Automated Establishment of a Secured Network for Providing a Distributed Container Cluster

Marcel Großmann, Andreas Eiermann
Faculty of Information Systems and Applied Computer Science
Otto-Friedrich University,
Bamberg, Germany
Email: marcel.grossmann@uni-bamberg.de, andreas@hypriot.com

***Abstract*—Modern service providers use virtualization in order to feasibly scale their applications. Their approaches exploit virtual machines to master the quality of service requirements, primarily redundancy, reliability and security. Accordingly, the services provided are hosted in data centers that fulfill the customers' needs. The standards of entry burden small enterprises by requiring them to establish their own distributed service delivery cloud. Therefore, we developed an energy-efficient solution, Hypriot Cluster Lab (HCL), that adapts Docker technologies in order to run on ARM powered single board computers. Herein, we demonstrate how HCL works with redundancy and replication between several hosts on different locations via wide area networks. Accordingly, we use a mesh network based on virtual private LANs to enable encrypted communication over the Internet between distributed hosts. Our result presents a fault tolerant, reliable and secure extension to connect several independent hosts with HCL to achieve the QoS-requirements.**

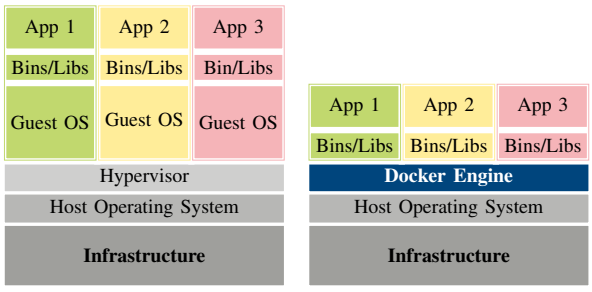
***Keywords*—Container Virtualization; Cluster Computing; Distributed Computing; Mesh Networking; Privacy; Reliability; Redundancy; QoS;**

I. Introduction

History has demonstrated that new technologies embed themselves in new areas of application in ways that complement existing infrastructures. For example, personal computers did not disestablish mainframes, but became a standard accessory on desktops for accessing the functional features of mainframes. Nowadays, *single board computers (SBC)* show similar behavior as they become established within the computer market as energy-efficient and inexpensive devices. They allow vendors to build manifold appliances that are part of the *Internet of Things (IoT)*. However, more devices in the IoT endorse unnecessary data communications that burden core networks and data centers in the *cloud*. Moreover, requesting services from the cloud "creates high latency to application, congestion and bottleneck to the network" [1]. As proposed by Aazam and Huh [2], fog computing is one possible solution for relocating virtualized services from the cloud to the edge network.

To empower edge nodes, SBCs are a perfect complement to relieve core networks of unnecessary traffic between IoT devices and centralized services in the cloud. Each SBC is able to aggregate, filter, and analyze data from connected IoT devices and pass only the relevant information to cloud services. Therefore, virtualized functionality of the data center is shifted to the edge nodes, thereby reducing latency times of IoT devices that request services in the cloud. Fog computing is based on the same paradigm as cloud computing, *virtualization*, which enables it to fluently migrate services.

Setting up SBCs at the edges of a network changes the way virtualized services are provided because they are unable to run *virtual machines (VM)* for services in a similar manner to the cloud. However, lightweight solutions already exist. For example, *LinuX Container (LXC)* behaves similar to a VM, but is challenging to use. Felter *et al.* [3] explain that containers are very beneficial, especially for small SBCs, because they "offer the control and isolation of VMs with the performance of bare metal" [3]. Fortunately, a more user-friendly option, *Docker*, was programmed three years ago.



(a) Virtual machines (b) Container virtualization

Figure 1. Comparison of virtualization approaches

Direct comparison of VMs and container virtualization in Figure 1 illustrate differences between the approaches. Docker containers are more lightweight compared to VMs without a guest *operating system (OS)*, but are restricted to the functionality of the host OS. For example, a container built on Windows is not able to run on a Linux host. Nevertheless, Docker packages a piece of software and its dependencies in a filesystem called *image*, which contains everything the application requires for execution. Each image's filesystem consists of layers that track various operations, similar to a versioning tool. This is beneficial if several containers use the same image because all layers of this image are started once, and for each container only the difference is loaded into its container environment.

Former cluster solutions that come with security features are primarily built for services hosted on VMs. For example,

978-0-9883045-1-2/16 $31.00 © 2016 ITC

Eucalyptus [4] uses VTun interfaces to connect various cluster front-ends via a tunnel, so that VM instances are connected by bridging their virtual interfaces to the VTun one. In contrast, the educational cloud computing platform called Seattle [5] offers a lightweight virtualization approach based on sandboxes that run inside Vessels and can be installed on ARM-powered devices. However, Seattle is only programmable with a subset of the Python programming language and is therefore not an alternative for the delivery of existing services.

