middle-sock: Linux namespace へのパケット宛先変換による DHCP サーバコンテナの移植性の向上

森井 佑誠[1]　大野 有樹[2]　串田 高幸[1]

概要：コンテナを用いた仮想化は, 従来の仮想化と比べ移植性が高い特徴をもつ. しかし, DHCP サーバコンテナではコンテナの移植性の特徴を活かせず, 必要な手順が増加する課題が存在する. 本研究では, ネットワーク分離とパケット宛先変換に行うソフトウェア middle-sock を提案する. middle-sock は, Linux カーネルに搭載されている namespace 機能を使用して DHCP サーバコンテナ内でネットワークを分離, そのネットワーク内でホスト環境を再現し DHCP サーバを起動させる. 同時に, DHCP サーバコンテナに送信された DHCP パケットの宛先を変換し作成した namespace に送信することによって, 増加する手順を減らす. 実験および評価では 50 回の起動実験を繰り返し, 平均起動時間を評価した. コンフィグを手動で書き換える既存手法では約 18.8 ms に対し, 提案手法では約 43.5 ms であった. 提案手法では, DHCP サーバの起動に加え, middle-sock の起動を含むため既存手法より起動時間は遅くなる.

1. はじめに

背景

仮想化技術の一つに, コンテナを用いた仮想化があげられる [1]. コンテナを用いることで, 従来のホスト型やハイパーバイザ型の仮想化に比べ, 容易にアプリケーションの実行環境を作成することができる. また, コンテナの廃棄, 再作成もこれらの仮想化環境と比べ容易であるため, 移植性が高い [2, 3]. 加えて, コンテナは軽量であるとともに, 仮想化によるオーバヘッドが少ない [4–7]. そのため, コンテナはクラウドだけでなく, IoT システムにも取り入れられている [8]. コンテナによる仮想化を実現するソフトウェアとして, Docker[*1] があげられる. Docker は, コンテナイメージからコンテナを作成, 実行できる環境をもつ [9–11]. Docker を例としたコンテナイメージをもとに, コンテナを作成するソフトウェアをコンテナランタイムと呼ぶ. 本研究では, このようなソフトウェアを Runtime と表記する. また, コンテナイメージやコンテナランタイムの標準仕様は Open Container Initiative (OCI)[*2] によって策定が進められている. コンテナを用いた仮想化では, 基本的にコンテナネットワークと呼ばれるネットワークをコンテナを作成した内に作成する. Runtime も同様にコンテナネットワークを作成する. 図 1 に Runtime を使用した場合のコンテナネットワークとホストマシンの関係を示す.

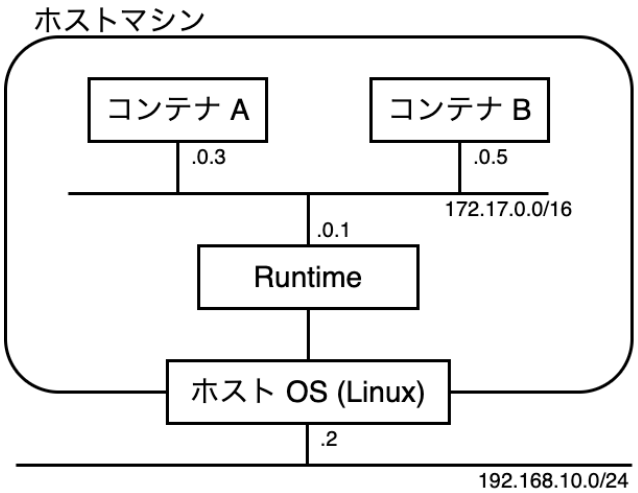

図 1　コンテナネットワークとホストマシンの関係

図 1 において, ホストマシンは 192.168.10.0/24 のネットワークに所属している. なお, IP アドレスは 192.168.10.2 である. Runtime は 172.17.0.0/16 のコンテナネットワークを作成し, そのネットワーク内にコンテナを配置する. 図 1 中のコンテナ A の IP アドレスは 172.17.0.3 であり, コンテナ B の IP アドレスは 172.17.0.5 である. それらは Runtime から自動的に割り当てられている. また, Runtime の IP アドレスは 172.17.0.1 であり, ホスト OS

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[2] 東京工科大学院バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1
[*1] https://www.docker.com
[*2] https://opencontainers.org

やコンテナネットワーク内から通信可能である．コンテナ A とコンテナ B の通信はデフォルトでは Runtime を経由して通信を行っている．ホスト OS とコンテナ A およびコンテナ B の通信も Runtime を経由する．これらの通信は Linux カーネルに搭載されている iptables を用いた Network Address Translation (NAT) と namespace による空間分離によって実現している．図 2 にコンテナ作成時における Docker と iptables の関係を示す．

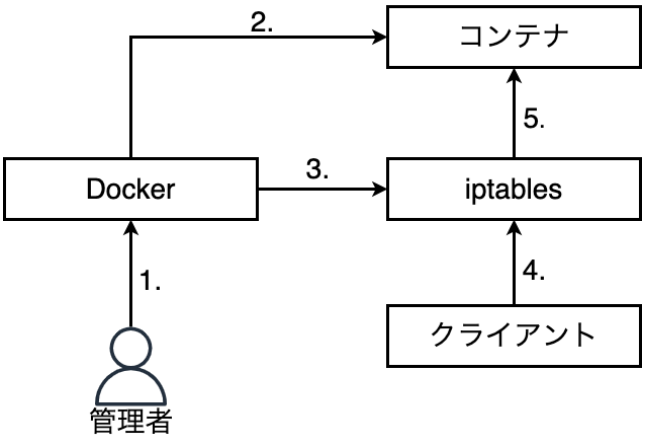

図 2　Docker と iptables の関係

図 2 では, 以下のことが行われている:

1. 管理者が Docker を用いてコンテナを作成する命令を送信する．
2. Docker がコンテナを作成し, コンテナに IP アドレスを割り当てる．
3. Docker が iptables にコンテナに割り当てた IP アドレスに NAT の設定を行う．
4. クライアントはコンテナのアプリケーションにアクセスするために iptables にパケットを送信する．
5. iptables はクライアントが送信したパケットを NAT によりコンテナに送信する．

Docker は iptables を用いることで, コンテナネットワークを通るパケットを管理している．

一方 namespace は, Linux カーネルに搭載されているプロセスの空間を分離する機能である．ここでの空間とは, プロセス ID やネットワークデバイスを例とした Linux カーネルで提供されている 8 個の空間のことを示す．空間の分離を実現するコマンドとして unshare がある．unshare は Linux のシステムコールでも用意されており, アプリケーションから呼び出すことも可能である．この機能を用いることで, 同じマシン上で動作しているアプリケーションをあたかも別々のマシンで動作させているように分離させることが可能となる．これらの機構を用いることでコンテナの安全性を高めている [12].

