WYŻSZA SZKOŁA ZARZĄDZANIA I BANKOWOŚCI W KRAKOWIE

WYDZIAŁ ZARZĄDZANIA, FINANSÓW I INFORMATYKI

KIERUNEK: INFORMATYKA

PRACA DYPLOMOWA INŻYNIERSKA

Piotr Czyż

Emulacja sieci LAN/WAN w środowisku wirtualnym przy użyciu technologii kontenerów

PROMOTOR
dr inż. Jacek Kosiński

KRAKÓW 2016

Wstęp

W procesie wytwarzania oprogramowania zawsze występują etapy testowania poszczególnych jego modułów lub też końcowego systemu. Weryfikacji podlegają wtedy stawiane przed nim wymagania, lokalizowane są błędy. O ile aplikacja jest niezależna i działająca na pojedynczych stanowiskach, testowanie jej nie stwarza problemów. Sytuacja zmienia się w przypadku systemów typu klient-serwer, zwłaszcza gdy docelowo mają pracować w wielu rozproszonych fizycznie lokalizacjach. Aplikacje takie oczywiście do działania wymagają komunikowania się między sobą. Sposób w jaki jest to wykonywane uzależniony jest od wielkości obszaru działania danego systemu. W przypadku systemów działających w obrębie budynku lub ich grupy jak na przykład w zakładach pracy, komunikacja będzie przebiegać zwykle w sieciach lokalnych zapewniających stosunkowo dobre warunki do ich funkcjonowania. Często jednak systemy te łączą ze sobą wiele oddziałów danej firmy rozproszonych w obrębie danego miasta, kraju. Wymaga to użycia łączy telekomunikacyjnych mających bardzo zróżnicowane parametry transmisji danych, zarówno pod względem dostępnego pasma jak i samej jakości sygnału. Wpływa to bezpośrednio na poprawność działania systemów typu klient-serwer. Z drugiej strony firmy produkujące oprogramowanie wykonują to w obrębie własnych sieci lokalnych. To zróżnicowanie pomiędzy środowiskiem produkcyjnym a środowiskiem testowym, będzie przyczyną spadku wiarygodności przeprowadzanych testów. Rozwiązaniem tego problemu byłoby przeprowadzanie testów w rzeczywistych sieciach, jednak po pierwsze jest to nie ekonomiczne z powodu konieczności utrzymywania dodatkowych zewnętrznych systemów. Poza tym utrudnione byłyby wprowadzanie zmian w strukturze takich sieci testowych oraz niemożliwe byłoby swobodne definiowanie w nich odpowiednich warunków testowych. Aby te niedogodności wyeliminować konieczne jest uruchomienie w lokalnej sieci oprogramowania emulującego działanie rzeczywistych sieci internetowych. W czasie gdy wirtualizacja jest już technologią dojrzałą o rozbudowanych funkcjach, możliwe stało się wykorzystanie jej w tym celu. Dzięki niej w szybszy i tańszy sposób tworzone są wirtualne komputery z szeroką gamą dostępnych systemów operacyjnych, jak i wirtualne sieci z możliwością ich elastycznej

i skalowalnej rekonfiguracji. Użycie takiej infrastruktury automatycznie przyspiesza proces testowania oprogramowania a zwłaszcza udostępnia programowalne środowisko o zadanych parametrach.

Celem niniejszej pracy jest stworzenie rozwiązania umożliwiającego uruchamianie i testowanie aplikacji sieciowych w definiowanych programowo sieciach wirtualnych o zadanych parametrach. Dodatkowym celem jest przedstawienie możliwości wykorzystania rozwijającej się technologii kontenerów. Stworzone rozwiązanie oparto wyłącznie na wolnym oprogramowaniu wykorzystując do tego standardową konfiguracje komputerową. Dzięki temu oprogramowanie użytkować można bez dodatkowych kosztów na przykład w celach edukacyjnych, a użytkownikami przedstawionego rozwiązania mogą być indywidualne osoby.

W pierwszym rozdziale przedstawiony został początkowy rozwój technologii wirtualizacji oraz zaprezentowano poszczególne jej typy. W rozdziale drugim przedstawiono technologie, które wykorzystane zostały w niniejszej pracy. Począwszy od kontenerów Docker na których oparto rozwiązanie, poprzez oprogramowanie Guagga wirtualizujące funkcje sieciowe po narzędzia automatyzujące proces konfiguracji sieci. W kolejnym trzecim rozdziale szczegółowo omówiono stworzone rozwiązanie, jego wymagania systemowe oraz sposób konfiguracji poszczególnych składników. Przedstawiona jest również architektura rozwiązania i przebieg działania skryptu na podstawie konfiguracji przykładowego łącza. Ostatni czwarty rozdział przedstawia metody i wyniki testów przykładowego wirtualnego łącza. Poprzez utworzenie kilku sieci wirtualnych o różnym stopniu złożoności, przetestowane zostały również pewne aspekty ich pracy. Wymienić można między innymi funkcjonowanie wirtualnego przełącznika, routera oraz udostępnianie usług internetowych.

1. Technologia wirtualizacji

Pojęcie wirtualizacji ma już długą historię a początki jej sięgają lat 60 XX w. kiedy to firma IBM uruchomiła pierwszy stabilnie działający system komputerowy pod nazwą System/360 wraz systemem operacyjnym CP-67 korzystającym z wirtualnej pamięci oraz procesora[1]. W ciągu kilku następnych lat rozwinięto projekt do etapu umożliwiającego uruchomienie na pojedynczym systemie komputerowym typu Mainframe wielu niezależnie działających systemów operacyjnych.

Ze względu jednak na ogromne koszty związane z projektowaniem i budową tych systemów, zakres ich funkcjonowania przez wiele lat ograniczony był wyłącznie do instytutów naukowych oraz największych korporacji. Rozpowszechnienie się tej terminologii możliwy był dopiero po stworzeniu programów wirtualizacyjnych w latach 1999 do 2007 dostępnych na komputery osobiste takie jak Workstation firmy VMWare, VirtualBox firmy Oracle, Microsoft VirtualPC, czy Xen firmy Citrix.

Termin wirtualizacja odnosi się do ogólnej idei rozdzielania fizycznych zasobów danego systemu komputerowego jednocześnie dla wielu programów, systemów operacyjnym. Wymagane jest przy tym takie stworzenie abstrakcji za pomocą dodatkowego oprogramowania lub rozwiązań sprzętowych, aby uruchamiane w nich programy odnosiły wrażenie pracy na fizycznym sprzęcie, jak i w całkowitej izolacji od pozostałych programów. Powodem rozwijania tej technologii są przede wszystkim oszczędności finansowe[2] mogące sięgać 50% poprzez lepsze wykorzystanie zasobów serwerów. Zastosowanie wirtualizacji zwiększa również elastyczność rozwiązań umożliwiając szybsze dostosowywanie się do potrzeb rynku[3].

W trakcie rozwoju rozwiązań wirtualizacyjnych opracowano kilka typów wirtualizacji. Różnią się między sobą pod względem wymagań systemowych i wydajnością.

---

[1] E. Amrehn, J. Elliott, „45 (40) Years of Mainframe Virtualization: CP-67/CMS and VM/370 to z/VM", IBM Corporation, Internet 2012, s.12

[2] Oficjalna strona firmy VMWare, https://www.vmware.com/pl/virtualization, (dostęp 19-03-2016)

[3] P. Przybylak, „*WIRTUALNA INFRASTRUKTURA – NOWE PODEJŚCIE DO SYSTEMÓW*" Zeszyt Naukowy 4/2010, Warszawska Wyższa Szkoła Informatyki, Warszawa 2010, s.17

Typami wirtualizacji między innymi mogą być:

- **Emulacja** – Zastosowanie jej daje możliwość uruchomienia innych systemów o innej architekturze niż architektura systemu gospodarza. Powoduje to zwiększenie liczby instrukcji do wykonania przez system gospodarza na rzecz jednej instrukcji emulowanej. Wynika z tego oczywiście spadek wydajności tego typu wirtualizacji.
- **Parawirtualizacja** – System podlegający wirtualizacji musi być o tej samej architekturze co sprzęt na którym jest wirtualizowany. Wadą jej jest konieczność modyfikacji systemu gościa, tak aby miał dostęp do części zasobów udostępnianych przez hypervisora[4] typu 1. Hypervisor tego typu instalowany jest bezpośrednio na warstwie sprzętowej a jego przykładem jest VMWare ESX Server lub Xen Server.
- **Pełna wirtualizacja** – Sposób działania najbardziej zbliżony do emulacji, jednak uzyskano zwiększenie wydajności poprzez bezpośrednie wykonywanie przez procesor części instrukcji.
- **Wirtualizacja na poziomie systemu operacyjnego** – Ze wszystkich typów jest najwydajniejszą wirtualizacją poprzez rezygnację z hypervisora, a zarazem najmniej elastyczną. Rodzaj systemu wirtualnego jest ograniczony do rodzaju architektury sprzętu i systemu gospodarza.

Spośród przedstawionych typów wirtualizacji omówiona zostanie wirtualizacja na poziomie systemu operacyjnego, która jest podstawą dla tworzonego rozwiązania w tym opracowaniu.

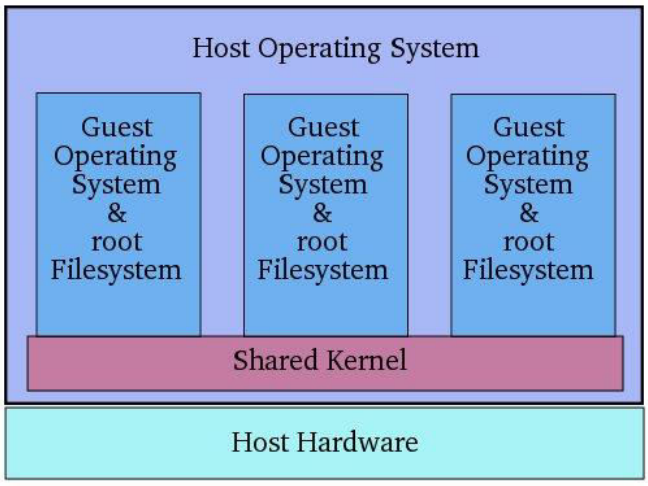
1.1 Wirtualizacja na poziomie OS

Wirtualizacja na poziomie systemu operacyjnego[5] oparta jest na architekturze systemów Linux/Unix. Cechą charakterystyczną tego typu wirtualizacji jest bezpośrednia komunikacja systemów wirtualnych z zasobami sprzętowymi bez udziału dodatkowego oprogramowania hypervisora. Zależności pomiędzy poszczególnymi elementami systemu przedstawia rysunek 1.

[4] Hypervisor (tłum. *Zarządca*) – Oprogramowanie instalowane w systemie komputerowym odpowiedzialne za udostępnianie zasobów sprzętowych dla systemów wirtualnych.

[5] Inna funkcjonująca nazwa to „wirtualizacja na poziomie jądra".

Rysunek 1. Diagram zależności elementów systemu w wirtualizacji OS



Źródło:http://www.virtuatopia.com/index.php/An_Overview_of_Virtualization_Techniques, (dostęp 18-03-2016)

Możliwe jest to ponieważ dla każdego nowego uruchamianego systemu wirtualnego, jądro systemu bazowego wydziela osobne środowisko wirtualne. Podstawą funkcjonowania tych środowisk są dwa elementy systemów Linux/Unix: cgroup oraz namespace.

Funckcjonalność namespace została zaimplementowana w Linuksie[6] od wersji 2.6.26. Ma za zadanie izolowanie poszczególnych procesów między sobą oraz od głównego systemu. Wykonuje to poprzez powielanie poszczególnych struktur zasobów głównego systemu i przydzielanie ich do odpowiednich przestrzeni nazw. Określonych jest sześć obszarów[7], które podlegają wirtualizacji:

- **Mount namespace** – Odpowiada za wydzielenie w systemie plików miejsca na strukturę katalogów i plików wraz z katalogiem „/” (`root`) na potrzeby uruchamianego w niej procesu. Proces uruchomiony wewnątrz tej przestrzeni ma dostęp jedynie to tej struktury i nie ma możliwości ingerencji w pliki systemu głównego. Nowo utworzona struktura może składać się z wybranych plików konfiguracyjnych, bibliotek wyłącznie na potrzeby danego procesu, lub też może być pełnym systemem plików działającym na Kernelu gospodarza.
- **UTS namespace** – Izoluje od głównego systemu dwa identyfikatory odpowiedzialne za nazwę hosta oraz nazwę domeny. Możliwe jest przez to uruchamianie i testowanie procesów zależnych od tych nazw, na przykład skryptów czy serwerów www.

[6] http://lxc.sourceforge.net/old/index.php/about/kernel-namespaces/ (dostęp 27-02-2016)
[7] N. Khare, „*Docker Cookbook*”, Packt Publishing Ltd, Birmingham 06-2015, s.4-6

- **IPC namespace** – W przestrzeni tej komunikaty przesyłane pomiędzy procesami zostają w niej izolowane i nie ma możliwości przejęcia ich przez inny proces.
- **PID namespace** – Izolowanie identyfikatorów procesów. Możliwe jest funkcjonowanie w każdej przestrzeni nazw procesu głównego `init` o identyfikatorze „1".
- **NET namespace** – Izoluje zasoby sieciowe między innymi interfejsy, adresy IP, tablice routingu numery portów. Umożliwia to funkcjonowanie wielu serwerom www w osobnych przestrzeniach nazw na tych samych portach na przykład 80.
- **USER namespace** – W przestrzeni tej zawarte są identyfikatory ID użytkowników i grup. Możliwe jest istnienie w przestrzeniach użytkowników `root` o identyfikatorze „0"

Analizując zakres przestrzeni nazw widać, że procesy uruchomione wewnątrz nich mają wrażenie funkcjonowania na niezależnych systemach. Mimo to wszystkie korzystają w dalszym ciągu z jednego Kernela[8] systemu bazowego.

Drugim elementem wykorzystywanym wtej wirtualizacji jest cgroup[9] (skr. ang. Control Group) odpowiedzialny za ograniczanie dostępu do zasobów systemu. W zakres jego działania wchodzi podsystem CPU odpowiedzialny za kontrolę czasu procesora, który ustala przydzielanie odpowiednich priorytetów do poszczególnych grup. Kernel na tej podstawie utrzymuje podział czasu procesora aby zachować te proporcje. Niezależny od innych grup, przydział czasu możliwy jest przez podanie konkretnych limitów czasowych określających maksymalny czas pracy dla danej grupy w zdefiniowanym okresie czasu. Innym podsystemem jest podsystem MEMORY odpowiedzialny za limitowanie dostępu do pamięci. Uniemożliwia przejęcie całej dostępnej pamięci na przykład przez błędnie działający proces. Limitowane są również operacje wejścia/wyjścia do pamięci masowych w podsystemie BLKIO.

Powyższe elementy są podstawą działania wirtualizacji opartej o kontenery LXC (skr. ang. LinuX Containers) umożliwiającej tworzyć wydzielone zasoby dla procesów lub usług.

[8] Kernel – Program będący głównym składnikiem systemu operacyjnego mający pełną kontrolę nad wszystkimi elementami systemu komputerowego.

[9] N. Khare, „*Docker Cookbook*"…, poz. cyt., s. 7

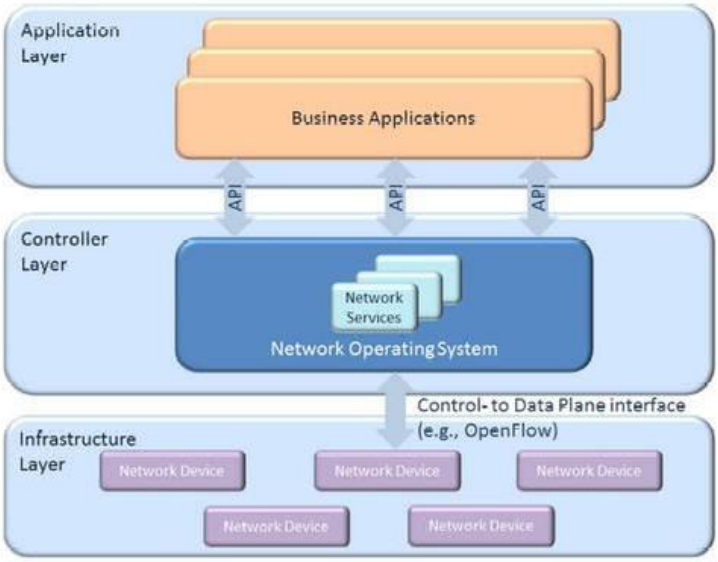
1.2 Wirtualizacja sieci

Tematyka wirtualizacji sieci jest ogólnym pojęciem zawierającym w sobie różne koncepcje i metody wirtualizacji. W trakcie ich rozwoju wyodrębniono dwa główne obszary. W rozdziale tym przedstawię ich charakterystykę, obszary zastosowań oraz różnice pomiędzy nimi.

1.2.1 SDN

Sieci definiowane programowo SDN (ang. Software Definition Network) tworzone pierwotnie w sieciach akademickich, miały na celu usprawnienie w przygotowywaniu środowisk testowych dla nowych rozwiązań. Charakteryzują się oddzieleniem warstwy logicznej, w której ustalana jest struktura sieci od warstwy sprzętowej bezpośrednio przekazującej strumienie danych. Na poniższym rysunku przedstawione są zależności pomiędzy poszczególnymi warstwami sieci.

Rysunek 2. Architektura sieci SDN



Źródło:http://www.networkcomputing.com/networking/7-essentials-software-defined-networking/1587046750

W warstwie infrastruktury funkcjonują przełączniki, routery i inne urządzenia sieciowe przesyłające strumienie danych. Sterowane są one centralnie za pomocą kontrolerów sieci, poprzez tzw. mostki południowe. Z drugiej strony kontrolery komunikują się z aplikacjami przez mostki północne i zdefiniowane interfejsy API.

Przez organizację ONF[10] opracowany został otwarty protokół OpenFlow umożliwiający komunikację kontrolerów z warstwą danych. Stał się on aktualnie standardem, który implementowany jest w coraz większą liczbę produktów sieciowych w technologii SDN. Jednym z zadań tego protokołu jest przekazywanie informacji o pozycjach przepływu do przełączników OpenFlow, które to z kolei pozycje grupowane są w tabele. Przełączniki na podstawie danych z tabel przepływu analizują przesyłane pakiety i podejmują decyzje o wykonaniu odpowiedniej akcji. Najczęściej jest to dalsze przekazanie pakietu do odpowiedniego portu. Innymi akcjami może być odrzucenie pakietu lub przekazanie go do kontrolera celem jego dokładnej analizy. Protokół OpenFlow umożliwia również monitorowanie przepływu danych, tworzenie strumieni zapewniających odpowiedni poziom usług QoS lub przesyłanie informacji do kontrolera o zdarzeniach występujących w sieci.

Dzięki rozdzieleniu funkcjonalności sieci na niezależne od siebie warstwy umożliwiono współdziałanie rozwiązań sprzętowych i programowych różnych producentów pod warunkiem kompatybilności z protokołem OpenFlow. Przykładem mogą być kontrolery sieci. Z pośród dostępnych, poniżej wymienionych jest kilka z nich[11]:

- POX – Kontroler zaimplementowany w języku Phyton. Udostępnia administratorom graficzny interfejs konfiguracyjny. Dostępny na platformę Linux,Windows oraz Mac. Obsługuje protokół OpenFlow w wersji1.0 oraz OpenvSwitch.
- Ruy – Kontroler zaimplementowany w języku Phyton. Obsługuje protokóły zarządzania siecią OpenFlow v1.5, NetConf, OF-Config. Rozpowszechniany jest na bezpłatnej licencji Apache2.0
- Floodlight – Kontroler współpracujący zarówno z przełącznikami sprzętowymi jak i wirtualnymi. Posiada architekturę modułową, zarządzanie prowadzone jest przez interfejs WWW, dane o strukturze sieci przechowywane są w pamięci, bazach SQL lub no-SQL. Jest kontrolerem skalowalnym obsługującym duże sieci. Zaimplementowany w języku Java i rozpowszechniany na licencji Apache2.
- OpenDaylight – Kontroler współpracujący zarówno z protokołem OpenFlow jak i innymi standardami open source. Udostępnia interfejs API dla aplikacji poprzez

[10] Oficjalna strona organizacji ONF (ang. Open Network Fundation) promującej sieci SDN, https://www.opennetworking.org/about/onf-overview

[11] Thomas D.Nadeau & Ken Gray "SDN Software Defined Network", O'Reilly Media, Sebastopol 2013, s.71-114

mostek północny umożliwiając im analizę i konfigurację sieci. Kontroler ten działa w maszynie wirtualnej JAVA[12].

Cisco opracowuje własny standard wirtualnych sieci. Koncepcja działania ich sieci jest nieco odmienna od tej realizowanej w SDN. Architektura określana jako ACI (ang. Application Centric Infrastructure) opiera się również na kontrolerze sieci, w tym wypadku na APIC (ang. Application Policy Infrastructure Controller). Nie centralizuje on całkowicie kwestii konfiguracji sieci, lecz wyznacza jedynie polityki według których urządzenia mają funkcjonować. Powoduje to częściowe przekazanie konfiguracji na podstawie tych polityk do urządzeń sieciowych w warstwie danych. W kontrolerze wykorzystywany jest dedykowany protokół Cisco o nazwie OpFlex pracujący w modelu deklaratywnym. Model ten związany jest wyłącznie z przekazywaniem wymagań do urządzeń sieciowych oczekując, że ustalą odpowiednią konfigurację spełniającą te wymagania[13]. Istotne jest również to, że przy konfiguracji urządzeń sieciowych w warstwie danych architektura ACI wykorzystuje inne protokoły wirtualizacyjne na przykład VXLAN czy IEE802.1Q tworzące logiczne połączenia pomiędzy poszczególnymi hostami[14].

Głównym obszarem zastosowań sieci SDN są centra danych firmy hostingowe. Związane jest to z wymogiem konkurencyjności i skróceniem czasu pomiędzy zamówieniem a uruchomieniem danej usługi dla klienta. Stało się to możliwe dzięki oddzieleniu sieci wirtualnej od fizycznej oraz związanej z tym możliwej szybkiej i zróżnicowanej programowej konfiguracji.

1.2.2 NFV

Drugim obszarem rozwoju sieci jest wirtualizacja funkcji sieciowych NFV (ang. Network Function Virtualization), które do tej pory były wyłącznie w zakresie rozwiązań sprzętowych. Zaliczyć do nich można między innymi routery, ściany ogniowe, równoważenie obciążenia, akceleratory WAN czy serwery DNS. Założeniem NFV jest całkowita izolacja funkcji sieciowych od warstwy sprzętowej sieci. Poprzez

[12] https://www.sdxcentral.com/resources/sdn/sdn-controllers/opendaylight-controller/", (dostęp 26-03-2016)

[13] SDxCentral – firma B2B dostarczająca wiadomości i badań na temat oprogramowania wirtualizacyjnego https://www.sdxcentral.com/resources/cisco/cisco-openflow/ (dostęp 26-03-2016)

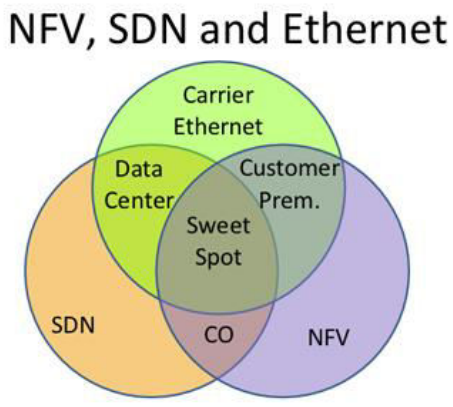
[14] http://www.tomsitpro.com/articles/software-defined-networking-solutions,2-835-2.html (dostęp 26-03-2016)

ich implementacje jako usługi w zwirtualizowanym środowisku, możliwe jest lepsze wykorzystanie standardowych sprzętowych przełączników oraz serwerów zamiast dedykowanych rozwiązań. Poniżej wymienione są niektóre korzyści wynikające ze stosowania NFV:

- Dotychczasowy etap projektowania i testowania dedykowanych układów jest pomijany. Powoduje to skrócenie czasu wdrażania nowych funkcjonalności elastycznie dostosowując się do szybko zmieniających się potrzeb na przykład w usługach chmurowych
- Udostępnianie usług nowym klientom poprzez zastosowanie gotowych szablonów i środowisk wirtualizacyjnych.
- Tworzenie środowiska wysokiej dostępności nie wymaga uruchamiania kilku fizycznych urządzeń. Możliwe jest przez wykorzystanie zasobów serwerowych i migracji systemów wirtualnych zachowanie ciągłości działania usług dla klienta. Przykładem może być rozwiązanie High Availability firmy VMWare.
- Zapewnienie bezpieczeństwa klientom przez udostępnianie indywidualnych zapór ogniowych czy systemów Proxy.

Na rysunku 3 przedstawione są zależności pomiędzy poszczególnymi rodzajami wirtualizacji. Sieci SDN działają na poziomie warstwy 2-4 modelu ISO/OSI udostępniając elastycznie zasoby sieciowe. Natomiast wirtualizacja NFV działa na poziomie warstw 4-7. Widać więc, że oba obszary wzajemnie się uzupełniają.

Rysunek 3. Zależności pomiędzy SDN, NFV i siecią fizyczną



Źródło: http://www.convergedigest.com/2014/03/blueprint-nfv-sdn-and-carrier-ethernet.html (dostęp 26-03-2016)

SDN działa w centrach danych w oparciu o wirtualne przełączniki korzystając z protokołów OpenFlow lub OpFlex tworząc połączenia pomiędzy wirtualnymi hostami. Natomiast NFV działa na styku usługodawców internetowych i klientów.

1.2.3 Sieci nakładkowe

Przedstawione zagadnienia SDN i NFV są stosunkowo nowymi rozwiązaniami wymagającymi całkowicie nowego podejścia do projektowania i zarządzania sieciami. Wymagają również specjalistów w tej dziedzinie, dlatego według badań Quinstreet Enterprise[15] duża część firm w dalszym ciągu będzie korzystać z tradycyjnej metody wirtualizacji. Funkcjonujące określenie „nakładki sieci" związane jest z tym, że nie zmieniają one fizycznej struktury sieci, lecz umożliwiają jednoczesne funkcjonowanie wielu logicznym sieciom na bazie pojedynczej sieci fizycznej. Poniżej przedstawionych jest kilka najbardziej znanych protokołów.

