AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Marcin Fabrykowski

kierunek studiów: **informatyka stosowana**

System zautomatyzowanego zarządzania farmą serwerów aplikacji WWW

Opiekun: **dr inż. Piotr Gronek**

Kraków, wrzesień 2014

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

.....
(czytelny podpis)

Na kolejnych dwóch stronach proszę dołączyć kolejno recenzje pracy popełnione przez Opiekuna oraz Recenzenta (wydrukowane z systemu MISIO i podpisane przez odpowiednio Opiekuna i Recenzenta pracy). Papierową wersję pracy (zawierającą podpisane recenzje) proszę złożyć w dziekanacie celem rejestracji.

Spis treści

Wstęp

Niniejsza praca opisuje aplikację ułatwiającą administratorowi tworzenie oraz administrację klasterm dla aplikacji WWW.
Została ona podzielona na pięć rozdziałów tworzących trzy grupy.
Pierwsza grupa skupia się na rozważaniach teoretycznych. W jej skład wchodzą rozdziały:

"Wstęp do klastrowania"
: Opisuje on powody z których zaczęto stosować klastry WWW. Przybliża podział klastrów oraz cechy każdego rodzaju. Dodatkowo zarysowywuje problem wynikający z trudności zarządzania konfiguracją na klastrach.

"Metody klastrowania" Rozdział ten opisuje najbardziej znane metody służące klastrowaniu aplikacji. Opisuje on metody takie jak *DNS round robin*, *Nginx upstream*, *Haproxy*, *LSV*. Każda z powyższych metod zostałą krótko opisana, sposób jej działania oraz sposób konfiguracji.

"Zarządzanie konfiguracja"
: Ten rozdział przedstawia problem konsystentnej konfiguracji dużej ilości serwerów. Opisuje on podstawowy, ręczny sposób konfiguracji maszyn. Jego zalety i wady, a następnie kolejne metody usprawniające i eliminujące wady ręcznej konfiguracji. Zostały opisane aplikacje takie jak: *Fabric*, *Puppet*, *CFEngine*, *Ansible*, ze szczególnym naciskiem na ostatni.

W drugiej części pracy, znajduje się jeden rozdział, opisujący praktyczne testy wydajnościowe opisywanych wcześniej metod klastrowania. Skupia się on zarówno na testowaniu możliwości zwiększania wydajności jak i zapewnianiu wysokiej dostępności.

Trzecia część opisuje tytułowy *System zautomatyzowanego zarządzania konfiguracją farmy serwerów aplikacji WWW*. Opisuje on powody wyboru konkretnych technologii, opis ich konfiguracji oraz sposobu współdziałania.

Rozdział 1

Wstęp do klastrowania

1.1 Idea klastrowania

W czasach kiedy powstał internet, narodziła się potrzeba udostępniania informacji. Początkowo był to zwykle czysty tekst, który umieszczano na wtedy standardowych komputerach. Liczba odbiorców również nie była nie była duża, co wynikało z dopiero tworzącej się sieci internet. W miarę upływu czasu, strony internetowe zaczęły się rozrastać. Zaczęto dodawać obrazy a następnie animacje i inne elementy poprawiające możliwości stron oraz odczucia wizualne. Zaczęła również, w związku z coraz łatwiejszym dostępem do internetu, liczba użytkowników chcących uzyskać dostęp do stron.
Wymusiło to potrzebę używania coraz to mocniejszych maszyn do serwowania treści. Niestety, liczba użytkowników oraz wymagań stawianych stronom internetowym rosła szybciej niż postępował rozwój mocy obliczeniowych komputerów. Zaczęto używać dedykowanych serwerów dla aplikacji WWW zamiast zwykłych komputerów domowych. Jednak i to okazało się niewystarczające w obliczu wymaganiom stawianym przez dzisiejszy świat.

Aby rozwiązać ten problem, narodziła się idea połączenia kilku maszyn w jeden twór, tak aby zwiększyć całkowitą moc przeznaczoną na serwowanie treści.

1.2 Rodzaje klastrów

Istnieją dwa główne rodzaje klastrów.

- klastry wysokiej dostępności
- klastry wysokiej wydajności

ch Podział ten nie jest jednak bardzo sztywny. Niektóre rzeczywiste klastry należą tylko do jednej grupy, jednak najczęściej mają one cechy obu tych grup.

1.2.1 Klastry wysokiej dostępności

Klastry wysokiej dostępności (ang. *high availability, HA*) tworzone są głównie po to, aby zapewnić jak najwyższy poziom dostępności danej usługi. Konstrukcje te składają się zwykle z wielu maszyn, z których jakaś część nie uczestniczy w serwowaniu danych, a jedynie czeka w gotowości i w przypadku awarii maszyn aktywnych, przejmuje ich zadanie. Efektem jest ciągła dostępność usługi dla klienta.
Brak dostępności portalu może wiązać się z kosztami bądź brakiem zysków dla właściciela, dlatego często stosuje się klastry wysokiej dostępności.