LAN 内の端末に IP アドレスを自動的に割り当てるプロトコルとして RFC2131 によって勧告されている Dynamic Host Configuration Protocol (DHCP) がある [13]. DHCP はクライアントサーバモデルを採用しており, 要求を行う DHCP クライアントと応答を返す DHCP サーバの二つが必要となる．DHCP クライアントと DHCP サーバは User Diagram Protocol (UDP) を使用して通信をしている．DHCP クライアントが DHCP サーバから IP アドレスを取得し, IP アドレスを開放するまでの通信を図 3 に示す．

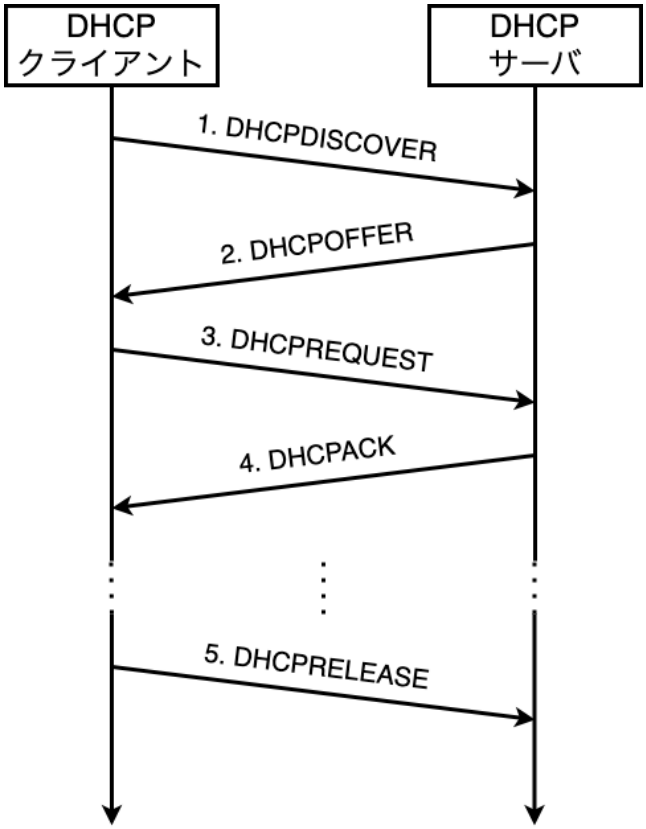

図 3　クライアントとサーバ間との通信

図 3 はクライアントとサーバ間で以下の通信が行われている:

1. DHCP クライアントは DHCPDISCOVER パケットをブロードキャストで送信する．
2. DHCP クライアントからの DHCPDISCOVER パケットを受信した DHCP サーバは DHCP クライアントに, DHCPOFFER パケットを送信する．
3. DHCP サーバからの DHCPOFFER パケットを受信した DHCP クライアントは DHCPREQUEST パケットをブロードキャストで送信する．
4. DHCP クライアントからの DHCPDISCOVER パケットを受信した DHCP サーバは DHCP クライアントに, IP アドレスが含まれている設定パラメータつきの DHCPACK パケットを送信する．DHCPACK パケットを受信した DHCP クライアントは設定パラメータ

をもとに自身の IP アドレスを設定する.

5. DHCP クライアントが IP アドレスを返却する際, そのことを DHCP サーバに伝えるために, DHCPRELEASE パケットを DHCP サーバに送信する.

また, DHCP パケットには Option と呼ばれる任意のパケットを埋め込むことができる [14]. Option の例として, ルータのアドレスを示す Option 3 や DNS のアドレスを示す Option 5 があげられる.

課題

DHCP サーバコンテナは, 他のコンテナと比べ, コンテナの移植性が低いという課題があげられる. 本研究におけるコンテナの移植性は, アプリケーションコンテナを動作させるまでユーザが必要とする手順を基準としている. 移植性の例示として, 図 4 に nginx におけるコンテナの作成を示す.

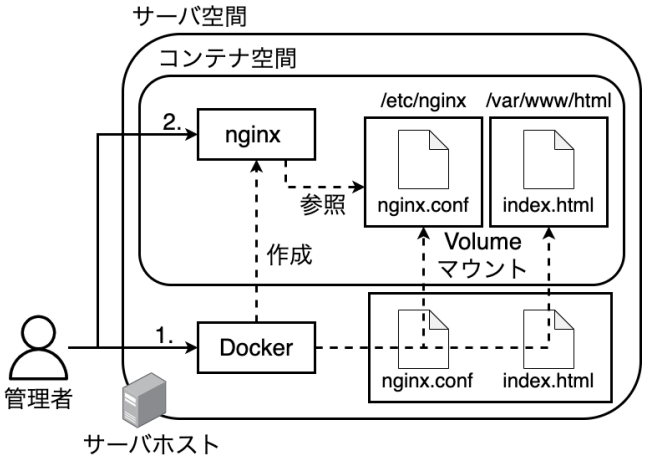

図 4 nginx におけるコンテナの作成

図 4 では, nginx の設定ファイルである nginx.conf や配信するコンテンツである index.html は既に用意されている. また, コンテナの作成には Docker を使用している. 図 4 におけるサーバ空間とは, サーバ内のファイル, ディレクトリ, アプリケーションが存在する空間を示している. コンテナ空間は, サーバ空間と同様にコンテナ内のファイル, ディレクトリ, アプリケーションが存在する空間である. サーバ空間内にコンテナ空間は存在し, コンテナ空間とサーバ空間の間で直接的なアクセスは基本的にはできない. 直接的なアクセスの例として, ファイル転送やアプリケーションの通信があげられる. その上でユーザ, すなわち管理者は nginx コンテナを動作させるために以下の手順を必要とする:

1. docker コマンドを用いて, Docker に nginx コンテナを起動させる命令を送信する. このとき Docker は受けた命令に従い, nginx コンテナを作成し, サーバ空間にある nginx.conf をコンテナ空間内の /etc/nginx ディレクトリに, index.html を /var/www/html ディレクトリに Volume マウントする. コンテナ空間の nginx はそれらのファイルを参照する.
2. nginx コンテナが起動しているか確認するため, curl コマンドやブラウザを用いて HTTP リクエストを送信し, レスポンスを受け取るか確認を行う.