VLAN

Jedną z najstarszych i najbardziej rozpowszechnionych metod są wirtualne sieci VLAN (ang. Virtual Local Area Network). Sieci te konfigurowane są w przełącznikach warstwy 2 modelu ISO/OSI i wykorzystują identyfikator w postaci liczby całkowitej przypisywany do portów przełącznika. 12-bitowa wartość identyfikatora ogranicza ilość sieci VLAN do 4096. Przełącznik blokuje transmisje pomiędzy portami o różnych numerach VLAN. Dzięki temu prostemu mechanizmowi zmniejsza się domenę rozgłoszeniową protokołu IP, uniezależnia się strukturę sieci od struktury organizacyjnej firmy oraz zwiększa się bezpieczeństwo już przez samo zastosowanie routingu pomiędzy fragmentami sieci.

Konfiguracja sieci VLAN może być statyczna, gdzie administrator na stałe przypisuje numer VLAN do portu. Ogranicza to dostęp do danej podsieci z góry określonych lokalizacji. Zwiększa to bezpieczeństwo wrażliwych danych z określonych działów firmy. W dynamicznej metodzie przełącznik automatycznie konfiguruje numer VLAN portu na podstawie adresu MAC podłączonego hosta. Umożliwia zmianę lokalizacji stanowiska pracownika bez wpływu na zakres dostępu do zasobów sieci.

W przypadku sieci złożonej z więcej niż jednego przełącznika konieczne jest zastosowanie połączeń typu trunk (ang. VLAN trunk). Pakiety przekazywane pomiędzy

[15] Portal o tematyce witualizacyjnej, https://virtualizationreview.com/articles/2016/02/03/sdn-vlan-survey.aspx, (dostęp 28-03-2016)

przełącznikami oznaczane są dodatkowo numerem ID danej sieci VLAN. Protokół odpowiedzialny za obsługę VLAN trunk to IEEE802.1Q.

VXLAN

Dotychczasowa liczba sieci VLAN stała się niewystarczająca dla stale powiększających się centrów danych. Z tego względu zostały wprowadzone sieci wirtualne VXLAN[16] (ang. Virtual eXtensible LAN) obsługujące 24-bitowe numery VNI (ang. VXLAN Network Identifier) umożliwiające adresację ponad 16 milionów sieci. W przypadku tych sieci również występuje problem powstawania pętli, który może być rozwiązany poprzez zastosowanie protokołu STP IEEE802.1d. Wiąże się to jednak z blokowaniem części portów w przełącznikach i pogorszeniem efektywności ich wykorzystania. Aby temu przeciwdziałać wprowadzono protokół SPB (ang. Shortest Path Bridging) oraz TRILL. Oba protokoły wykorzystują informacje o stanie łącza znajdując najkrótszą ścieżkę i oznaczają pakiety na krawędziach ścieżki.

MPLS

Protokół MPLS[17] (ang Multi Protocol Label Switching) wykorzystywany przez routery do przekazywania pakietów opierając się na ich etykietowaniu. Przyśpiesza przesyłanie pakietów ponieważ analizowane są krótkie etykiety zamiast stosowania trasowania na podstawie adresów sieci i analizy rozbudowanych tablic routingu. Nagłówki protokołu dołączane są pomiędzy warstwę 2 a 3 dzięki czemu możliwe jest przekazywanie dowolnego protokołu na zasadzie tunelowania przez obszar działania routerów MPLS. W obszarze takim działają dwa rodzaje routerów. Router LER (ang. Label Edge Router) działa na brzegu obszaru łącząc go z sieciami o innych protokołach. W tym routerze pakiety są klasyfikowane np. na podstawie adresów docelowych IP. Wewnątrz obszaru działają routery LSR (ang. Label Switching Path) przekazujące pakiety wyłącznie na podstawie etykiet. Rozgłaszane są one w obrębie obszaru przez protokół LDP (ang. Label Distribution Protocol). W przypadku sprzętowych routerów decyzja o wyborze trasy może zostać podjęta już na etapie

[16] M.Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, C. Wright, „ VXLAN: A Framework for overlaying virtualized layer 2 network over layer 3 networks", RFC7348, ISSN:2070-1721, Sierpień 2014

[17] E.Rosen, R. Callon, „Multiprotocol Label Switching Architekcture", RFC3031 Cisco System Inc. Styczeń 2001

odczytu etykiety bez potrzeby analizowania nagłówka zagnieżdżonych protokołów. Zmniejsza to opóźnienia w przekazywaniu pakietów.

Mosty linuksowe

Mosty programowe w systemie Linux (ang. bridge) pod względem funkcjonalności odpowiadają rzeczywistym przełącznikom pracującym na warstwie 2. Służą do łączenia ze sobą różnych typów interfejsów sieciowych np. fizycznych Ethernet, wirtualnych VLAN, programowych TAP. Ograniczają domenę rozgłoszeniową korzystając z tablicy adresów źródłowych MAC oraz mają możliwość wykrywania i usuwania zapętleń pakietów poprzez zastosowanie protokołu drzewa rozpinającego STP (ang. Spanning Tree Protocol). Dla protokołów wyższych warstw np. IP są niewidoczne. Wykonywanie testów trasy pakietów poleceniem traceroute nie wykaże ich jako dodatkowe punkty przeskoków.

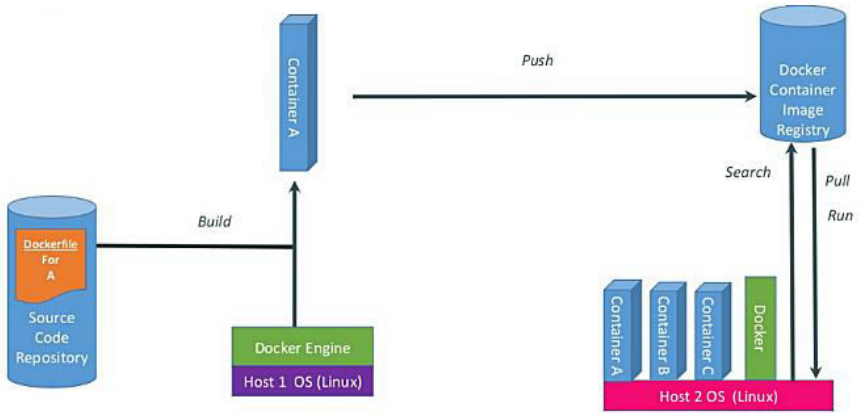
2. Zastosowane technologie

W rozdziale tym przedstawione zostaną technologie wykorzystane w niniejszej pracy. Pierwsze omówione zostaną tytułowe kontenery będące podstawą omawianego rozwiązania. Kolejne będą zawierać się w tematyce routingu, a na koniec omówione zostanie pojęcie kształtowania pasma.

2.1 Kontenery Docker

Docker jest oprogramowaniem opierającym swoje działanie na funkcjonalności kontenerów LXC i jest rozpowszechniane na licencji *License Apache 2.0* [18]. Udostępnia użytkownikom rozszerzonego i ujednoliconego interfejsu do zarządzania kontenerami. Ułatwia ich tworzenie, instalowanie w nich usług i aplikacji, zapisywanie w repozytorium i wysyłanie do zewnętrznego rejestru. Dzięki temu możliwe jest wyszukanie tych kontenerów przez inne osoby i uruchomienie ich kopii na własnych systemach. Sytuacja ta przedstawiona jest na poniższym rysunku.

Rysunek 4. Zależności pomiędzy elementami systemu Docker



Źródło: Ł. Piątkowski, „*Kontener aplikacyjny nie tylko dla programistów*", Instytut Informatyki Politechnika Poznańska – Koło Naukowe SKiSR, 26-11-2014, s. 20

Oprogramowanie Docker jest aplikacją typu klient-serwer. Wszystkie polecenia do zrealizowania na kontenerach użytkownik wprowadza przez interfejs klienta, który po zweryfikowaniu przekazuje je do silnika Docker. Część serwerowa odpowiedzialna

[18] https://docs.docker.com/v1.5/#about-this-guide, (Dostęp 02-2016)

jest za tworzenie i obsługę kontenerów. Może pracować na innym systemie komputerowym a komunikować się z klientem poprzez gniazda[19]. Jest to podstawą do tworzenia klastrów Docker zarządzanych przez jednego zarządcę.

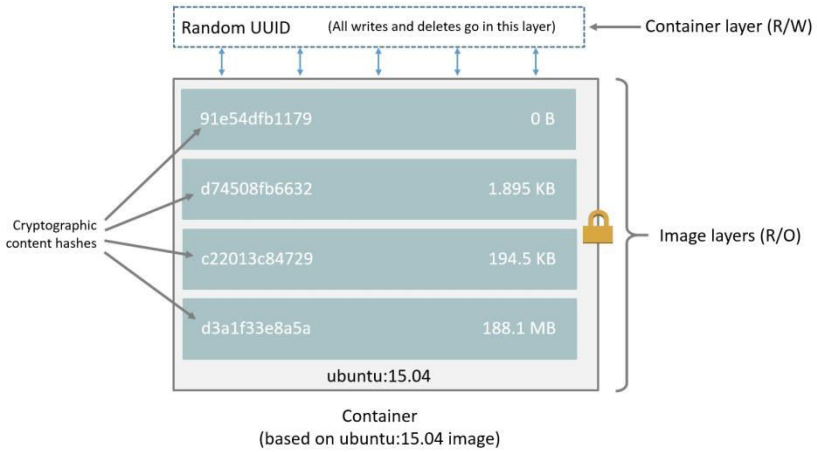
Dodatkowymi elementami działającymi w strukturze Dockera są rejestry umożliwiające przechowywanie informacji o obrazach kontenerów. Wraz z instalacją Dockera tworzony jest w systemie rejestr lokalny będący do dyspozycji danej jego instancji. Przechowywane w nim obrazy identyfikowane są poprzez 64-bajtowe indywidualne numery ID wybierane losowo lub na podstawie skrótu kryptograficznego z obrazu. Możliwe jest również określanie ich nazw składających się z nazwy repozytorium, nazwy własnej kontenera i znacznika TAG określającego numer wersji. Drugi rejestr jest rejestrem zdalnym funkcjonującym na serwerach firmy Docker przechowując informacje o wszystkich repozytoriach. Dostępne są w nich oficjalne wersje oprogramowania różnych producentów oraz obrazy tworzone przez użytkowników Docker.

Podstawowym elementem na którym wykonywane są działania w kontenerze to obraz. Składa się z logicznych warstw, które dokładane są przy zapisie kontenera. Każda nowo dołożona warstwa zawiera w sobie zmiany dokonane od ostatniego uruchomienia kontenera. Jest kilka zalet takiego rozwiązania. Najważniejsza to olbrzymia oszczędność miejsca w przestrzeni dyskowej. Przykładem może być potrzeba uruchomienia kilku identycznych systemów z różnymi usługami. Zakładając, że w repozytorium znajduje się obraz systemu np. CentOS oznaczony indywidualnym numerem ID, uruchomione zostają trzy kontenery. Po wprowadzeniu w nich zmian, każdy z nich zostaje zapisany powtórnie do repozytorium otrzymując nowy numer ID. Nowe obrazy składają się wówczas z dwóch warstw logicznych, lecz plik fizycznie zawiera tylko informacje o zmianach dokonanych w kontenerze. Dzięki temu możliwe jest przechowywanie w repozytoriach dużej ilości obrazów.

Na rysunku 5 przedstawiona jest specyfika obrazów pośrednich, które są zawsze plikami tylko do odczytu a zmiany wprowadzane są wyłącznie w ostatniej warstwie. Zaletą tego jest możliwość odrzucenia w przypadku błędów wszystkich wprowadzonych zmian poprzez jego usunięcie bez jego zapisu.

[19] N. Khare, „*Docker Cookbook*”…, poz. cyt., s. 12

Rysunek 5. Organizacja warstw w działającym kontenerze



Źródło: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/

Użytkowanie Dockera wymaga uprawnień administratora, aby to ominąć dodajemy danego użytkownika do grupy „docker". Mając dostęp do programu, jedną z pierwszych czynności będzie wyszukanie i pobranie oprogramowania z repozytorium poprzez wykonanie poleceń.

```
# docker search fedora
# docker pull Fedora
```

Na poniższym raporcie widoczna jest informacja o pobraniu obrazu „fedora:latest" oraz obecność gotowego obrazu pośredniego.

Rysunek 6. Wyszukiwanie i pobieranie obrazu z repozytorium

```
[root@localhost ~]# docker search fedora
NAME                      DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
fedora                    Official Docker builds of Fedora                325       [OK]

[root@localhost ~]# docker pull fedora
Using default tag: latest
latest: Pulling from library/fedora
a3ed95caeb02: Already exists
236608c7b546: Pull complete
Digest: sha256:1fa98be10c550ffabde65246ed2df16be28dc896d6e370dab56b98460bd27823
Status: Downloaded newer image for fedora:latest
```

Można zweryfikować jego i wyświetlić zawartość repozytorium poleceniem

```
# docker images
```

Na podstawie tego obrazu uruchomić można kontener w tym wypadku o nazwie "zeus" z obecnym wewnątrz systemem Fedora. Opcja „-i" oznacza pracę kontenera w trybie interaktywnym, natomiast „-t" uruchomienie wirtualnego terminala. Po uruchomieniu kontenera włączana jest powłoka „BASH" mająca dostęp wyłącznie do wewnętrznej struktury plików kontenera.

```
# docker –t –i run ––name zeus  fedora  /bin/bash
```

Zapisanie działających kontenerów po wprowadzeniu w nich zmian polega na wykonaniu polecenia z podaną jego nazwą lub numerem ID.

```
# docker commit [name | ID]
```

2.2 Router Quagga

W sieci Internet, w przeciwieństwie do sieci lokalnych, nie ma możliwości stosowania statycznego routingu. Konieczne jest użycie protokołów routingu automatyzujących proces wyszukiwania odpowiedniej trasy. Ze względu na duże wymagania wydajnościowe, w sieciach tych stosowane są głównie rozwiązania sprzętowe. Pewnym ich uzupełnieniem, możliwym do zastosowania w mniej wymagających środowiskach lub do celów szkoleniowych są programowe implementacje routerów. Jednym z przykładów jest oprogramowanie Quagga rozpowszechniane na licencji GPL.

Quagga wyróżnia się wśród tego typu rozwiązań zaawansowaną obsługą routingu w sieciach TCP/IP obejmującą protokoły RIP, OSPF, IS-IS, BGP wraz z protokołem OSPF w wersji v6. Architekturą oprogramowania zorganizowaną w taki sposób aby można było elastycznie dostosowywać ją do aktualnych wymagań. W związku z tym każdy protokół routingu obsługiwany jest przez osobne usługi, które uruchamiane są w miarę potrzeby. Konfiguracja poszczególnych usług możliwa jest po zalogowaniu się do nich poprzez telnet na odpowiednio zdefiniowane numery portów. Za routing statyczny odpowiedzialna jest usługa „zebra" pracująca na porcie 2601, natomiast routing OSPF obsługuje usługa „ospfd" na porcie 2604. Z uwagi na różne zakresy parametrów konfiguracyjnych poszczególnych usług, zarządzanie w odrębny sposób poszczególnymi usługami Quagga może być kłopotliwe. Ułatwieniem pracy jest zatem wprowadzenie ujednoliconego interfejsu dostępnego poprzez polecenie *vtysh*. Analogiczna sytuacja jest w przypadku plików konfiguracyjnych. Domyślnie pakiet Quagga skonfigurowany jest na użytkowanie osobnych plików konfiguracyjnych dla

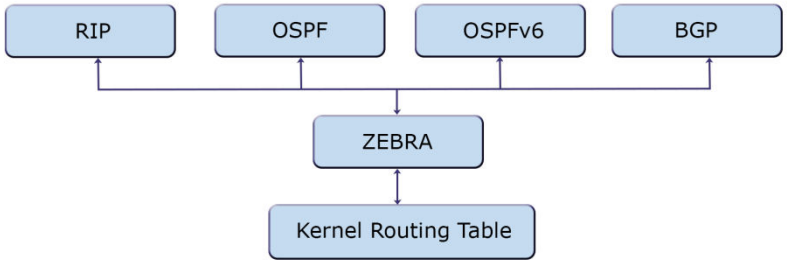
poszczególnych protokołów. Można to zmienić i używać pojedynczego pliku wydając poniższe polecenie w terminalu konfiguracyjnym Quagga[20]:

```
Router(config)# service integrated-vtysh-config
```

Użytkowanie ułatwia również zastosowanie przez twórców oprogramowania składni poleceń oraz struktury menu na wzór rozwiązań stosowanych przez firmę Cisco.

Architektura Quagga przedstawiona jest na rysunku 7. Widoczne są na nim poszczególne usługi protokołów komunikujące się z nadrzędną usługą zebra, która jako jedyna ma możliwość komunikować się z systemem hosta. Poprzez nią również wprowadzane są zmiany w tablicy routingu na podstawie danych otrzymywanych od pozostałych usług. Quagga zatem działa w zakresie komunikacji między routerami, ustala najkorzystniejsze trasy routingu według odpowiednich algorytmów. Właściwe przetwarzanie strumienia danych pozostaje nie zmienione i wykonywane jest na poziomie hosta korzystającego z własnych tablic routingu dostosowywanych wyłącznie przez usługę „zebra".

Rysunek 7. Architektura oprogramowania Quagga



Skrypt omawiany w niniejszej pracy tworzący sieci wirtualne wykorzystuje protokół OSPF. Jego zastosowanie podyktowane było koniecznością ustalania tras nie na podstawie wektora odległości jak to ma miejsce w protokole RIP, lecz uwzględniając parametry poszczególnych łączy. W protokole tym najkorzystniejsza trasa ustalana jest według algorytmu SPF (ang. Shortest Path First) zwanego inaczej algorytmem Dijkstry. Opiera się on na tworzeniu drzewa rozpinającego w grafie będącego odpowiednikiem sieci komputerowej, w skutek czego blokowana jest możliwość powstawania pętli routingu.

Cechą charakterystyczną protokołu OSPF jest możliwość definiowania obszarów w obrębie których routery posiadają pełny obraz sieci wraz z parametrami łączy,

[20] http://www.nongnu.org/quagga/docs/quagga.html#VTY-shell-integrated-configuration (Dostęp 20-12-2015)

a protokół OSPF działa jako protokół stanu łącza. Obszary numerowane są od 1 do 65535 i wszystkie łączone są z obszarem 0. Jest to wyróżniony obszar działający jako sieć szkieletowa poprzez który wymieniane są informacje pomiędzy poszczególnymi obszarami. Aby komunikacja pomiędzy routerami przebiegała prawidłowo zdefiniowano różne typy komunikatów LSA (ang. Link State Area):

- **LSA typ 1 (Router LSA)**: Komunikaty wysyłane przez każdy router informujące o sobie i swoich interfejsach rozchodzące się w zakresie danego obszaru. Umożliwiają poznanie routerom pełnej topologii obszaru i obliczenie w nim najkrótszych ścieżek do pozostałych routerów.
- **LSA typ 2 (Network LSA)**: W każdym obszarze zostaje wytypowany router desygnowany DR (ang. Desygnated Router), który jest odpowiedzialny za zbieranie komunikatów typu 1 i rozsyłanie jednego zbiorczego komunikatu typu 2[21]. Zmniejsza to obciążenie sieci od pakietów rozgłoszeniowych. Wybierany jest również zapasowy router BDR (ang. Backup Designated Router). Oba typowane są na podstawie ich priorytetów, a jeżeli są one równe to na podstawie wartości IP interfejsów.
- **LSA typ 3 (Summary LSA)**: Komunikaty te wysyłane są przez routery brzegowe ABR. Zawierają w sobie informacje ze wszystkich komunikatów LSA typu 1 i 2 z danego obszaru i wysyłane są do pozostałych obszarów. Zmniejsza to znacznie ilość pakietów LSA transmitowanych pomiędzy obszarami.
- **LSA typ 4 (ASBR Summary LSA)**: Komunikaty wysyłane z routera połączonego z zewnętrznym routerem o innym protokole np. RIP. Rozsyłane są do wszystkich obszarów z informacją o lokalizacji routera brzegowego dla całego systemu autonomicznego.
- **LSA typ 5 (External LSA)**: Komunikat wysyłany przez router ASBR z informacją o domyślnej trasie do zewnętrznych systemów.

2.3 Pipework

Docker domyślnie przydziela adresy interfejsów w swoich kontenerach ze zdefiniowanego wcześniej zakresu sieci. Możliwe jest ustalanie własnych adresów interfejsów wewnątrz kontenerów, jednak łączone są one do wirtualnego

[21] http://tech-itcore.pl/cisco-labs/ospf/ospf-wybor-dr-designated-router-oraz-bdr-backup-designated-router/ (Dostęp 10-02-2016)

interfejsu „docker0”[22] Ogranicza to możliwość konfigurowania w nich wielu interfejsów o dowolnych adresach. W artykule[23] opisano krok po kroku sposób utworzenia połączenia pomiędzy dwoma kontenerami, jednak jego ręczne dostosowywanie do różnych warunków konfiguracyjnych systemu hosta byłoby uciążliwe. Poza tym rozwiązanie łączenia kontenerów wyłącznie między sobą jest mało elastyczne. Jerome Petazzoni stworzył skrypt Pipework[24], który w pełni automatyzuje proces konfiguracji połączeń. Skrypt ten ma funkcjonalność umożliwiającą współpracę z mostami linuksowymi, przełącznikami OPVSwitch a także fizycznymi interfejsami systemu hosta. Dzięki temu mamy możliwość w łatwy i dowolny sposób łączenia kontenerów między sobą.

Wywołanie skryptu przedstawione poniżej umożliwia połączenie kontenera <guest> o wewnętrznym interfejsie <containerinterface> z interfejsem hosta <hostinterface>. Do wewnętrznego interfejsu zostanie przypisany adres IP <ipaddr> z dowolnie zdefiniowaną maską <subnet> z zakresu $< 0, 32 >$ i z opcją ustawienia domyślnego routingu <@defaultgateway>. Mamy również możliwość zdefiniowania własnego adresu MAC i przypisania danego połączenia do odpowiedniego numeru vlan. Funkcjonalność taka w znacznym stopniu przybliża sposób konfiguracji do sprzętowych rozwiązań przełączników.

```
# Pipework <hostinterface> [-i containerinterface] [-l
       localinterfacename] <guest> <ipaddr>/<subnet>[@default_gateway]
       [macaddr][@vlan]
```

Poniżej przedstawiona jest przykładowa konfiguracja dwóch kontenerów z uruchomionymi terminalami bez zdefiniowanej sieci. Kontenery te uruchamiane są na podstawie obrazów <host:v1> wykorzystywanych w niniejszej pracy

```
# docker run -ti --name host0 --net none chefronpc/host:v1 /bin/bash
# docker run -ti --name host1 --net none chefronpc/host:v1 /bin/bash
```

Kontenery te można w prosty sposób połączyć ze sobą poprzez most linuksowy <br1> po wykonaniu poniższych poleceń:

---

[22] P. Raj, J.S.Challadhurai, V. Singh, *Learning Docker*, Packt Publishing, Birmingham 2015, s.90,s 102.

[23] Oficjalna strona oprogramowania Docker, https://docs.docker.com/v1.5/articles/networking/ (dostęp 11.09.2015)

[24] Skrypt udostępniany na licencji Apache License 2.0 umożliwiającej wykorzystanie go w niniejszym opracowaniu, https://github.com/jpetazzo/pipework (dostęp 02-11-2015)

```
# pipework br1 -i eth1 host0 10.0.0.10/24
# pipework br1 -i eth1 host1 10.0.0.11/24
```

Pomijając etap działania skryptu w którym następuje weryfikacja danych wejściowych oraz wymagań systemu, proces łączenia jednego z powyższych kontenerów do mostu linuksowego przebiega w następujący sposób[25] [26].

Na wstępie tworzony jest link symboliczny do przestrzeni nazw umożliwiający wykonywanie w niej poleceń w granicach uprawnień – w tym wypadku konfiguracji sieciowych.

```
# ln -s /proc/29244/ns/net /var/run/netns/29244
```

Wykonane jest łączenie dwóch wirtualnych interfejsów hosta i kontenera.

```
# ip link add name veth1pl29244 mtu 1500 type veth peer name
       veth10pg29244 mtu 1500
```

Przyłączenie wirtualnego interfejsu do mostu jako jednego z jego portów i uruchomienie interfejsu.

```
# ip link set veth10pl29244 master brlink6
# ip link set veth10pl29244 up
```

Przypisanie interfejsu do przestrzeni nazw

```
# ip link set veth10pg29244 netns 29244
```

Na koniec wykonana jest konfiguracja sieciowa interfejsu wewnątrz przestrzeni nazw do której należy kontener z poziomu przestrzeni root hosta

```
# ip netns exec 29244 ip link set veth10pg29244 name eth10
# ip netns exec 29244 ip addr add 50.0.0.1/24 brd 50.0.0.255 dev eth10
# ip netns exec 29244 ip link set eth10 up
```

[25] https://docs.docker.com/v1.5/articles/networking/ (Dostęp 02-2016)
[26] http://baturin.org/docs/iproute2/ (Dostęp 10-2015)

2.4 Traffic Control

Narzędzie do kształtowania ruchu TC (ang. Traffic Control) w sieciach Ethernet, jest częścią bardzo rozbudowanego pakietu iproute2. Pakiet ten do Linuksa wprowadzony został od wersji jądra 2.2 zastępując poprzednie narzędzie net-tools zawierające m. in. polecenia route, ifconfig, arp, netstat.

Iproute2 zaprojektowany i napisany od nowa całkowicie zmienia sposób podejścia do obsługi sieci w systemie Linux. Zawiera polecenia ip, tc oraz ss zarządzającymi i konfigurującymi w sposób zintegrowany zagadnieniami takimi jak:

- Interfejsy sieciowe
- Tablice routingu, ARP
- Protokoły IPv4 i IPv6
- Tunelowanie protokołów
- Kształtowanie ruchu QoS
- Informacje o stanie sieci

Polecenie tc umożliwia konfiguracje klasyfikowania ruchu za pomocą filtra u32 na podstawie informacji zawartych w nagłówkach pakietów, filtra route według danych zawartych w tablicy routingu lub według znaczników jakimi są oznaczane pakiety przy przejściu przez reguły filtrujące iptables. Konfiguruje również klasy oraz kolejki mogące tworzyć hierarchiczne struktury do których kierowane są sklasyfikowane wcześniej pakiety.

