | System model

We assume an asynchronous distributed system that consists of a set Π of n processes connected on an arbitrary topology (multi-hop) network. Processes and communication links can fail by crashing, ie, by prematurely halting. An FD is used as a monitor that informs the state of processes and links as correct (fault-free) or suspected to have crashed. The output of the FD is one of the following possible states: **S** (suspect): "FD suspects that monitored process (p) has crashed"; **T** (trust): "FD trusts that p is correct"; or **U** (unreachable): "FD suspect that p is unreachable." A process is suspected to have crashed (state **S**) whenever the timeout expires before a liveness reply from p is received. On the other hand, if the reply from p arrives before the respective timeout expires, the corresponding state is set to **T**. State **U** may occur after a communication link crashes, leaving process p unreachable.

We assume that the controller does not crash. Furthermore, crashes do not disrupt the communication between the FD and the network controller. In addition, the communication channel does not create, alter, or loose messages.

2.2 | NFV-FD architecture

The architecture proposed for the implementation of NFV-FD is shown in Figure 1. The figure shows the OpenFlow controller and also the FD, which consists of 2 modules: NFV-FD itself and FDMod, described next. The OpenFlow controller executes several functions; for example, the controller manages rules of forwarding tables of the switches, keeps information about the topology and network devices, among other tasks. It is possible to extend the functionality of the controller by using Module Applications—originally from the Floodlight controller[†]—which enables the creation of new features. The controller offers a REpresentational State Transfer (REST) interface that facilitates the configuration of the Module Applications.

To integrate the controller with NFV-FD, we propose a module called FDMod. The FDMod works as a packet filter that parses packet header information. The objective of FDMod is to assist NFV-FD to obtain information about monitored processes. This information is configured by the specification of which header fields are to be checked. Information is then extracted from the packet headers; an example field is the Internet Protocol address or the port used by the monitored process.

Monitoring starts after NFV-FD receives the specification of which information should be collected from a monitored process. To receive monitoring information from the controller, it is necessary to install rules, which is done through a REST interface. A rule specifies application attributes, for example, addresses, ports, and protocols it uses.

Consider, for example, a rule based on the following fields: source address (*nw_src=10.0.0.1*), destination address (*nw_dst=230.0.0.1*), User Datagram Protocol (UDP) transport protocol (*nw_proto=17*), and destination port (*tp_dst=4446*). The following packet would be captured by FDMod by using this rule:

[in_port=1,dl_dst=01:00:5e:00:00:01,dl_src=00:00:00:00:00:01, nw_dst=230.0.0.1, nw_src=10.0.0.1,nw_proto=17, nw_tos=0,tp_dst=4446,tp_src=4446]

The FDMod forwards the captured packet to NFV-FD, which can, for example, begin monitoring the process. In synthesis, all packets are forwarded to FDMod (see Figure 1), which applies a filter and only forwards to NFV-FD a copy of messages that match the rules. Note that packet headers are not modified in any way.

2.3 | Process failure detection

The NFV-FD monitors processes using liveness request messages. Periodically (Δ_i) NFV-FD sends liveness requests to monitored process. At a Δ_i time interval, NFV-FD sends an `AreYouAlive?` message and waits for the monitored process to reply `YesIAm!`. If a message is not received within a specific timeout Δ_{to} interval, NFV-FD starts to suspect the monitored process. In this case, the state of monitored process is set to **S**. Otherwise, if NFV-FD receives a reply, the state of monitored process is set to **T**.