1.2.2 Klastry wysokiej wydajności

Klastry wysokiej wydajności (ang. *high performance, HP*) tworzone są głównie po to, aby zapewnić jak najwyższy poziom wydajności danej usługi. Celem tego typu klastrów jest zapewnienie komfortu korzystania z serwisu dla klienta. W tej architekturze, pracują wszystkie maszyny, oraz każda z nim wykonuje jakąś część zadania w celu jak najszybszego skonstruowania odpowiedzi dla klienta.
W wersji ideowej, awaria jakiejś maszyny, unieruchamia klaster, ponieważ nie jest możliwe uzyskanie części odpowiedzi za którą odpowiadała maszyna która uległa awarii. W praktyce rzadko spotyka się czyste klastry wysokiej wydajności.

1.2.3 Klastry mieszane

Klastry mieszane są najczęściej spotykanymi klastrami. Posiadają one cechy obu powyższych grup, czyli najczęściej pracują wszystkie maszyny w klastrze, jednak ich konfiguracja pozwala im na wykonywanie dowolnego zadania (z puli przewidzianych zadań), w taki sposób, że podczas pracy wszystkich maszyn w klastrze, następuje szybsza odpowiedź do klienta - cecha wysokiej wydajności - jednak po awarii którejś z maszyn, pozostałe są w stanie przejąć jej obowiązki pozwalając odpowiedzieć na zapytanie - wysoka dostępność.
Można zauważyć, że podczas awarii węzłów w takim rodzaju klastra, spada wydajność, lecz zachowana jest ciągłość dostępu usługi, bo zwykle daje czas administratorowi na usunięcie usterki, bądź - jak zostanie pokazane w dalszej części tej pracy - skonfigurowanie nowych węzłów w celu zastąpienia tych które uległy awarii.

1.3 Zarządzanie konfiguracją

W czasach gdy strony były serwowane przez ich twórców na ich własnych komputerach, oni sami dbali o konfigurację swojej maszyny aby spełniała swoje zadanie.
W miarę jak zaczęto potrzebować coraz to większych mocy obliczeniowych, zaczęto wynajmo-

wać mocne serwery z dobrymi łączami, aby to one serwowały dane dla klientów. Konfiguracją takiego serwera zajmował się twórca strony, bądź zatrudniony administrator.

Jednak, gdy zaczęto używać klastrów pojawił się problem ich konfiguracji. Gdy klaster składał się z kilku węzłów, było bardzo mozolną pracą skonfigurowanie każdego węzła osobno oraz pilnowanie, aby konfiguracje na każdym węźle były odpowiednie. Po przypadkowej zmianie parametrów konfiguracji na jednej maszynie, trudno jest później znaleźć błąd.

Problem pojawił się również, gdy klastry zaczęły mieć więcej niż kilka węzłów. Skonfigurowanie kilkuset maszyn nie było prostym zadaniem, jak również prowadziło do wielu pomyłek. Dlatego administratorzy klastrów starali się ułatwić sobie pracę a zarazem uniknąć przypadkowych błędów.
Tak też powstały narzędzia do kontrolowania konfiguracji na wielu maszynach.

Rozdział 2

Metody klastrowania

W rozdziale tym przedstawię podstawowe metody wykorzystywane przy tworzeniu klastrów pod aplikacje internetowe.

2.1 DNS round robin

2.1.1 Czym jest DNS

DNS (ang *Domain Name System*) jest to system nazw domenowych. Usługa której najważniejszą funkcją jest przyporządkowanie nazwom domenowym (czytelnym dla człowieka) adresów IP. Oznacza to, że system taki, jest w stanie zamienić nazwę `www.ftj.agh.edu.pl` na adres `149.156.110.3`. Funkcjonalność taka w znacznym stopniu ułatwia korzystanie z sieci Internet, ponieważ przeciętnemu człowiekowi jest prościej zapamiętać mnemonik `www.ftj.agh.edu.pl` bądź `www.duckduckgo.com` niż ciągi czterech liczb. Do zamiany nazwy domenowej na adres IP służą rekordy `A` oraz `AAAA`, wykorzystywane odpowiednio do adresów IPv4 oraz IPv6. Przykład zastosowania rekordu `A` oraz `AAAA` przedstawiam na poniższym listingu:

Listing 2.1: rekord A oraz AAAA

```
nazwa4.domenowa.pl.      A       1.2.3.4
nazwa6.domenowa.pl.      AAAA    2001:db8::1428:57ab
```

w powyżej konfiguracji widzimy, że odpytując server DNS o wartość `nazwa4.domenowa.pl` otrzymamy informację, że dana nazwa wskazuje na adres `1.2.3.4`.
Innymi, często spotykanymi rekordami są rekordy `CNAME`, `MX` oraz `TXT`.