Domyślnie w systemie dla każdego interfejsu tworzona jest jednak główna kolejka pfifo_fast. Wysyła pakiety przez interfejs zgodnie z kolejnością ich przyjmowania, lecz z możliwością uwzględnienia priorytetów ToS poszczególnych pakietów. Nie umożliwia ona jednak kształtowania ruchu na interfejsie. Funkcjonalność taką udostępniają bardziej rozbudowane kolejki, które można podzielić na dwie grupy:

Bezklasowe:

- **TBF** – (ang. *Token Bucket Filter*) Kolejka określana potocznie jako filtr z wiadrem żetonów. Stosowana jest przy ograniczaniu przepustowości do ściśle określonego poziomu.
- **SFQ** – (ang. *Stochastic Fairness Queueing*) Kolejka tworząca wiele kolejek fifo do których losowo kierowane są potoki odpowiadające poszczególnym

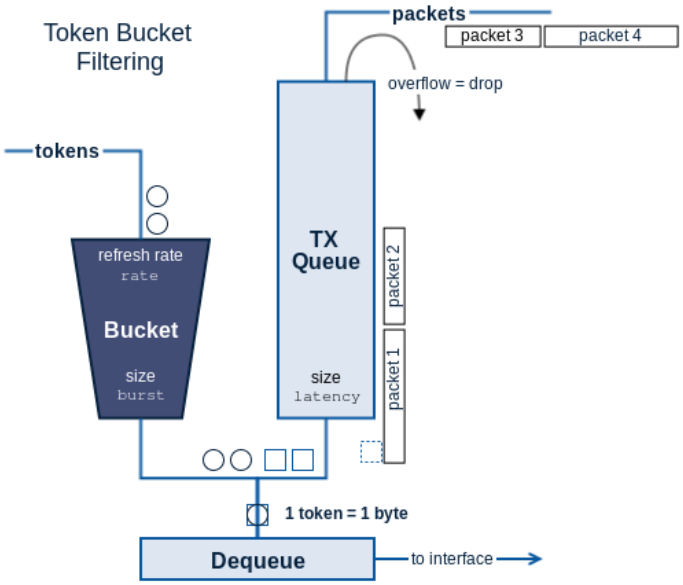
sesjom TCP lub strumieniom UDP. Pakiety wysyłane są kolejno rotacyjnie z każdej kolejki fifo dzieląc sprawiedliwie pasmo.

Klasowe:

- **CBQ** – (ang. *Class Based Queueing*) Kolejka oparta o klasy korzystająca ze skomplikowanego algorytmu analizującego czas bezczynności łącza. Jest najbardziej wymagającą obliczeniowo kolejką a zarazem najmniej dokładną i z tego względu aktualnie nie stosowaną.
- **HTB** – (ang. Hierarchical Token Backet) Kolejka tworząca w swojej strukturze wiele równoległych łączy działających w oparciu o kolejkę TBF. Dla każdego z nich ustala się parametry graniczne a algorytm dostosowuje je w tych granicach do aktualnej zajętości łącza. Jest najbardziej popularną kolejką ze względu na dokładność, prostotę konfiguracji i stosunkowo nieduże wymagania obliczeniowe.

Do symulacji jednocześnie wielu łączy LAN/WAN w jednym hoście, wymagana jest kolejka zapewniająca dużą i powtarzalną dokładność, a zarazem mało wymagającą obliczeniowo. Jedynym wyborem do takiego zastosowania jest kolejka TBF. Jej działanie przedstawione na rysunku 8 i polega na utworzeniu tzw. wiadra z określoną ilością żetonów definiowaną przez parametr *burst*.

Rysunek 8. Działanie kolejki TBF (ang. Tocken Bucket Filtering)



Źródło:https://www.excentis.com/sites/excentis/files/styles/full-width/public/linux_tc_tbf.png?itok=2F8lC_7h, (Dostęp 14-03-2016)

Każdy transmitowany pakiet pobiera z wiadra ilość żetonów odpowiadającą długości pakietu, a po jego wysłaniu żetony zwracane są do wiadra. W przypadku obecności w wiadrze odpowiedniej ilości żetonów pakiety wysyłane są bez żadnego opóźnienia. Jednak w przypadku ich braku pakiet zostaje zatrzymany w kolejce do czasu pojawienia się żetonów. Aby nie stracić na interaktywności transmisji i zwiększenia opóźnień stosuje się parametr *lantency* ograniczający czas przetrzymywania pakietu w kolejce albo parametr *limit* określający wielkość bufora kolejki. Po przekroczeniu tych parametrów pakiety zostają odrzucone.

W celu wiarygodnego emulowania połączeń sieciowych, zaimplementowano w jądrze Linuksa od wersji 2.6 moduł NetEm. W systemie funkcjonuje jako dodatkowa kolejka którą można przyłączać do interfejsów, a konfiguracje wykonywać za pomocą polecenia tc z omówionymi poniżej opcjami.

Opóźnienie (ang. D*elay*) – Wartość podawana w milisekundach wprowadza stałe opóźnienie w przekazywaniu pakietów. Wiarygodniejsze emulowanie łączy internetowych, możliwe jest przez wprowadzenie zmienności parametru po zdefiniowaniu jego zakresu $\pm t[ms]$. Dodatkowo zmiany te można skorelować z czasem opóźnienia z poprzedniego pakietu podając w procentach stopień zależności. Przykładowo w poniższym poleceniu wprowadzamy 20ms opóźnienia zmieniającego się losowo w zakresie $\pm 4ms$ i uzależnionego w 20% od czasu opóźnienia poprzedniego pakietu:

```
# tc qdisc add dev eth0 root netem delay 20ms 4ms 20%
```

Utrata pakietów (ang. *Loss*) – Parametr określający procentowo ilość usuniętych pakietów z transmisji. Również w tym mamy możliwość skorelowania tej wartości z poprzednim pakietem

Powielanie (ang. *Duplicate*) – Parametr konfigurowany identycznie jak poprzedni. Powtarzanie transmisji pakietów powoduje zmniejszenie pasma łącza dla właściwych danych. Sytuacja mogąca wystąpić np. w sieciach WiFi podczas współpracy kilku punktów dostępowych ze sobą. Poniżej pokazane jest polecenie

konfigurujące filtr powielający pakiety w 2% z korelacją na poziomie 25% z jednoczesną utratą pakietów na poziomie 1%

```
# tc qdisc add dev eth0 root netem duplicate 2% 25% loss 1%
```

3. Narzędzie QoSLink

W tym rozdziale przedstawiony zostanie projekt rozwiązania opartego na skrypcie określanego dalej jako QoSLink, umożliwiającego symulację sieci LAN/WAN. Elementami składowymi sieci tworzonymi przez skrypt są wirtualne przełączniki, routery oraz łącza pomiędzy nimi. Dodatkowo dla każdego łącza niezależnie, możliwe jest ustalanie parametrów transmisji danych uwzględniających wymaganą przepustowość, opóźnienie, utratę oraz duplikowanie pakietów. Działanie skryptu oparte jest na wykorzystaniu technologii kontenerów Docker, routingu programowego Quagga, kontroli ruchu pakietów TC, modułu NetEM symulującego parametry sieci WAN oraz skryptu pipework automatyzującego proces łączenia kontenerów.

3.1 Założenia systemu

Zaprojektowane rozwiązanie QosLink ma w założeniu wiernie symulować zadane parametry łącza, aby umożliwić wiarygodną ocenę testowanego oprogramowania. W minimalnym stopniu ingerować w konfigurację sieciową systemu bazowego przenosząc całą konfigurację kontroli ruchu pakietów do wnętrza kontenerów Docker.

Umożliwiać konfigurację sieci wirtualnej z poziomu terminala.

Opierać się wyłącznie na działaniu skryptu.

Posiadać uproszczoną metodę wprowadzania parametrów sieci.

3.2 Instalacja i konfiguracja

W bieżącym rozdziale przedstawione zostaną działania konieczne do uruchomienia skryptu Qoslink oraz powiązanych z nim narzędzi. Ze względu na projektowanie rozwiązania całkowicie od podstaw nie istnieją gotowe wypracowane procedury automatyzujące ten proces. Przygotowanie systemu wykonywane jest zatem ręcznie a mając na uwadze założenia systemu, działania są uproszczone i zajmują niewiele czasu.

3.2.1 Wymagania wstępne

Skrypt wykorzystuje do komunikacji mosty linuksowe, których oprogramowanie do zarządzania instalujemy za pomocą polecenia

```
# yum install bridge-utils
```

Fukcjonalność skryptu dotycząca udostępniania połączenia internetowego do wnętrza sieci oraz uruchomionych usług na zewnątrz, wymaga również aktywacji usługi iptables. Związane jest to z bezpośrednim działaniem skryptu na adresach i portach sieciowych. W nowych systemach Linux - w przypadku systemu Fedora od wersji 19 – domyślnie instalowaną usługą jest firewalld, która nastawiona jest na obsługę predefiniowanych stref oraz usług[27]. Przystosowanie systemu wymaga wyłączenia usługi firewalld i załączenia iptables poprzez wykonanie poniższych poleceń[28]:

```
#systemctl disable firewalld
#systemctl stop firewalld
#yum install iptables-services
#systemctl enable iptables
#systemctl start iptables
```

3.2.2 Docker

Skrypt Qoslink do uruchomienia wymaga działającego oprogramowania Docker. Wymusza to użycie systemów zgodnych ze specyfikacja zawartą w dokumentacji Docker[29]. Testy przeprowadzane były na systemach CentOS 7.0 oraz Fedora 22. W przypadku systemów Linux minimalna wersja jądra Kernel obsługującego kontenery to wersja 3.10.

Oba testowane systemy umożliwiają instalacje pakietu Docker za pomocą gotowych repozytoriów po uprzedniej aktualizacji systemów.

---

[27] Dokumentacja on-line systemu Fedora, https://docs.fedoraproject.org/en-US/Fedora/19/html/Security_Guide/sec-Understanding_Network_Zones.html, (dostęp 12-05-2016)

[28] https://docs.fedoraproject.org/en-US/Fedora/19/html/Security_Guide/sec-Disabling_firewalld.html, poz. cyt. (dostęp 12-05-2016)

[29] Oficjalna strona dokumentacji Docker, https://docs.docker.com/engine/installation/, (dostęp 23-03-2016)

```
# yum upgrade
# yum install dokcer-io
```

Docker'a uruchamiamy za pomocą polecenia

```
# systemctl start docker.service
```

Automatyczne uruchamianie usługi Docker możliwe jest po jej włączeniu w menadżerze „systemd"

```
# systemctl enable docker.service
```

3.2.3 Pipework

Skrypt wymagany do tworzenia połączeń kontenerów pobieramy z repozytorium GitHub[30]. Uruchomienie skryptu pipework bez podawania ścieżki możliwy jest po przeniesieniu go do poniższego foldera i nadaniu mu uprawnień wykonywania.

```
# wget https://github.com/jpetazzo/pipework/blob/master/pipework
# mv pipework /usr/local/sbin/pipework
# chmod +x /usr/local/sbin/pipework
```

3.2.4 Qemu/KVM

Opcjonalną częścią przystosowywania systemu jest Uruchomienie maszyny wirtualnej KVM wymaga zainstalowania środowiska wirtualizacyjnego poprzez instalacje poniższych pakietów - w tym przykładzie na bazie systemu rodziny RedHat:

```
# dnf install libvirt-daemon-vbox  RemoteBox qemu-system-x86
```

Maszyny wirtualne można utworzyć na dwa sposoby. Po pierwsze poprzez polecenia wydawane z poziomu terminala, podając wielkość obrazu systemu, ilość przyznanej pamięci RAM oraz lokalizacje plików obrazu dysku i instalatora a następnie zaimportować do virt-managera.

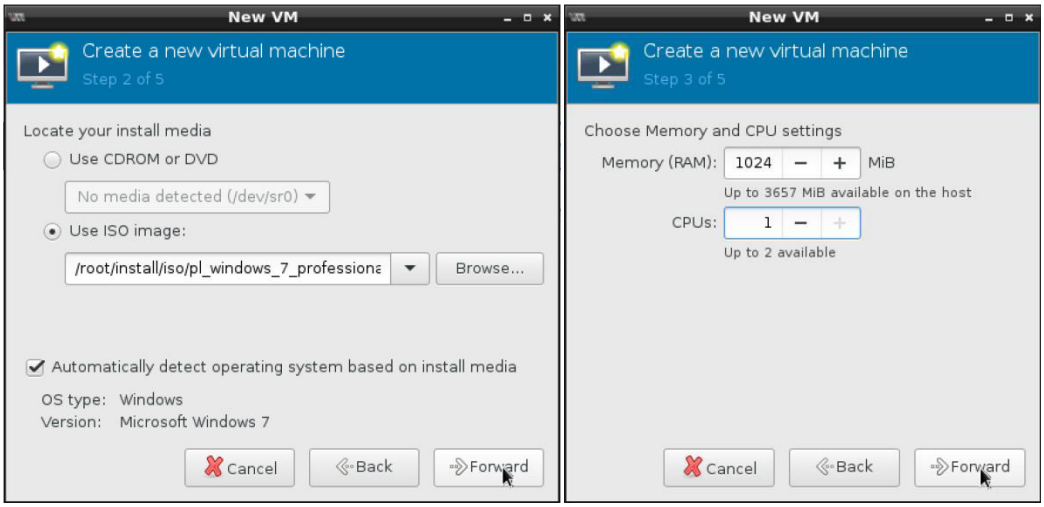
```
# qemu-img create -f qcow2 /mnt/iso/win7/win7.img 10G
```

[30] Repozytorium Docker - Jerome Petazzoni, https://github.com/jpetazzo/pipework (dostęp 02-11-2015)

```
# qemu-system-x86_64 -m 728 -hda /mnt/iso/win7/win7.img -cdrom
/root/install/iso/pl_windows_7_professional_with_sp1_x86_dvd_u_6
77091.iso
```

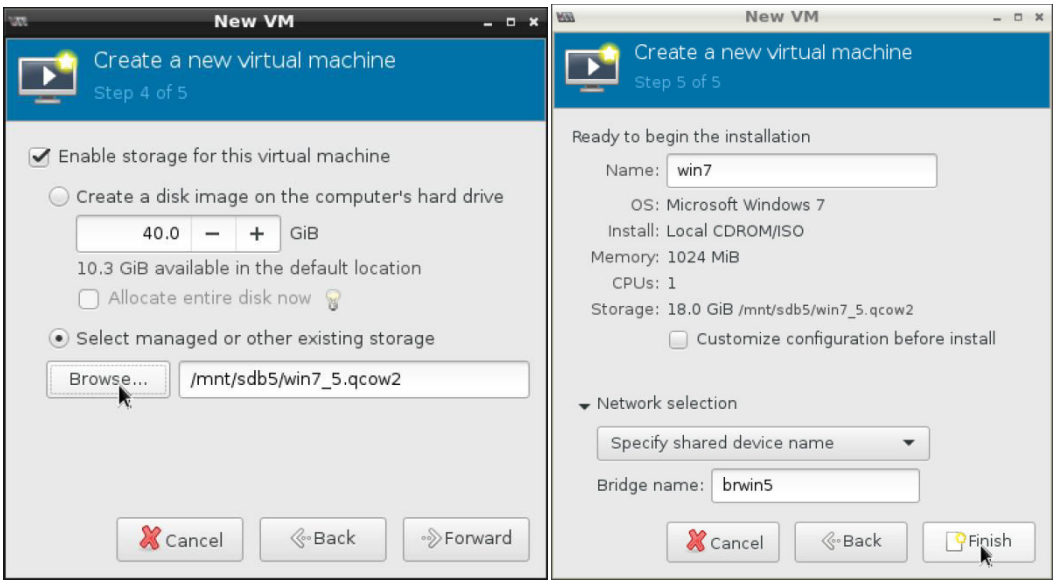
Można również uruchomienie maszyny w całości przeprowadzić z poziomu virt-managera. Podając kolejno źródło obrazu instalatora systemu jaki zamierzamy zainstalować – w tym przypadku Windows 7, ustalając ilość pamięci RAM i wirtualnych procesów przeznaczonych dla systemu.

Rysunek 9. Instalacja maszyny wirtualnej KVM



W następnej kolejności ustalamy wielkość i lokalizację obrazu dysku. Ważnym parametrem umożliwiającym komunikację z naszą siecią jest rodzaj interfejsu. Wybieramy opcję „*Specify shared device name*" i wpisujemy nazwę mostu utworzonego skryptem Qoslink do pola o nazwie „*bridge name*" jak na rysunku 10. Po zatwierdzeniu powyższych danych następuje uruchomienie maszyny wirtualnej, a wraz z nią instalatora systemu.

Rysunek 10. Instalacja maszyny wirtualnej KVM

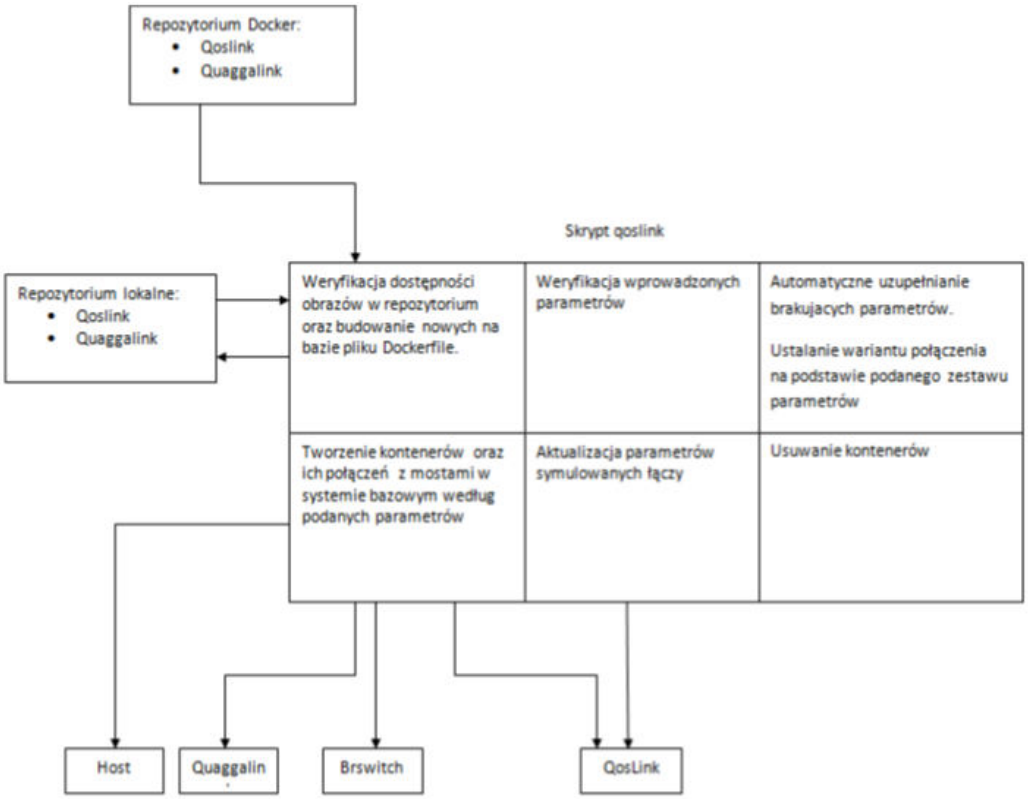


3.3 Architektura

Działanie skryptu opiera się na komunikacji poszczególnych elementów systemu poprzez tworzone wirtualne interfejsy veth oraz mosty linuksowe. W ten sam sposób podłączane są wszystkie inne systemy mające być testowane w tworzonej sieci wirtualnej. Dzięki przypisywaniu do interfejsów adresów IP oraz tworzeniu odpowiedniej struktury połączeń pomiędzy tymi elementami, umożliwiamy emulację warstwy sieciowej. Zakres funkcjonalny skryptu obejmuje modyfikację parametrów transmisji protokołu IP poszczególnych łącz oraz umożliwia uruchomienie protokołu routingu. Zapewnia to odwzorowanie zarówno parametrów sieci lokalnych jak i łącz internetowych.

Głównym elementem prezentowanego rozwiązania jest skrypt, który zawiera w sobie kilka części funkcjonalnych zaprezentowanych na rysunku 11 w znacznym stopniu automatyzujących i upraszczających wykonywanie postawionych przed nim zadań.

Rysunek 11. Podział funkcjonalny skryptu

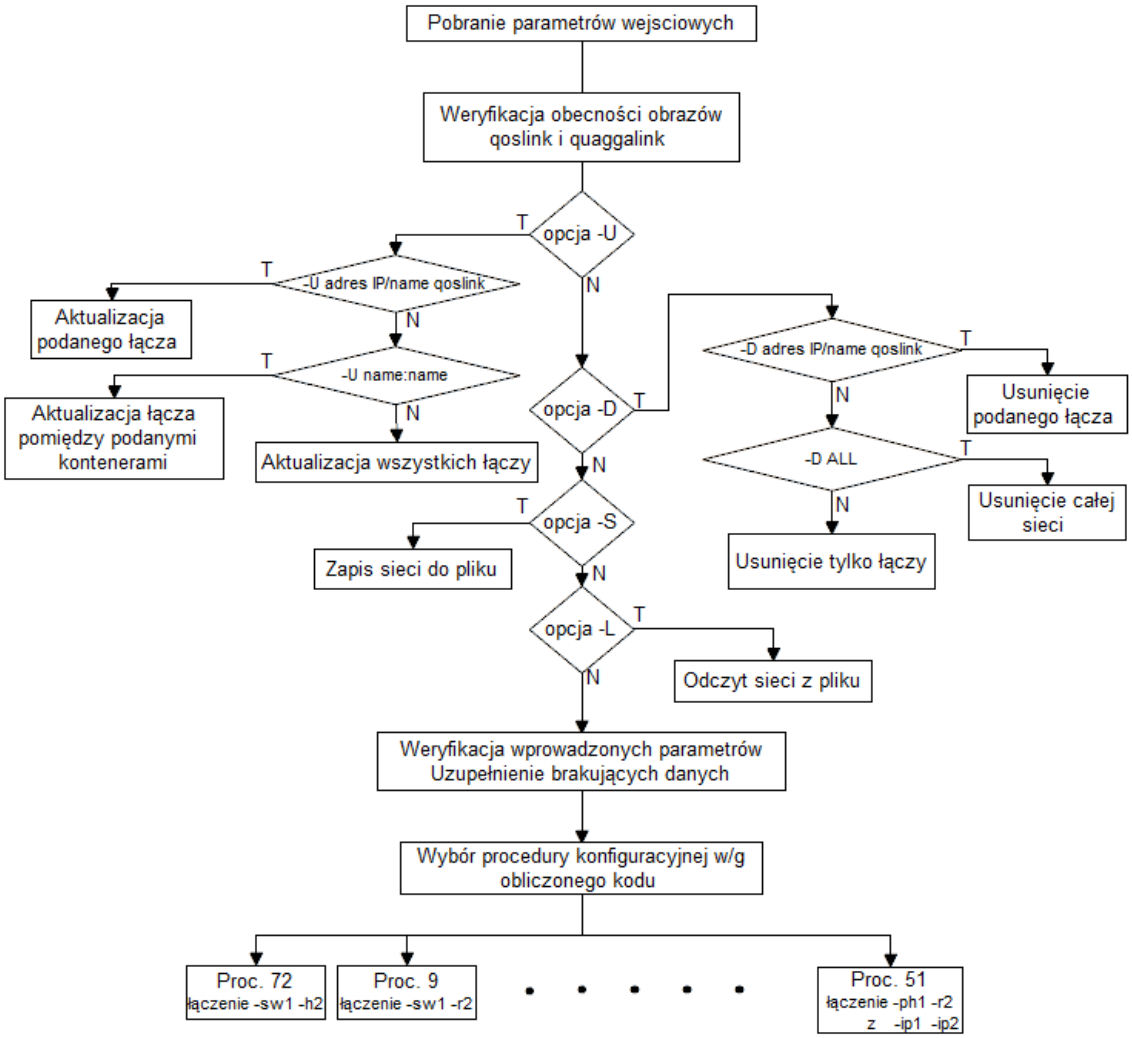


Dodatkowymi elementami rozwiązania są obrazy dockera przechowywane w lokalnym repozytorium. Określane są one w dalszej części pracy jako “qoslink” symulujący jednostkowe łącze, “quaggalink” pracujący jako router oraz “host” uruchamianego jako domyślny system klienta w symulowanej sieci.

Na rysunku 12 przedstawiono diagram przebiegu programu na którym można wyszczególnić pięć logicznie odrębnych etapów:

- Pobieranie parametrów wejściowych
- Sprawdzenie dostępności obrazów Dockera
- Obsługa opcji konfiguracyjnych sieci
- Weryfikacja parametrów wejściowych
- Wybór procedury konfiguracyjnej

Rysunek 12. Diagram przebiegu skryptu



Parametry konfiguracyjne przekazywane są do skryptu wyłącznie poprzez jego argumenty w wierszu poleceń. Lista parametrów opisana jest w dodatku B. Ze względu na ich dużą ilość, czterdziestu trzech, opracowałem procedurę zapisującą je do zmiennej tablicowej `$CFG[]`, Dzięki temu możliwe jest podawanie parametrów w dowolnej kolejności, zmiany ich nazewnictwa oraz stosunkowo łatwą implementację dodatkowych funkcji. Zmienna ta wykorzystywana jest na każdym etapie działania skryptu, a dane zawarte w niej opisują każdy aspekt tworzonej sieci.

Kolejnym etapem działania skryptu jest sprawdzenie dostępności obrazów dockera w repozytorium lokalnym. W przypadku ich braku, skrypt automatycznie pobierze je ze zdalnego repozytorium lub utworzy na bazie nowego systemu CentOS.

Weryfikacja poprawności danych wejściowych polega na sprawdzeniu ich obecności i ewentualnym automatycznym ich uzupełnieniu do pełnego zestawu

widocznego poniżej wraz z odpowiednią parą parametrów z tab.1 wymaganą do zestawienia łącza.

(-c –if1 –if2 –if3 –if4 –ip1 –ip2 –ip3 –ip4 –br1 –br2 –band1 –band2 –loss1 –loss2 –delay1 –delay2 –duplic1 –duplic2) - opis dodatek B

Sprawdzeniu podlegają również zależności pomiędzy innymi opcjami. Brane są pod uwagę następujące parametry.

Tabela 1. Parametry skryptu decydujące o rodzaju połączenia

Opcje skryptu	Opis
h1 h2	Nazwy kontenerów wirtualnych hostów, które zamierzamy podłączyć do wirtualnej sieci.
r1 r2	Nazwy kontenerów w których uruchomione zostaną routery Quagga.
sw1 sw2	Nazwy switchy oparte na mostach linuxowych.
ph1 ph2	Nazwy kontenerów “phlink” funkcjonujących jako routery brzegowe ASBR połączonych z fizyczną siecią hosta.
ip1 ip2	Adresy IP odpowiednio przypisywane do tworzonych kontenerów.

Powyższe parametry z Tabela 1 ułożone w słowo binarne według tabeli przedstawionej w dodatku A umożliwiają obliczenie kodu - obecność parametru oznacza wartość 1. Według wartości kodu skrypt wykonuje odpowiednio zdefiniowaną sekwencje procedur wymaganą do utworzenia połączenia.

Skrypt wszystkie dane o parametrach sieci przechowuje w postaci plików tekstowych wewnątrz kontenerów qoslink. Nie wymaga do działania żadnych dodatkowych baz danych oraz plików w systemie hosta. W przypadku zmiany topologii sieci, usuwane są lub tworzone nowe kontenery qoslink a wraz z nimi powyższe pliki. Dzięki temu stale utrzymujemy aktualne informacje o sieci. Zaletą tego rozwiązania jest brak konieczności aktualizowania zewnętrznych baz danych i weryfikacji ich spójności w stosunku do utworzonej sieci.