[†]http://www.projectfloodlight.org/

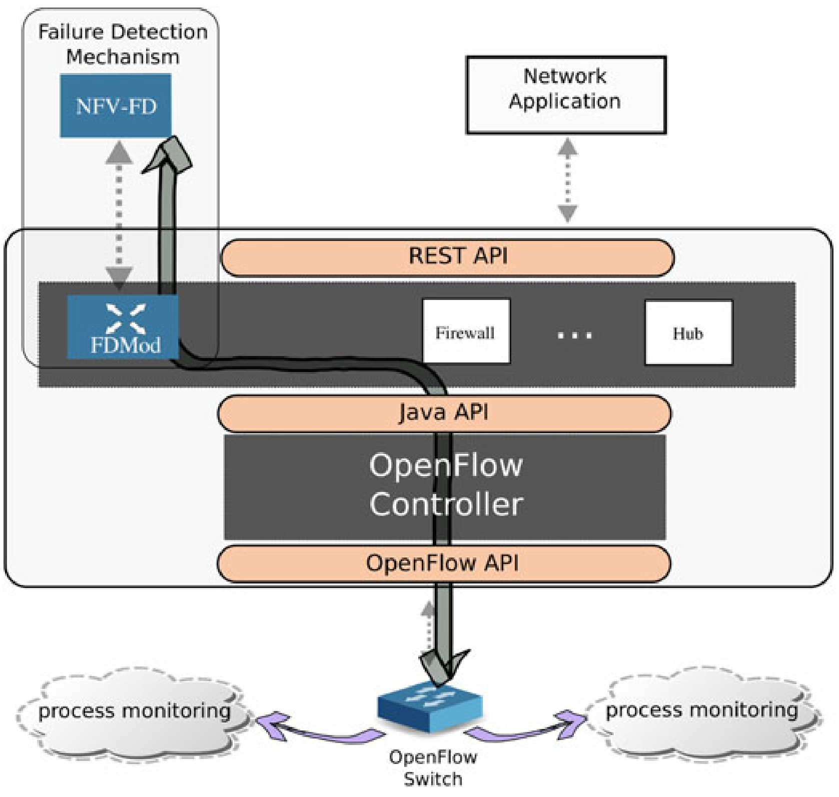

FIGURE 1 Integration of NFV-FD and an OpenFlow controller. API, Application Programming Interface; FD, failure detector; NFV, network function virtualization

A key problem to implement effective FDs is to compute a precise timeout interval. If the timeout interval is too short, crashes are quickly detected, but the likeliness of wrong suspicions is higher. Conversely, if the timeout interval is too long, wrong suspicions will be rare, but this comes at the expense of long detection times.[17] An adaptive FD can automatically update the timeout intervals, and for this reason, it is generally considered to be the best option for real networks.[18]

In this work, we use an adaptive strategy that forecasts the expected instant of time when the next `YesIAmAlive!` message should be received. The expected message arrival date is called EA and uses a dynamic safety margin α, as proposed in one study.[19] Let us consider 2 processes: p and q, p monitors q. Periodically (Δ_i) p sends liveness requests to q. Let $m_1, m_2, \dots, m_k$ be the k most recent messages received by p. Let T_i be the time interval that each message took to be received by p, measured with the local clock. EA can be estimated as follows:

$$EA_{k+1} = \frac{1}{k}\left(\sum_{i=1}^{k} T_i - \Delta_i\right) + (k+1)\Delta_i. \tag{1}$$

EA_{k+1} is the expected time instant at which the next message (the $k+1$-*th* message) will arrive. The next timeout will expire at

$$\tau_{(k+1)} = EA_{(k+1)} + \alpha_{(k+1)}. \tag{2}$$

The safety margin ($\alpha_{(k+1)}$ in expression (2)) is based on the TCP algorithm by Jacobson,[20] and it is used to correct the error of the estimated arrival time of the next message:

$$\begin{aligned} Diff &= Timestamp\_msg_k - Timestamp\_msg_{k-1} \\ Mean &= \phi * Mean + (1-\phi) * Diff \\ Var &= \phi * Var + (1-\phi) * |Mean - Diff| \\ \alpha_{k+1} &= Mean + \beta * Var. \end{aligned} \tag{3}$$

In expression (3), *Diff* corresponds to the difference between the local time instant at which the last monitoring message was received ($Timestamp\_msg_k$) and the local time instant at which this monitoring message had been sent ($Timestamp\_msg_{k-1}$). *Mean* maintains a weighted mean of the differences. *Var* reflects the dispersion of intervals from the average and is computed using a weighted mean. The higher the *Var*, the larger is the α. ϕ and β were assigned values 0.9 and 4, respectively, approximations of the original values proposed by Jacobson often used in implementations. Using expressions (1), (2), and (3), the timeout can be adapted to different communication scenarios.

2.4 | Link failure detection

Besides detecting process crashes, NFV-FD also detects crashes of communication links. To enable this function, NFV-FD makes use of a topology discovery mechanism that is performed by OpenFlow switches with the Link Layer Discovery Protocol (LLDP).[21] The topology discovery protocol uses 2 types of OpenFlow messages: *packet in* and *packet out*. The protocol is executed among OpenFlow switches that periodically exchange *packet out* messages carrying LLDP packets. After the OpenFlow switch receives such a message, it propagates the message to the other switches to communicate events related to topology information.

OpenFlow switches periodically exchange *packet out* messages. When a switch receives a *packet out*, LLDP packets are propagated. When a switch receives a LLDP packet, it sends the *packet in* message to the OpenFlow controller to inform about the detected link.[22]

There are 3 types of events that cause an OpenFlow controller to update the status of a given link[23]:

- **Link up (Controller←Switch)**: Whenever a switch detects a link to a new switch, it sends a new message to the controller to inform about the new link.
- **Link down (Controller←Switch)**: Whenever a switch detects a link is down (no new LLDP packets received for some period of time), it sends a message to the controller to inform about the loss of the link.
- **Get Links (Controller→Switch)**: Ask a switch to send *Link up* messages for all known links. Useful when a controller first connects to the switch.