Upon summing up the former developments, this paper outlines Dockers' (version $\geq$ 1.10) and SBCs' influence on fog computing. We demonstrate how to establish a distributed and secured cluster setup on the Raspberry Pi (RPi) 3 Model B by distributing them at different locations. It is the basis for making vision fog computing a reality.

In Section II we describe our setup for the demonstration infrastructure. Subsection II-A focuses on the establishment of a secure connection, followed by the initialization of a Docker cluster in Subsection II-B. The security for the proposed configuration is evaluated briefly in Section III. Subsequently, Docker-specific security features are compared to this approach in Subsection III-A. Current restrictions are then shown in Subsection III-B. The paper is concluded with Section IV.

II. Fog Computing with Docker

A. Initialization of a Distributed Mesh Infrastructure

Private transmissions over the Internet call for *Virtual Private Networks (VPN)*, which use tunneling and encryption to establish a secure connection between two hosts attached to the Internet. Traditionally, VPNs are created between two endpoints and additional tunnels are added for larger networks with more sites. According to Khanvilkar and Khokhar [6], an open source Linux based VPN solution, *tinc*[1], offers all four VPN security aspects: confidentiality, data integrity, authentication, and anti-replay. Moreover, it provides a built-in routing daemon that configures a full mesh network and eases the establishment of single-domain VPNs. Therefore, only endpoints are specified and tinc itself creates tunnels, thereby allowing an easier configuration and improved scalability [7]. First, for each tinc endpoint, several configuration files need to be generated that contain

- the `name` of the private network with its corresponding
- `tinc-up` and `tinc-down` scripts, to configure the host's network interface,
- the configuration file of the host itself, and
- a generated RSA key pair.

To connect the mesh network, tinc configurations are exchanged and stored on each endpoint including each node's public RSA key.

Our demonstration uses the *tincregistrar*[2] framework to administrate nodes of a private network and to automate the initialization of the mesh, which is publicly available on Github. It provides a script to generate the tinc configuration on each host and communicates with a Django[3] application to store and distribute endpoint configurations to ease the exchange of the necessary files. The server application runs inside a Docker container and can therefore start on each Docker-enabled Linux host. The basic mechanism of the communication between the registrar and one endpoint, cf. Figure 2:

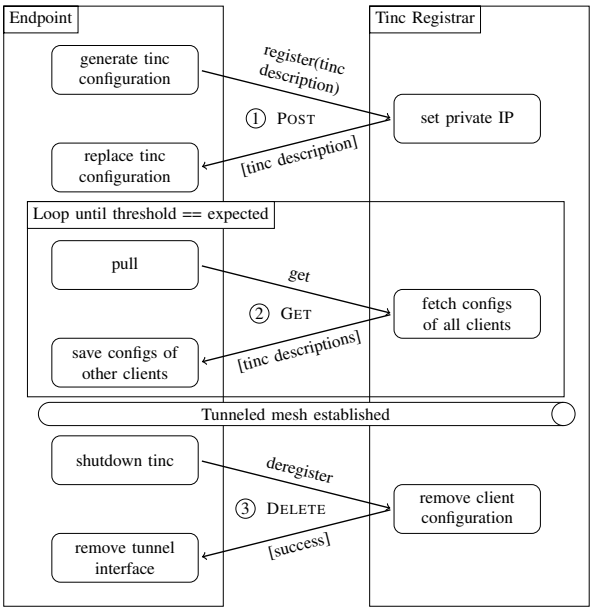


Figure 2. Communication mechanisms between the registrar[2] and one endpoint to setup a private secured infrastructure.

1) With POST an endpoint's tinc configuration is sent to the server and stored there.
2) By using GET, tinc configurations are retrieved for all registered endpoints.
3) DELETE removes endpoints from registered nodes.

After announcing its own configuration to the registrar, an endpoint performs a loop until it receives the configuration from all attached nodes.Within the receipt, the tinc deamon is started and data communication is possible within the tunneled mesh network.

For our demonstration, the mesh infrastructure is setup as shown in Figure 3, where each node is connected to the Internet and reaches a tincregistrar instance without being restricted by *network address translation (NAT)* or a firewall. Though tinc offers *session traversal utilities for NAT (STUN)* protocols to circumvent restrictions, our ongoing work examines this functionality.