W skrypt zaimplementowane są również funkcje zapisu i odczytu do pliku aktualnie skonfigurowanej sieci przedstawione na rysunku 13. Pliki te z informacjami o parametrach i topologii sieci umożliwiają późniejsze jej odtworzenie z niezmienioną adresacją lub przeniesienie jej na inny system komputerowy.

Rysunek 13. Zapis sieci do pliku.

```
[root@localhost qoslink]# ./qoslink -S
Zapis: 11 węzłów
Nazwa pliku: testsdn17.dat
[root@localhost qoslink]#
```

W funkcjonalności skryptu nie zawarto możliwości bezpośredniego kasowania interfejsów wirtualnych tworzonych w systemie hosta, ze względu na to, że są one kasowane automatycznie po usunięciu poszczególnych kontenerów. Usunięcie zbędnych elementów sieci można uzyskać zatem poprzez jej zapis i ponowny jej odczyt - zapisywane są wyłącznie aktywne kontenery qoslink. Na rysunku 14 przedstawiono działanie polecenia kasującego całą skonfigurowaną sieć. Powoduje to automatyczne usunięcie z naszego systemu wszystkich nieaktualnych konfiguracji interfejsów, przełączników i routerów.

Rysunek 14. Kasowanie sieci z systemu.

```
[root@localhost qoslink]# ./qoslink -D ALL
Usuwanie kontenera phlink0
Usuwanie kontenera quaggalink0
Usuwanie kontenera qoslink1
Usuwanie kontenera host1
Usuwanie kontenera host0
Usuwanie kontenera qoslink0
Usuwanie bridga brlink0
Usuwanie bridga brlink1
Usuwanie bridga brlink2
Usuwanie bridga brlink3
Gotowe
[root@localhost qoslink]#
```

3.4 Elementy składowe

Podstawą tworzonych elementów sieci jest kontener docker. Wymagało to wyboru systemu uruchomionego w jego wnętrzu. Ze względu na charakter działania skryptu, tworzącego sieć wirtualną niedostępną z poziomu hosta i zewnętrznej sieci oraz wykorzystywaną jako tymczasowe środowisko testowe, możliwe było zastosowanie wcześniejszych wersji systemów nieobsługujących najnowszych zabezpieczeń. W związku z tym bezpieczeństwo zapewniane jest przez hermetyczny charakter kontenerów. Ważnym kryterium był rozmiar obrazu docker systemu bazowego dla routera. Konieczność uruchomienia jednocześnie wielu instancji wymusza ich optymalizacje pod względem wielkości. Uwzględniając powyższe warunki, wybrany został system CentOS 6.6. Określenie konkretnej wersji systemu zapewnia odporność skryptu na niezgodności związane z przyszłą aktualizacją do nowszych wersji systemu.

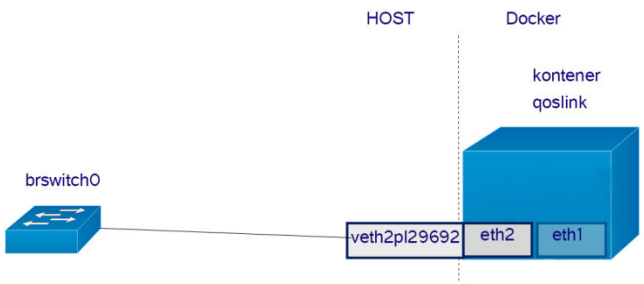
W związku z operacjami na plikach oraz uruchamianiem usług, kontenery uruchamiane są z włączoną powłoką interaktywną (-ti) oraz działające w tle (-d) za pomocą poniższego polecenia[31]:

```
# docker run -d -ti --name ${CFG[18]} --hostname ${CFG[18]} --net
      none --cap-add NET_ADMIN  chefronpc/qoslink:v1 /bin/bash
```

3.4.1 Switch

Switche w wirtualnej sieci oparte są na mostach linuksowych. Tworzone są one automatycznie za pomocą skryptu pipework konfigurującego ich połączenie z wirtualnymi interfejsami w kontenerach Docker. Jednocześnie włączony zostaje w mostach protokół drzewa rozpinającego STP. Dzięki temu protokołowi switche rozpoznają topologię sieci i likwidują możliwość zapętlania się ścieżek pakietów w wirtualnej sieci. Na rysunku 15 widać przykładowy sposób podłączenia switcha do kontenera qoslink. Skrypt pipework tworzy parę powiązanych ze sobą wirtualnych interfejsów „eth2" w kontenerze oraz „veth2pl29692" w systemie hosta. Dodatkowo interfejs „veth2pl29692" łączony jest ze switchem „brswitch0" jako jeden z jego portów. Przesłanie do switcha jakiegokolwiek pakietu powoduje automatyczne przekazanie go do interfejsu „eth2" wewnątrz kontenera

Rysunek 15. Podłączenie switcha z kontenerem



Na rysunku 16 pokazana jest konfiguracja mostu po utworzeniu jednostkowego łącza poleceniem „`# ./qoslink -h1 -sw2`"

[31] https://docs.docker.com/engine/reference/commandline/run/, (Dostęp 09-2015)

Rysunek 16. Konfiguracja mostu

```
[root@localhost qoslink]# brctl show
bridge name     bridge id               STP enabled     interfaces
brlink0         8000.222c90f2fff0       yes             veth1pl29692
                                                        veth1pl29760
brswitch0               8000.3665d4dfee57       yes             veth2pl29692
docker0         8000.0242c52677b6       no
[root@localhost qoslink]#
```

Wyjaśnienia wymaga nazewnictwo interfejsów wirtualnych. Pełna nazwa składa się kilku członów v|xxxx|pl|yyyyy, gdzie xxxx to nazwa interfejsu wewnątrz kontenera a yyyyy to numer procesu w systemie odpowiadający powłoce BASH uruchomionej w kontenerze. Parametry te można odczytać z plików konfiguracyjnych odpowiedniego kontenera poniższymi poleceniami:

```
# docker exec „kontener” ip a | grep eth
# docker inspekt „kontener” | grep Pid
```

Efekt działa poleceń przedstawiony jest na poniższym rysunku.

Rysunek 17. Parametry interfejsu sieciowego

```
[root@localhost lab1]# docker exec lab1 ip a | grep eth
691: eth1@if692: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state
UP qlen 1000
    link/ether 4e:d8:63:5e:1a:84 brd ff:ff:ff:ff:ff:ff
    inet 21.0.0.1/16 scope global eth1
[root@localhost lab1]# docker inspect R0 | grep Pid
        "Pid": 4587,
        "PidMode": "",
```

Dzięki temu łatwiejsza staje się lokalizacja poszczególnych interfejsów sieciowych, a tym samym monitorowanie sieci w trybie tekstowym.

3.4.2 Qoslink

Kontener qoslink jest podstawowym elementem sieci występującym w każdym skonfigurowanym łączu. W kontenerze tym w postaci pliku zapisane są wszystkie parametry wejściowe oraz te automatycznie tworzone w trakcie działania skryptu - Rysunek 18. Umożliwia to późniejszą analizę sieci oraz zmianę właściwości poszczególnych połączeń.

Rysunek 18. Zapis parametrów łącza w pliku

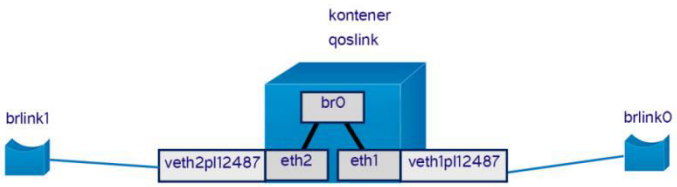
```
:qoslink0:host0:host1:eth1:eth1:10.0.0.1/24:10.0.0.2/24:brlink0:brlink1:100mbit:1
00mbit:0%:0%:0ms:0ms:.:_:_:_:_:_:_:10.0.0.3/24:10.0.0.4/24:eth1:eth2:_:0%:0%:_:
_:_:_:100mbit:0%:0ms:0%:_:_:_:_
~
~
"buffor_cfg.dat" 1L, 194C
```

Podczas tworzenia kontenera qoslink przyznawane są mu uprawnienia[32] do rekonfiguracji ustawień sieciowych opcją --cap-add NET_ADMIN. Wymagane jest to ze względu na wykonywanie poleceń tc. Jednocześnie za pomocą opcji --net NONE wyłączona jest w nim konfiguracja sieci i możliwość bezpośredniego dostępu do sieci fizycznej hosta przez domyślny most "docker0". Dostęp taki możliwy jest wyłącznie poprzez tworzoną sieć wirtualną oraz router brzegowy omówiony w dalszej części pracy.

Konfiguracja qoslinka podzielona jest na dwa etapy. W pierwszym podczas wywołania procedury *crt_brinqos()* następuje utworzenie mostu br0 pomiędzy dwoma wirtualnymi interfejsami za pomocą poniższych poleceń. Natomiast na rysunkach 19 i 20 widoczne jest pełne zestawienie interfejsów wraz powiązaniem mostów brlink tworzonych przez skrypt z odpowiednimi wirtualnymi interfejsami.

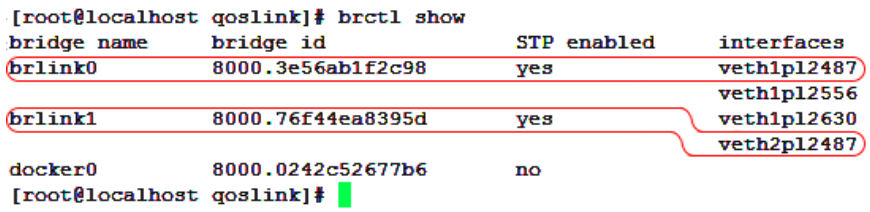
```
# docker exec qoslink brctl addbr br0
# docker exec qoslink brctl addif br0 eth1
# docker exec qoslink brctl addif br0 eth2
# docker exec qoslink ip addr add 10.0.0.3/24 dev br0
# docker exec qoslink ip link set dev br0 up
```

Rysunek 19. Schemat kontenera qoslink – interfejsy



[32] http://linux.die.net/man/7/capabilities, (Dostęp 18-12-2015)

Rysunek 20. Konfiguracja mostów – interfejsy



W drugim etapie w procedurze *set_link()* ustawiane są parametry symulacyjne łącza[33] [34] dla obu interfejsów niezależnie.

```
# docker exec qoslink tc qdisc add dev eth1 root handle 1: tbf rate
      100mbit limit 1 burst 10k mpu 64
# docker exec qoslink tc qdisc add dev eth2 root handle 1: tbf rate
      100mbit limit 1 burst 10k mpu 64
# docker exec qoslink tc qdisc add dev eth1 parent 1: handle 2: netem
      delay 10ms loss 0.1% duplicate 0%
# docker exec qoslink tc qdisc add dev eth2 parent 1: handle 2: netem
      delay 10ms loss 0.1% duplicate 0%
```

Ważnym elementem konfiguracji jest prawidłowe ustalenie wartości parametrów pasma, opóźnienia, utraty i powtarzania pakietów - parametry opisane w dodatku B - niezależnie dla obu kierunków transmisji. W rzeczywistych sieciach podczas testów mierzymy sumaryczną w obu kierunkach wartość powyższych parametrów. Aby symulować odpowiadające im własności łącza w wirtualnej sieci należy przeliczyć je dla obu kierunków niezależnie. Dla parametru utraty pakietów, według poniższego wzoru:

$$loss1 = loss2 = \left(10 - \sqrt{100[\%] - loss[\%]}\right) \cdot 10 \qquad (5.1)$$

Natomiast dla duplikowania pakietów ze wzoru:

$$duplic1 = duplic2 = \left(\sqrt{\frac{duplic[\%]}{100\%} + 1} - 1\right) * 100\% \qquad (5.2)$$

[33] http://linux.die.net/man/8/tc-tbf, (Dostęp 09-2015)
[34] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, (Dostęp 09-2015)

3.4.3 Router

Kontener quaggalink opiera się na tym samym systemie co qoslink. Umożliwia to wykorzystanie już wcześniej utworzonych przy okazji kontenera qoslink obrazów pośrednich. Obraz quaggalink skonfigurowany został w oparciu o plik dockerfile przedstawiony poniżej.

```
FROM centos:6.6
RUN yum -y install bridge-utils net-tools mtr tar nmap telnet wget
       tcpdump quagga
RUN echo "hostname quaggalink" > /etc/quagga/zebra.conf \
&& echo "hostname quaggalink" > /etc/quagga/ospfd.conf \
&& echo "password zebra" >> /etc/quagga/zebra.conf \
&& echo "password zebra" >> /etc/quagga/ospfd.conf \
&& echo "enable password zebra" >> /etc/quagga/ospfd.conf \
&& chmod 640 /etc/quagga/zebra.conf \
&& chmod 640 /etc/quagga/ospfd.conf \
&& chown quagga:quagga /etc/quagga/zebra.conf \
&& chown quagga:quagga /etc/quagga/ospfd.conf
```

Są to podstawowe ustawienia umożliwiające uruchomienie usług zebra oraz ospfd. Pozostałe czynności wykonuje się już w trakcie tworzenia łączy i zawierają w sobie konfigurację:

- adresacji IP interfejsów
- okresów rozsyłania komunikatów protokołu OSPF
- obszaru do którego jest przypisany router

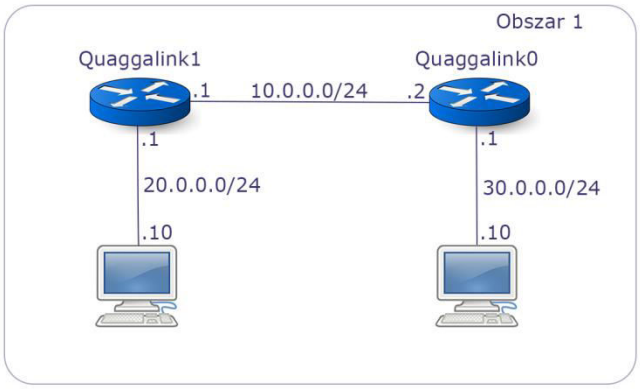
Działania te wykonywane są w procedurze *crt_linkif1r1()* lub *crt_linkif2r2()* za pomocą zintegrowanego terminala vtysh w trybie poleceń wsadowych przedstawionych poniżej:

```
# docker exec ${CFG[18]} vtysh -c "configure terminal" -c "interface
       ${CFG[3]}" -c "ip address ${CFG[5]}" -c "description to-
       ${CFG[0]}" -c "ip ospf hello-interval 5" -c "ip ospf dead-
       interval 10" -c "no shutdown" -c "exit" -c "exit" -c "write"
# docker exec ${CFG[18]} vtysh -c "configure terminal" -c "router
       ospf" -c "network $NEWNET area 1" -c "exit" -c "exit" -c
       "write"
```

Okres rozgłoszeniowy komunikatów protokołu OSPF ustalony został na 10 sekund, na zasadzie kompromisu pomiędzy ilością komunikatów OSPF w sieci,

a czasem ich propagacji na obszarze całej wirtualnej sieci. Podyktowane to było możliwością wykonywania częstych testów w sytuacji zmiennych parametrów sieci. Ważnym parametrem jest przypisanie każdego routera quaggalink do jednego z obszarów określających zakres kalkulacji kosztów łączy. Domyślnie jest to obszar 1 widoczny na rysunku 21.

Rysunek 21. Schemat połączeń routerów w obszarze 1



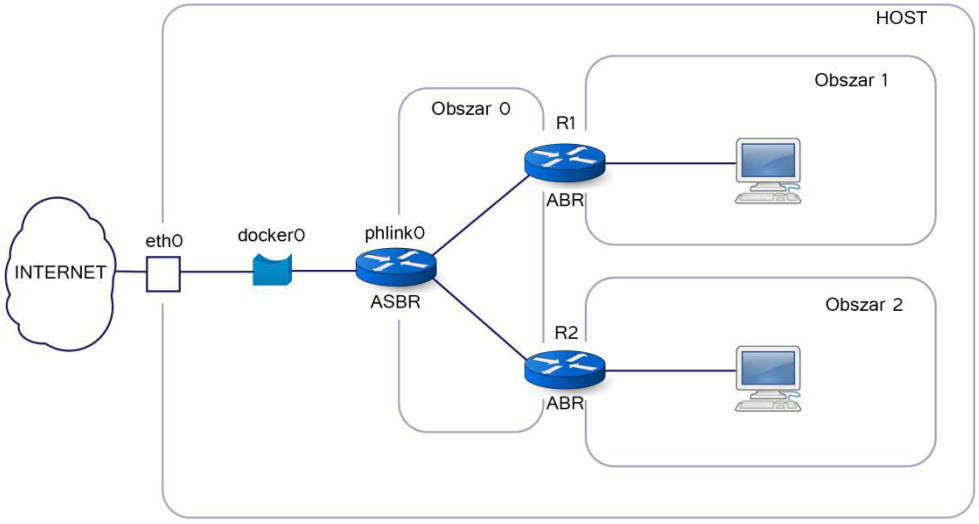
3.4.4 Router brzegowy ASBR

Router brzegowy o nazwie domyślnej „phlink” (skr. ang. physic link) posiada częściowo zmienioną konfigurację, w stosunku do zwykłego routera, umożliwiającą podłączanie sieci wirtualnej do fizycznej sieci hosta. Na rysunku 22 przedstawiono umiejscowienie routera ASBR w przykładowej sieci. Proces tworzenia połączenia przebiega podobnie jak dla routera z obszaru różnego od zera. Różnica uwidacznia się w konfiguracji interfejsu wyjściowego. Tworzony jest jako domyślny interfejs podczas uruchamiania kontenera w trybie “--net bridge”. Powoduje to automatyczne połączenie kontenera z mostem “docker0” i przypisaniem domyślnego adresu z zakresu 172.17.0.0/16. Poniższymi poleceniami przypisujemy interfejs zewnętrzny do sieci szkieletowej (obszar 0 OSPF) oraz włączamy domyślny routing dla danego obszaru OSPF. Dodatkowo w firewallu definiujemy SNAT umożliwiając poprawną komunikację wszystkich hostów w wirtualnej sieci z Internetem.

```
# docker exec ${CFG[32]} vtysh -c "configure terminal" -c "router
       ospf" -c "network $NEWNET area 0" -c "default-information
       originate always" -c "exit" -c "ip route 0.0.0.0 0.0.0.0
       ${IP5}" -c "exit" -c "write"
```

```
# docker exec ${CFG[32]} iptables -t nat -A POSTROUTING -s 0.0.0.0/0 -
       o eth0 -j SNAT --to-source ${IP5}
```

Rysunek 22. Przykładowa struktura połączeń sieci dla routera w funkcji ASBR

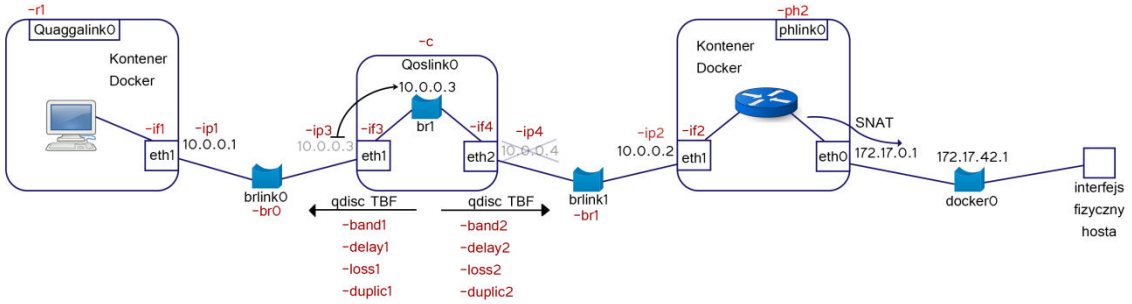


3.5 Procedura konfiguracji łącza

W punkcie tym zaprezentowany zostanie przebieg działania skryptu. Na jego przykładzie będzie można przeanalizować proces zestawiania wirtualnych połączeń sieci. Użyteczność skryptu wymaga, aby była możliwość łączenia ze sobą poszczególnych typów wirtualnych urządzeń sieciowych, tak jak ich odpowiedników fizycznych. Na wstępnym etapie projektowania skryptu wybrane zostały parametry, które mają znaczenie przy ustalaniu typu połączenia tab.1. To z kolei pozwoliło na przegląd ich dostępnych kombinacji połączeń i wybranie tych, które mają odzwierciedlenie w rzeczywistości tab. 2 - Dodatek A. Poniżej przedstawiona zostanie jedna z wielu dostępnych procedur pokazująca możliwie najwięcej funkcji zawartych w skrypcie.

Schemat przedstawiony na rysunku 23 przedstawia szczegółowo parametry konfiguracyjne łącza.

Rysunek 23. Szczegółowa konfiguracja łącza



Jako przykład przedstawione zostanie połączenie routera (-r1) do sieci fizycznej hosta poprzez router brzegowy (-ph2). Umożliwi to udostępnienie Internetu wewnątrz wirtualnej sieci. Polecenie z pełną lista opcji wymaganych do konfiguracji, zawierające dane o interfejsach, adresach IP, nazwach kontenerów i parametrach sieci przedstawia się następująco:

```
# ./qoslink -ph2 phlink0 -if1 eth1 -ip1 10.0.0.1/24 -r1 quaggalink0 -
       if2 eth1 -ip2 10.0.0.2/24 -ip3 10.0.0.3/24 -ip4 10.0.0.4/24 -
       br1 brlink0 -br2 brlink1 -band1 100Mbit -band2 100Mbit -loss1
       0% -loss2 0% -delay1 0ms -delay2 0ms -duplic1 0% -duplic2 0%
```

Ze względu na dużą ilość parametrów, w skrypcie zawarte zostały procedury auto konfiguracyjne ułatwiające jego użytkowanie. Polecenie to w efekcie można zapisać w najbardziej skróconej formie ograniczając się do wskazania jakie elementy sieci zamierzamy za sobą połączyć.

```
# ./qoslink  -r1 -ph2
```

Na podstawie Tabela 9. Tabela parametrów wejściowych skryptu. w dodatku A z parametrów r1 oraz ph2 uzyskujemy kod “258” oraz listę funkcji do wykonania w celu zestawienia łącza.

```
freenet(), freeip() IP1, freeip() IP2, freeip() IP3, freeip() IP4,
       set_c(), set_br1(), set_br2(), crt_c(), set_r1(), crt_r1(),
       crt_ph2(), crt_ph2(), set_if1r1(), crt_linkif1r1(),
       set_if2ph2(), crt_linkif2ph2(), set_if3(), crt_linkif3(),
       set_if4(), crt_linkif4(), crt_brinqos(), set_link(),
       crt_linkif2docker0();
```

Poniżej omówione zostały chronologicznie poszczególne funkcje:

freenet():

Skrypt uruchomiony bez podanych adresów IP, na wstępie ustali nowy nieużywany jeszcze adres sieci, w której będą konfigurowane interfejsy kontenerów. Zwalnia to użytkownika z analizowania sieci i pilnowania aby uniknąć np. konfliktu adresów. Przydatne jest to zwłaszcza przy konfiguracji sieci routerów gdzie adresacja poszczególnych segmentów nie ma czasami większego znaczenia. Wyszukiwanie wolnej sieci rozpoczyna się od domyślnego adresu zapisanego w zmiennej $DEFAULTNET. Funkcja w pierwszej fazie wykonuje listę wszystkich adresów IP z kontenerów, a następnie poszukiwane i pomijane są kolejne użyte już adresy sieci. Zaimplementowana funkcja obsługuje maski sieci w szerokim zakresie od 2 do 29 bitów. Długość 29 bitów jest graniczną ze względu na konieczność dostępu w danej sieci do czterech adresów IP dla pojedynczego połączenia. Wynikiem działania funkcji jest adres nowej sieci.

freeip() IP1-4:

Funkcja wyszukiwania wolnego adresu IP, podobnie jak poprzednia podzielona jest na dwa etapy. W pierwszym etapie, z kontenerów pobierane są ustawienia sieciowe i tworzona jest z lista adresów IP zgodnych z zadaną siecią. Następnie lista ta jest przeszukiwana w celu wyszukania wolnego adresu IP. Dodatkową wartością zwracaną przez funkcję jest adres bramy domyślnej. Ustalany jest on jako ostatni dostępny adres IP dla danej sieci, a wykorzystywany do ewentualnej późniejszej konfiguracji hostów. Korzyścią z takiego rozwiązania jest możliwość tworzenia hostów z ustawionym adresem bramy zanim zostanie jeszcze utworzony router.

W obu omówionych funkcjach kontenery do analizy były wybierane na podstawie nazwy obrazu na bazie którego powstały, czyli chefronpc/qoslink:v1, chefronpc/quaggalink:v1 oraz chefronpc/host:v1. Zaletą takiego rozwiązania jest brak ingerencji skryptu w kontenery nienależące do sieci, a utworzone przez użytkownika na podstawie innych obrazów. Brak interakcji pomiędzy systemem hosta a tworzoną siecią, było jednym z wymagań stawianych temu skryptowi.

set_c():

Ustalanie nazwy kontenera qoslink. Możliwe jest nadawanie własnych nazw dla tych kontenerów, których później możemy używać do wskazywania połączeń celem ich

aktualizacji lub wyświetlania. Poniżej podałem przykład wykorzystania własnej nazwy „kra-war" do wskazywania połączenia pomiędzy routerami w Krakowie i Warszawie.

Utworzenie połączenia pomiędzy dwoma routerami:

```
# ./qoslink -c kra-war -r1 r-kra -r2 r-war
```

Aktualizacja parametrów łącza

```
# ./qoslink -U kra-war -link ASDL3/8
```

Wyświetlenie parametrów pojedynczego łącza

```
# ./qoslink -P kra-war
```

set_br1(), set_br2():

Ustalanie nazw dla mostów linuksowych w systemie hosta, pośredniczących w przekazywaniu pakietów pomiędzy daną parą kontenerów. Raz utworzony i podłączony most nie jest już podłączany do żadnego innego kontenera. Pomimo możliwości ich nazywania jest to zbyteczne. Mosty te działają w wirtualnej sieci w tle i ich nazwy nie są wykorzystywane do żadnych późniejszych konfiguracji.