CNAME jest to rekord będący wskaźnikiem. Dla przykładu:

Listing 2.2: rekord CNAME

```
nazwa.domenowa.pl.       A       1.2.3.4
www.nazwa.domenowa.pl.   CNAME   nazwa.domenowa.pl.
```

widzimy, że w powyższym przykładzie `nazwa.domenowa.pl` wskazuje na 1.2.3.4. Chcąc aby, `www.nazwa.domenowa.pl` również wskazywała w to samo miejsce, moglibyśmy również zdefiniować ją jako rekord `A` z takim samym adresem. Jednak, w przypadku migracji serwera z adresu `1.2.3.4` na `1.2.3.5`, należałoby zmieniać ten adres w obu rekordach. Zastosowanie rekordu `CNAME`, pozwala powiedzieć "nazwa `www.nazwa.domenowa.pl` wskazuje w to samo miejsce, w które wskazuje `nazwa.domenowa.pl`". Prowadzi to do zmniejszenia ryzyka pomyłki przy wpisywaniu adresów IP, jak również zmniejsza liczbę miejsc w których należy zmienić adresację w przypadku migracji serwera.

MX rekord wskazujący na adres serwera pocztowego obsługującego daną domenę. Strefa może zawierać kilka rekordów `MX` w celu dystrybucji ruchu na kilka serwerów pocztowych.

Listing 2.3: rekord MX

```
        IN MX   5       mail
mail    IN A    1.1.1.1
```

Widzimy, że wpis definiujący rekord `MX` nie posiada nazwy. Zwykle rekord ten definiowany jest na początku definicji strefy, dlatego pominięcie nazwy powoduje, ze rekord ten odnosi się do nazwy tej strefy. Jest to kolejna rzecz które uogólnia konfigurację i ułatwia migrowanie. Wartość `5` oznacza poziom preferencji danego serwera. Mając zdefiniowanych kilka rekordów `MX`, poczta jest dystrybuowana przy pomocy algorytmu ważonego round robin.

TXT rekord który zgodnie z założeniami DNS miał zawierać dane tekstowe czytelne dla człowieka. Obecnie rzadko zawiera dane dla użytkowników. Wykorzystywany jest głównie do konfiguracji SPF, co wykracza poza tematykę tej pracy.

2.1.2 Opis metody

Metoda ta polega na odpowiednim skonfigurowaniu strefy na serwerze DNS, w taki sposób, aby pod jedna nazwa rozwiązywała się na kilka adresów IP. W efekcie, gdy serwer otrzyma zapytanie o daną nazwę domenową, zostanie mu zwrócona pula adresów zamiast jednego. Aplikacja która otrzyma listę adresów IP, powinna połączyć się na losowy z nich. Niestety nigdy nie ma pewności, że aplikacja posiada zaimplementowaną obsługę wielu adresów zwracanych przez serwer DNS, dlatego serwer wprowadza zabezpieczenie przed takim zachowaniem, a mianowicie tytułowy algorytm *round robin*, który zwraca adresy IP, jednak za każdym razem ich permutację.
Dla przykładu, poniżej zamieszczone trzy zapytania wykonane po sobie.

Listing 2.4: dig rr.mgr.fabrykowski.pl +short

```
$ dig rr.mgr.fabrykowski.pl +short
```

```
10.13.0.100
10.13.0.101
10.13.0.102
$ dig rr.mgr.fabrykowski.pl +short
10.13.0.101
10.13.0.102
10.13.0.100
$ dig rr.mgr.fabrykowski.pl +short
10.13.0.102
10.13.0.100
10.13.0.101
```

Widzimy, że przy każdym zapytaniu, jako pierwszy adres zwracany jest kolejny adres z puli. Zapewnia to prawidłowe balansowanie ruchu, nawet przy aplikacjach nie potrafiących obsłużyć wielu adresów i łączących się na pierwszy otrzymany.

Metoda ta jest metodą wysokiej wydajności, ponieważ pozwala w sposób niewidoczny dla użytkownika rozdzielić ruch na kilka serwerów, a tym samym rozłożyć obciążenie, to skutkować będzie szybszą odpowiedzią klientowi na zapytanie. Metoda ta nie zapewnia natywnie wykrywania niedostępności któregoś z serwerów, dlatego nie może służyć bezpośrednio jako metoda wysokiej dostępności. Pośrednio występuje tutaj jednak mechanizm broniący przed niedostępnością któregoś z serwerów. W przypadku gdy aplikacja próbować się będzie połączyć z losowym adresem z puli, a połączenie nie będzie mogło być nawiązane, osiągnięty zostanie limit czasu połączenia (tzw. *timeout*. W takiej sytuacji, dobrze napisana aplikacja, będzie próbować połączyć się na kolejny adres z puli, w nadziei, że będzie on dostępny. W takiej sytuacji, połączenie zostanie nawiązane i klient otrzyma odpowiedź, jednak do czasu generowania odpowiedzi, trzeba doliczyć czas potrzebny na osiągnięcie *timeout-u*. Może on wynieść od kilku, do kilkudziesięciu sekund.