The controller receives information about these events through a function called *Link Discovery*. The FDMod receives all the information about the network topology that is provided by/to the OpenFlow controller. In turn, NFV-FD receives this information via FDMod. For example, whenever there are events of the types **Link up** or **Link down**, the controller forwards this information to FDMod.

The FDMod forwards to NFV-FD detailed information about the event that has occurred. For example, *"port s2-eth2 changed: DOWN"* warning that the switch (s2) port (eth2) is connectionless. If there are processes that belong to the respective switch port, NFV-FD updates the state of process to **U** unreachable.

As shown in this section, SDN technology enables the integration of protocols or devices, which usually require specific communication interfaces (for example, LLDP protocol). Another work that has explored this fact is the *Bidirectional Forwarding Detection* protocol, which has been proposed[24] to enable the *fast recovery* of SDN networks.

3 | EXPERIMENTAL EVALUATION

In this section, we describe several strategies to implement NFV-FD and experimental results. We implemented the VNF in the same host of the controller and also on a separate host; furthermore, we evaluated the use of both traditional VMs and containers. We also evaluated the quality of the service provided by the proposed NFV-FD from the point of view of an application that uses a reliable broadcast algorithm and the detection service; this algorithm is described in the first subsection. We used the Floodlight OpenFlow controller. We used containers running on the Docker platform[‡] and VMs running on the Virtualbox[§] hypervisor. The experiments were executed on a host based on the Intel Core i5 processor at 2.50 GHz with 4 cores. A virtualized network was created using Mininet[¶] and running the Linux Mint 17 kernel 3.13.0-24 (first set of experiments) and Ubuntu 14.04 (second set of experiments).

3.1 | Application: reliable broadcast

Reliable broadcast ensures essentially that all correct processes deliver the same messages and that messages broadcast by correct processes are eventually delivered.[25] Moreover, it is of course good that messages are delivered as fast as possible.[26]

[‡]https://www.docker.com/
[§]https://www.virtualbox.org/
[¶]http://mininet.org

Reliable broadcast can be defined by 2 primitives: *broadcast(m)* and *deliver(m)*, where *m* is a message. When a process executes *broadcast(m)*, we say that it broadcasts *m*. When a process executes *deliver(m)*, we say that it delivers *m* to the application that is communicating with reliable broadcast. Algorithm 1 implements these primitives.

Algorithm 1: Reliable Broadcast Algorithm [27]

```
upon event ⟨ _Init ⟩ do
    delivered := $\emptyset$;
    correct := $\Pi$;
    forall $p_i \in \Pi$ do
        from[$p_i$] := $\emptyset$;

upon event ⟨ _Broadcast | $m$⟩ do
    trigger ⟨ broadcast | [Data, self, $m$] ⟩

upon event ⟨ _Deliver | $p_i$, [Data, $s_m$, $m$]⟩ do
    if ($m \notin$ delivered) then
        delivered := delivered ∪ {$m$}
        trigger ⟨ deliver | $s_m$, $m$ ⟩;
        from[$p_i$] := from[$p_i$] ∪ {($s_m$, $m$)}
        /* (Case 2: redistribute m) */
        if ($p_i \notin$ correct) then
            trigger ⟨ broadcast | [Data, $s_m$, $m$] ⟩;

upon event ⟨ _Crash | $p_i$ ⟩ do
    correct := correct ∪ \ {$p_i$}
        /* (Case 1: redistribute m) */
    forall ($s_m$, $m$) $\in$ from[$p_i$] do
        trigger ⟨ broadcast | [Data, $s_m$, $m$] ⟩;
```

Algorithm 1 treats 4 types of events. First, the *_Init* event causes the initialization of the following variables: *delivered* (for filtering out duplicate messages), *correct* (processes that have not been detected as crashed by NFV-FD) and *from[p_i]* (each entry contains the messages that have been delivered from each possible sender). When a message is transmitted by reliable broadcast, the process needs to invoke the *_Broadcast* event. The *_Deliver* event causes the process to deliver *m*, which is of course a message that has already been transmitted by a process belonging to Π. In this event, message *m* is delivered and the state of process p_i is verified. Whether $p_i \notin correct$ then *_Broadcast* event is invoked retransmitting the same message already stored in *from[p_i]*. Finally, the *_Crash* event is invoked if process p_i that has sent *m* crashed before *m* has been delivered.

Two kinds of events that are dependent on NFV-FD can force a process to retransmit a message:

- Case 1: the process receives a message from a given source and detects that this source has crashed.
- Case 2: the process delivers a message and realizes that the source has crashed (ie, the source does not belong anymore to set *correct*).