このように, nginx では, 上記の手順で nginx コンテナを動作させることが可能である. nginx 以外のコンテナも同様に nginx コンテナを動作させるために行った手順を必要とする. これを踏まえ, DHCP サーバにおけるコンテナの作成を図 5 に示す.

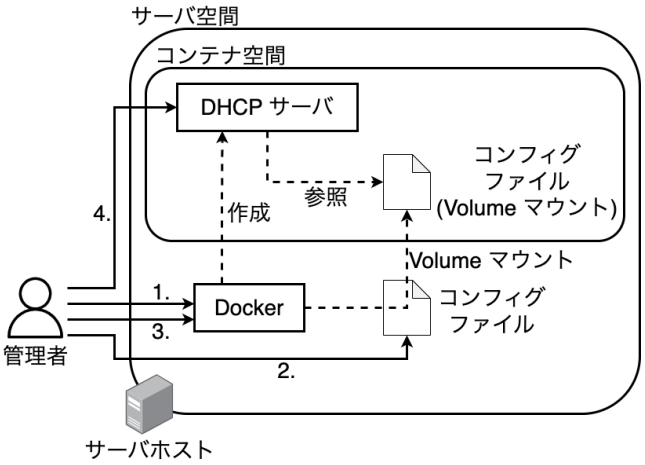

図 5 DHCP サーバにおけるコンテナの作成

図 5 では, 管理者は起動のために以下の手順を必要とする:

1. DHCP サーバの動作に必要なコンテナネットワークやコンテナ内で使用されるネットワークインターフェースを調べる.
2. 1. をもとにコンフィグファイルを書き換える.
3. docker コマンドを用いて, Docker に DHCP サーバコンテナを起動させる命令を送信する. このとき Docker は受けた命令に従い, DHCP サーバコンテナを作成すると同時に, サーバ空間にあるコンフィグファイルをコンテナ空間内に Volume マウントする. コンテナ空間の DHCP サーバは Volume マウントされたコンフィグファイルを参照する.
4. DHCP サーバが正常起動しているか確認するため, DHCP クライアントを用いて, コンフィグファイルに設定された IP アドレスが DHCP クライアントに送信されているか確認を行う.

DHCP サーバの構築は nginx と比べ手順が多く, 容易にコンテナ化できないことがわかる. また, このことは基礎実験にて実際に示す.

基礎実験

前項にて述べた移植性の課題を示すために, 実際にコンテナを用いた DHCP サーバを構築し, 列挙した手順が必要かどうかの基礎実験を行う. 基礎実験では図 6 に示す環境で実験を行う.

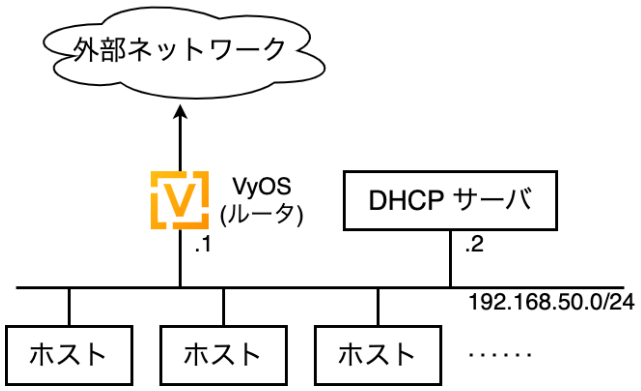

図 6　基礎実験の環境

図 6 ではルータに VyOS[*3] を使用し, 192.168.50.0/24 のネットワークを作成した. また, 192.168.50.0/24 のネットワーク上にはホスト及び DHCP サーバが存在する. ホストには IP アドレスは割り当てられておらず, DHCP サーバのみ 192.168.50.2 の IP アドレスが割り当てられている. また, DHCP サーバの OS は Ubuntu 22.04 を使用し, ソフトウェアは ISC DHCP を使用した.

まず, コンテナを使用せずに ISC DHCP を動作させる. 使用する ISC DHCP は Ubuntu のパッケージレジストリ[*4] で提供されているものを使用した. また, この環境で使用するコンフィグファイル dhcpd.conf をファイル 1 に示す.

```
default-lease-time 600;
max-lease-time 7200;

authoritative;

subnet 192.168.50.0 netmask 255.255.255.0 {
    range 192.168.50.30 192.168.50.200;
    server-identifier 192.168.50.2;
    option routers 192.168.50.1;
    option domain-name-servers 192.168.50.1;
    option subnet-mask 255.255.255.0;
}
```

ファイル 1: dhcpd.conf

ファイル 1 では, 192.168.50.0/24 のネットワークでの設定を記述している. 192.168.50.0/24 のネットワークは, DHCP サーバが所属しているネットワークである. この環境で ISC DHCP を起動し, ホスト上の DHCP クライアントを用いて, ISC DHCP がホストに IP が割り当てるか確認をした. IP を割り当てたかの判断は, DHCP サーバからホストに DHCPACK パケットを送信したかで判断を行う. また, パケットの確認は Wireshark を用いて行った. その結果, この環境では, ISC DHCP はホストに IP アドレスを割り当てたことを確認した.

次に, コンテナを使用して ISC DHCP を動作させる. コンテナイメージの作成及び Runtime には Docker を使用した. ファイル 2 にコンテナイメージの作成に利用した Dockerfile を示す.

```
FROM debian:bookworm-slim AS BUILDING

WORKDIR /root

RUN apt update && apt install -y wget make gcc file && \
    wget https://downloads.isc.org/isc/dhcp/4.4.3-P1/dhcp-4.4.3-P1.tar.gz && \
    tar xzf dhcp-4.4.3-P1.tar.gz && cd dhcp-4.4.3-P1 && \
    ./configure && make

FROM debian:bookworm-slim AS RUNNING

WORKDIR /root

COPY --from=BUILDING /root/dhcp-4.4.3-P1/server/dhcpd /root/dhcpd

RUN mkdir -p /var/lib/dhcp && touch /var/lib/dhcp/dhcpd.leases && \
    mkdir -p /etc/dhcp && touch /etc/dhcp/dhcpd.conf && \
    mkdir -p /run/dhcp-server && touch /run/dhcp-server/dhcpd.pid && \
    chmod 775 /var/lib/dhcp && chmod 664 /var/lib/dhcp/dhcpd.leases

ENTRYPOINT ["./dhcpd", "-f", "-4", "-pf", "/run/dhcp-server/dhcpd.pid", "-cf", "/etc/dhcp/dhcpd.conf", "-lf", "/var/lib/dhcp/dhcpd.leases"]
```

ファイル 2: Dockerfile

コンテナ環境では, コンテナネットワーク内でコンテナを動作させる. この環境におけるコンテナネットワークは 172.17.0.0/16 である. また, サーバ内のインターフェースの IP アドレスの関係について, 図 7 に示す.