Metoda ta jest również zależna od działania serwera DNS. Najprostszym sposobem ochrony przed awarią tego systemu dystrybucji ruchu jest skonfigurowanie *Secondary DNS*. To jednak wykracza poza tematykę tej pracy.

2.1.3 Konfiguracja

Konfiguracja DNS round robin jest stosunkowo prosta. W konfiguracji strefy, należy umieścić wpis z wieloma rekordami *A* dla jednej nazwy. Przykład takiej strefy zamieszczony został poniżej

Listing 2.5: mgr.fabrykowski.pl.zone

```
$TTL 3600
;
@       IN SOA  mgr.fabrykowski.pl.     root.mgr.fabrykowski.pl. (
                2014071700      ;serial
                2H              ;refresh
```

```
                30M             ;retry
                2w1d            ;expiry
                1h )            ;minimum
;
        IN NS   fabrykowski.pl.

;
@       IN A    95.85.57.200
rr      IN A 10.13.0.100
        IN A 10.13.0.101
        IN A 10.13.0.102
```

W powyższym przykładzie, dla nazwy `rr.mgr.fabrykowski.pl` zostały zdefiniowane trzy adresy IP.

2.2 Nginx

2.2.1 Czym jest nginx

Nginx jest serwerem proxy oraz serwerem treści statycznych. Wykorzystywany jest zwykle w połączeniu z serwerem Apache który serwuje treści PHP, podczas gdy sam dostarcza pliki statyczne (JavaScript, CSS, JPEG itp). Drugim często wykorzystywanym modelem wykorzystania Nginx-a jest serwowanie treści statycznych oraz wykonywanie *fastcgi pass*

2.2.1.0.1 Fastcgi pass

Moduł ten pozwala na komunikacje z procesami FastCGI. Wykorzystanie FastCGI daje dużą niezależność w technologi opracowania aplikacji, która może zostać wykonana w PHP, Pythonie bądź Rubym. Istnieje również możliwość zmiany wersji aplikacji, bądź technologii jej wykonania bez zmian w konfiguracji serwera, jeżeli aplikacja udostępnia to samo api FastCGI.

Do obsługi języka PHP zostanie wykorzystany php-fpm (*PHP FastCGI Process Manager*. Jest to alternatywna implementacja PHP FastCGI. Pozwala ona na większą kontrolę w zakresie pul procesów - ich liczby oraz sposobu uruchamiania, jak również dowolność w kwestiach sieciowych - adres oraz port do nasłuchiwania.

Porównanie testów wydajności PHP-fpm oraz mod_php do Apache jak również szybkość serwowania treści statycznych zostanie przedstawione w późniejszych rozdziałach.

2.2.2 Opis metody

Metoda klastrowania przy pomocy Nginx-a polega na zdefiniowaniu sekcji `upstream`. Pozwala to na skonfigurowanie puli adresów do których będą przekazywane zapytania. Aby dodać serwer do puli, należy podać jego adres IP bądź nazwę domenową oraz port.

Zapytania do serwerów wykonujących (*workerów*) rozdzielane są równomiernie pomiędzy wszystkie serwery w puli.
Pozwala to na obsługiwanie zapytań na wielu maszynach, dlatego metoda ta pozwala na tworzenie klastrów **wysokiej wydajności**.

Ponadto, zaimplementowany jest również mechanizm sprawdzający stan poszczególnych serwerów w puli i w przypadku wykrycia awarii, oznaczany jest on jaki *failure* i zapytanie nie są do niego kierowane.
Jest to zachowanie typowe dla klastrów **wysokiej dostępności**

Istnieje możliwość modyfikacji domyślnego algorytmu używanego przez Nginx-a.

- zmiana sposobu dystrybucji zapytań dodając opcjonalny parametr `weight` mówiący o wadze danego węzła. Dla przykładu, w poniższej konfiguracji:

  Listing 2.6: nginx upstream

  ```
  upstream pula1 {
          server server1:9000 weight=5;
          server server2:9000;
  }
  ```

  na każde 6 zapytań do `pula1`, 5 zostanie przekazanych do `server1` a jedno do `server2`. Opcja ta wykorzystywana jest głównie tam, gdzie poszczególne serwery różnią się parametrami bądź obciążeniem nie wynikającym z obsługiwania tej puli.