Wyjątek może stanowić sytuacja, w której "przyjazna" nazwa może być wykorzystana do łatwiejszego odszukania danego połączenia i analizy transmisji pakietów jednym z programów diagnostycznych np. wireshark.

crt_c():

```
W funkcji tej następuje uruchomienie kontenera qoslink z wcześniej
                        określoną nazwą.
```

set_r1():

Ustalanie nazwy dla routera. W przypadku podania opcji -r1 lub -r2 bez argumentu, dodatkowa funkcja *freerouter()* wyszuka pierwszą wolną nazwę dla routera. Sytuacja ta występuje w podanym poleceniu dla routera r1

```
# ./qoslink -r1  -r2  router
```

Działanie automatycznie uzupełniające będzie funkcjonowało również przy innych parametrach zawartych w tabeli 1.

crt_r1():

Założeniem pracy routera jest możliwość konfigurowania wielu interfejsów dla różnych sieci. Funkcja ta rozróżnia sytuację tworzenia nowego routera od dołączania nowej sieci do już istniejącego routera. W pierwszym przypadku uruchamiany jest kontener quaggalink, a następnie usługi zebra oraz ospfd. W przeciwnym wypadku funkcja pomija całą konfiguracje wyświetlając tylko komunikat o istniejącym już routerze.

set_ph2(), crt_ph2():

Struktura funkcji jest identyczna co dwóch poprzednich. Różnica polega jedynie na korzystaniu z innych danych konfiguracyjnych tablicy $CFG[] oraz konieczności uruchomienia kontenera z obsługą domyślnej sieci zapewnianej przez system Docker (opcja --net bridge). W efekcie router ABR automatycznie podpinany jest do mostu "docker0" z systemu hosta. Jeżeli nie zostały wprowadzone zmiany w konfiguracji Dockera, powinien zostać przypisany adres interfejsu z zakresu 172.17.0.0/16..

set_if1r1():

Ustalanie nazwy interfejsu do utworzenia w routerze dla nowej sieci. Możliwe jest podanie tej nazwy w opcji "-if1"

crt_linkif1r1():

Po przygotowaniu w poprzednich funkcjach wstępnych danych, możliwe jest utworzenie połączenia pomiędzy routerem -r1 a mostem -br1 za pomocą skryptu pipework. Właściwa konfiguracja routera wykonana jest poprzez uruchomienie wewnątrz kontenera terminala zintegrowanego vtysh w trybie polecenia wsadowego[35].

```
# docker exec ${CFG[18]} vtysh -c "configure terminal" -c "interface
       ${CFG[3]}" -c "ip address ${CFG[5]}" -c "description to-
       ${CFG[0]}" -c "ip ospf hello-interval 5" -c "ip ospf dead-
       interval 10" -c "no shutdown" -c exit -c exit -c write
```

[35] https://docs.docker.com/engine/reference/commandline/exec/, (Dostęp 11-2015)

set_if2ph2(), crt_linkifph2():

Funkcje te są identyczne w działaniu co dwie poprzednie. Różnica polega na odwoływaniu się do innych pozycji konfiguracyjnych.

set_if3(), crt_linkif3(), set_if4(), crt_linkif4():

Zestaw tych funkcji jest stałym elementem konfiguracji połączenia. Ustalają nazwy interfejsów w kontenerze qoslink oraz łączą je za pomocą skryptu "pipework" do sąsiadujących mostów -br1 i -br2. Wyjątek stanowią połączenia do przełączników kiedy to w zamian stosowane są funkcje *crt_linkif3sw1() lub crt_linkif4sw2()* korzystające z danych sw1 lub -sw2.

crt_brinqos(), set_link():

Funkcje te łączą ostatecznie obie części sieci za pomocą mostu oraz konfigurują parametry łącza. Szczegółowy opis przedstawiony został w pkt. 5.3.2.

crt_linkif2docker0():

W ostatniej funkcji konfigurowane jest połączenie z fizyczną siecią hosta poprzez obszar 0 sieci szkieletowej routingu OSPF. Opis przedstawiony w pkt. 5.3.4

4. Testowanie sieci

W bieżącym rozdziale przedstawię kilka wybranych konfiguracji sieci, możliwych do utworzenia za pomocą skryptu qoslink. Będzie to m. in. bezpośrednie połączenie dwóch kontenerów będące podstawą testów łącza, połączenie kilku sieci routerami prezentujące protokół OSPF, komunikację sieci wirtualnej z internetem. Zaprezentowana zostanie również możliwość podłączania do wirtualnej sieci innych systemów niż kontenery docker, a mianowicie maszyn wirtualnych KVM/QEMU. Zwiększa to znacznie zakres wykorzystania omawianego skryptu.

Testy skryptu przeprowadzone zostały na komputerze Fujitsu ESPRIMO 5925 opartym o procesor Intel Core2Duo 2.33Ghz, 4GB pamięci RAM oraz dysku SSD 40GB Sandisk-C50. Do wykonania testów porównawczych sieci fizycznej z rozwiązaniem wirtualnym wykorzystany był również drugi komputer o parametrach Intel Core2Duo 2.0Ghz, 2GB pamięci RAM oraz dysku 128GB Sandisk-C120. Oba komputery były łączone do testów przełącznikiem D-Link 10/100Mbps. Testy te miały na celu wykazać na ile rozwiązanie wirtualne odzwierciedla parametry rozwiązań sprzętowych.

4.1 Generowanie i monitorowanie danych

Przeprowadzenie wiarygodnych testów sieci wymaga stworzenia w niej kontrolowanych i powtarzalnych warunków pracy. Narzędzia zaprojektowane w tym celu to generatory pakietów. Tworzą one strumienie danych o określonej charakterystyce, uwzględniającej prędkość przesyłu danych, rozmiar pakietu oraz typ protokołu. Na potrzeby tej pracy wykorzystany został generator pakietów Iperf[36]. Dostępny jest na platformy Linux oraz Windows, dzięki czemu możliwe jest przeprowadzenie jednolitych testów zarówno z poziomu kontenerów Docker jak i maszyn wirtualnych KVM z powyższymi systemami.

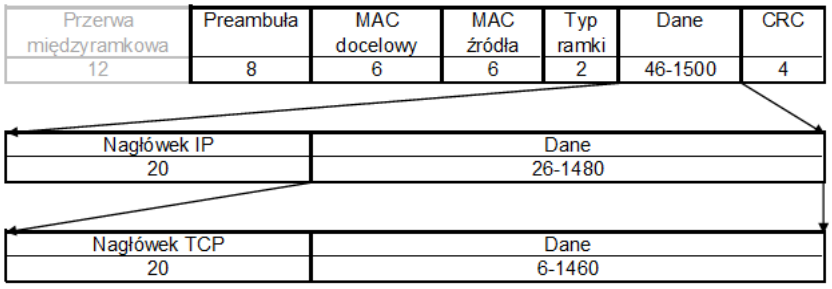
Architektura programu Iperf jest typu klient-serwer. Pozwala to na jednoczesne testowanie łączy pomiędzy dwoma lub większą ilością hostów. Parametry możliwe do zmierzenia to m. in. dostępne pasmo łącza, opóźnienie oraz utrata pakietów. Należy

[36] Strona domowa oprogramowania IPerf, https://iperf.fr/, (Dostęp 09-2015)

zaznaczyć, iż jednostki prędkości łącza jakimi posługujemy się przy konfiguracji skryptu qoslink to pochodne mbit/s (1mbit/s = 1000x1000 bit/s). Rozwiązania sprzętowe oraz rzeczywiste łącza internetowe również wykorzystują te jednostki, dzięki czemu zachowujemy odpowiednią skalę porównawczą.

Do przeprowadzenia testów wykorzystywany jest protokół TCP. Przedstawię strukturę pakietu TCP/IP[37], aby wyjaśnić wpływ jego wielkości na efektywną skuteczność transmisji danych.

Rysunek 24. Struktura pakietu TCP



Na powyższym rysunku widoczne jest, iż przesłanie danych wymaga dołączenia nagłówków TCP, IP oraz Ethernet o łącznej długości 58 bajtów. Doliczyć należy również 20 bajtów na preambułe i przerwę między ramkową do każdego transmitowanego pakietu. Efektywność transmisji można zatem obliczyć (6.1) ze stosunku ilości danych transmitowanych w pojedynczym pakiecie do całkowitej ilości bajtów wymaganych do jego przesłania.

$$skuteczność = \frac{ilość\ danych\ w\ pakiecie}{ilość\ danych + nagł.IP + nagł.TCP + nagł.\ ethernet + 20\text{B}} \qquad (6.1)$$

Dla najdłuższego pakietu Ethernet przy $MTU = 1500$ bajtów skuteczność wynosi:

$$skuteczność = \frac{MTU - 40\ bajtów}{MTU + 18bajtów + 20bajtów} = \frac{1460}{1538} = 94{,}93\% \qquad (6.2)$$

W przypadku pakietu UDP sytuacja jest analogiczna z jedną różnicą. Rozmiar nagłówka pakietu UDP jest równy 8 bajtów i umożliwia przesłanie 12 bajtów więcej danych w jednym pakiecie niż w przypadku protokołu TCP/IP. Efektywna skuteczność protokołu UDP wyliczamy ze wzoru (6.3)

[37] http://www.staff.amu.edu.pl/~ttomek/sik/cwiczenia7.html, (Dostęp 12-2015)

$$skuteczność = \frac{MTU - 28\ bajtów}{MTU + 18bajtów + 20bajtów} = \frac{1472}{1538} = 95{,}71\% \qquad (6.3)$$

Do monitorowania obciążenia sieci wykorzystany był program „bmon" na platformę Linux. Umożliwia równoczesny zapis do pliku w formacie tekstowym danych z dowolnej liczby interfejsów dostępnych w systemie. Poniższe przykładowe wywołanie programu bmon tworzy plik wynikowy jak na rysunku 25.

```
# bmon -b -p veth1pl4587,veth2pl4587,veth3pl4587,veth2pl18543 -o ASCII
```

Rysunek 25. Program bmon – raport obciążenia sieci

```
[root@localhost lab1]# bmon -b -p veth1pl4587,veth2pl4587,veth3pl4587,veth2pl18543 -o ascii
Interfaces                          RX bps        pps      %    TX bps        pps      %
  veth1pl4587                       281.90Kb      385           10.69Mb       495
  veth2pl4587                         5.19Mb      247          149.89Kb       205
  veth2pl18543                      152.58Kb      208            5.10Mb       245
  veth3pl4587                         5.49Mb      247          132.84Kb       181
Interfaces                          RX bps        pps      %    TX bps        pps      %
  veth1pl4587                       239.86Kb      321            8.92Mb       378
  veth2pl4587                         4.29Mb      188          128.39Kb       172
  veth2pl18543                      131.89Kb      176            4.33Mb       193
  veth3pl4587                         4.62Mb      188          111.67Kb       149
[root@localhost lab1]#
```

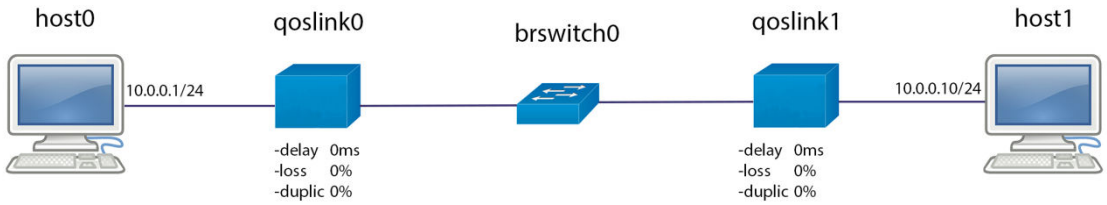
Dane z tych plików są podstawą do wizualizacji obciążenia sieci w postaci wykresów tworzonych w programie Gnuplot.

4.2 Testy łącza

Głównym parametrem skryptu qoslink określającym przydatność rozwiązania jest dokładność symulacji łącza w stosunku do zadanych parametrów. Przedstawione zostaną zatem charakterystyki przepustowości łącza oraz opóźnienia, gubienia i duplikowania pakietów, czyli parametry na które bezpośrednio mamy wpływ podczas konfiguracji sieci. Testy przeprowadzone zostały za pomocą połączonych dwóch hostów poprzez przełącznik brswitch0, łączem o parametrach domyślnych, czyli gubienie i duplikowanie pakietów na poziomie 0% a opóźnienie 0ms. Przepustowość będzie parametrem badanym w zakresie od 9,6kbit do 2,75gbit. Polecenie konfigurujące łącze jest następujące:

```
# ./qoslink -h1 –sw2 brswitch0 –ip1 10.0.0.1/24
# ./qoslink –h1 –sw2 brswitch0 –ip1 10.0.0.10/24
```

Rysunek 26. Konfiguracja testowa łącza



4.2.1 Przepustowość

Tabela 2 przedstawia zależność uzyskiwanego poziomu ograniczania pasma w stosunku do zadanej wartości. Dla porównania zawarta jest również wartość teoretyczna przepustowości obliczona na podstawie wzoru (6.2). Rozbieżności w górnym paśmie wynikają ograniczeń związanych z parametrami technicznymi sprzętu na którym prowadzone były testy. Pomimo rozbieżności uzyskiwanego pasma na końcach zakresów pomiarowych, w szerokim zakresie od 32kbit do 1,5gbit uzyskano liniowe odwzorowanie pasma w stosunku do założeń teoretycznych.

Tabela 2. Przepustowość łącza Qoslink – Protokół TCP/IP

Zadane pasmo łącza [Mbps]	przepustowość qoslink [Mbps]	przepustowość teoretyczna [Mbps]	Przepustowość qoslink w stosunku do wart. teoretycznej [%]	
2750,0000	2217,0000	2609,0000	84,98	
2500,0000	2150,0000	2373,0000	90,60	
2250,0000	1908,0000	2135,0000	89,37	
2000,0000	1674,0000	1898,0000	88,20	
1750,0000	1639,0000	1660,0000	98,73	
1500,0000	1435,0000	1423,0000	100,84	
1000,0000	956,0000	949,3000	100,71	
512,0000	489,0000	485,9000	100,64	
256,0000	245,0000	242,9000	100,86	
100,0000	95,7000	94,9300	100,81	
64,0000	61,2000	60,7400	100,76	
32,0000	30,6000	30,3700	100,76	
16,0000	15,3000	15,1800	100,79	
10,0000	9,5700	9,4930	100,81	
4,0000	3,8200	3,7960	100,63	
1,0000	0,9570	0,9490	100,84	
0,5120	0,4900	0,4860	100,82	
0,2560	0,2450	0,2430	100,82	
0,1280	0,1230	0,1215	101,23	
0,0640	0,0614	0,0607	101,15	Średnia [%]
0,0320	0,0305	0,0304	100,33	100,80

. Nieprawidłowością jest uzyskiwanie pasma powyżej wartości teoretycznej, z tego względu wprowadzony został do skryptu wskaźnik korygujący ten parametr.

Jego wyliczenie opiera się na podstawie pomiarów przesyłu w sprzętowym przełączniku 10/100mbit oraz średniej wartości parametrów z liniowej części charakterystyki Tabela 2.

Tabela 3. Testy przepustowości przełącznika sprzętowego

Pasmo switcha [mbit]	Przepustowość teoretyczna [mbit]	Przepustowość przełącznika [mbit]	Wartość przepustowości przełącznika do wart. Teoretycznej [%]
10	9,493	9,41	99,13
100	94,93	94,14	99,17

Wskaźnik o jaki należy skorygować wartość zadaną pasma w skrypcie Qoslink obliczony został jako stosunek średniej ze strat przesyłu w sprzętowym przełączniku do średniej procentowej rozbieżności z Tabela 2. Przepustowość łącza Qoslink – Protokół TCP/IP.

$$wskaźnik\ pasma = \frac{\frac{(99{,}13\% + 99{,}17\%)}{2}}{100{,}8\%} = 0{,}9836 \qquad (6.4)$$

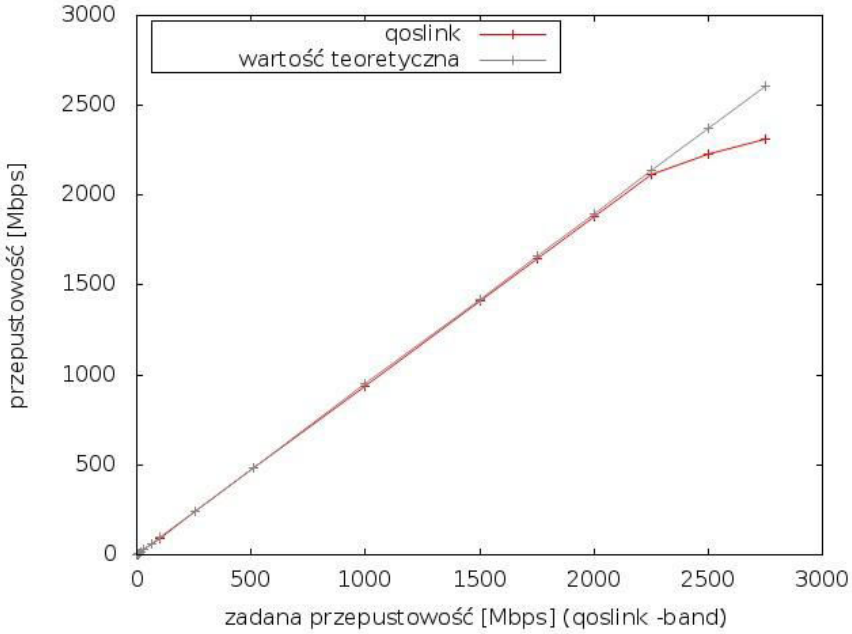
Po obniżeniu zadanego pasma w skrypcie o ten wskaźnik, ponowne testy łącza wykazały prawidłowe ograniczanie pasma w zakresie od 32kbit do 2,25gbit. Wyniki przedstawione zostały w Tabela 4. Obliczona wartość średnia jest zbieżna z tą uzyskiwaną w sprzętowym przełączniku.

Tabela 4. Przepustowość łącza z uwzględnieniem wskaźnika pasma – protokół TCP

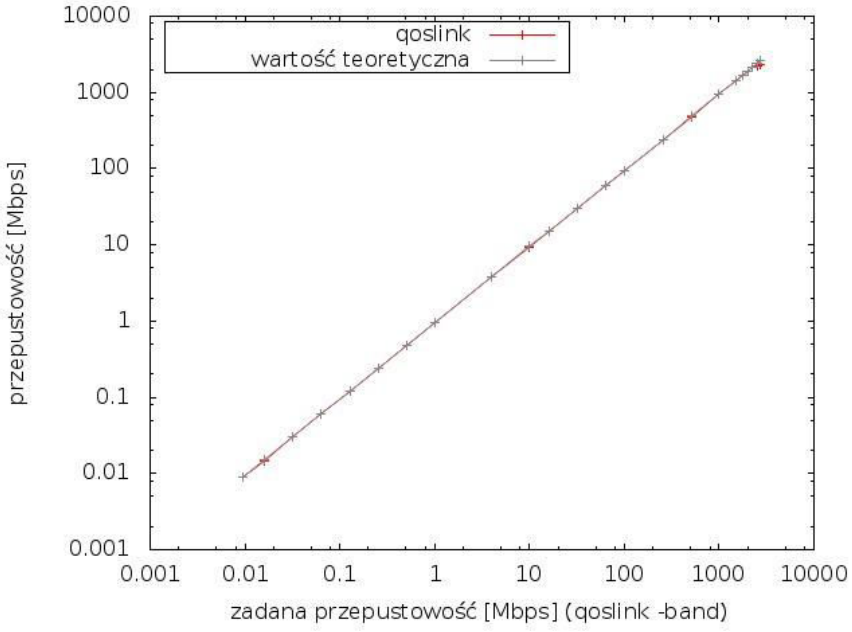
Zadane pasmo łącza [Mbps]	przepustowość qoslink [Mbps]	przepustowość teoretyczna [Mbps]	Wartość przepustowości qoslink w stosunku do wart. teoretycznej [%]	
2750	2312	2609	88,62	
2500	2229	2373	93,93	
2250	2116	2135	99,11	
2000	1881	1898	99,10	
1750	1646	1660	99,16	
1500	1411	1423	99,16	
1000	940,8	949,3	99,10	
512	481,7	485,9	99,14	
256	240,8	242,9	99,14	
100	94,08	94,93	99,10	
64	60,21	60,74	99,13	
32	30,11	30,37	99,14	
16	15,05	15,18	99,14	
10	9,408	9,493	99,10	
4	3,764	3,796	99,16	
1	0,941	0,949	99,16	
512	0,482	0,486	99,18	
0,256	0,241	0,243	99,18	
0,128	0,121	0,1215	99,59	
0,064	0,0604	0,0608	99,34	Wart. Średnia [%]
0,032	0,0301	0,0304	99,01	99,17

Na rysunkach 27 i 28 przedstawiono wykresy wykonane na podstawie danych z Tabela 4.

Rysunek 27. Przepustowość łącza qoslink z uwzględnieniem wskaźnika - skala liniowa

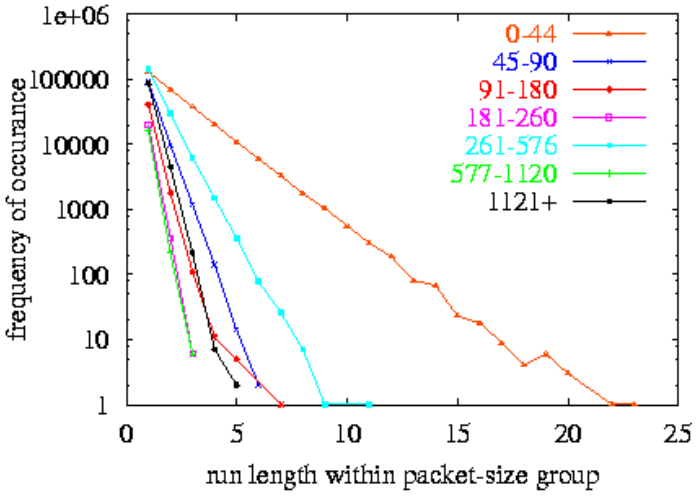


Rysunek 28. Przepustowość łącza qoslink z uwzględnieniem wskaźnika - skala logarytmiczna



Po przeprowadzeniu analiz przez grupę INET na sieciach szkieletowych[38] wykazano, iż duża część transmisji danych składa się małych pakietów jak pokazano na rysunku poniżej.

Rysunek 29. Częstość występowania pakietów w danej grupie pakietów



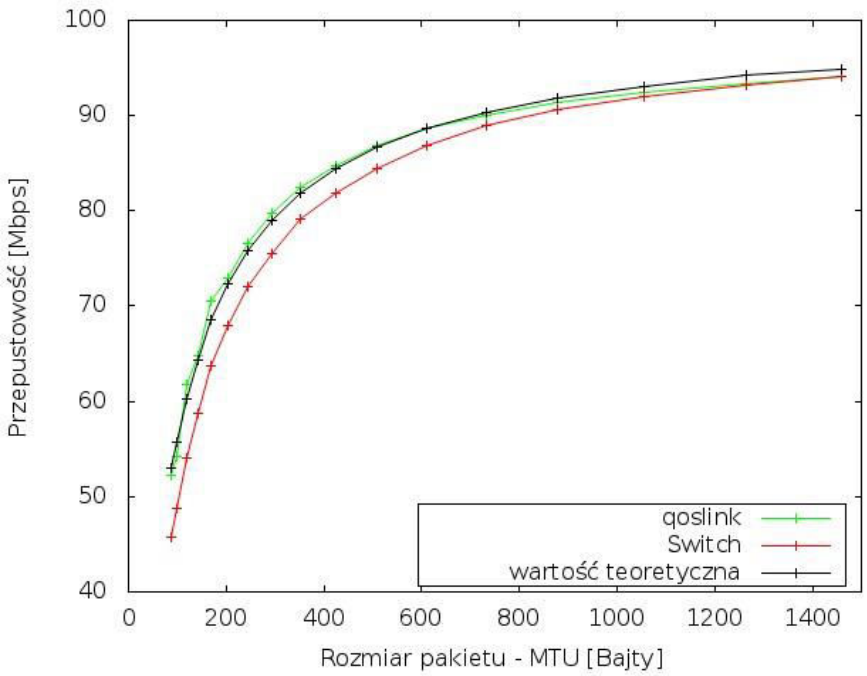
Z tego powodu przeprowadzone zostały testy łącza uwzględniające rozmiar pakietu. Wyniki przedstawione są na rysunku 30. Widoczny jest gwałtowny spadek wydajności transmisji przy małych rozmiarach pakietów. Jest to jeden z powodów wprowadzania serializacji danych i łączenia ich w większe bloki do transmisji. Widoczne jest również osiąganie wartości przepustowości wyższych niż w przełączniku sprzętowym jednak nie przekraczających znacząco wartości teoretycznej.

[38] https://www.isoc.org/inet98/proceedings/6g/6g_3.htm#micetroops, (Dostęp 18-01-2016)

Tabela 5. Przepustowość łącza w funkcji wartości MTU

MTU	Pasmo teoretyczne Mbps	Qoslink Mbps	Switch Mbps
1500	94,92	94,10	94,10
1305	94,19	93,40	93,20
1094	93,10	92,50	92,00
918	91,84	91,40	90,60
772	90,37	90,10	88,90
650	88,66	88,60	86,90
548	86,68	86,80	84,50
463	84,43	84,70	81,90
393	81,90	82,40	79,10
434	79,03	79,70	75,50
285	75,85	76,60	72,00
245	72,34	73,00	68,00
210	68,55	70,60	63,70
181	64,38	64,80	58,80
158	60,20	61,70	54,00
138	55,68	54,20	48,70
128	53,01	52,30	45,70

Rysunek 30. Przepustowość łącza w funkcji zmiennego rozmiaru MTU – protokół TCP/IP

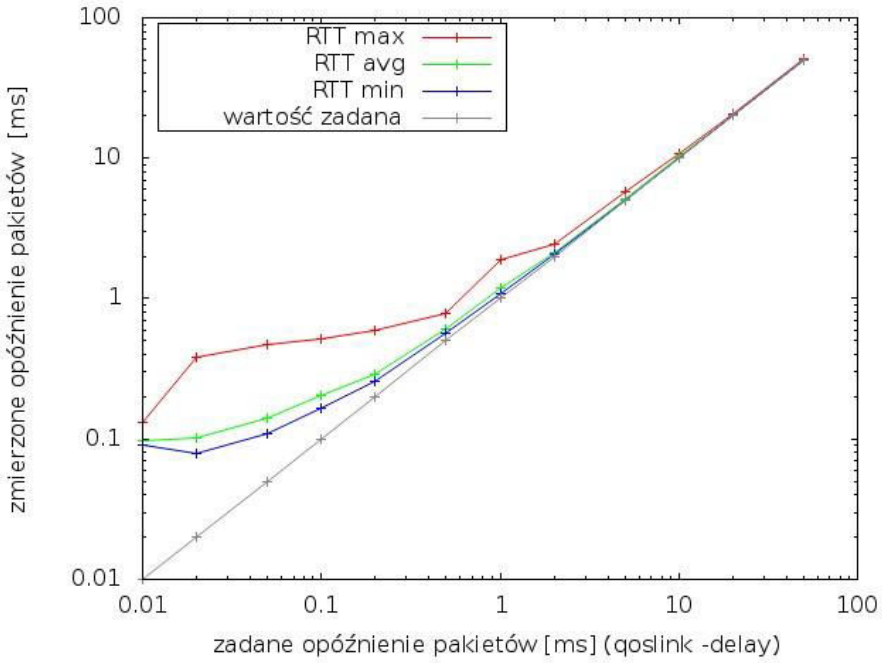


4.2.2 Opóźnienie

Testy opóźnień przeprowadzane były na łączu o symulowanym paśmie 100mbit. Za pomocą polecenia ping mierzony był całkowity czas przesłania pakietu od nadawcy do odbiorcy i z powrotem tj czas RTT. Na rysunku 31 w początkowym zakresie

pomiarów widoczna jest stała czasowa rzędu ok. 0.1ms. Związana jest ona z opóźnieniami wprowadzanymi przez samo łącze qoslink. Wraz ze wzrostem testowanego opóźnienia stała czasowa ma coraz mniejszy wpływ na wartość całkowitego opóźnienia. Przy wartości rzędu 2ms rozbieżności czasowe w pomiarach można uznać za pomijalne.