3.2 | Evaluation of the failure detection service

In our first experiment, we report results of implementation of the FD both within an on a separate host from that of the OpenFlow controller. For this experiment, the virtualized network consists of a single topology with a switch, a controller, and 4 hosts running the reliable broadcast algorithm based on an FD.[4] The NFV-FD is configured to send liveness request messages

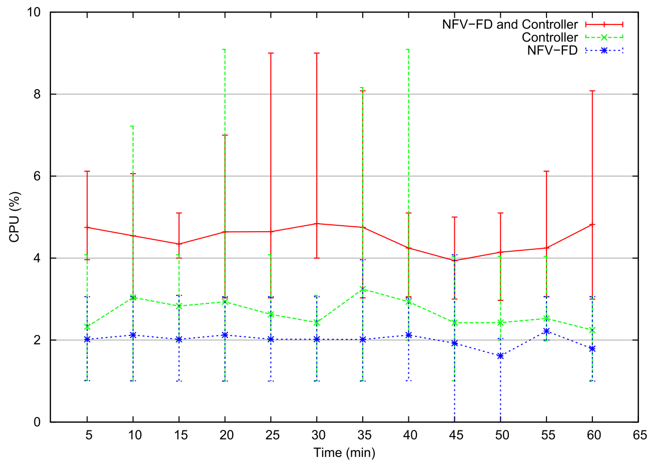

FIGURE 2 CPU usage: performance as NFV-FD is executed with and separated from the controller. FD, failure detector; NFV, network function virtualization

with $\Delta_i = 1000$ milliseconds. The hosts that are running the reliable broadcast algorithm are configured to trigger the broadcast every 1000 milliseconds.

As mentioned above, one of the goals of the first experiment was to measure the impact of NFV when it is executed within/on a separate host of the controller. Some authors including Batalle et al[9] mention that placing an application on the controller brings benefits; eg, the application can have a full view of the network. To perform this experiment, we evaluated the CPU usage taking into account the following: (1) the performance of the controller alone, (2) the performance of NFV-FD as a separate process, and (3) the performance of both (NFV-FD and controller) running on the same CPU. Each experiment was performed within a 1-hour interval. Results presented in Figure 2 are mean of 5 executions; the maximum and minimum values computed are also shown.

We can see in Figure 2 that when NFV-FD is run with the controller (NFV-FD and Controller curve), the CPU usage is up to 4.49%. If the Controller is run alone (curve labelled "Controller"), CPU usage is approximately 2.67%; in this case NFV-FD is executed outside the controller (curve labelled "NFV-FD") and the average CPU usage is about 2%. We can thus conclude that placing the function with the controller simply adds the function's resource requirements to the controller resource requirements. Thus, this strategy is not scalable: There is clearly a limit on the number of functions that can be implemented in this way. As the number of functions grows, the impact on the system will surely soar.

For the second experiment, we measured how quickly failures are detected by simulating crashes of both hosts/processes and communication links. In this case, the virtualized network topology was composed of 3 switches, a remote controller and 4 hosts running the reliable broadcast algorithm. We show the average detection time obtained for a total of 10 crashes. Crashes were simulated by blocking monitoring messages[‖] and thus causing a timeout. The failure detection time (T_D) is the time elapsed from the moment the crash was simulated until all processes/hosts are notified of the incident. The notification occurs only when the reliable broadcast algorithm gets informed about the crash.

The average host failure detection time measured is 1256 milliseconds while the maximum value observed is 1454.23 milliseconds. Note that the maximum detection time also includes message transmission delays and task processing delays, including tasks related to the reliable broadcast algorithm itself. We observed that there is a larger variation of the T_D, with the mean at about 800 milliseconds. The minimum T_D measured was 598 milliseconds; in this case, the crash probably happened immediately before the other hosts were notified. We also measured the T_D considering that the crash is notified to a single host, and in this case it is equal to 22.68 milliseconds. We believe that these detection times are good (fast enough) and they are also a consequence of the adaptive strategy used to forecast the timeout. In synthesis, based on the experimental results, we believe it is possible to conclude that it is perfectly feasible to implement the FD service as a VNF.

In the link failure scenario, the T_D is the time elapsed since a link crashes until the NFV-FD is aware of the crash. As mentioned in Section 2.4, the LLDP protocol is used to detect link failures. We evaluated the detection time with the NFV-FD hosted both with the controller and running at a separate host. For these experiments, we have observed that when the NFV-FD is run with the controller, it presented a slightly smaller T_D. This difference of about 120 milliseconds corresponds to the communication

[‖] Messages from LLDP and NFV-FD.