[*3] https://vyos.io
[*4] https://packages.ubuntu.com/jammy/isc-dhcp-server

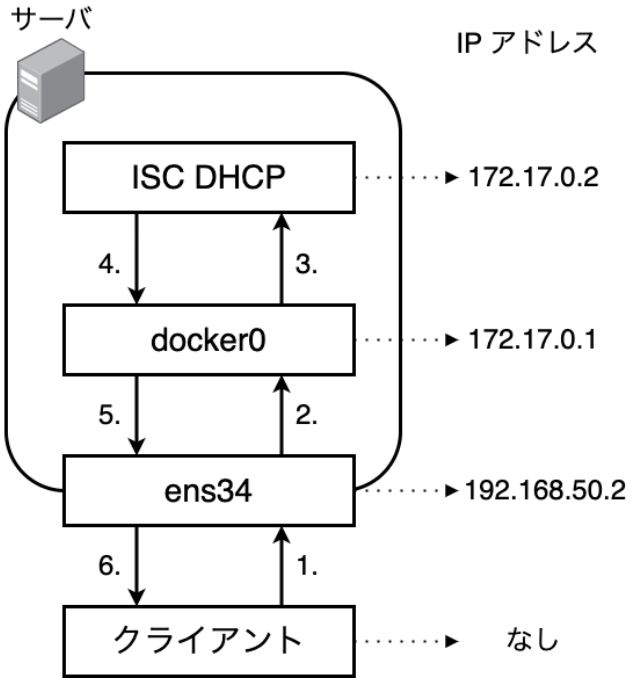

図 7 サーバ内のインターフェースの IP アドレスの関係

図 7 では, クライアントの DHCP パケットは 1. から 3. の順に ISC DHCP に送信される. 一方, ISC DHCP の DHCP パケットは 4. から 6. の順にクライアントへ送信される. また, ens34 はサーバのネットワークインターフェース, docker0 は Runtime が提供するネットワークインターフェースである. この環境において, ファイル 1 を用いて ISC DHCP コンテナを起動させたところ, コンテナ内の ISC DHCP プロセスが終了し, コンテナも起動せずに終了した. これは, コンテナネットワークである 172.17.0.0/16 の定義がファイル 1 内にされていないために発生した. このことから, 管理者は dhcpd.conf の書き換えを行う必要があり, 前項で示した移植性の課題があることがわかる.

各章の概要

本研究は, 次のように構成される. 第 1 章では, 本研究の背景・課題について述べる. 第 2 章では, 本研究の関連研究について述べる. 第 3 章では, 第 1 章で述べた課題を解決するための提案手法, ユースケース・シナリオについて述べる. 第 4 章では, 提案手法を実現するための実装と実験方法について述べる. 第 5 章では, 基礎実験とその分析について述べる. 第 6 章では, 本研究における議論について述べる. 第 7 章では, 本研究のまとめについて述べる.

2. 関連研究

Network Function Virtualization (NFV) と呼ばれる物理ネットワークから分離してソフトウェアから操作する方法がある [15, 16]. NFV は 2012 年に提唱され, European Telecommunications Standards Institute (ETSI) によって, 策定が進められている. NFV を用いることで, 従来, ユーザ側で必要となる DHCP サーバや Firewall をサービスプロバイダ側に委譲することができ, ユーザ側は管理をする必要がなくなる. サービスプロバイダ側に委譲された機能は仮想化され, 物理的に操作することなく設定を変更することが可能である. 加えて, この例の他に, Long Term Evolution (LTE) の中心ネットワークである Evolved Packet Core (EPC) に適用できる例がある. NFV を適用することで, EPC で使われる機能を仮想化し, データセンタ側に委譲できるため, 容易にスケーリングできる利点が存在する. しかし, コンテナ環境での NFV については追加の研究が必要とされている. そのため, コンテナを前提とした本研究では NFV による仮想化は適用できない.

NFV に関連して, サーバレス環境で NFV を用いた DHCP 環境を構築し, 性能測定をした研究がある [17]. この研究では, サーバレス環境を構築できるソフトウェア OpenWhisk を用いて, コンテナ化した DHCP サーバをコンテナとしてサーバレス環境で動作させている. この環境とサーバレスではない従来の環境での DHCP サーバと NAT の性能比較をこの研究は行っている. 結論として, DHCP サーバのサーバレス環境での有用性を示せたが, 必ずしもサーバレス環境での実行が優れているということではないと述べられている. 本研究では, DHCP サーバコンテナの移植性に焦点をおいているため, 性能評価となるこの研究は適用できない.

コンテナによる仮想化や NFV とは違った手段として OpenFlow がある [18–20]. OpenFlow では, スイッチやルータを例としたネットワーク機器の設定を Controller と呼ばれるリモートサーバに委譲させる. ネットワーク機器の設定を Controller に委譲させることで各機器に対する設定を一つのサーバに集約でき, 各機器で設定する必要性がなくなる. これは DHCP サーバ機能に限定したネットワーク機器も同様で, OpenFlow プロトコルさえ実装していればよい. しかし, 本研究では, ソフトウェアを用いた DHCP サーバを利用しているため, ネットワーク機器に適用する OpenFlow はこの研究では適用できない.

3. 提案

提案方式

提案方式では, ネットワーク分離とパケット宛先変換に行うソフトウェア middle-sock を提案する. middle-sock はコンテナ内で Linux カーネルに搭載されている namespace 機能を利用し, ネットワークを分離させることでホスト環境と同様な環境を namespace 内に作成する. 作成した namespace 内で DHCP サーバを起動させることで, DHCP サーバはホスト環境と同様な環境で動作できる. 同時に, コンテナに送信された DHCP パケットの宛先を変換し作成した namespace に送信する. これにより, 課題で示した

DHCP サーバコンテナの構築で必要な 2. までの手順を減らすことで移植性を向上させることが可能である. 図 8 に middle-sock と Runtime, DHCP サーバとの関係を示す.