Rysunek 31. Opóźnienie łącza - skala logarytmiczna



Aby zorientować się na ile istotne jest to opóźnienie, przeprowadziłem również test przełącznika sprzętowego 100mbps. Wyniki przedstawione są w **Tabela 6**. Widoczne kilkukrotnie mniejsze średnie czasy łącza wirtualnego qoslink, umożliwiają swobodne symulowanie połączeń w sieciach lokalnych.

Tabela 6. Opóźnienie qoslink oraz switcha - pasmo 100Mbps

RTT	Switch sprzętowy [ms]	łącze Qoslink [ms]
min	0,25	0,069
avg	0,428	0,112
max	0,627	0,302

4.2.3 Utrata pakietów

Parametr określający procentową utratę pakietów wykorzystywany będzie zwłaszcza przy symulacji zawodnych łączy np. WiFi. W rzeczywistości sytuacja taka

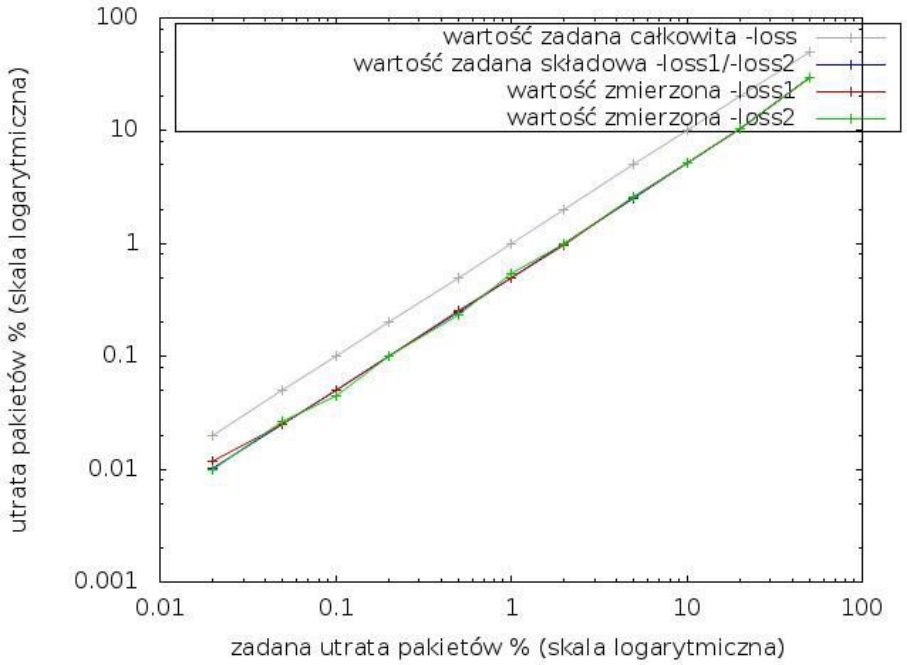
występuje gdy duża liczba sieci bezprzewodowych na danym obszarze konkuruje ze sobą na tych samych kanałach komunikacyjnych.

Pierwsze testy przeprowadzane były programem iperf za pomocą protokołu UDP. Analizowane były oba kierunki transmisji niezależnie za pomocą poniższego polecenia:

```
# ./iperf -c 10.0.0.2 -u -b 100mbit -r -t x
```

Wyniki przedstawione są na rysunku 32. Wartości „-loss1" i „-loss2" otrzymane z pomiarów zbieżne są z wartościami obliczonymi według wzoru (5.1) z rozdział 5.3.2. Na wykresie o skali logarytmicznej widoczne jest prawidłowe działanie skryptu w szerokim zakresie zmian parametrów "-loss".

Rysunek 32. Utrata pakietów w obu kierunkach niezależnie - iperf



Drugie testy wykonane były za pomocą polecenia ping.

```
# ping x.x.x.x -f -c "liczba testowych pakietów"
```

Aby pomiary były wiarygodne liczba pakietów, powinna być odpowiednio duża w stosunku do poszczególnych prawdopodobieństw utraty pakietów. Dokonując serii testów ustaliłem, iż 100-krotność ilości pakietów, w której statystycznie występuje utrata jednego pakietu dla danej wartości parametru "-loss", jest wystarczająca dla

dokładności obliczeń. Dalsze zwiększanie liczby pakietów nie poprawiało znacząco dokładności testów.

$$liczba\ pakietów\ testowych = \frac{100\%}{loss\%} \cdot 100 \qquad (6.5)$$

Wykres z rysunku 33 przedstawia zmierzoną wypadkową wartość utraty pakietów na podstawie danych z tabeli 7. Na wynik ten składa się działanie dwóch interfejsów o parametrach loss1 i loss2. Średnia wartość rozbieżności na poziomie ok. 4% potwierdza poprawność obliczeń na podstawie wzoru (5.1) oraz prawidłowość działania modułu netem podłączonego do kolejek qdisc na interfejsach w kontenerze qoslink.

Rysunek 33. Utrata pakietów - wartość wypadkowa z obu kierunków – ping

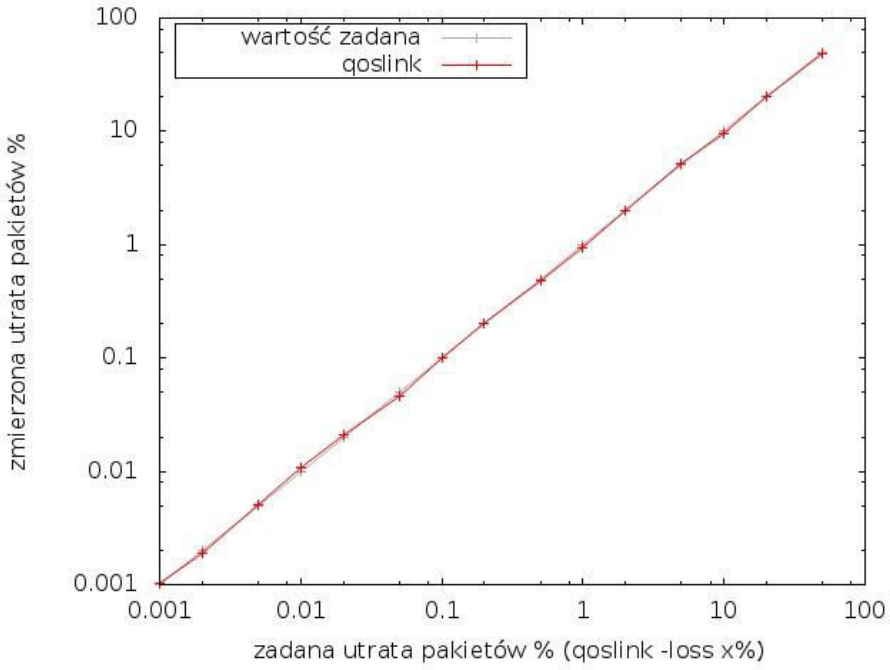


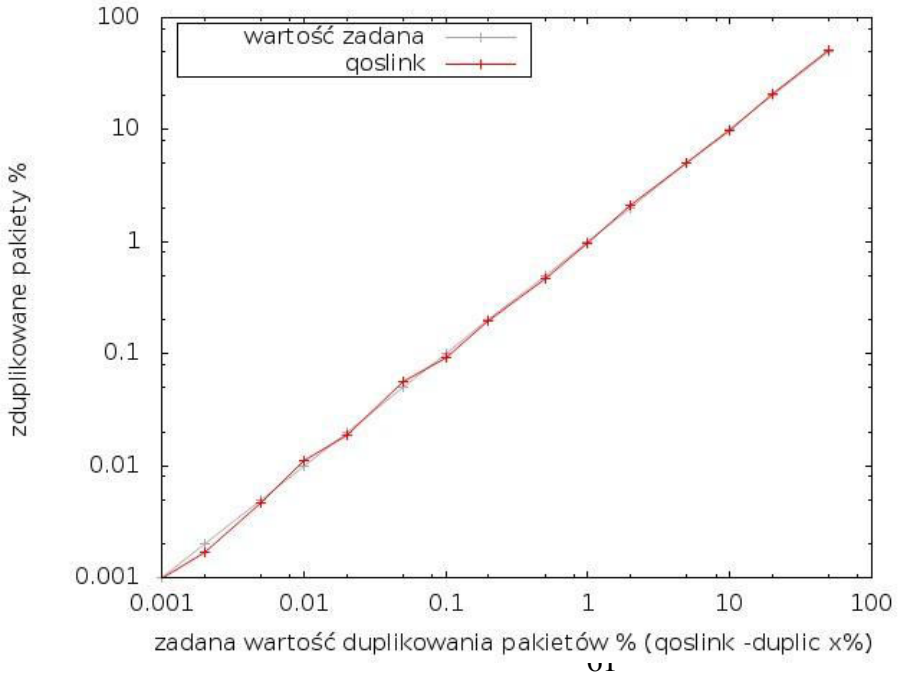
Tabela 7. Utrata pakietów – testy ping

Wart. Zadana [%] - loss	Wart. obliczona [%] -loss1/-loss2	Wart. Zmierzona [%]	Róznica [%]
0,001	0,00060	0,00103	3,00%
0,002	0,00110	0,00192	4,00%
0,005	0,00260	0,00508	1,60%
0,010	0,00510	0,01090	9,00%
0,020	0,01010	0,02100	5,00%
0,050	0,02510	0,04600	8,00%
0,100	0,05010	0,10300	3,00%
0,200	0,10010	0,20200	1,00%
0,500	0,25040	0,48000	4,00%
1,000	0,50130	0,94000	6,00%
2,000	1,00510	2,04000	2,00%
5,000	2,53000	5,30000	6,00%
10,000	5,13000	9,70000	3,00%
20,000	10,56000	20,40000	2,00%
50,000	29,29000	49,00000	2,00%
		Średnia =	3,97%

4.2.4 Duplikowanie pakietów

Testowanie duplikowania pakietów przeprowadzane było analogicznie do poprzedniego testu - poleceniem ping. Również w tym przypadku widoczne jest na wykresie z rysunku 34 prawidłowe odwzorowanie w szerokim zakresie charakterystyki zadanego parametru.

Rysunek 34. Duplikowanie pakietów



4.3 Test sieci LAN

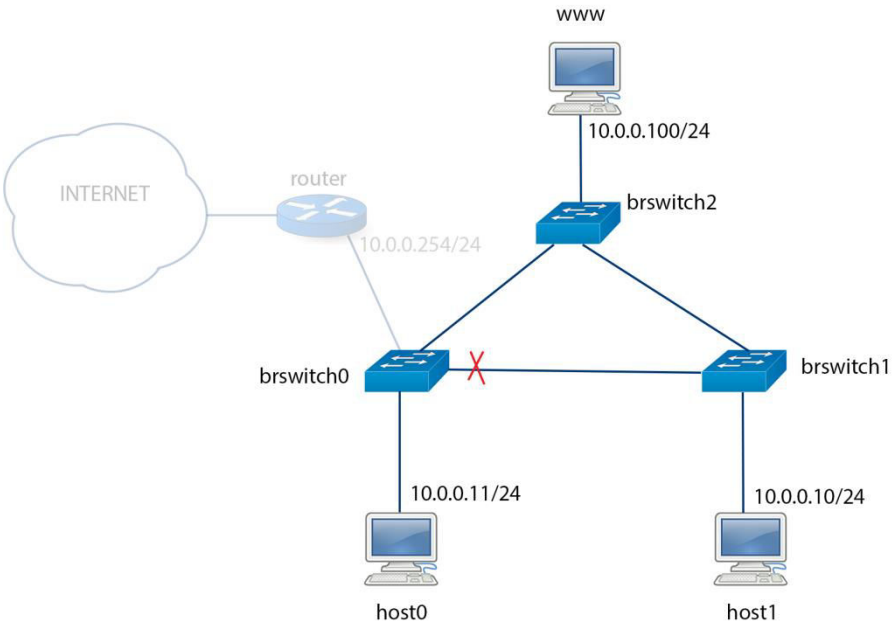
W części tej przedstawię symulację sieci LAN oraz działanie protokołu STP. Sposób konfiguracji podstawowej sieci za pomocą skryptu qoslink. Przetestuję wzajemną dostępność hostów i transmisję danych pomiędzy nimi.

4.3.1 Konfiguracja sieci – LAN

Do utworzenia hostów w symulowanej sieci LAN wykorzystane zostały domyślne kontenery przechowywane w repozytorium: chefronpc/host:v1. Jeden z nich będzie pełnił funkcję serwera WWW, natomiast pozostałe jako komputery klienckie. Testową sieć tworzą trzy przełączniki połączone ze sobą w formę trójkąta, umożliwiając nadmiarowość połączeń w przypadku awarii jednego z nich. Utworzenie sieci przedstawionej na rysunku 35 przeprowadzamy za pomocą poniższych poleceń:

```
# ./qoslink –h1 host0 –sw2 brswitch0 –ip1 10.0.0.10/24 –gw1
# ./qoslink –h1 host1 –sw2 brswitch1 –ip1 10.0.0.11/24 –gw1
# ./qoslink –sw1 brswitch0 –sw2 brswitch1 –ip1 10.0.0.3/24
# ./qoslink –sw1 brswitch0 –sw2 brswitch2 –ip1 10.0.0.4/23
# ./qoslink –sw1 brswitch1 –sw2 brswitch2 –ip1 10.0.0.5/24
# ./qoslink –sw1 brswitch2 –h2 www –ip2 10.0.0.100/24 –gw2
```

Rysunek 35. Struktura wirtualnej sieci LAN



Pomimo testów w tym punkcie wyłącznie sieci LAN, udostępniono połączenie do Internetu dla hostów w wirtualnej sieci. Wymagane to było ze względu na instalowanie w nich dodatkowego oprogramowania. Połączenie to utworzono za pomocą poniższego polecenia, a omówionego w dalszej części pracy.

```
# ./qoslink -sw1 brswitch0 -ph2 -ip2 10.0.0.254/24
```

Na potrzeby testów zainstalowany i uruchomiony został serwer Apache wraz z plikiem testowym w kontenerze „www”.

```
# docker exec www yum –y install httpd
# docker exec www sed –e ‘/#ServerName/c\ServerName www.localhost.pl
      80’ /etc/httpd/conf/httpd.conf
# docker exec www dd if=/dev/zero of=/var/www/html/file_200MB bs=1024
      count=204800
# docker exec www /usr/sbin/apachectl
```

4.3.2 Testy dostępności hostów i usług w sieci

Po utworzeniu sieci warto upewnić się czy wszystkie hosty są prawidłowo podłączone i mogą komunikować się ze sobą wzajemnie. Efektywnym sposobem na to

jest uruchomienie skanera sieciowego. Jednym z nich jest Nmap[39], umożliwiającym również wykrywanie usług uruchomionych na poszczególnych hostach. Na skróconym raporcie z rysunku 36 widać dostępne adresy IP poszczególnych hostów oraz uruchomioną usługę na porcie 80 pod numerem IP 10.0.0.100.

Uwidacznia się w tym momencie bardzo ważna właściwość konfigurowanej sieci. Dotyczy ona uruchamianych usług w kontenerach docker. Dzięki zastosowaniu połączeń za pomocą skryptu pipework, usługi nie wymagają mapowania portów pomiędzy kontenerami. Wirtualna sieć staje się dla nich jednolicie dostępna i funkcjonalna jak w rzeczywistości.

Rysunek 36. Raport skanowania sieci – Nmap

```
[root@host1 /]# nmap 10.0.0.0/24
Starting Nmap 6.40 ( http://nmap.org ) at 2016-02-20 22:42 UTC

Nmap scan report for 10.0.0.1
Host is up (0.00045s latency).
All 1000 scanned ports on 10.0.0.1 are closed
MAC Address: BE:FA:2C:BE:BF:97 (Unknown)

Nmap scan report for 10.0.0.2
Nmap scan report for 10.0.0.3
Nmap scan report for 10.0.0.4
Nmap scan report for 10.0.0.5
Nmap scan report for 10.0.0.6
Nmap scan report for 10.0.0.7
Nmap scan report for 10.0.0.10

Nmap scan report for 10.0.0.100
Host is up (0.00066s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
80/tcp open  http
MAC Address: 42:B5:0D:E6:2E:C6 (Unknown)

Nmap scan report for 10.0.0.254
Nmap done: 256 IP addresses (11 hosts up) scanned in 2.93 seconds
```

Po sprawdzeniu dostępności hostów mamy możliwość pobrania z serwera pliku testowego o rozmiarze 200MB. Prędkość pobierania pliku ograniczona jest przepustowością sieci do 100mbit co potwierdza raport programu wget przedstawiony na rysunku 37. Po przeliczeniu jednostek uzyskujemy prędkość zgodną z tą otrzymaną w testach łącza.

$$Przepustowość = \frac{11{,}4\,MB \cdot 1024 \cdot 1024 \cdot 8}{10^6} = 95{,}63Mbps$$

[39] https://nmap.org/, (Dostęp 12-2015)

Rysunek 37. Pobieranie pliku z serwera – sieć LAN

```
[root@host0 /]# wget 10.0.0.100/file_200MB
--2016-02-21 14:50:13--  http://10.0.0.100/file_200MB
Connecting to 10.0.0.100:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 209715200 (200M)
Saving to: 'file_200MB'

100%[======================================>] 209,715,200 11.4MB/s   in 18s

2016-02-21 14:50:30 (11.4 MB/s) - 'file_200MB' saved [209715200/209715200]
```

4.3.3 Testy protokołu STP

W testowej sieci LAN wykorzystując kilka przełączników umożliwilibyśmy powstawanie pętli, co byłoby niekorzystne zwłaszcza dla pakietów rozgłoszeniowych. Mogłoby to spowodować obciążenie sieci i brak możliwości działania w niej. Dzięki zastosowaniu protokołu STP następuje analiza sieci i utworzenie w niej drzewa rozpinającego poprzez zablokowanie odpowiednich portów przełączników. Na schemacie sieci z rysunku 35 zaznaczony jest na czerwono zablokowany port przełącznika brswitch0 przerywający powstałą pętlę. Potwierdzeniem jest poniższy raport konfiguracji przełącznika.

Rysunek 38. Stan portów switcha „brswitch0" po uzgodnieniu drzewa rozpinającego

```
[root@localhost qoslink]# brctl showstp brswitch0
brswitch0
 bridge id              8000.8624c1da3131
 designated root        8000.1e7fca6b3a33
 root port                 3                    path cost                  2
 max age                  20.00                 bridge max age            20.00
 hello time                2.00                 bridge hello time          2.00
 forward delay            15.00                 bridge forward delay      15.00
 ageing time             300.00
 hello timer               0.00                 tcn timer                  0.00
 topology change timer     0.00                 gc timer                 235.75
 flags


veth1pl3918 (2)
 port id                8002                    state                  blocking
 designated root        8000.1e7fca6b3a33       path cost                  2
 designated bridge      8000.52b7c92a011a       message age timer         19.30
 designated port        8002                    forward delay timer        0.00
 designated cost           2                    hold timer                 0.00
 flags

veth1pl6427 (3)
 port id                8003                    state                forwarding
 designated root        8000.1e7fca6b3a33       path cost                  2
 designated bridge      8000.1e7fca6b3a33       message age timer          1.32
 designated port        8001                    forward delay timer        0.00
 designated cost           0                    hold timer                 0.00
 flags

veth2pl31671 (1)
 port id                8001                    state                forwarding
 designated root        8000.1e7fca6b3a33       path cost                  2
 designated bridge      8000.8624c1da3131       message age timer          0.00
 designated port        8001                    forward delay timer        0.00
```

Wystąpienie sytuacji awaryjnej w jednym z połączeń powoduje wykrycie tego faktu przez protokół. Symulujemy to poprzez ustawienie 100% utraty pakietów pomiędzy przełącznikiem brswitch0 a brswitch2.

```
# ./qoslink -U brswitch0:brswitch2 -loss 100%
```

Skutkiem tego jest ponowne uruchomienie analizy sieci i próba obejścia uszkodzonego połączenia. W trakcie tego procesu zablokowany wcześniej port przechodzi w tryb „listening" nasłuchując informacji o dostępności pozostałych przełączników w sieci i ich kosztach trasy do mostu głównego. Gdy port nasłuchujący będzie miał najkorzystniejszą ścieżkę, przechodzi w tryb „learning", w którym następuje uczenie się adresów MAC znajdujących się w sieci. Po określonym czasie określonym przez parametr „forward delay" następuje ponowne przejście w tryb przekazywania pakietów – „forwarding" [40].

4.4 Test protokołu routingu OSPF

W poprzednim rozdziale testowana była sieć o jednolitej adresacji. Nie zapewnia to możliwości tworzenia odpowiednich warunków do testowania komunikacji pomiędzy aplikacjami. Zatem kolejnym krokiem będzie łączenie różnych sieci za pomocą routerów. Zastosowanie protokołu routingu OSPF umożliwi dostosowywanie się do ewentualnych zmian jakości pracy łączy, aby zachować transmisję danych na jak najlepszym poziomie.

4.4.1 Konfiguracja sieci – protokół OSPF

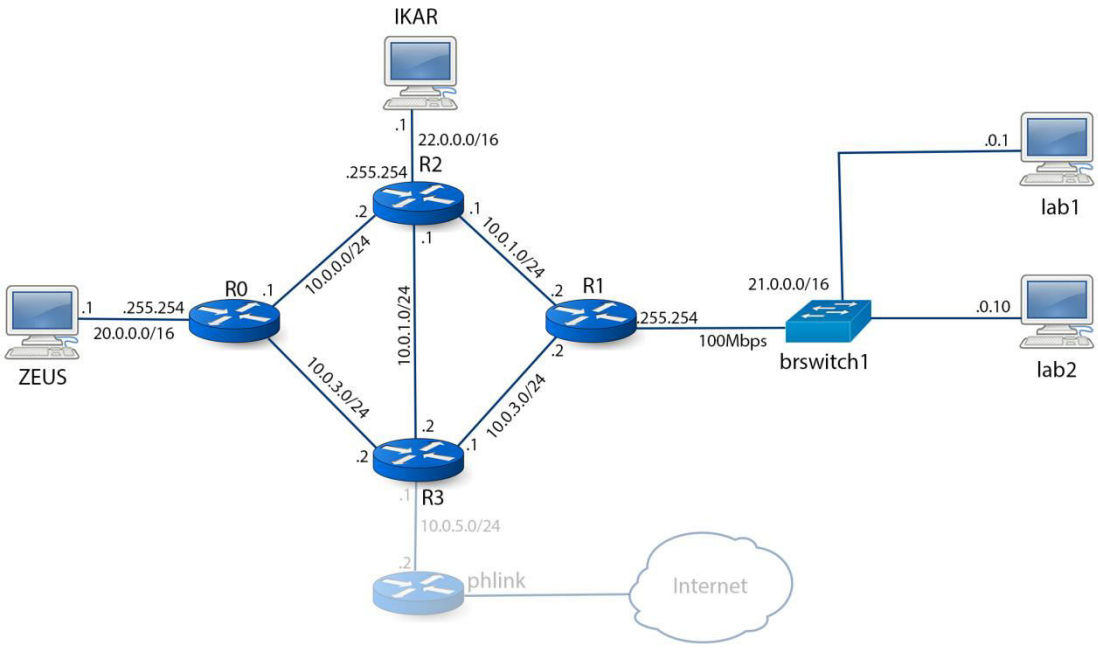
W bieżącym eksperymencie zastosowano cztery routery, dwa serwery www oraz dwa hosty. Struktura sieci z rysunku 39 umożliwia analizę wpływu jakości łącz na transmisje danych z serwera oraz wzajemne oddziaływanie dwóch niezależnych strumieni danych wytworzonych pomiędzy serwerami „Zeus" i „Ikar" a hostami „Lab1" i „Lab2". Ich konfiguracja jest identyczna jak w poprzedniej testowej sieci. Hosty podłączone zostały do routerów poprzez standardową sieć 100Mbps, natomiast

[40] http://cisco.sadzer.pl/rip/stp/, (Dostep 02-2016)

wzajemne połączenia routerów za pomocą wolniejszych łącz z różnymi opóźnieniami. Ma to na celu ustalenie na wstępie różnych kosztów tras routingu. Wartości parametrów zawarte są w poleceniach konfigurujących sieć przedstawionych poniżej:

```
# ./qoslink -h1 zeus -gw1 -ip1 20.0.0.1/16 -r2 R0 -ip2 20.0.255.254/16
# ./qoslink -h1 ikar -gw1 -ip1 22.0.0.1/16 -r2 R2 -ip2 22.0.255.254/16
# ./qoslink -h1 lab1 -gw1 -ip1 21.0.0.1/16 -sw2 brswitch1
# ./qoslink -h1 lab2 -gw1 -ip1 21.0.0.10/16 -sw2 brswitch1
# ./qoslink -sw1 brswitch1 -r2 R1 -ip2 21.0.255.254/16
# ./qoslink -r1 R0 -r2 R2 -band 16mbit –delay 4ms
# ./qoslink -r1 R2 -r2 R1 -band 8mbit –delay 6ms
# ./qoslink -r1 R0 -r2 R3 -band 16mbit –delay 4ms
# ./qoslink -r1 R3 -r2 R1 -band 4mbit –delay 2ms
# ./qoslink -r1 R2 -r2 R3 -band 16mbit –delay 2ms
# ./qoslink -r1 R3 -ph2 phlink
```

Rysunek 39. Struktura sieci – protokół OSPF



Po uruchomieniu sieci, logując się za pomocą terminala zintegrowanego na router R1 możemy sprawdzić działanie protokołu OSPF.

```
# vtysh
R1# show ip route
```

Na przykładowym raporcie z rysunku 40 zaznaczone sieci tworzą kompletną listę tych istniejących w naszej wirtualnej sieci. Widoczne są interfejsy bezpośrednio podłączone do routera oraz dalsze sieci dostępne poprzez wytypowane interfejsy na podstawie algorytmu wyszukiwania optymalnej ścieżki. Do niektórych sieci jak widać na raporcie, może prowadzić kilka tras, ze względu na ich porównywalny koszt. Potwierdza to poprawność działa wszystkich routerów i ich interfejsów.