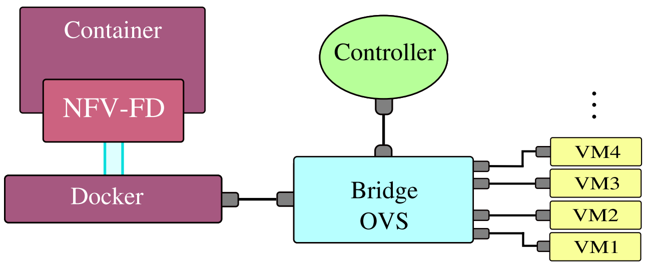

FIGURE 3 NFV-FD running on a container. FD, failure detector; NFV, network function virtualization; VM, virtual machine

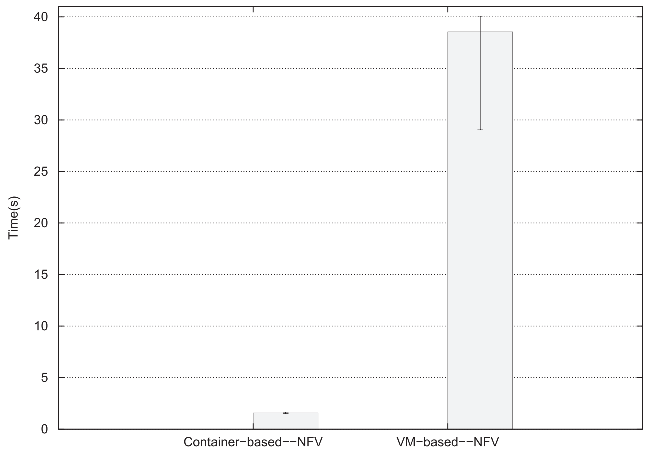

FIGURE 4 Container vs VM: instantiation time. NFV, network function virtualization; VM, virtual machine

time between the controller and the host. On the other hand, we can see in Figure 2 that the NFV-FD running with the controller is an alternative that cannot be used by an arbitrary number of functions at the same time.

3.3 | Experimental results: implementation of NFV-FD with containers

In this experiment, we evaluated the benefits of implementing NFV-FD with containers. Container-based virtualization is an alternative to hypervisors where the virtualization layer runs as an application of the local operating system level.[28] Since applications in containers run directly on the operating system and do not use hardware indirections, they should be more efficient than their VM-based counterparts and allow higher application density on the same host.[13] The comparison is made between a traditional VM running on the Virtualbox hypervisor and a container running on the Docker platform.

To implement NFV-FD with containers, we used the architecture described in Figure 3. This figure shows VMs that exchange information with each other through an OpenFlow switch (OVS: Open vSwitch[**]). Traffic is redirected to NFV-FD as shown in Figure 1 with FDMod (see Section 2.2), which is an OpenFlow controller module that forwards the packets that match predefined rules to NFV-FD; those rules specify which processes and links are monitored.

For the first experiment of this series, we measured the instantiation time for each system; eg, we measure the time that a container or a VM takes to start and stop its execution. To perform this task, we have used the *time* command to compute the start and stop times. A *script* included in *rc.local* file is used to shutdown the system. Results presented in Figure 4 are mean, minimum and maximum values, and were repeated 10 times for each virtualization systems. Both virtualization systems are prepared to execute the NFV-FD and also both running the Linux Ubuntu 14.04.

We can see in Figure 4 that when NFV-FD uses a container, we have the shortest instantiation time, which is up to 95% lower than the VM counterpart. Furthermore, as we check the minimum and maximum values, we can confirm that the container-based implementation presents low variation. We can confirm this fact as the standard deviation of the instantiation time is equal to only 0.04 second for the container-based NFV-FD, while for the VM-based NFV-FD, it grows to 3.35 seconds.