B. Running Hypriot Cluster Lab on the Distributed Mesh

Hypriot Cluster Lab (HCL) [8] was initially designed to be void of a configuration, while running it on a local cluster

[1]https://www.tinc-vpn.org/
[2]https://github.com/whatever4711/tincregistrar.git
[3]https://www.djangoproject.com/

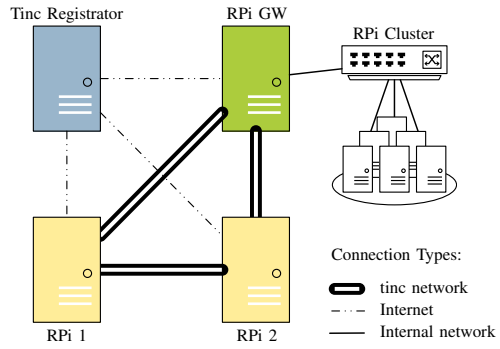


Figure 3. In this sample setup, RPi 1, RPi 2, and RPi GW (gateway) span up the tinc network. Behind the gateway, a subnet declaration allows a cluster to run that is reachable within the tinc network.

of RPis and performing all necessary steps to configure the cluster to communicate in a *virtual LAN (VLAN)* with its own *dynamic host configuration protocol (DHCP)* server. With tinc, it is unnecessary to create an additional VLAN and start a DHCP service. Thus, HCL only operates on the infrastructure generated by tinc. The reduced set of activities required for running HCL is depicted in Figure 4.

Leaving out the address configuration, HCL boots up and uses the tunnel interface directly. Thereafter, it re-configures the Docker engine to advertise the functionality of Docker Swarm[4] and initiates a Consul[5] container on a *leader* node. This node is a *key-value (KV)* store with a built-in DNS server that can easily discover and track the health of various services throughout an infrastructure. Subsequently, all other nodes join the existing Consul service by starting their own Consul containers. Similarly, all nodes connect with a Docker Swarm[4] container and start another instance as a manager or replica of the cluster. Docker-Compose[4], Consul[5], Docker Swarm[4] and other suitable tools can all be used to run services on the configured cluster. However, concerns may arise from the architecture of Docker images when running them on ARM machines.

III. Evaluation of Demonstration Approach

A. Existing Docker Security Features vs. Tinc

In order to communicate with Docker in a safe manner, *transport layer security (TLS)* can be enabled so that the daemon only allows connections from authenticated clients with a certificate signed by a *certificate authority (CA)*. The Consul service also offers a TLS encryption of all of its network traffic. It is setup by establishing an internal CA and distributing all necessary certificates and key files among all connected Consul hosts. By enabling all provided security features, the control communication for Swarm and its necessary KV store are secured by TLS.

[4]https://docs.docker.com/
[5]https://github.com/hashicorp/consul

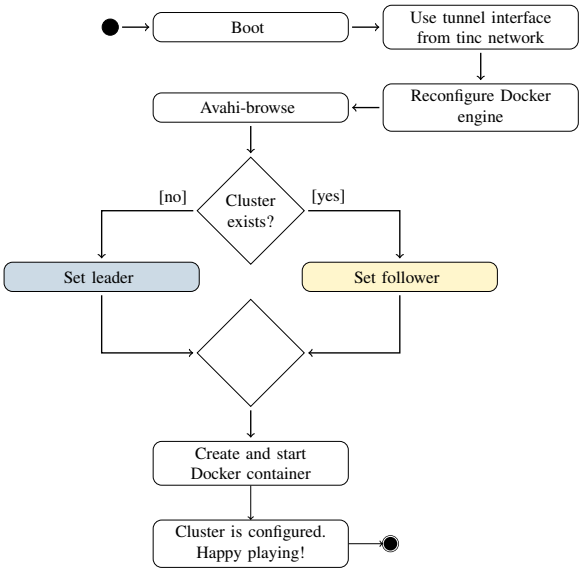


Figure 4. Activity diagram of HCL. It shows the reduced set of activities performed, if a tinc network already exists.

In conclusion, the establishment of natively integrated security features requires two adjustments and is therefore painful to organize and concerns only the control mechanisms of a Swarm. In contrast, our approach avoids provided security features and their inability to secure the inter-container communication. In fact, secured connections are established by tinc before the HCL cluster is booted up. As proof, all communications of a Swarm over a tinc connection are analyzed next. We start a HCL cluster on two nodes and reconfigure it on both nodes. Before changing the configuration of HCL, we stop it with:

```
VERBOSE=true cluster-lab stop
```

Modifying the following configuration parameter in `/etc/cluster-lab/cluster.conf` instructs HCL to disable VLAN and DHCP, and operate on tinc's `tun0` interface.

```
INTERFACE="tun0"
ENABLE_VLAN="false"
ENABLE_DHCP="false"
```