- zmiana sposobu określania serwera jako niedostępnego. Służą do tego parametry `max_fails`, `fail_timeout` oraz `slow_start`.

  `max_fails` określa liczbę nieudanych prób komunikacji z serwerem w czasie `fail_timeout` nim serwer zostanie oznaczony jako niedostępny. Domyślna wartość tego parametru wynosi 1, natomiast wartość 0 wyłącza oznaczanie serwerów jako niedostępne.

  `fail_timeout` określa czas w jakim musi nastąpić `max_fails` nim serwer zostanie uznany za niedostępny. Określa również interwał czasowy co który będzie sprawdzana dostępność serwera. Wartość domyślna dla tego parametru wynosi 10 sekund.

  `slow_start` określa czas w jakim będzie zwiększana wartość `weight` od zera do docelowej po przejściu serwera ze stanu niedostępnego do stanu dostępnego. Wartość domyślna wynosi 0, co oznacza wyłączone płynne włączanie serwera do puli.

- oznaczenie konkretnych serwerów, jako serwery zapasowe. Powoduje to nieprzekazywanie zapytań do tych serwerów jeżeli wszystkie serwery podstawowe odpowiadają. W przypadku, gdy któryś z podstawowych serwerów zostanie oznaczony jako niedostępny, zapytania zostają przekazywane do któregoś z serwerów zapasowych. Powoduje to zachowanie **wysokiej wydajności** oraz **wysokiej dostępności**.

2.2.3 Konfiguracja

Listring 2.7 przedstawia zmodyfikowany na potrzeby tego listingu, przykład konfiguracji Nginx-a z wykorzystaniem rozdzielania połączeń przez serwer WWW.

Listing 2.7: nginx.conf

```
error_log  /var/log/nginx/error.log;
access_log  /var/log/nginx/access.log  main;
user       nginx;
pid        /var/run/nginx.pid;
events {
               worker_connections  1024;
}
http {
            include       mime.types;
            ssl_certificate     /etc/ssl/diasp.clug.pl.chained.crt;
            ssl_certificate_key /etc/ssl/clug.key;

            upstream clug_fcgi {
                server 162.251.114.75:17303;
                server 162.251.144.77:17303;
            }
            server {
                listen 443 ssl;
                server_name clug.pl www.clug.pl;
                rewrite ^(.*[^/])$ $1/;
                location / {
                        uwsgi_pass clug_fcgi;
                        include uwsgi_params;
                }
                access_log  /var/log/nginx/clug.log main;
                error_log   /var/log/nginx/clug_error.log error;
        }
}
```

2.3 Haproxy

2.3.1 Czym jest haproxy

Haproxy jest serwerem proxy wysokiej dostępności (ang. High Availability Proxy). Posiada on dwie główne funkcjonalności, które czynią go powszechnie używanym narzędziem. Są nimi:

- możliwość dystrybuowania ruchu na kilka maszyn, dając tym samym zwiększone możliwości obliczeniowe
- wykrywanie awarii serwerów *backendowych* i nieprzekazywaniem do nich zapytań aż do czasu naprawy

2.3.1.1 Funkcjonalność wysokiej wydajności

Haproxy pozwala na zdefiniowanie tzw. *backendu*, czyli grupy serwerów pełniących tą samą funkcje. Decyzja o wyborze serwera dla danego zapytania może być podjęta na podstawie jednego z kilku algorytmów. Poniżej znajduje się lista kilku najpopularniejszych. Pełną listę można znaleźć w dokumentacji

Round robin
: najpopularniejszy algorytm. Polega na rozdzielaniu zapytam do poszczególnych serwerów "po kolei". Kryterium modulującym działanie tego algorytmu jest parametr `weight`, który jak bardzo dany serwer ma być preferowany. Domyślna wartość `weight` wynosi 1. W przypadku, gdy wszystkie serwery mają takie same wartości, połączenia przekazywane są równo do każdego z nich.

Leastconn
: wybór serwera podejmowany jest na podstawie ilości aktywnych połączeń do każdej maszyny. Wybierany jest serwer z najmniejszą ilością połączeń

Source
: serwer docelowy wybierany jest na podstawie adresu nadawcy. Powoduje to, że jeden klient będzie zawsze obsługiwany przez tą samą maszynę. Pozwala to na uproszczenie obsługi sesji pomiędzy maszynami.