[**]http://openvswitch.org/

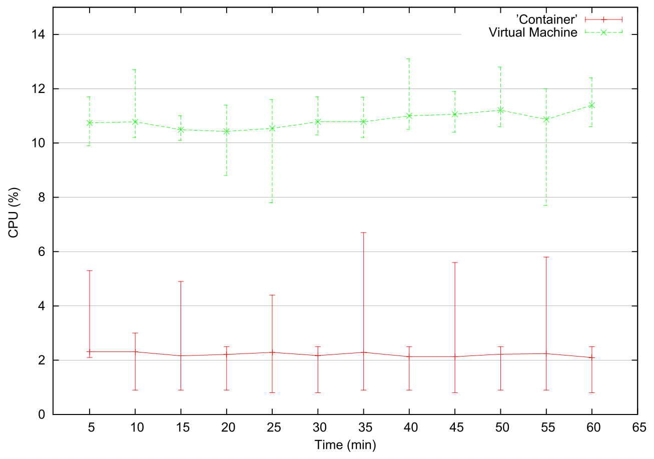

FIGURE 5 Container vs virtual machine: CPU usage

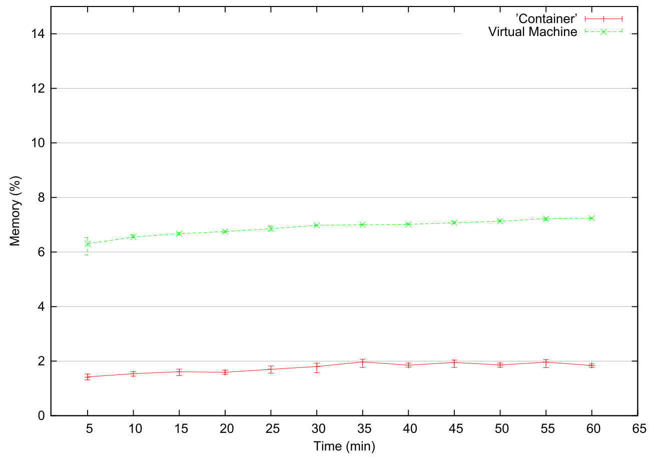

FIGURE 6 Container vs virtual machine: memory usage

The second experiment in this series was executed to measure CPU and memory usage. For these experiments, the virtualized network consists of a single topology composed of one switch as shown in Figure 3, one remote controller and 4 hosts running the reliable broadcast algorithm. The NFV-FD is configured to send liveness request messages with $\Delta_i = 1000$ milliseconds. The hosts that are running the reliable broadcast algorithm are used to generate a flow of messages into the network. They are also configured to transmit messages every 1000 milliseconds.

Figures 5 and 6 show the results for an execution that lasted 1 hour. It is easy to see the advantage of using containers—in particular, CPU usage was reduced 89.3% on average. A similar result can be seen in Figure 6, in which the containers also reduce the average memory usage about 74.5%. We believe these results are as expected, as containers do not need to virtualize the hardware and they are a lightweight virtualization technology that share host kernel.[12]

4 | RELATED WORK

First of all, we have previously published a preliminary version of NFV-FD.[14] In that paper, we first investigated the feasibility of implementing an FD as a virtual network function. We compared 2 implementations: within and outside the OpenFlow controller. In the current paper, we explored another implementation alternative that uses containers and proved to be better than the previous implementations both in terms of instantiation time and the amount of resources required. Furthermore, we also describe in the current paper how a specific application runs on top of NFV-FD. An application is described that runs a reliable broadcast algorithm that uses information from the FD to reduce the number of messages required.

In Cerrato et al,[29] an efficient algorithm to handle the communication between a virtual switch and multiple network functions that are executed on a single server is proposed. Data exchange ensures traffic isolation between network functions. The proposal adopts a master-worker architecture, in which the master corresponds to the virtual switch that determines which network function (worker) processes the data. The proposal is designed to present high speed.

In Vilalta et al,[10] NFVs are implemented on a cloud infrastructure to provide the services of a *Path Computation Element* (PCE). A PCE is an entity (eg, component, application, or network node) that is capable of computing a network path or route based on a network graph and applying computational constraints. A PCE can calculate paths in transport networks. To overcome scalability limitations, the authors extend the concept of NFV to transport networks by creating a vPCE. A vPCE is a virtual PCE instance, which is run as a software application on a cloud computing environment. Experimental results show that the vPCE was able to guarantee a mean processing time for the computation of the paths, even during the peak requests.

In previous study,[13] the authors investigate the feasibility of using technologies designed to accelerate VM networking with containers. In addition, they quantify the network performance of a container-based VNF. The goal is to identify the factors that have an influence on the packet delay and throughput of container-based applications and VMs. In this context, the authors have concluded that containers have lower latency cost and lower variability than equivalent Xen VMs running the same software.

In Nakagawa et al,[30] the authors propose an architecture for lightweight virtualized network creation using container-based virtualization environments. A dynamic virtualized network configuration method for container-based NFV is proposed. When the container starts on a host, the physical switch identifies a switch port where the host connects to and locally applies a Virtual Local Area Network (VLAN) configuration to the switch port to construct the virtualized network. When the container stops running the switch removes the virtualized network by deleting the VLAN configuration on the switch port without any communication with the SDN controller. However, to decide where to run each container, a manager is used. In fact, the NFV can be configured without any direct communication with the SDN controller. Instead, to build the VLAN, it is first necessary to perform some communication between the container manager with each the switch.

In Cziva et al,[28] a framework (called GLANF—*Glasgow Network Functions*) is proposed to create, deploy, and manage VNFs in OpenFlow networks. The authors exploit container-based network functions to achieve low performance overhead and fast deployment in the NFV context. In the approach proposed, the traffic is rerouted to the GLANF Server hosting the network function. The GLANF relies on OpenFlow to match and forward traffic to an NFV host. The GLANF Router is necessary to route the traffic to the GLANF Servers hosting the network functions. To achieve this, OpenFlow rules are inserted in each switch. In the experimental results, the authors show a significant improvement in throughput and function instantiation time by using container. The main difference to our work is that we do not have to install rules on each switch. Instead, we have a module on the OpenFlow controller that forwards the packet to the corresponding network functions.