Additionally, the avahi configuration file `/etc/avahi/avahi-daemon.conf` needs to allow point-to-point discovery (`allow-point-to-point=yes`), and a restart of the service to activate this configuration. Then, we start HCL with:

```
VERBOSE=true cluster-lab start
```

On the first node, we execute these commands to create an overlay network called 'wireshark' and start a webserver named 'web'.

```
docker network create -d overlay \
--ip-range=172.29.254.0/24 \
--subnet 172.29.254.0/24 wireshark

docker run -it --rm --name web -v \
/var/run/docker.sock:/var/run/docker.sock \
--net wireshark firecyberice/whalesay:web
```

On the second node, tcpdump is initiated in order to capture the payload of tinc on UDP port 655 and UDP port 4789. The payload is used by Docker for its *Virtual Extensible LAN (VXLAN)* overlay networks. Afterwards, a web call is made against the webserver on the first node. The 'web' can be resolved in the 'wireshark' overlay network by the internal *Domain Name System (DNS)* of Docker.

```
docker run -it --rm --net wireshark \
 alpine:3.2 /bin/sh -c "apk add \
 --update curl && curl -L \
 'http://web:5000/message/Hello'"
```

The captured results are:

```
#vxlan
tcpdump: listening on eth1 [...]
0 packets captured [...]
#tinc
tcpdump: listening on eth1 [...]
5972 packets captured [...]
#vxlan
tcpdump: listening on tun0 [...]
12 packets captured [...]
```

In conclusion, packages are traversing the `tun0` device and not the `eth1` device, as the traffic is already encrypted by tinc. The number of tinc packets is far greater compared to those of VXLAN on `tun0`. This arises because the only VXLAN traffic are the HTTP request and response, while the tinc traffic contains additional control messages for Swarm and Consul.

These results affirm that our approach boots up a VPN-based mesh network between the nodes Swarm is running on and secures the control channel. Even inter-container communication traverses the Internet in an encrypted manner that is impossible by adopting Docker's and Consul's security mechanisms.

B. Current Restrictions

The tincregistrar from section II-A is currently proof-of-concept for how a tinc network could be initialized by using a central instance for managing the information of each node in the private network. At the moment, it does not provide the necessary security for the control channel, but it is on our future work agenda.

IV. Conclusion & Future Work

In our demonstration we spawned a private, secured, and geographically distributed cluster to run binned services with a secured link. This allows connection of Docker Swarm instances over the Internet with no traffic leaving the secured tinc mesh network. Moreover, we contribute open source tools to easily setup a distributed HCL in a fog environment.

In our ongoing work, we focus on securing the tincregistrar[2] framework. Until now, it only supported unencrypted control traffic for registering and communicating with the endpoints of a tinc mesh network.

Future extensions could include frameworks that support Docker and are able to manage containers inside a cluster, such as Kubernetes[6] or Apache Mesos[7]

References

[1] B. I. Ismail, E. M. Goortani, M. B. A. Karim, W. M. Tat, S. Setapa, J. Y. Luke, and O. H. Hoe, "Evaluation of docker as edge computing platform", in *Open Systems (ICOS), 2015 IEEE Confernece on*, Aug. 2015, pp. 130–135. DOI: 10.1109/ICOS.2015.7377291.

[2] M. Aazam and E.-N. Huh, "Fog computing and smart gateway based communication for cloud of things", in *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, Aug. 2014, pp. 464–470. DOI: 10.1109/FiCloud.2014.83.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers", *technology*, vol. 28, p. 32, 2014.

[4] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system", in *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, May 2009, pp. 124–131. DOI: 10.1109/CCGRID.2009.93.

[5] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: a platform for educational cloud computing", in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '09, Chattanooga, TN, USA: ACM, 2009, pp. 111–115, ISBN: 978-1-60558-183-5. DOI: 10.1145/1508865.1508905. [Online]. Available: http://doi.acm.org/10.1145/1508865.1508905.

[6] S. Khanvilkar and A. Khokhar, "Virtual private networks: an overview with performance evaluation", *IEEE Communications Magazine*, vol. 42, no. 10, pp. 146–154, Oct. 2004, ISSN: 0163-6804. DOI: 10.1109/MCOM.2004.1341273.

[7] I. Timmermans and G. Sliepen, *Tinc manual*. [Online]. Available: http://tinc-vpn.org/documentation-1.1/tinc.pdf.

[8] M. Großmann, A. Eiermann, and M. Renner, *Hypriot Cluster Lab: An ARM-Powered Cloud Solution Utilizing Docker*, presented at 23rd International Conference on Telecommunications (ICT2016), May 2016. [Online]. Available: http://ict-2016.org/pdf/Demo1.pdf.

[6] http://kubernetes.io/

[7] http://mesos.apache.org/