Ponieważ Haproxy działa w warstwie siódmej modelu OSI - czyli w warstwie aplikacji, możliwe jest również decydowanie o wyborze serwera na podstawie nagłówków zapytać HTTP. Na podstawie np: wartości `host` w nagłówku HTTP, haproxy jest w stanie nasłuchując na jednym porcie dystrybuować ruch na odpowiednie *backendy* odpowiedzialne za różne strony. Przykłady takiego zastosowania zostaną przedstawione w podrozdziale "2.3.2 Konfiguracja"

2.3.1.2 Funkcjonalność wysokiej dostępności

Możliwości wykrywania niedostępności usługi oraz zapewnienia wysokiej dostępności były głównym celem twórców. Świadczyć może o tym nazwa - *HAProxy*, pochodzącą od *High Availability* czyli wysoka dostępność.
Domyślnie haproxy nie sprawdza dostępności serwerów. Aby włączać tą funkcjonalność należy

użyć parametru `check`.
`Check` sprawdza pod adresem i portem zdefiniowanymi dla danego serwera udaje się ustanowić połączenie TCP. Jeśli tak jest, usługa jest uznawana za działającą i połączenia są kierowane na daną maszynę.
Istnieją również inne predefiniowane funkcje sprawdzające, np: `httpcheck` służący do sprawdzania odpowiedzi serwera WWW pod zadanym `uri`, `smtpcheck` - sprawdza usługę `smtp`, `mysql-check` oraz `pgsql-check` do baz danych. Istnieje również możliwość stworzenia własnych mechanizmów sprawdzających działanie usługi, bazujące na technologii `expect`.

2.3.2 Konfiguracja

Na listingu 2.8 przedstawiono przykładową konfigurację HAProxy obsługującą zarówno wiele adresów url jak również stanowi *frontend* dla `php-fpm`.

Listing 2.8: haproxy.cfg

```
global
        log 127.0.0.1 local0 notice
        maxconn 2000
        user haproxy
        group haproxy

defaults
        log global
        mode http
        option httplog
        option dontlognull
        retries 3
        option redispatch

frontend http-in
        bind 0.0.0.0:80
        acl is_test.pl hdr_end(host) -i test.pl
        use_backend test.pl if is_test.pl
        acl is_test.ru hdr_end(host) -i test.ru
        use_backend test.ru if is_test.ru
        acl is_test2.pl hdr_end(host) -i test2.pl
        use_backend test2.pl if is_test2.pl

backend test.pl
        balance roundrobin
        option forwardfor except 127.0.0.1
        server mgr02 10.13.0.12:80 check
        server mgr03 10.13.0.13:80 check
```

```
backend test.ru
        balance roundrobin
        option forwardfor except 127.0.0.1
        server mgr03 10.13.0.13:80 check

backend test2.pl
        balance roundrobin
        option forwardfor except 127.0.0.1
        server mgr03 10.13.0.13:80 check


frontend php_test2.pl-in
        bind 0.0.0.0:9007
        default_backend php_test2.pl
frontend php_test.pl-in
        bind 0.0.0.0:9005
        default_backend php_test.pl
frontend php_test.ru-in
        bind 0.0.0.0:9006
        default_backend php_test.ru

backend php_test.pl
        balance roundrobin
        server mgr04 10.13.0.14:9005 check
        server mgr05 10.13.0.15:9005 check
        server mgr06 10.13.0.16:9005 check
        server mgr07 10.13.0.17:9005 check
backend php_test.ru
        balance roundrobin
        server mgr06 10.13.0.16:9006 check
        server mgr07 10.13.0.17:9006 check
backend php_test2.pl
        balance roundrobin
        server mgr06 10.13.0.16:9007 check
        server mgr07 10.13.0.17:9007 check

listen stats 0.0.0.0:9000
    mode http

    stats uri /haproxy_stats
    stats realm HAProxy\ Statistics
    stats auth admin:admin
    stats admin if TRUE
```

W konfiguracji *haproxy* wyróżniamy następujące ważne sekcje:

`global`

zawiera konfigurację ustawień dla procesu haproxy. Umieszczane są tutaj informacje o ilości maksymalnych połączeń do procesu, `uid`-dzie bądź nazwie użytkownika i grupy z jakim należy uruchomić aplikacje, ścieżka do pliku z numerami `PID` procesów haproxy.

`defaults`

sekcja ta zawiera wartości domyślne dla innych sekcji. Pozwala to na umieszczenie dużej części konfiguracji w jednym miejscu, co ułatwia zarządzanie nią. Wartości zdefiniowane w sekcji `defaults` mogą być nadpisane w konkretnej sekcji wartością specyficzną dla danej sekcji.

Dla przykładu, można zdefiniować domyślną wartość `timeout` na 10 sekund, natomiast dla pewnego serwisu, który wykonuje długotrwałe obliczenia, można tą wartość nadpisać wartości większa. Podejście takie pozwala na zachowanie zabezpieczenia przed nieodpowiadającymi procesami dla wszystkich serwisów, jednocześnie zezwalając aby serwis wykonujący długotrwałe obliczenia nie był anulowany przed uzyskaniem wyniku.