5 | CONCLUSION

In this work, we described NFV-FD, an NFV-based implemention of an FD, a classical distributed systems abstraction that can be used to monitor processes of a distributed application. The NFV-FD uses information obtained from an OpenFlow controller to monitor processes and determine their state. In addition, NFV-FD takes advantage of OpenFlow's LLDP to monitor links and provides information about link state and process reachability. We describe several alternatives to implement NFV-FD: The network function can be implemented with the controller or on a separate host. Furthermore, we also compared implementations using traditional VMs and containers. Our first experiment shows that although the implementation of a network function with the controller brings benefits—in our case an improved failure detection time—it does not scale, as the resource requirements of the function sum up with the requirements of the controller. The results of the comparison between NFV-FD running on a traditional VM and running on a container confirm that containers are an extremely attractive technology in terms of resource requirements.

As future work, we can clearly see that there is a lack of tools to support the development and deployment of NFVs, in particular taking advantage of containers. As we can conclude from our experience that NFV technology was perfect to the implement FD abstraction, the implementation of other classical distributed services within the network using the same technology comes as a natural next step. In this work, we explored FDs at the application level, but they can also be applied to monitor virtualized network infrastructure itself that would be a first step towards developing reliable, fault-tolerant virtualized networks based on NFV.

ACKNOWLEDGMENT

This work was partially supported by grants 309143/2012-8 and 141714/2014-0 from the Brazilian Research Agency (CNPq).

ORCID

Rogerio C. Turchetti http://orcid.org/0000-0002-5242-5057

REFERENCES

1. Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed systems. *J ACM*. 1996;43(2).
2. Freiling FC, Guerraoui R, Kuznetsov P. The failure detector abstraction. *ACM Comput Surv*. 2011;43(2):9:1-9:40.
3. Turchetti RC, Duarte EP, Arantes L, Sens P. A QoS-configurable failure detection service for Internet applications. *JISA*. 2016;7(1).
4. Cachin C, Guerraoui R, Rodrigues L. *Introduction to Reliable and Secure Distributed Programming*. 2nd ed. Berlin, Heidelberg: Springer Publishing Company Incorporated; 2011.
5. Charron-Bost B, Pedone F, Schiper A. *Replication: Theory and Practice*. Berlin, Heidelberg: Springer; 2010.
6. Han B, Gopalakrishnan V, Ji L, Lee S. Network function virtualization: challenges and opportunities for innovations. *IEEE Commun Mag*. 2015;53(2):90-97.
7. Sherry J, Hasan S, Scott C, Krishnamurthy A, Ratnasamy S, Sekar V. Making middleboxes someone else's problem: network processing as a cloud service. In: Symposium on Communications Architectures and Protocols (SIGCOMM'12). New York, NY, USA; 2012.
8. Mijumbi R, Serrat J, Gorricho JL, Bouten N, Turck FD, Boutaba R. Network function virtualization: state-of-the-art and research challenges. *IEEE Commun Surv Tut*. 2016;18(1):236-262.
9. Batalle J, Ferrer Riera J, Escalona E, Garcia-Espin J. On the implementation of nfv over an openflow infrastructure: routing function virtualization. In: IEEE SDN for Future Networks and Services (SDN4FNS'13). Trento, Italy; 2013.
10. Vilalta R, Munoz R, Mayoral A, Casellas R, Martinez R, Lopez V, Lopez D. Transport network function virtualization. *J Lightwave Technol*. 2015;33(8):1557-1564.
11. Kreutz D, Ramos FMV, Veríssimo PJE, Rothenberg CE, Azodolmolky S, Uhlig S. Software-defined networking: a comprehensive survey. *Proc IEEE*. 2015;103(1):14-76.
12. ETSI. ETSI GS NFV-EVE 004: Network Functions Virtualisation (NFV); Virtualisation Technologies; Report on the application of Different Virtualisation Technologies in the NFV Framework. http://www.etsi.org/deliver/etsi_gs/NFV-EVE/001_099/004/01.01.01_60/gs_NFV-EVE004v010101p.pdf, Accessed September 26, 2016.
13. Anderson J, Hu H, Agarwal U, Lowery C, Li H, Apon A. Performance considerations of network functions virtualization using containers. In: International Conference on Computing, Networking and Communications (ICNC'16). Kauai, HI, USA; 2016.
14. Turchetti RC, Duarte EP Jr. Implementation of failure detector based on network function virtualization. In: IEEE International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2015, June 22-25, 2015; 2015; Rio de Janeiro, Brazil.
15. Turek J, Shasha D. The many faces of consensus in distributed systems. *IEEE Comput*. 1992;25(6):8-17.
16. Felber P, Dfago X, Guerraoui R, Oser P. Failure detectors as first class objects. In: International Symposium on Distributed Objects and Applications (DOA'99). Edinburgh, United Kingdom; 1999.
17. Hayashibara N, Defago X, Katayama T. Two-ways adaptive failure detection with the ϕ-failure detector. In: Intl. Workshop on Adaptive Distributed Systems. Sorrento, Italy; 2003.
18. Falai L, Bondavalli A. Experimental evaluation of the QoS of failure detectors on wide area network. In: International Conference on Dependable Systems and Networks (DSN'05). Yokohama, Japan; 2005.
19. Bertier M, Marin O, Sens P. *Performance Analysis of a Hierarchical Failure Detector*, *International Conference on Dependable Systems and Networks (DSN'03)*. San Francisco, USA; 2003.
20. Jacobson V. Congestion avoidance and control. In: Symposium on Communications Architectures and Protocols (SIGCOMM'88). Standford, USA; 1988.
21. Authors are not available for this document. IEEE standard for local and metropolitan area networks—station and media access control connectivity discovery. In: IEEE Std 802.1AB-2009 (Revision of IEEE Std 802.1AB-2005); 2009.
22. Sharma S, Staessens D, Colle D, Pickavet M, Demeester P. Enabling fast failure recovery in OpenFlow networks. In: International Workshop on the Design of Reliable Communication Networks (DRCN'11). Krakow, Poland; 2011.
23. Openflow. Available at: http://archive.openflow.org/wk/index.php/Openflow_1.X_Discussion, Accessed September 26, 2016.
24. van Adrichem NLM, van Asten BJ, Kuipers FA. Fast recovery in software-defined networks. In: European Workshop on Software Defined Networks (EWSDN'14). London, United Kingdom; 2014.
25. Correia M, Neves NF, Verissimo P. From consensus to atomic broadcast: time-free Byzantine-resistant protocols without signatures. *Comput J*. 2006;49(1):82-96.
26. Garcia-Molina H, Kogan B, Lynch NA. Reliable broadcast in networks with nonprogrammable servers. In: International Conference on Distributed Computing Systems; 1988; San Jose, California, USA.
27. Guerraoui R, Rodrigues L. *Introduction to Reliable Distributed Programming*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.; 2006.
28. Cziva R, Jouet S, Pezaros D. Container-based network function virtualization for software-defined networks. In: International Symposium on Computers and Communications (ISCC'15). Larnaca, Cyprus; 2015.
29. Cerrato I, Marchetto G, Risso F, Sisto R, Virgilio M. *An Efficient Data Exchange Algorithm for Chained Network Functions*, *International Conference on High Performance Switching and Routing (HPSR'14)*. Vancouver, Canada; 2014.
30. Nakagawa Y, Lee C, Hyoudou K, Kobayashi S, Shiraki O, Tanaka J, Ishihara T. *Dynamic Virtual Network Configuration Between Containers Using Physical Switch Functions for NFV Infrastructure*, *Conference on Network Function Virtualization and Software Defined Network (NFV-SDN'15)*. San Francisco, USA; 2015.