Rysunek 40. Tablica routingu – router R1

```
R1# show  ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       I - ISIS, B - BGP, > - selected route, * - FIB route

O>* 0.0.0.0/0 [110/1] via 10.0.3.1, eth3, 22:46:03
O>* 10.0.0.0/24 [110/20] via 10.0.1.1, eth2, 16:47:12
O   10.0.1.0/24 [110/10] is directly connected, eth2, 16:47:12
C>* 10.0.1.0/24 is directly connected, eth2
O>* 10.0.2.0/24 [110/20] via 10.0.3.1, eth3, 22:46:04
O   10.0.3.0/24 [110/10] is directly connected, eth3, 1d10h42m
C>* 10.0.3.0/24 is directly connected, eth3
O>* 10.0.4.0/24 [110/20] via 10.0.1.1, eth2, 16:47:12
  *                      via 10.0.3.1, eth3, 16:47:12
O>* 10.0.5.0/24 [110/20] via 10.0.3.1, eth3, 22:46:04
O>* 20.0.0.0/16 [110/30] via 10.0.1.1, eth2, 16:47:12
  *                      via 10.0.3.1, eth3, 16:47:12
O   21.0.0.0/16 [110/10] is directly connected, eth1, 1d10h43m
C>* 21.0.0.0/16 is directly connected, eth1
O>* 22.0.0.0/16 [110/20] via 10.0.1.1, eth2, 16:47:12
C>* 127.0.0.0/8 is directly connected, lo
O>* 172.17.0.0/16 [110/30] via 10.0.3.1, eth3, 22:46:04
```

4.4.2 Testy dostępności hostów w sieci – ping, mtr

Na bazie poprawnie działającego routingu można wykonać dalsze testy polegające na dostępności poszczególnych hostów w testowej sieci. Sprawdzamy dostępność serwerów „ZEUS” i „IKAR” z hostów „LAB1” oraz „LAB2” za pomocą komendy ping. Na raporcie z rysunku 41 warto zwrócić uwagę na dwa różne czasy odpowiedzi na zapytania z hosta „LAB1”. Spowodowane jest to dostępnością dwóch różnych tras do zdalnego hosta.

Rysunek 41. Dostępność serwerów - ping

```
[root@localhost qoslink]# docker exec lab1 ping 20.0.0.1
PING 20.0.0.1 (20.0.0.1) 56(84) bytes of data.
64 bytes from 20.0.0.1: icmp_seq=1 ttl=61 time=6.67 ms
64 bytes from 20.0.0.1: icmp_seq=2 ttl=61 time=10.5 ms
64 bytes from 20.0.0.1: icmp_seq=3 ttl=61 time=6.55 ms
64 bytes from 20.0.0.1: icmp_seq=4 ttl=61 time=10.5 ms

[root@localhost qoslink]# docker exec lab2 ping 22.0.0.1
PING 22.0.0.1 (22.0.0.1) 56(84) bytes of data.
64 bytes from 22.0.0.1: icmp_seq=1 ttl=62 time=6.61 ms
64 bytes from 22.0.0.1: icmp_seq=2 ttl=62 time=6.48 ms
64 bytes from 22.0.0.1: icmp_seq=3 ttl=62 time=6.49 ms
64 bytes from 22.0.0.1: icmp_seq=4 ttl=62 time=6.49 ms
```

Potwierdzeniem tej sytuacji jest sprawdzenie dostępnych tras z hosta „LAB1” do serwera „ZEUS” za pomocą polecenia mtr. Również w tym wypadku widoczna jest na rysunku 42 transmisja obiema trasami bez żadnych strat pakietów.

Rysunek 42. Dostępne trasy do serwera

```
                        My traceroute  [v0.85]
lab1 (0.0.0.0)                                  Wed Feb 24 11:12:31 2016
Keys:  Help   Display mode   Restart statistics   Order of fields   quit
                                 Packets               Pings
 Host                          Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. 21.0.255.254               0.0%    57    0.4   0.4   0.3   0.5   0.0
 2. 10.0.3.1                   0.0%    57    5.9   2.8   2.3  10.1   1.3
    10.0.1.1
 3. 10.0.0.1                   0.0%    57    8.5   8.5   8.4   8.6   0.0
    10.0.2.1
 4. 20.0.0.1                   0.0%    57   10.6  10.4   6.5  10.6   0.6
```

4.4.3 Testy transmisji danych

Testy wykonywane do tej pory nie odzwierciedlały w pełni właściwości tworzonej sieci wirtualnej. Parametry, które będą nas interesować będzie można zaobserwować dopiero w przypadku obciążenia sieci odpowiednim strumieniem danych. Do przeprowadzenia kolejnego testu wykorzystany zostanie ponownie serwer WWW „ZEUS” z którego będzie pobierany plik do hosta „LAB1”.

Rysunek 44 przedstawia przebieg transmisji danych przez poszczególne segmenty sieci. Wykres „Router R0-ZEUS” jest to całkowity transfer danych odczytywany bezpośrednio z wyjścia serwera. Drugim punktem monitorowania sieci jest router R1 z przyłączonymi do niego dwoma segmentami sieci R1-R3 oraz R1-R2. Ostatni wykres to transfer w segmencie sieci pomiędzy routerami R2-R3. W trakcie transmisji danych następowały zmiany w parametrach połączeń co powodowało konieczność dostosowywania się routerów do nich i szukania nowych optymalnych tras.

W pierwszym etapie transmisji strumień danych rozdzielał się symetrycznie w routerze R0 zgodnie z dostępnymi trasami routingu:

Rysunek 43. Trasy dostępne z routera R0 do sieci 21.0.0.0/16 (Host LAB1)

```
[root@R0 /]# ip route
21.0.0.0/16  proto zebra  metric 30
        nexthop via 10.0.0.2  dev eth2 weight 1
        nexthop via 10.0.2.2  dev eth3 weight 1
```

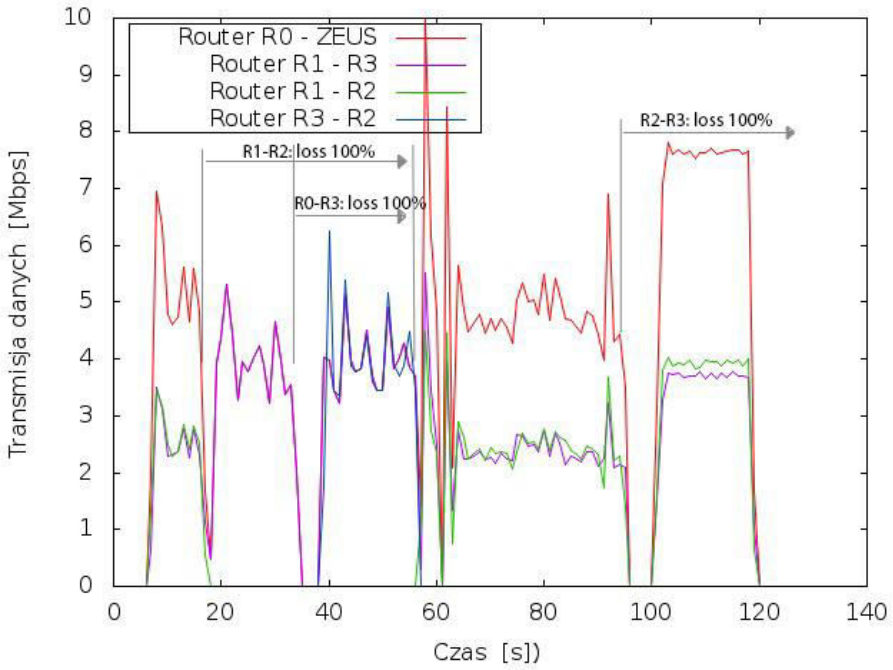
W czasie 20-stej sekundy zasymulowana została awaria łącza pomiędzy routerami R1-R2 poleceniem:

./qoslink –U R1:R2 –loss100%

Spowodowało to tylko częściowy spadek całkowitego transferu na rzecz łącza R1-R3. Awaria drugiego połączenia R0-R3 i ostatniej znanej trasy pomiędzy hostami spowodowała kilkusekundowe zatrzymanie przesyłu danych. Długość tej przerwy to wynik czasu wymaganego przez routery do odświeżenia informacji o nowym stanie sieci. Po tym czasie transmisja zostaje wznowiona przez ostatnie dostępne i niewykorzystywane do tej pory łącze pomiędzy routerami R2-R3 utrzymując dotychczasowy transfer. Po usunięciu awarii obu połączeń routery otrzymują ponownie pakiety rozgłoszeniowe HELLO OSPF od swoich sąsiadów. Na tej podstawie wznawiają transmisje na dwóch poprzednich trasach, ponieważ mają one w tablicy routingu przypisane mniejsze koszty.

Należy zwrócić uwagę na sposób dzielenia strumienia danych pomiędzy różne trasy. Pomimo różnych maksymalnych transferów tras, dla ścieżki R0-R2-R1 wynosi 8Mbit, a dla R0-R3-R1 wynosi 4mbit, to i tak router dzielił pakiety naprzemiennie wysyłając dwa strumienie pakietów o paśmie 4mbit. Bardzo dobrze widoczne jest to na poniższym wykresie w czasie od setnej do sto dwudziestej sekundy.

Rysunek 44. Przebieg transmisji danych - protokół OSPF



4.5 Testy złożonych sieci - routery OSPF (ABR/ASBR) + KVM/QEMU

Przedstawiany tutaj skrypt będzie spełniał swoje funkcje jeżeli będzie go można wykorzystać nie tylko w przypadku kontenerów Docker. Możliwość podpięcia innych systemów wirtualnych zdecydowanie zwiększa jego praktyczność. Elementem wspólnym systemów wirtualnych jest możliwość łączenia się z sieciami zewnętrznymi poprzez mosty. Dzięki nim w łatwy sposób możemy łączyć te systemy poprzez naszą sieć. W punkcie tym przedstawione zostanie podłączanie systemów opartych na wirtualizacji KVM (ang. Kernel Virtual Machine).

Cechą skryptu qoslink jest możliwość bezpośredniej konfiguracji sieci wyłącznie w kontenerach Docker. Natomiast inne systemy można łączyć z naszą siecią poprzez mosty. W przypadku maszyn wirtualnych KVM można je uruchamiać z interfejsami skonfigurowanymi w trybie mostu pod warunkiem, że one w systemie już działają. Wynika z tego konieczność uruchomienia na wstępie sieci wirtualnej, a dopiero później podłączania do niej maszyn wirtualnych KVM. Aby jednak przyszłe rekonfiguracje naszej sieci a tym bardziej usuwanie i ponowne jej wczytywanie nie zakłócało konfiguracji sieciowych zewnętrznych systemów, wskazane jest użycie innych nazw przełączników niż domyślne „brswitch”. W sieci testowej użytej w tym rozdziale utworzono dodatkowe mosty o nazwach „brwin1” i „brwin2”, w skutek czego podłączone do nich maszyny wirtualne KVM z systemami Windows są stale gotowe do pracy w różnych konfiguracjach sieci.

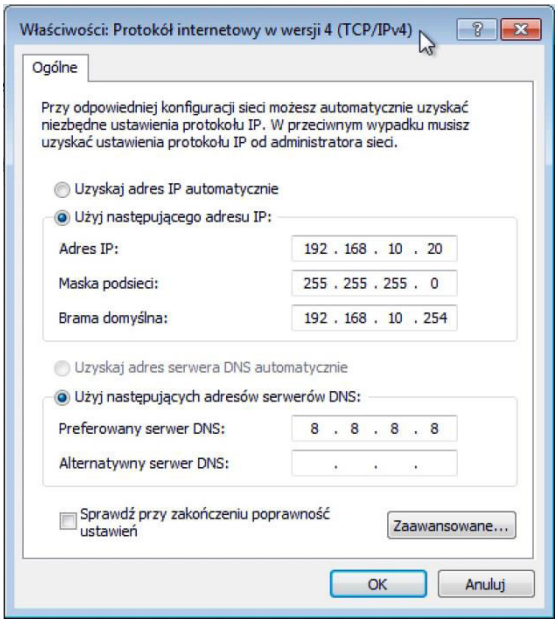
4.5.1 Testy sieci WAN

W części tej przedstawię symulację bardziej rozbudowanej sieci mogącej odwzorowywać wzajemne połączenie np. dwóch oddziałów firmy, z ich jednoczesnym dostępem do sieci Internet. Konfiguracja ta może być podstawą testów aplikacji typu klient-serwer, uwzględniającą rzeczywiste parametry łączy internetowych. Przetestuję wzajemną dostępność hostów oraz transmisję danych pomiędzy serwerem a komputerami klienckimi znajdującymi się w różnych oddziałach. Wykaże również wpływ jakości łączy na testową transmisję danych.

4.5.2 Konfiguracja emulowanej sieci WAN

W prezentowanym przykładzie do symulowania niezależnych sieci LAN połączonych poprzez łącza internetowe wykorzystano możliwości protokołu OSPF. Dzięki tworzeniu osobnych obszarów OSPF ograniczono działanie algorytmu SPF (and. Shortest Path Firts) do symulowanych sieci LAN w zakresie danego oddziału. W tym przykładzie jest to uproszczone ponieważ każdy z oddziałów używa po jednym routerze dostępowym odpowiednio R1 i R2, a adresacja LAN jest ograniczona do jednej podsieci. Natomiast symulacja łączy dostępowych WAN przeniesiona jest do obszaru zero jako połączenia routerów R1-R0 oraz R2-R0. Jako systemy testowe w sieci wykorzystane zostały domyślne kontenery przechowywane w repozytorium: chefronpc/host:v1. Jeden z nich będzie pełnił funkcję serwera WWW pod nazwą „zeus", natomiast drugi jako komputer kliencki o nazwie „lab2". Uruchomione zostały również maszyny wirtualne KVM z systemami Windows 7 jako „Win7_1" i „Win7_2" z odpowiednio skonfigurowanymi interfejsami sieciowymi. Jeden z interfejsów przedstawiony jest poniżej.

Rysunek 45. Konfiguracja sieciowa maszyny wirtualnej Win7_1



Utworzenie sieci przedstawionej na rysunku 46 przeprowadzone zostało za pomocą poniższych poleceń:

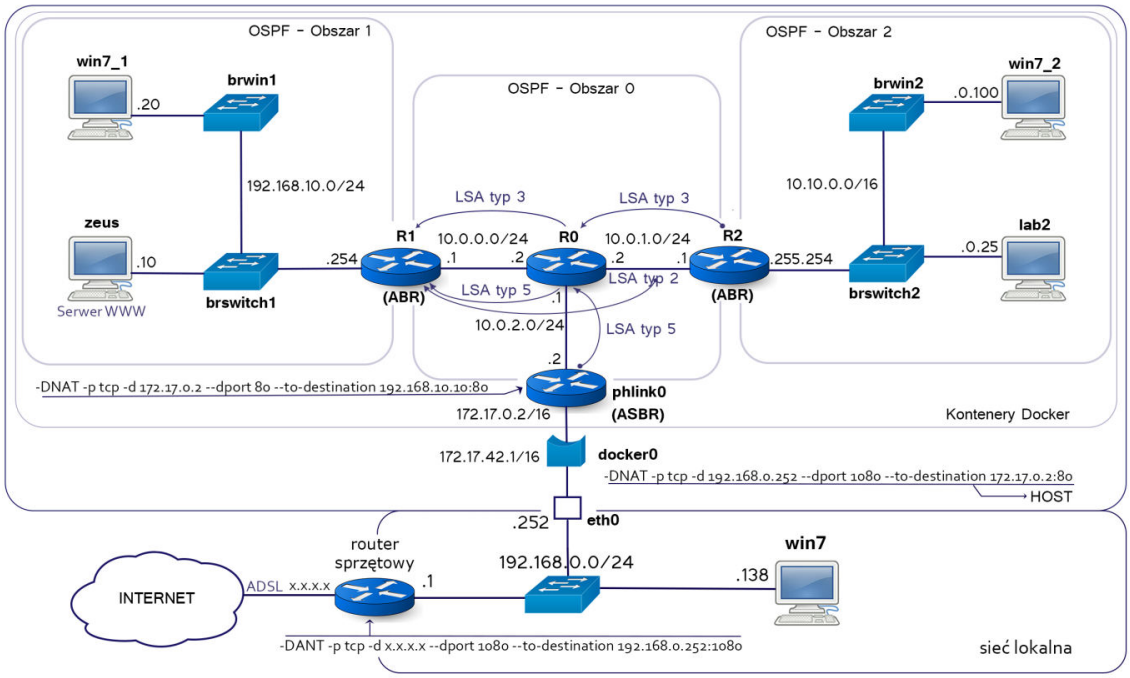
```
#./qoslink -h1 zeus -sw2 brswitch1 -ip1 192.168.10.10/24 -gw1
#./qoslink -sw1 brswitch1 -r2 R1 -ip2 192.168.10.254/24 -A 1
```

```
#./qoslink -sw1 brswitch1 -sw2 brwin1 -ip1 192.168.10.21/24

#./qoslink -h1 lab2 -sw2 brswitch2 -ip1 10.10.0.25/16 -gw1
#./qoslink -sw1 brswitch2 -r2 R2 -ip2 10.10.255.254/16 -A 2
#./qoslink -sw1 brswitch2 -sw2 brwin2 -ip1 10.10.0.10/16

#./qoslink -r1 R1 -r2 R0 -band 8mbit -delay 12ms -loss 0.2% -A 0
#./qoslink -r1 R2 -r2 R0 -band 4mbit -delay 30ms -loss 1% -A 0
#./qoslink -r1 R0 -ph2 -A 0
```

Rysunek 46. Struktura sieci – obszary OSPF



4.5.3 Testy dostępności hostów oraz weryfikacja protokołu OSPF

Na wstępie przetestujemy komunikacje pomiędzy poszczególnymi hostami. Ze względu na uruchomionych kilka różnych zakresów sieci najlepszym sposobem będzie użycie komendy ping. Na rysunku 47 widoczna jest prawidłowa komunikacja pomiędzy hostami „zeus” oraz „lab2”. Dodatkowo w raporcie widoczne jest uzyskanie średnich opóźnień zgodnych z zadanymi $\sim 42ms$ oraz utraty pakietów na poziomie 1,15% w stosunku do zadanego poziomu 1,2%. Zatem błąd utraty pakietów wynosi ~4,3% co mieści się w zakresie wykazanym w punkcie 6.2.3.

Rysunek 47. Komunikacja pomiędzy kontenerami

```
[root@lab2 /]# ping 192.168.10.10 -f -c 10000
PING 192.168.10.10 (192.168.10.10) 56(84) bytes of data.
.....................................................................................
.................................
--- 192.168.10.10 ping statistics ---
10000 packets transmitted, 9885 received, 1% packet loss, time 139429ms
rtt min/avg/max/mdev = 42.144/42.212/42.813/0.229 ms, pipe 5, ipg/ewma 13.944/42
.197 ms
```

Podczas testów dostępności hosta wirtualnego Windows „Win7_1" komunikacja działała wyłącznie z zakresu sieci lokalnej, natomiast próba połączenia się z hosta „lab2" kończyła się niepowodzeniem według raportu z rysunku 48. Okazało się, że przyczyną była aktywna zapora systemowa hosta Win7_1 z domyślnymi ustawieniami blokującymi zapytania ICMP z sieci zdalnych. Po wyłączeniu zapory problem ustąpił. Ograniczyłem się do takiego działania, aby mieć pewność, że ten i przyszłe testy nie będą zależne od konfiguracji systemów a jedynie od parametrów testowanej sieci.

Rysunek 48. Komunikacja z systemem wirtualnym Windows

```
[root@lab2 /]# ping 192.168.10.20 -c 1
PING 192.168.10.20 (192.168.10.20) 56(84) bytes of data.
--- 192.168.10.20 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

[root@R1 /]# ping 192.168.10.20 -c 1
PING 192.168.10.20 (192.168.10.20) 56(84) bytes of data.
64 bytes from 192.168.10.20: icmp_seq=1 ttl=128 time=1.04 ms
--- 192.168.10.20 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 1ms
rtt min/avg/max/mdev = 1.046/1.046/1.046/0.000 ms
```

Przetestowana została również dostępność z hosta wirtualnego „lab2" systemów znajdujących się poza wirtualną siecią a mianowicie serwera DNS TP s.a. oraz komputerem w fizycznej sieci lokalnej. Wyniki widoczne są na raporcie poniżej.

Rysunek 49. Komunikacja z zewnętrznymi systemami: dns TP s.a. oraz host w sieci lokalnej

```
[root@lab2 /]# ping 194.204.159.1 -c 1
PING 194.204.159.1 (194.204.159.1) 56(84) bytes of data.
64 bytes from 194.204.159.1: icmp_seq=1 ttl=243 time=41.0 ms
--- 194.204.159.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 41.048/41.048/41.048/0.000 ms

[root@lab2 /]# ping 192.168.0.138 -c 1
PING 192.168.0.138 (192.168.0.138) 56(84) bytes of data.
64 bytes from 192.168.0.138: icmp_seq=1 ttl=124 time=31.2 ms
--- 192.168.0.138 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 31.230/31.230/31.230/0.000 ms
```

Należy zaznaczyć, że ze względu na uruchomiony SNAT na routerze phlink0, inicjowanie połączeń i dostęp do usług uruchomionych wewnątrz sieci wirtualnej z zewnętrznych systemów domyślnie jest niemożliwy. Sytuacja jest analogiczna do sieci podłączonej poprzez router do jednego publicznego adresu IP. Rozwiązaniem problemu niedostępności usług uruchomionych wewnątrz sieci wirtualnej jest konfiguracja translacji adresów docelowych DNAT na odpowiednich routerach. W przypadku omawianej sieci testowej, wpisy do tablicy nat należy wykonać ręcznie kolejno na trzech routerach: wirtualnym „phlink0", systemie hosta oraz w routerze sprzętowym zgodnie z rysunkiem 50.

```
# iptables -t nat -A PREROUTING –p tcp -d 172.17.0.2 --dport 80 -j
       DNAT --to-destination 192.168.10.10:80
# iptables -t nat -A PREROUTING -p tcp -d 192.168.0.252 --dport 1080 -
       j DNAT --to-destination 172.17.0.2:80
```

Rysunek 50. Translacja adresów DNAT – router sprzętowy

Nazwa usługi	Zakres portów	Lokalny IP	Port lokalny	Protokół	Protokół nr	
				TCP		
SDN-zeus-apache	1080	192.168.0.252	1080	TCP		

Dzięki tym wpisom uzyskujemy dostęp z Internetu do naszej usługi WWW w kontenerze zeus poprzez adres http://x.x.x.x:1080. Port inny niż 80 w tym wypadku 1080, wykorzystano ze względu na nie blokowanie stron internetowych dla innych użytkowników sieci lokalnej.

Na poniższych raportach programu mtr widoczne są przebiegi tras do hosta „Win7_1" o adresie 192.168.10.20 wewnątrz sieci wirtualnej

Rysunek 51. Test trasy z hosta „lab2" do „Win7_1

```
                        My traceroute  [v0.85]
lab2 (0.0.0.0)                                   Sat Mar 12 22:08:51 2016
Keys:  Help   Display mode   Restart statistics   Order of fields   quit
                                 Packets                Pings
 Host                          Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. 10.10.255.254               0.0%     6    0.4   1.2   0.4   5.1   1.7
 2. 10.0.1.2                    0.0%     6   30.5  30.5  30.5  30.5   0.0
 3. 10.0.0.1                    0.0%     6   42.6  42.6  42.5  42.6   0.0
 4. 192.168.10.20               0.0%     6   43.4  43.2  43.0  43.4   0.0
```

oraz jednoczesny dostęp do serwera dns TP s.a. w sieci Internet poprzez router brzegowy ASBR o adresie 10.0.2.2. Na rys.46 zaznaczone jest również prawidłowo działające łącze R2:R0 zgodnie z zadanymi parametrami $-loss\ 1\%$ oraz $-delay\ 30ms$.

Rysunek 52. Test trasy z hosta „lab2" do dns TP s.a.

```
                                My traceroute  [v0.85]
lab2 (0.0.0.0)                                          Sat Mar 12 22:17:30 2016
Keys:  Help   Display mode   Restart statistics   Order of fields   quit
                                     Packets               Pings
 Host                              Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. 10.10.255.254                   0.0%   101    0.4   0.5   0.3   4.9   0.6
 2. 10.0.1.2                        1.0%   101   30.5  30.4  30.4  30.6   0.0
 3. 10.0.2.2                        0.0%   100   30.6  30.6  30.5  30.7   0.0
 4. 172.17.0.1                      0.0%   100   30.6  30.6  30.5  30.7   0.0
 5. 192.168.0.1                     0.0%   100   31.2  31.4  31.2  34.5   0.3
 6. srv129.exnor.eu                 1.0%   100   37.8  35.6  33.3  48.6   2.2
 7. srv1.exnor.eu                   0.0%   100   36.2  36.0  33.3  51.7   2.6
 8. rev-217.17.40.121.atman.pl      2.0%   100   40.1  42.9  39.1  69.6   4.6
 9. 80.50.143.89                    0.0%   100   40.9  43.6  39.4  72.3   5.4
10. war-ar3.tpnet.pl                1.0%   100   39.9  43.9  39.4  87.1   7.4
11. 80.50.113.22                    0.0%   100   44.4  42.7  39.7  47.9   1.9
12. dns.tpsa.pl                     0.0%   100   41.8  41.8  40.0  55.2   2.0
```

Po zweryfikowaniu komunikacji pomiędzy hostami, sprawdzona została poprawność działania protokołu OSPF. W tym celu odczytana została tablica routingu jednego z routerów brzegowych R1 łączącego obszar 0 i 1 - rysunek 53. W pierwszej sekcji „network routing" występują wszystkie skonfigurowane przez nas adresy sieci. Dwie z nich są bezpośrednio dołączone do routera (10.0.0.0/24 oraz 192.168.10.0/24). Jedna oznaczona jako „IA" (ang. Internal Area) jest siecią należącą do innego obszaru nienależącego do bieżącego routera. Informacja o tej sieci przekazana została przez router R2 pakietem LSA typu 3. Pozostałe sieci należą do obszarów przyłączonych do routera R1. W drugiej sekcji „router routing" zawarte są routery brzegowe ABR lub ASBR. Ostatnia sekcja skonfigurowana na podstawie pakietu LSA typ 5 wysłanego przez router „phlink" informuje o domyślnej trasie do zewnętrznych sieci.