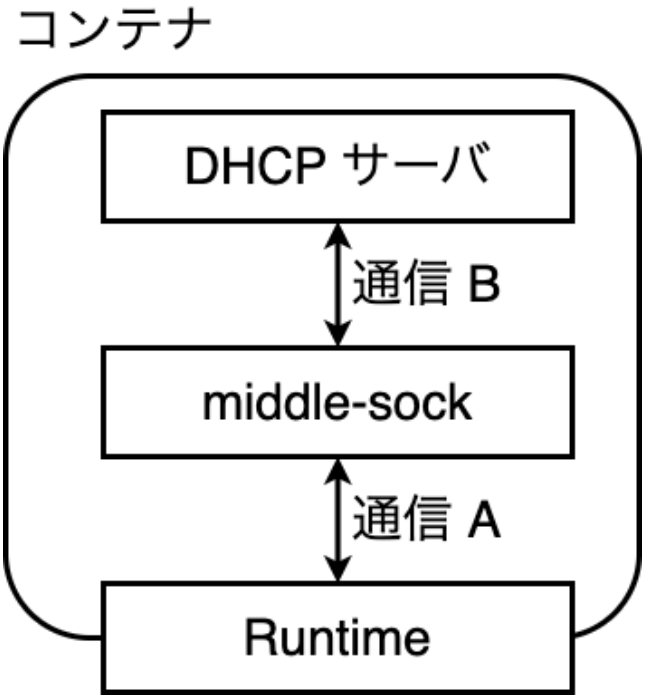

図 8 middle-sock と Runtime, DHCP サーバとの関係

図 8 では, middle-sock は Runtime と通信 A, DHCP サーバと通信 B を行う. Runtime はコンテナイメージからコンテナを作成, 実行できる環境をもつソフトウェアである. 通信 A 及び通信 B の内容は以下である:

通信 A: middle-sock と Runtime で DHCP パケットの送受信を行う. パケットの内容は, middle-sock がない場合の DHCP サーバと Runtime で送受信される DHCP パケットと同じである.

通信 B: middle-sock と DHCP サーバで DHCP パケットの送受信を行う. ここで, 本研究で提案する手法を適用させる.

提案ソフトウェアである middle-sock は以下の特徴をもつ:

- ホストが所属するネットワークとホストのネットワークインターフェースを取得
- ホスト環境を再現するための network namespace を作成
- コンテナと namespace との間に通信するための Virtual Ethernet (veth) ピアを作成
- veth ピアの片方にホストと同じ IP アドレスとネットワークインターフェース名を割当
- namespace 内で DHCP サーバを起動
- 受信した DHCP パケットを namespace に送信

ホストが所属するネットワークとホストのネットワークインターフェースの取得にはホストの /proc/net/route ファイルを利用する. /proc/net/route ファイルの内容の例をファイル 3 に示す.

```
1 Iface   Destination     Gateway         Flags   RefCnt
  Use     Metric  Mask            MTU     Window  IRTT
2 ens34   00000000        0164A8C0        0003    0
  0       100     00000000        0       0       0
3 ens34   2346C80A        0164A8C0        0007    0
  0       100     FFFFFFFF        0       0       0
4 docker0 000011AC        00000000        0001    0
  0       0       0000FFFF        0       0       0
5 ens34   0064A8C0        00000000        0001    0
  0       100     00FFFFFF        0       0       0
6 ens34   0164A8C0        00000000        0005    0
  0       100     FFFFFFFF        0       0       0
7 ens34   0664A8C0        00000000        0005    0
  0       100     FFFFFFFF        0       0       0
```

ファイル 3: /proc/net/route

/proc/net/route は procfs と呼ばれるファイルシステムに存在するファイルである. procfs はコンピュータを構成するハードウェアの情報をファイルとして読み込むことができるファイルシステムである. ハードウェアの例として CPU やメモリがあげられ, その情報の例には CPU コア数, 使用メモリ量があげられる. UNIX や UNIX 系の OS ではこの procfs によって, ハードウェア情報をファイルの一つとして扱うことが可能である [21]. /proc/net/route はこのうち, ルーティングに関する情報を格納している. /proc/net/route は 11 個の要素で構成されている. このうち DHCP サーバのコンフィグで必要となる要素は, Iface, Destination, Gateway, Flags, Mask である. これらの要素の各説明を表 1 に示す.

表 1 /proc/net/route における必要な要素の説明表

要素	説明
Iface	ネットワークインターフェース
Destination	宛先 IP アドレス
Gateway	ゲートウェイ
Flags	<linux/route.h>に定義されたフラグ
Mask	サブネットマスク

また, <linux/route.h>に定義されているフラグを Linux ソースコード上の include/uapi/linux/route.h[*5] から抜粋しファイル 4 に示す.

表 1 とファイル 4 を使用して, ファイル 3 の内容について説明する. ファイル 3 の 3 行目の内容を例として説明する. 内容の要素と値を表 2 に示す.

表 2 において, Destination, Gateway, Mask は 16 進数

[*5] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/route.h

```
51 #define RTF_UP          0x0001          /* route
   usable                 */
52 #define RTF_GATEWAY     0x0002          /* destination
   is a gateway     */
53 #define RTF_HOST        0x0004          /* host entry
   (net otherwise)   */
54 #define RTF_REINSTATE   0x0008          /* reinstate
   route after tmout  */
55 #define RTF_DYNAMIC     0x0010          /* created
   dyn. (by redirect)   */
56 #define RTF_MODIFIED    0x0020          /* modified
   dyn. (by redirect)  */
57 #define RTF_MTU         0x0040          /* specific
   MTU for this route  */
58 #define RTF_MSS         RTF_MTU         /*
   Compatibility :-(            */
59 #define RTF_WINDOW      0x0080          /* per route
   window clamping    */
60 #define RTF_IRTT        0x0100          /* Initial
   round trip time      */
61 #define RTF_REJECT      0x0200          /* Reject
   route                 */
```

ファイル 4: include/uapi/linux/route.h

表 2 内容の要素と値

要素	値
Iface	ens34
Destination	2346C80A
Gateway	0164A8C0
Flags	0007
Mask	FFFFFFFF

表記のリトルエンディアンで表されている．これらを変換した結果を表 3 に示す．

表 3 変換結果

要素	値
Destination	10.200.70.35
Gateway	192.168.100.1
Mask	255.255.255.255

また, Flags は<linux/route.h>に示されているフラグのビット演算である．そのため, 0007 を変換すると, RTF_UP, RTF_GATEWAY, RTF_HOST の 3 つのフラグが存在することがわかる．よって, これらの結果から 3 行目の内容を説明することができる．/proc/net/route ファイルではこの内容が一行ごとに記載されている．提案手法では, ホストマシンが所属するネットワークを判別するために, Gateway が存在し, かつ自身のネットワークがわかるネットワークを /proc/net/route ファイルから複数選択する．ファイル 3 では, 2 行目と 5 行目が該当する．また, ネットワークインターフェースが複数個存在する場合, ネットワークインターフェースごとに同じ処理を行う．このアルゴリズムをアルゴリズム 1 に示す．