Rogerio C. Turchetti received the MSc degrees in production engineering with emphasis on information systems from Federal University of Santa Maria (UFSM), Santa Maria, Brazil, in 2006. He is currently an adjunct professor of computer network at the UFSM and PhD student at Federal University of Parana, Curitiba, Brazil, where he is also a student member of the Computer Networks and Distributed Systems Lab (LaRSis). His research interests include Computer Network and

Distributed Systems, their Dependability, and Algorithms. His recent research is focused on the dependability in Network Function Virtualization and Software Defined Network.

Elias P. Duarte Jr. is a Full-time Professor at Federal University of Parana, Curitiba, Brazil, where he is the leader of the Computer Networks and Distributed Systems Lab (LaRSis). His research interests include Computer Networks and Distributed Systems, their Dependability Management, and Algorithms. He has published more than 150 peer-reviewed papers and has chaired several conferences and workshops. He received a PhD degree in computer science from Tokyo Institute of Technology, Japan, 1997, the MSc degree in telecommunications from the Polytechnical University of Madrid, Spain, 1991, and the BSc and MSc degrees in computer science from Federal University of Minas Gerais, Brazil, 1987 and 1991, respectively. He chaired the Special Interest Group on Fault Tolerant Computing of the Brazilian Computing Society (2005-2007); the Graduate Program in Computer Science of UFPR (2006-2008); and the Brazilian National Laboratory on Computer Networks (from 2012). He is a member of the Brazilian Computer Society and a Senior Member of the IEEE.

How to cite this article: Turchetti RC, Duarte Jr EP. NFV-FD: Implementation of a failure detector using network virtualization technology. *Int J Network Mgmt*. 2017;27:e1988. https://doi.org/10.1002/nem.1988