Rysunek 53. Tablica routingu – router brzegowy ABR

```
R1# show  ip ospf  route
============ OSPF network routing table ============
N    10.0.0.0/24           [10] area: 0.0.0.0
                           directly attached to eth2
N    10.0.1.0/24           [20] area: 0.0.0.0
                           via 10.0.0.2, eth2
N    10.0.2.0/24           [20] area: 0.0.0.0
                           via 10.0.0.2, eth2
N IA 10.10.0.0/16          [30] area: 0.0.0.0
                           via 10.0.0.2, eth2
N    172.17.0.0/16         [30] area: 0.0.0.0
                           via 10.0.0.2, eth2
N    192.168.10.0/24       [10] area: 0.0.0.1
                           directly attached to eth1

============ OSPF router routing table =============
R    10.0.2.2              [20] area: 0.0.0.0, ASBR
                           via 10.0.0.2, eth2
R    10.10.255.254         [20] area: 0.0.0.0, ABR
                           via 10.0.0.2, eth2

============ OSPF external routing table ===========
N E2 0.0.0.0/0             [30/1] tag: 0
                           via 10.0.0.2, eth2
```

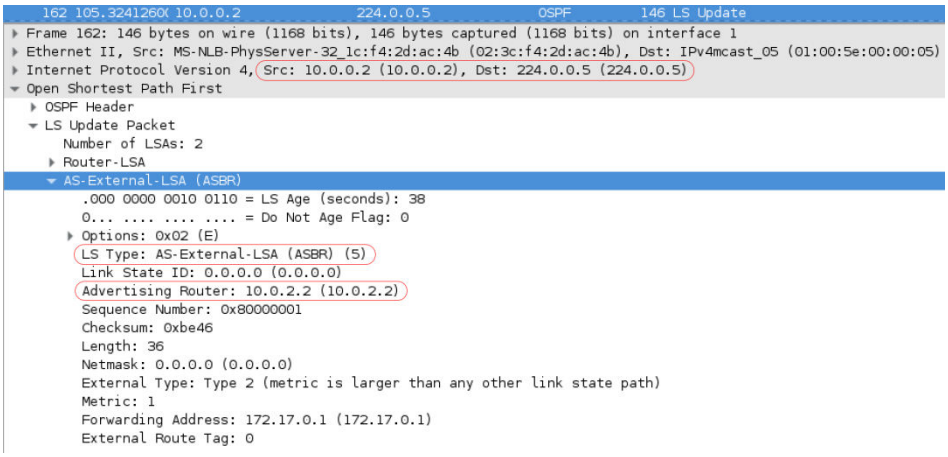
Dokładną analizę można przeprowadzić na podstawie zawartości bazy danych LSDB z rysunku 54. Na wydruku widać dwie grupy wpisów dla dwóch obszarów 0 i 1, które zawierają różne zestawy danych. Związane jest to z różnym widokiem sieci z punktu widzenia dwóch interfejsów routera brzegowego. Od strony interfejsu obszaru 0 widoczne są wszystkie cztery routery (I) należące do sieci szkieletowej wraz z omawianym routerem R1, w sekcji II są trzy adresy sieci z obszaru 0. Inny obraz sieci jest z punktu widzenia obszaru 1. Ze względu na obecność tylko jednego routera R1 w tym obszarze w sekcji IV jest tylko jeden wpis oraz brak informacji o dodatkowych sieciach w tym obszarze. Informacje z komunikatów LSA typu 3 zawarte są w sekcji III i V. W pierwszym przypadku są to sieci z obszarów 1 i 2 widoczne z sieci szkieletowej, natomiast w drugim przypadku to sieci z obszaru 0 i 2 widoczne z obszaru 1. Na końcu przedstawione są dwa wpisy o routerze brzegowym do zewnętrznych systemów wraz adresem domyślnej bramy dla testowanej sieci.

Rysunek 54. Zawartość LSDB protokołu OSPF – router R1

```
R1# show  ip ospf database
       OSPF Router with ID (192.168.10.254)
                Router Link States (Area 0.0.0.0)
Link ID      I  ADV Router      Age  Seq#       CkSum  Link count
10.0.0.2        10.0.0.2        794 0x8000000e 0x7334 3
10.0.2.2        10.0.2.2       1294 0x80000006 0x1f14 2
10.10.255.254   10.10.255.254   794 0x80000008 0xb251 1
192.168.10.254  192.168.10.254 1303 0x80000004 0xd773 1
                Net Link States (Area 0.0.0.0)
Link ID      II ADV Router      Age  Seq#       CkSum
10.0.0.1        192.168.10.254 1303 0x80000002 0x0c51
10.0.1.1        10.10.255.254   790 0x80000003 0x58c1
10.0.2.2        10.0.2.2       1294 0x80000002 0x5dc5
                Summary Link States (Area 0.0.0.0)
Link ID      III ADV Router     Age  Seq#       CkSum  Route
10.10.0.0       10.10.255.254  1624 0x80000002 0x2efc 10.10.0.0/16
192.168.10.0    192.168.10.254  884 0x80000002 0x610b 192.168.10.0/24
                Router Link States (Area 0.0.0.1)
Link ID      IV ADV Router      Age  Seq#       CkSum  Link count
192.168.10.254  192.168.10.254 1339 0x80000004 0xfeed 1
                Summary Link States (Area 0.0.0.1)
Link ID      V  ADV Router      Age  Seq#       CkSum  Route
10.0.0.0        192.168.10.254   914 0x80000002 0xfed6 10.0.0.0/24
10.0.1.0        192.168.10.254  1184 0x80000002 0x5872 10.0.1.0/24
10.0.2.0        192.168.10.254  1044 0x80000002 0x4d7c 10.0.2.0/24
10.10.0.0       192.168.10.254   783 0x80000001 0x5167 10.10.0.0/16
172.17.0.0      192.168.10.254   304 0x80000003 0xb656 172.17.0.0/16
                ASBR-Summary Link States (Area 0.0.0.1)
Link ID      VI ADV Router      Age  Seq#       CkSum
10.0.2.2        192.168.10.254   934 0x80000002 0x2b9b
                AS External Link States
Link ID      VII ADV Router     Age  Seq#       CkSum  Route
0.0.0.0         10.0.2.2       1427 0x80000003 0xba48 E2 0.0.0.0/0 [0x
```

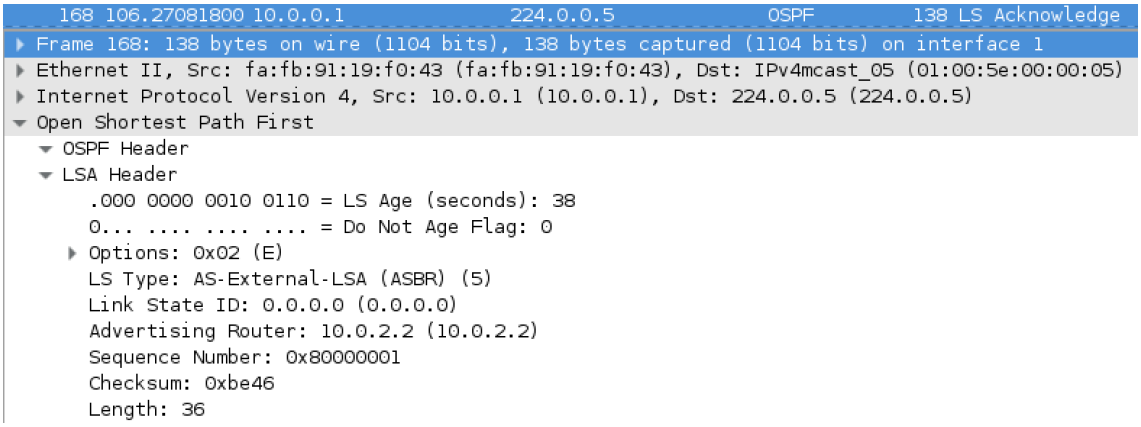
Informacje z rysunku 54 z sekcji VII przekazane zostały komunikatem LSA typ 5 przedstawionym na rysunku 55. Wysłano go z routera „phlink” przez interfejs o adresie 10.0.2.2, a przekazanego dalej w głąb sieci przez interfejs 10.0.0.2 na adres rozgłoszeniowy 224.0.0.5.

Rysunek 55. Komunikat LSA typ 5



W programie Wireshark zarejestrowana została odpowiedź routera R1, który wysłał potwierdzenie przyjęcia komunikatu również na adres rozgłoszeniowy. Szczegóły widoczne są na poniższym rysunku.

Rysunek 56. Potwierdzenie przyjęcia komunikatu LSA typ 5



Podsumowanie

Przeprowadzone praktyczne testy w przedstawionej pracy wykazały możliwość tworzenia elastycznych środowisk testowych dla aplikacji internetowych. Pomimo zaimplementowania rozwiązania wyłącznie w postaci skryptu BASH osiągnięto funkcjonalności obejmujące tworzenie emulowanych łącz o określonych parametrach, rozbudowanych sieci z routingiem OSPF oraz łączenia ze sobą w wspólną sieć różnych systemów wirtualnych. Zapewniono również możliwość zapisu i przenoszenia pełnej konfiguracji na inne systemy co jest podstawą w efektywnych i powtarzających się testach.

Nie zakładaną wcześniej opcją jest udostępnianie do Internetu utworzonych usług wewnątrz wirtualnej sieci. Realna wówczas staje się sytuacja, w której użytkownik konfiguruje wirtualne laboratorium z działającym serwerem i klientami Windows. Połączonych za pomocą emulowanych łączy ADSL/WiFi odzwierciedlającymi na przykład podłączanie się zdalnego pracownika do sieci firmowej. Wymaga to częściowo ręcznej konfiguracji sieci omówionej w rozdziale 4.5.3 udostępniającej zdalne pulpity, jednak umożliwia zdalne testowanie na przykład mobilnych profili użytkowników czy testów replikacji serwera domeny. Wielu użytkowników z pewnością skorzysta również z dużej ilości oprogramowania dostępnego w technologii kontenerów jak serwery WWW (Apache, Nginx) lub oprogramowanie monitorujące ZABBIX, które uruchamiane w kontenerach mogą być bezpośrednio podłączane do wirtualnej sieci.

Pomimo tych zalet występują jednak też słabsze strony tego rozwiązania. Z uwagi na korzystanie z formy skryptu i procedur automatyzujących konfiguracje parametrów o złożoności przekraczającej złożoność logarytmiczną, rozwiązanie to nie jest skalowalne i skierowane jest bardziej do użytku akademickiego i szkoleniowego. Podczas jego projektowania występowały również problemy. Jednym z głównych był sposób przekazywania parametrów do skryptu oraz konieczność podjęcia na ich podstawie decyzji o możliwości połączenia i o ewentualnym jego rodzaju. Wymagało to przeanalizowania wszystkich kombinacji połączeń pomiędzy elementami sieci z tabeli 1. Innym problemem był sam proces konfiguracji łącza przy różnorodności jego typów. Wynikiem tego była konieczność rozdzielenia całego procesy na wiele jednostkowych etapów. Rozwiązanie powyższych problemów zawarte zostało w postaci procedur zamieszczonych w dodatku A.

Elementami, które można poprawić w tym rozwiązaniu to skrócenie czasu usuwania kontenerów z systemu oraz zakodowanie rozwiązania w języku niższego poziomu w celu jego przyspieszenia na przykład w C++. Aktualny projekt ma możliwość dalszego rozwoju. Funkcjonalność, którą można dodać to tworzenie niezależnych sieci jednocześnie przez wprowadzenie obsługi sieci VLAN. Kolejną funkcjonalnością możliwą do dodania to automatyczna konfiguracja dynamicznej translacji adresów w celu udostępniania usług na zewnątrz sieci. Dostępna aktualnie jako konfiguracja ręczna, możliwa byłaby na przykład jako dodatkowa opcja „-E" (export) z parametrami dotyczącymi adresów i portów TCP/IP analogicznie jak w translacji DNAT. Inna zmiana mogłaby dotyczyć wykorzystania przełączników OpenVSwitch zamiast mostów linuksowych co mogłoby zwiększyć wydajność sieci.

Przedstawione rozwiązanie może być zastosowane również w przypadku innych typów wirtualizacji systemów jednak pod warunkiem współpracy z mostami Linux. Prowadzone dodatkowe testy wykazały, że możliwe jest łączenie naszej wirtualnej sieci poprzez mosty z siecią uruchomioną w oprogramowaniu wirtualizacyjnym GNS3. Poszerza to o możliwość wykonywania testów z sieciami złożonymi z wirtualnych przełączników CISCO uruchomionych w maszynach wirtualnych VPCS.

Bibliografia

1. Amrehn E. , Elliott J., „45 (40)Years of Mainframe Virtualization: CP-67/CMS and VM/370 to z/VM", IBM Corporation, Internet 2012, s.12
2. Khare N., „*Docker Cookbook*", Packt Publishing Ltd, Birmingham 06-2015, s.4-6
3. Mahalingam M., Dutt D., Duda K., Agarwal P., Kreeger L., Sridhar T., Bursell M., Wright C., „ VXLAN: A Framework for overlaying virtualized layer 2 network over layer 3 networks", RFC7348, ISSN:2070-1721, Sierpień 2014
4. Nadeau Thomas D. & Gray Ken "SDN Software Defined Network", O'Reilly Media, Sebastopol 2013, s.71-114
5. Raj P., Challadhurai J.S., V. Singh, *Learning Docker*, Packt Publishing, Birmingham 2015, s.90,s 102
6. Rosen E., Callon R., „Multiprotocol Label Switching Architekcture", RFC3031 Cisco System Inc. Styczeń 2001
7. Sridharan M., Duda K., Ganga I., Greenberg A., Lin G., Pearson M., Thaler P., Tumuluri C., Venkataramiah N., Wang Y., "NVGRE, Network virtualization using generic routing encapsulation," Network Working Group , 08-2011
8. Oficjalna strona dokumentacji Docker, docs.docker.com/engine/installation/, (dostęp 23-03-2016)
9. www.vmware.com/pl/virtualization, (dostęp 19-03-2016)
10. www.sdxcentral.com/resources/cisco/cisco-openflow/ (dostęp 26-03-2016)
11. www.tomsitpro.com/articles/software-defined-networking-solutions,2-835-2.html
12. lxc.sourceforge.net/old/index.php/about/kernel-namespaces/ (dostęp 27-02-2016)
13. docs.docker.com/v1.5/#about-this-guide, (Dostęp 02-2016)
14. www.nongnu.org/quagga/docs/quagga.html#VTY-shell-integrated-configuration (Dostęp 20-12-2015)
15. tech-itcore.pl/cisco-labs/ospf/ospf-wybor-dr-designated-router-oraz-bdr-backup-designated-router/ (Dostęp 10-02-2016)
16. docs.docker.com/v1.5/articles/networking/ (dostęp 11.09.2015)
17. github.com/jpetazzo/pipework (dostęp 02-11-2015)
18. docs.docker.com/v1.5/articles/networking/ (Dostęp 02-2016)

19. baturin.org/docs/iproute2/ (Dostęp 10-2015)
20. docs.docker.com/engine/reference/commandline/run/, (Dostęp 09-2015)
21. linux.die.net/man/7/capabilities, (Dostęp 18-12-2015)
22. linux.die.net/man/8/tc-tbf, (Dostęp 09-2015)
23. www.linuxfoundation.org/collaborate/workgroups/networking/netem, (Dostęp 09-2015)
24. docs.docker.com/engine/reference/commandline/exec/, (Dostęp 11-2015)
25. iperf.fr/, (Dostęp 09-2015)
26. www.staff.amu.edu.pl/~ttomek/sik/cwiczenia7.html, (Dostęp 12-2015)
27. www.isoc.org/inet98/proceedings/6g/6g_3.htm#micetroops, (Dostęp 18-01-2016)
28. nmap.org/, (Dostęp 12-2015)
29. cisco.sadzer.pl/rip/stp/, (Dostep 02-2016)
30. Dokumentacja on-line systemu Fedora, docs.fedoraproject.org/en-US/Fedora/19/html/Security_Guide/sec-Understanding_Network_Zones.html, (dostęp 12-05-2016)

Spis ilustracji

Spis tabel

Dodatek A

Tabela 8. Tabela procedur skryptu Qoslink

kod	ph1	ph2	h1	h2	ip1	ip2	sw1	sw2	r1	r2	proc1	proc2	proc3	proc4	proc5	proc6	proc7	proc8	proc9	proc10	proc11
72				v			v				freenet()	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br2()	set_sw1()	crt_c()	set_h2()	crt_h2()	set_if2()
9							v			v	freenet()	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_sw1()	set_br2()	crt_c()	set_r2()	crt_r2()	set_if2r2()
132			v					v			freenet()	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_sw2()	set_br1()	crt_c()	set_h1()	crt_h1()	set_if1()
6								v	v		freenet()	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_sw2()	set_br1()	crt_c()	set_r1()	crt_r1()	set_if1r1()
28						v	v	v			checkipall() ip2	freeip() ip4	set_c()	set_sw1()	set_sw2()	crt_c()	set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4sw2()	crt_brinqos()
44					v		v	v			checkipall() ip1	freeip() ip4	set_c()	set_sw1()	set_sw2()	crt_c()	set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4sw2()	crt_brinqos()
66				v					v		freenet()	freeip() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_r1()	set_br1()	set_br2()	crt_c()	set_h2()
129			v							v	freenet()	freeip() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_r2()	set_br1()	set_br2()	crt_c()	set_h1()
164			v		v			v			checkipall() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_sw2()	crt_c()	set_h1()	crt_h1()	set_if1()	crt_linkif1()
224			v	v	v						checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()	set_h2()
161			v		v					v	checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_r2()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()
88				v		v	v				checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br2()	set_sw1()	crt_c()	set_h2()	crt_h2()	set_if2()	crt_linkif2()
208			v	v		v					checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()	set_h2()
82				v		v			v		checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_r1()	set_br1()	set_br2()	crt_c()	set_h2()	crt_h2()
38					v			v	v		checkipall() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_sw2()	crt_c()	set_r1()	crt_r1()	set_if1r1()	crt_linkif1r1()
98				v	v				v		checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()	set_h2()
35					v				v	v	checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()	set_r2()
25						v	v			v	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br2()	set_sw1()	crt_c()	set_r2()	crt_r2()	set_if2r2()	crt_linkif2r2()
145			v			v				v	checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()	set_r2()
19						v			v	v	checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()	set_r2()
177			v		v	v				v	comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1	crt_h1()
240			v	v	v	v					comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()
114				v	v	v			v		comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()
51				v	v				v	v	comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()
192			v	v							freenet()	freeip() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()
3									v	v	freenet()	freeip() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()
400		v	v			v					checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()	set_ph2()
432		v	v		v	v					comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_h1()	crt_h1()
264		v					v				freenet()	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_sw1()	set_br2()	crt_c()	set_ph2()	crt_ph2()	set_if2ph2()
280		v				v	v				checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_sw1()	set_br2()	crt_c()	set_ph2()	crt_ph2()	set_if2ph2()	crt_linkif2ph2()
258		v							v		freenet()	freeip() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()
290		v			v				v		checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()	set_ph2()
274		v				v			v		checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()	set_ph2()
306		v			v	v			v		comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_r1()	crt_r1()
608	v			v	v						checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_ph1()	crt_ph1()	set_h2()
624	v			v	v	v					comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_ph1()	crt_ph1()
516	v							v			freenet()	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_sw2()	crt_c()	set_ph1()	crt_ph1()	set_if1ph1()
548	v				v			v			checkipall() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_sw2()	crt_c()	set_ph1()	crt_ph1()	set_if1ph1()	crt_linkif1ph1()
513	v									v	freenet()	freeip() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_ph1()	crt_ph1()
545	v				v					v	checkipall() ip1	freeip() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_ph1()	crt_ph1()	set_r2()
529	v					v				v	checkipall() ip2	freeip() ip1	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_ph1()	crt_ph1()	set_r2()
561	v				v	v				v	comparenet() ip1 ip2	checkipall() ip1	checkipall() ip2	freeip() ip3	freeip() ip4	set_c()	set_br1()	set_br2()	crt_c()	set_ph1()	crt_ph1()

proc12	proc13	proc14	proc15	proc16	proc17	proc18	proc19	proc20	proc21	proc22	proc23	proc24
crt_linkif2()	set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()						
crt_linkif2r2()	set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()						
crt_linkif1()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4sw2()	crt_brinqos()	set_link()						
crt_linkif1r1()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4sw2()	crt_brinqos()	set_link()						
set_link()												
set_link()												
crt_h2()	crt_r1()	set_if1r1()	crt_linkif1r1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
crt_h1()	crt_r2()	set_if1()	crt_linkif1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
set_if3()	crt_linkif3()	set_if4()	crt_linkif4sw2()	crt_brinqos()	set_link()							
crt_h2()	set_if1()	crt_linkif1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
crt_r2()	set_if1()	crt_linkif1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()							
crt_h2()	set_if1()	crt_linkif1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
crt_r1()	set_if1r1()	crt_linkif1r1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
set_if3()	crt_linkif3()	set_if4()	crt_linkif4sw2()	crt_brinqos()	set_link()							
crt_h2()	set_if1r1()	crt_linkif1r1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
crt_r2()	set_if1r1()	crt_linkif1r1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()							
crt_r2()	set_if1()	crt_linkif1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
crt_r2()	set_if1r1()	crt_linkif1r1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()		
set_r2()	crt_r2()	set_if1()	crt_linkif1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
set_h2()	crt_h2()	set_if1()	crt_linkif1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
set_if1r1()	crt_linkif1r1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
set_r2()	crt_r2()	set_if1r1()	crt_linkif1r1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
set_h2()	crt_h2()	set_if1()	crt_linkif1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
set_r2()	crt_r2()	set_if1r1()	crt_linkif1r1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	
crt_ph2()	set_if1()	crt_linkif1()	set_if2ph2()	crt_linkif2ph2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()	
set_ph2()	crt_ph2()	set_if1()	crt_linkif1()	set_if2ph2()	crt_linkif2ph2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()
crt_linkif2ph2()	set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()					
set_if3()	crt_linkif3sw1()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()						
set_ph2()	crt_ph2()	set_if1r1()	crt_linkif1r1()	set_if2ph2()	crt_linkif2ph2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()
crt_ph2()	set_if1r1()	crt_linkif1r1()	set_if2ph2()	crt_linkif2ph2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()	
crt_ph2()	set_if1r1()	crt_linkif1r1()	set_if2ph2()	crt_linkif2ph2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()	
set_ph2()	crt_ph2()	set_if1r1()	crt_linkif1r1()	set_if2ph2()	crt_linkif2ph2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif2docker0()
crt_h2()	set_if1ph1()	crt_linkif1ph1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif1docker0()	
set_h2()	crt_h2()	set_if1ph1()	crt_linkif1ph1()	set_if2()	crt_linkif2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif1docker0()
crt_linkif1ph1()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4sw2()	crt_brinqos()	set_link()	crt_linkif1docker0()					
set_if3()	crt_linkif3()	set_if4()	crt_linkif4sw2()	crt_brinqos	set_link()	crt_linkif1docker0()						
set_r2()	crt_r2()	set_if1ph1()	crt_linkif1ph1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif1docker0()
crt_r2()	set_if1ph1()	crt_linkif1ph1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif1docker0()	
crt_r2()	set_if1ph1()	crt_linkif1ph1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif1docker0()	
set_r2()	crt_r2()	set_if1ph1()	crt_linkif1ph1()	set_if2r2()	crt_linkif2r2()	set_if3()	crt_linkif3()	set_if4()	crt_linkif4()	crt_brinqos()	set_link()	crt_linkif1docker0()

Dodatek B

Tabela 9. Tabela parametrów wejściowych skryptu.

Lp.	Opcja	Opis
1	-h1	Nazwa kontenera pracującego jako host w sieci wirtualnej - "host"
2	-h2	j.w.
3	-r1	Nazwa kontenera pracującego jako router - "quaggalink"
4	-r2	j.w.
5	-sw1	Nazwa mostu linuksa funkcjonującego jako przełącznik w sieci - "brswitch"
6	-sw2	j.w.
7	-ph1	Nazwa kontenera pracujacego jako router brzegowy do obszaru 0 - "phlink"
8	-ph2	j.w.
9	-A	Liczba określająca obszar w którym uruchomiony będzie interfejs router
10	-c	Nazwa kontenera symulujacego łącze - "qoslink"
11	-br1	Nazwa dla mostu linuksa łączącego poszczególne elementy sieci - "brlink"
12	-br2	j.w.
13	-ip1	Adres IP przypisany do interfejsu -if1 w kontenerach typu -h -r lub -ph
14	-ip2	j.w. dla interfejsu -if2
15	-ip3	Adres IP przypisany do interfejsu -if3 w kontenerze qoslink
16	-ip4	j.w. dla interfejsu -if4
17	-if1	Interfejs tworzony w kontenerach typu -h -r lub -ph
18	-if2	j.w
19	-if3	Interfejs tworzony w kontenerze qoslink
20	-if4	j.w.
21	-gw1	Adres domyślnej bramy dla hosta
22	-gw2	j.w.
23	-band1	(jednostki: bit,kbit,mbit)
24	-band2	j.w.
25	-loss1	Ilość gubionych pakietów dla jednego kierunku (jednostki: %)
26	-loss2	j.w.
27	-delay1	Wartość opóźnienia pakietów dla jednego kierunku (jednostki: ms)
28	-delay2	j.w.
29	-duplic1	Ilość duplikowanych pakietów dla jednego kierunku (jednostki: %)
30	-duplic2	j.w.
31	-band	Przepustowość łącza dla obu kierunków. Łącze symetryczne (bit,kbit,mbit)
32	-loss	Rzeczywista ilość gubienia pakietów. Przeliczana jest na poszczególne kierunki.
33	-delay	Sumaryczna wartość opóźnienia w obu kierunkach. Przeliczana na poszczególne kierunki przez podział na pół.
34	-duplic	Ilość duplikowanych pakietów w obu kierunkach. Przeliczana na poszczególne kierunki
35	-link	Zdefiniowana lista łącz o określonych parametrach. Np.: 100Mbps, ISDN, 802.11n, ADSL3/8 …

Tabela 10. Tabela parametrów wejściowych skryptu - c.d.

Lp.	Opcja	Opis
36	-U	Opcja umożliwia aktualizację parametrów łącza. Łącze wskazujemy poprzez adres IP/Maska, nazwa kontenera qoslink lub podanie dwóch elementów sieci połączonych danym łączem
37	-D	Kasowanie sieci. Łącze wskazujemy j.w. Dodatkowo argument ALL - kasowanie całej sieci.
38	-P	Wyświetlanie parametrami łącz: Łącze wskazujemy jak w "-U". Dodatkowo po fragmencie nazwy kontenera. Np. "Lab" -> wyświetli Lab11,Lab12, TenLab22 itp..
39	-S	Zapis sieci do pliku. Opcja pusta -> domyślna nazwa pliku "qosnetxxx.dat"
40	-L	Odczyt sieci z pliku
41	-s	Opcja pracy skryptu bez komunikatów konfiguracyjnych
42	-V	Tymczasowa opcja - pełne komunikaty - debug
43	-?	Instrukcja skryptu