アルゴリズム 1 ネットワークの判別アルゴリズム

Require: $lines$:= /proc/net/route の 2 行目以降
Require: RTF_UP := 0x0001
Require: $RTF\_GATEWAY$:= 0x0002
function GETNETINFO(lines)
  $results \leftarrow \{\}$
  for all $line \leftarrow lines$ **do**
    $element \leftarrow$ SPLIT($line$)
    $Iface \leftarrow element.Iface$
    $Destination \leftarrow element.Destination$
    $Gateway \leftarrow element.Gateway$
    $Flags \leftarrow element.Flags$
    $Mask \leftarrow element.Mask$
    $segments \leftarrow$ COUNT($Mask, 1$)
    if $Flags$ & RTF_UP **then**
      if $Gateway$ **and** $Flags$ & $RTF\_GATEWAY$ **then**
        if not $results[Iface]$ **then**
          $results[Iface] \leftarrow Iface$
        end if
        $results[Iface].Gateway \leftarrow Gateway$
      else if not $Destination >> segments$ **then**
        if not $results[Iface]$ **then**
          $results[Iface] \leftarrow Iface$
        end if
        $results[Iface].Destination \leftarrow Destination$
        $results[Iface].Mask \leftarrow Mask$
      end if
    end if
  end for
  return $results$
end function

middle-sock は, コンテナ内で network namespace を作成する．namespace の作成後, middle-sock は veth ピアを作成する．veth ピアの一つは, ホストと同様な環境にするためにホストの IP アドレスとアルゴリズム 1 をもとに取得したホストの所属ネットワークとネットワークインターフェースを使用して作成する．その veth ピアを作成した namespace に割り当てて, コンテナと namespace の間で疎通ができるようにする．その後, 作成した namespace 内で DHCP サーバを起動する．このとき, DHCP サーバは namespace に割り当てされた veth ピアの IP アドレスとネットワークインターフェースをもとに起動する．DHCP サーバの起動後, middle-sock は受信した DHCP パケットを namespace に存在する DHCP サーバに対し転送するため, パケットの変換を行う．DHCP パケットの変換アルゴリズムをアルゴリズム 2 に示す．

アルゴリズム 2 では, 受信したパケットの送信元アドレスがコンテナランタイムによって作成されたインターフェースの IP アドレスの場合, 宛先アドレスをホストの IP アドレスに変換を行う．また, コンテナと namespace の静的ルートを DHCP サーバの起動前に作成しているた

アルゴリズム 2 DHCP パケット変換アルゴリズム

Require: *packet* := DHCP のパケット
Require: *IP_HOST* := ホストの IP アドレス
Require: *IP_RUNTIME_IF* := コンテナランタイムインターフェースの IP アドレス

```
function TRANSFORM(packet)
    input ← packet
    if input.src == IP_RUNTIME_IF then
        input.dest ← IP_HOST
    end if
    output ← input
    return output
end function
```

め, コンテナ内におけるホストの IP アドレスのルートは namespace 宛てとなる. 一方, パケットの送信元アドレスがコンテナランタイムによって作成されたインターフェースの IP アドレスではない場合, 変換を行わない. このアルゴリズムにより, DHCP サーバ宛のパケットを DHCP サーバに転送することができる.

ユースケース・シナリオ

本提案を適用するユースケース環境は, 複数の LAN が存在する環境である. この提案を適用するユースケース環境の例を図 9 に示す.

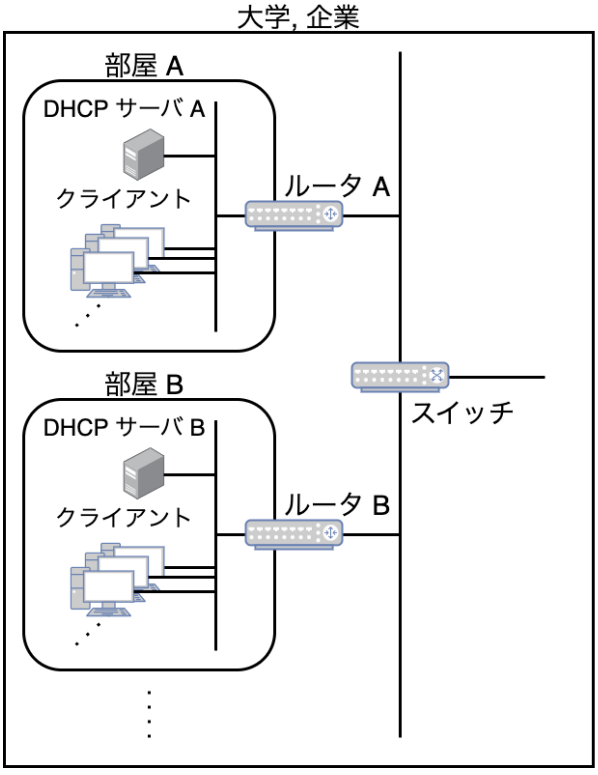

図 9 本提案におけるユースケース環境

図 9 では, 大学や企業を例として, 部屋ごとにネットワークを区切るために LAN が構成されている. また, 各部屋には DHCP サーバが用意されており, 各部屋のクライアントはこの DHCP サーバにリクエストを送信することで IP アドレスを取得する. 各部屋に DHCP サーバが設置されている理由として, 負荷分散があげられる. これは, 大学の講義を例とした人の出入りが一定時間に発生する場合, DHCP サーバへのリクエストが一時的に増える. このリクエストの分散のために各部屋に DHCP サーバを設置している. シナリオの具体例は, 既存の DHCP サーバをコンテナ内で動作させるように移行することである. 移行の理由として, コンテナを用いることでソフトウェアのバージョンを固定しやすいため, 管理者が環境を気にせずに管理できるということがあげられる. このとき, ユーザであるシステム管理者は DHCP サーバをコンテナ内で動作させるにあたって, コンフィグの書き換えや DHCP サーバ環境の調査を例とした作業を行われければならない. しかし, 本研究で提案した手法を適用することで, システム管理者が行う作業を減らすことが可能である.

4. 実装

実装では, DHCP サーバ起動とパケットの転送の 2 つの実装を行った.

まず, DHCP サーバ起動の実装について図 10 に示す.