`listen`

definiuje usługę wysokiej dostępności. Po słowie kluczowym `listen` następuje nazwa danej usługi a następnie adres IP oraz port na którym będzie nasłuchiwać dana usługa.

Słowo kluczowe `mode` definiuje w jakim trybie ma działać dana usługa. Wyróżniamy dwa tryby:

- `http`
  tryb ten działa w warstwie siódmej i pozwala na operowanie na zmiennych zawartych w nagłówkach HTTP
- `tcp`
  tryb ten działa w warstwie czwartej i powinien być stosowany do wszystkich połączeń nie będących połączeniami HTTP, tj. SSH, SSL i inne.

Historycznie istniał jeszcze tryb `health`, jednak jest już przestarzały i nie zalecane jest jego używanie.

W przykładzie na listingu 2.8 w usłudze `stats` znajdują się polecenia dające dostęp administratorowi do statystyk haproxy. W ich skład wchodzi m.in:

- ilość wszystkich połączeń przyjętych na dany *frontend*
- ilość aktywnych połączeń na *frontendach*
- stan serwerów obsługujących *backendy*
- ilości połączeń na każdy serwer w *backendach*

W przypadku większych instalacji, bądź potrzeby posiadania większej kontroli nad sposobem *load balncingu* stosuje się strukturę rozbijającą prosty `listen` na dwie sekcje: `frontend` oraz `backend`.

`frontend`

`Frontend` odpowiedzialny jest za przyjmowanie i analizę połączeń od użytkownika. W sekcji tej znajduje znajduje się podzbiór poleceń z sekcji `listen` dotyczących opcji nasłuchiwania, takich jak adres oraz port, sposobu traktowania ruchu (http, tcp) jak również polecenia mogące analizować ruch w celu odpowiedniego jego obsłużenia. Noszą one nazwę *ACL (and. Access Control List)*. Reguły ACL potrafią analizować ruch począwszy od warstwy czwartej do warstwy siódmej.

Dla ruchu HTTP istnieje możliwość analizy nagłówków oraz rozdział połączań na różne serwery w zależności od pola `Host`, adresu `url` bądź metody żądania. Dla ruchu HTTPS istnieje możliwość konfiguracji ACL dla żądanego hosta dzięki technologi SNI *ang. Server Name Indication* - technologia pozwalająca na odczytywanie wartości żądanego hosta w połączeniach HTTPS. Wykorzystywana w celu uruchamiania wielu adresów WWW na jednym porcie przy szyfrowaniu SSL.

Po zdefiniowaniu ACL, istnieje możliwość zdefiniowania użycia konkretnego `backend`-u w zależności od przypisania do ACL.

Dodatkowo, oprócz wyboru odpowiedniego `backend`-u istnieje możliwość odrzucania połączeń spełniających warunki ACL - np: zbyt duża ilość połączeń.

`backend`

Sekcja ta definiuje zaplecze serwerowe, tj. listę serwerów do których ma być kierowany ruch. Są tutaj w mocy wszystkie polecenia które mogą znaleźć się w sekcji `listen` które tyczą się serwerów obsługujących, czyli m.in. algorytm rozdziału połączeń czy specyficzne metody sprawdzania dostępności serwera.

2.4 LVS

2.4.1 Czym jest LVS

LVS (*ang. Linux Virtual Server*) jest technologią pozwalającą na tworzenie klastrów bazujących na systemach GNU/Linux. Metoda ta jest bardzo wysoko skalowalna przy małym obciążeniu procesora.

Trzeba mieć na uwadze, że LVS nie możliwości współbieżnego przetwarzania operacji, a jedynie dystrybucję połączeń pomiędzy wiele serwerów.

2.4.2 Opis metody

LVS jest technologią działającą w czwartej warstwie modelu OSI, tj. w warstwie transportowej. Zakłada ona istnienie dwóch typów serwerów w klastrze:

Director

Jest to serwer zarządzający. Występuje jeden w klastrze. To do niego kierowane są połączenia klienckie.

Real Server

Jest to serwer z właściwą usługą. W klastrze może występować ich wiele. Odpowiedzialne są za przetwarzanie zapytań od klienta.

LVS może działać w jednym z trzech trybów:

NAT

W tym trybie zapytania przychodzące od klienta do *Directora* zostają znatowane na adres jednego z *Real server*-ów. Po obsłużeniu zapytania, *real server* przekazuje odpowiedź do *director*-a który następnie przekazuje odpowiedź do klienta.
Tłumaczenie pakietów wymaga pewnej mocy obliczeniowej, ponadto *director* uczestniczy w przesyłaniu zapytania oraz odpowiedzi, co sprawia, że tryb NAT ma ograniczoną skalowalność ograniczoną mocą procesora oraz łącza internetowego.