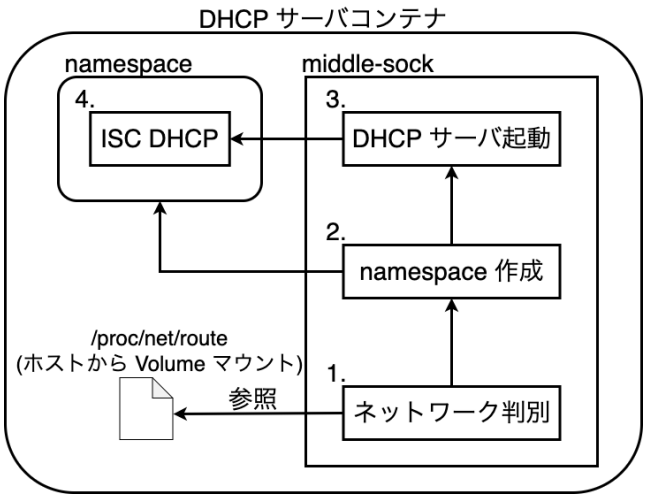

図 10 DHCP サーバ起動の実装

図 10 では, 1. から 4. に従って処理が行われる. 1. のネットワーク判別では, ホストから Volume マウントされた /proc/net/route ファイルを参照し, アルゴリズム 1 をもとにホストの所属ネットワークやネットワークインターフェースを判別する. 2. の namespace 作成では, 1. で取得したネットワークをもとに ISC DHCP を起動するための namespace を作成する. 3. の DHCP サーバ起動では, 2. で作成された namespace 内に ISC DHCP のプロセスを作成する. 最後の 4. にて, 3. で作成された ISC DHCP が起動する.

次にパケットの転送の実装について図 11 に示す.

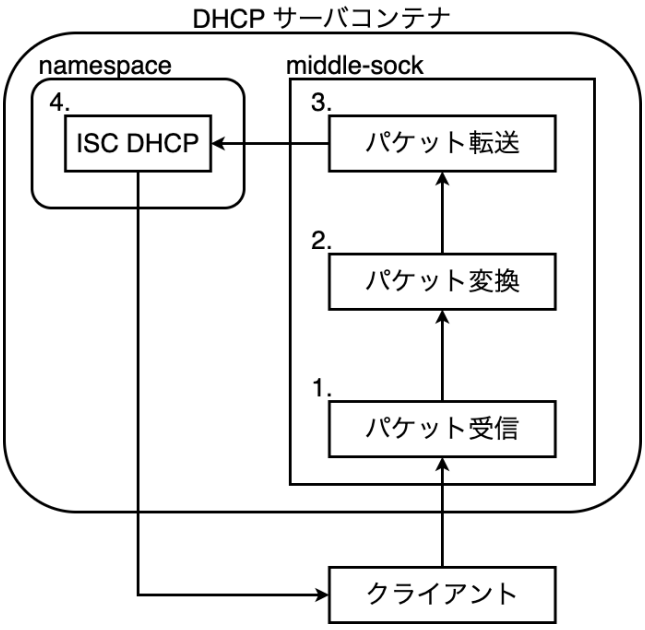

図 11 パケットの転送の実装

図 11 では, 図 10 と同様に 1. から 4. に従って処理が行われる. 1. のパケット受信では, クライアントからの DHCP パケットを受信する. 2. のパケット変換では, 1. で受信した DHCP パケットをアルゴリズム 2 をもとに変換を行う. 3. のパケット転送では, 2. で変換した DHCP パケットを namespace 内で起動している ISC DHCP に転送する. 最後の 4. にて, ISC DHCP は DHCP パケットに対する返答をクライアントに対して行う.

これら 2 つの実装は GitHub 上にて確認することができる[*6].

5. 評価実験

提案方式を適用した実験環境で実際に DHCP サーバコンテナの構築をする. このとき, DHCP サーバの起動までにかかる時間を計測し, 提案手法を適用せずに DHCP サーバコンテナを構築する方法と比較を行う.

実験環境

基礎実験で用いた図 6 と同様の実験環境で実験を行う. ルータは VyOS であり, 192.168.50.0/24 のネットワークが作成されている. DHCP サーバの IP アドレスは 192.168.50.2 であり, DHCP サーバに使用するソフトウェアは ISC DHCP である. また, DHCP サーバはコンテナとして起動を行い, 提案手法と適用したコンテナと, 手動で dhcpd.conf を書き換えたコンテナとで実験, 比較を行う. 加えて, 手動で dhcpd.conf を書き換える手法を既存手法と表記する. 書き換えた dhcpd.conf をファイル 5 に示す.

ISC DHCP が起動するまでの起動時間を既存手法と提案手法の両方で計測, 要した時間を評価として比較する. 起動時間の計測回数は 50 回とした.

[*6] https://github.com/cdsl-research/middle-sock

```
default-lease-time 600;
max-lease-time 7200;

authoritative;

shared-network container {
  # main network
  subnet 192.168.50.0 netmask 255.255.255.0 {
    range 192.168.50.30 192.168.50.200;
    server-identifier 192.168.50.2;
    option routers 192.168.50.1;
    option domain-name-servers 192.168.50.1;
    option subnet-mask 255.255.255.0;
  }

  # container network
  subnet 172.17.0.0 netmask 255.255.0.0 {
  }
}
```

ファイル 5: 書き換えた dhcpd.conf

実験結果と分析

図 12 に起動時間の結果を示す. 起動時間の単位はミリ秒 [ms] である.

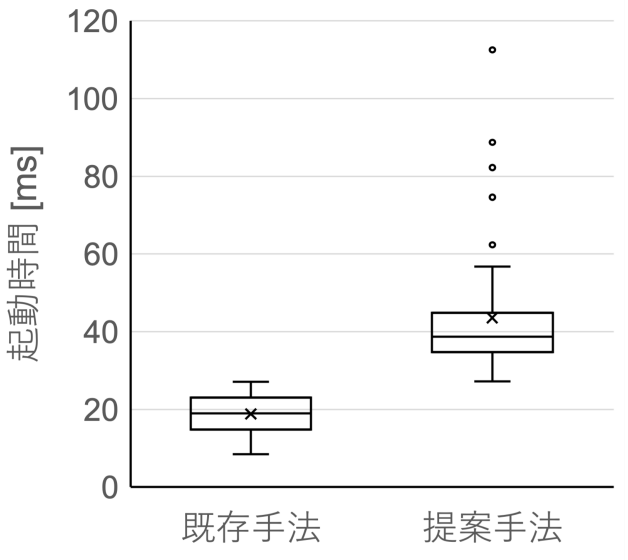

図 12 起動時間の結果

起動実験では, 既存手法では平均起動時間は約 18.8 ms であった. 一方, 提案手法では, 平均起動時間は約 43.5 ms であった. また, 最大起動時間では, 既存手法は約 27.1 ms であったのに対し, 提案手法は約 112.5 ms であった. 提案手法では, ISC DHCP のほかに提案したソフトウェアである middle-sock の起動が必要なため, 既存手法より長く起動時間を要している. しかし, 最大起動時間である 112.5

ms を除く起動時間は 100.0 ms 以内であった.

6. 議論

本研究では, 単一ホストマシンを対象に提案を行い, 実験では Docker を使用し, 評価および分析を行った. 実際の運用において, DHCP サーバの冗長性を高める場合, DHCP サーバをフェイルオーバーする手法があげられる. しかし, 本研究においては DHCP サーバコンテナ内で DHCP サーバを構築しているため, DHCP サーバコンテナ内でフェイルオーバーさせる必要がある. DHCP サーバコンテナ内でフェイルオーバーする方法として, DHCP サーバが起動している namespace のほかにもう一つフェイルオーバー先となる namespace を作成しその中で DHCP サーバを起動する方法がある. また, フェイルオーバーの検知処理は, middle-sock で行い宛先変換時にフェイルオーバー先のアドレスに宛先を変更することによって, DHCP サーバコンテナ内でフェイルオーバーをさせることが可能である.

7. おわりに

本研究では, DHCP サーバコンテナにおける移植性の低さを示し, ネットワーク分離とパケット宛先変換による DHCP サーバコンテナの移植性の向上の提案を行った. 実験では, 既存手法と提案手法における起動時間の比較を行った. その結果, 既存手法では平均起動時間は約 18.8 ms に対し, 提案手法では約 43.5 ms であった. また, 最大起動時間は既存手法では約 27.1 ms に対し, 提案手法では約 112.5 ms であった.

参考文献

[1] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A. and Peterson, L.: Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors, *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, New York, NY, USA, Association for Computing Machinery, p. 275–287 (online), DOI: 10.1145/1272996.1273025 (2007).

[2] Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P.: Cloud Container Technologies: A State-of-the-Art Review, *IEEE Transactions on Cloud Computing*, Vol. 7, No. 3, pp. 677–692 (online), DOI: 10.1109/TCC.2017.2702586 (2019).

[3] Kang, H., Le, M. and Tao, S.: Container and Microservice Driven Design for Cloud Infrastructure DevOps, *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 202–211 (online), DOI: 10.1109/IC2E.2016.26 (2016).

[4] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T. and De Rose, C. A. F.: Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments, *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240 (online), DOI: 10.1109/PDP.2013.41 (2013).

[5] Li, Z., Kihl, M., Lu, Q. and Andersson, J. A.: Performance Overhead Comparison between Hypervisor and Container Based Virtualization, *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pp. 955–962 (online), DOI: 10.1109/AINA.2017.79 (2017).

[6] Kozhirbayev, Z. and Sinnott, R. O.: A performance comparison of container-based technologies for the Cloud, *Future Generation Computer Systems*, Vol. 68, pp. 175–182 (online), DOI: https://doi.org/10.1016/j.future.2016.08.025 (2017).

[7] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J.: An updated performance comparison of virtual machines and Linux containers, *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172 (online), DOI: 10.1109/ISPASS.2015.7095802 (2015).

[8] Celesti, A., Mulfari, D., Fazio, M., Villari, M. and Puliafito, A.: Exploring Container Virtualization in IoT Clouds, *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 1–6 (online), DOI: 10.1109/SMARTCOMP.2016.7501691 (2016).

[9] Boettiger, C.: An Introduction to Docker for Reproducible Research, *SIGOPS Oper. Syst. Rev.*, Vol. 49, No. 1, p. 71–79 (online), DOI: 10.1145/2723872.2723882 (2015).

[10] Anderson, C.: Docker [Software engineering], *IEEE Software*, Vol. 32, No. 3, pp. 102–c3 (online), DOI: 10.1109/MS.2015.62 (2015).

[11] Rad, B. B., Bhatti, H. J. and Ahmadi, M.: An introduction to docker and analysis of its performance, *International Journal of Computer Science and Network Security (IJCSNS)*, Vol. 17, No. 3, p. 228 (2017).

[12] Dua, R., Raja, A. R. and Kakadia, D.: Virtualization vs Containerization to Support PaaS, *2014 IEEE International Conference on Cloud Engineering*, pp. 610–614 (online), DOI: 10.1109/IC2E.2014.41 (2014).

[13] Droms, R.: Dynamic Host Configuration Protocol, RFC 2131 (1997).

[14] Droms, R. and Alexander, S.: DHCP Options and BOOTP Vendor Extensions, RFC 2132 (1997).

[15] Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., De Turck, F. and Boutaba, R.: Network Function Virtualization: State-of-the-Art and Research Challenges, *IEEE Communications Surveys & Tutorials*, Vol. 18, No. 1, pp. 236–262 (online), DOI: 10.1109/COMST.2015.2477041 (2016).

[16] Han, B., Gopalakrishnan, V., Ji, L. and Lee, S.: Network function virtualization: Challenges and opportunities for innovations, *IEEE Communications Magazine*, Vol. 53, No. 2, pp. 90–97 (online), DOI: 10.1109/MCOM.2015.7045396 (2015).

[17] Savi, M., Banfi, A., Tundo, A. and Ciavotta, M.: Serverless Computing for NFV: Is it Worth it? A Performance Comparison Analysis, *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 680–685 (online), DOI: 10.1109/PerComWorkshops53856.2022.9767495 (2022).

[18] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. and Turner, J.: OpenFlow: Enabling Innovation in Campus Networks, *SIGCOMM Comput. Commun. Rev.*, Vol. 38, No. 2, p. 69–74 (online), DOI: 10.1145/1355734.1355746 (2008).

[19] Hu, F., Hao, Q. and Bao, K.: A Survey on Software-

Defined Network and OpenFlow: From Concept to Implementation, *IEEE Communications Surveys & Tutorials*, Vol. 16, No. 4, pp. 2181–2206 (online), DOI: 10.1109/COMST.2014.2326417 (2014).

[20] Lara, A., Kolasani, A. and Ramamurthy, B.: Network Innovation using OpenFlow: A Survey, *IEEE Communications Surveys & Tutorials*, Vol. 16, No. 1, pp. 493–512 (online), DOI: 10.1109/SURV.2013.081313.00105 (2014).

[21] Ritchie, D. M.: The UNIX System: A Stream Input-Output System, *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pp. 1897–1910 (1984).