Direct Routing

Tryb ten jest wolny od problemów z mocą obliczeniową oraz utylizacją łącza występujących w przypadku trybu NAT.
W przypadku *direct routing*-u *real server*-y posiadają dodatkowe adresy IP, takie same jak adresy IP używane do *load balancingu* na *directorze* jednak *real server* musi być tak skonfigurowany aby nie odpowiadał na zapytania ARP o te adresy.
Gdy pakiet dochodzi do *directora*, zostaje on przekazany w niezmienionej formie (od trzeciej warstwy wzwyż) do jednego z *real server*-ów. *Director* musi mieć możliwość bezpośredniego połączenia z *real server*-em aby móc opakować pakiet w odnowienie nagłówki warstwy drugiej. Ponieważ *real server* posiada dodatkowy adres IP, pakiet przekazany przez *director*-a jest akceptowany ponieważ adres docelowy w pakiecie zgadza się z adresem posiadanym przez *real server*. Po przyjęciu pakiety, *real server* odpowiada na niego na adres źródłowy zawarty w pakiecie, czyli bezpośrednio do klienta - omijając *director*-a. Następnie, klient wysyłając kolejne kieruje je do *director*-a, który śledząc połączenia, przekazuje je zawsze do odpowiedniego *real server*-a.

Powyższa procedura powoduje, iż *director* nie jest obciążany obliczaniem adresacji dla NAT, gdyż przekazuje pakiety w niezmienionej formie, oraz zmniejsza utylizacje łącza, ponieważ na łączy *director*-a przesyłane są jedynie pakiety z żądaniami (na wejściu - od

klienta, i na wyjściu - do *real server*-a), natomiast pakiety z odpowiedzią, np: HTTP, które są zwykle znacznie większe niż zapytania, są przesyłane łączami używanymi przez *real server*-y. W efekcie, metoda ta jest wysoko skalowalna.

IP tunneling

Tryb ten działa identycznie jak *direct routing*, z tą różnicą, ze *director* oraz *real server*-y nie znajdują się w jednej fizycznej sieci, lecz są spięte jakimś tunelem.

LVS wpiera kilka algorytmów rozdzielających zapytania. Najważniejsze z nich to:

round robin

Połączenia rozdzielane są po równo do każdego serwera.

weighted round robin

Połączenia rozdzielane są do każdego serwera w proporcji określonej wagami każdego z węzłów.

least-connection

Połączenia są przekazywane do serwera z najmniejszą liczbą aktywnych połączeń

LVS nie posiada żadnego wbudowanego systemu zapewniającego wysoką dostępność. Jest to metoda zapewniająca wyłącznie wysoką wydajność. Istnieją rozwiązania współpracujące z LVS dodające funkcjonalność wykrywania niedostępności usługi na *real server*-ach i wypinające je z konfiguracji LVS.
Te rozwiązania są jednak poza zakresem niniejszej pracy.

2.4.3 Konfiguracja

Konfiguracja LVS obejmuje stworzenie tablicy określającej adres oraz port działania usługi. Następnie należy dodać *real server*-y należące do tego LVS. Przykładową konfigurację LVS przedstawia listing 2.9.

Listing 2.9: LVS

```
ipvsadm -A -t 10.13.0.101:80 -s rr
ipvsadm -a -t 10.13.0.101:80 -r 10.13.0.12:80
ipvsadm -a -t 10.13.0.101:80 -r 10.13.0.13:80
ipvsadm -a -t 10.13.0.101:80 -r 10.13.0.14:80
ipvsadm -a -t 10.13.0.101:80 -r 10.13.0.15:80
```

Konfiguracja 2.9 przedstawia utworzenie usługi nasłuchującej na adresie `10.13.0.101` oraz porcie `80`. Oraz wykorzystujący algorytm *round robin* do rozdzielania połączeń.
Następnie do nowo utworzonej usługi dodane zostają cztery *real erver*-y. Domyślnym trybem dodawania *real server*-ów jest *direct routing*.

Należy jeszcze pamiętać o dodaniu odpowiednich adresów do interfejsów sieciowych. Dla *director*-a listing 2.10

Listing 2.10: konfiguracja adresacji dla directora

```
ip addr add 10.13.0.101 dev eth0
```

Dla *real server*-a należy jeszcze pamiętać o problemie z ARP. Jednym ze sposobów jest użycie `ARPtables`, listing 2.11

Listing 2.11: konfiguracja adresacji dla real servera

```
ip addr add 10.13.0.101 dev lo
arptables -A OUTPUT -s 10.13.0.101 -j DROP
```

Zapis konfiguracji adresacji *na stałe* jest zależny od systemu operacyjnego.

Rozdział 3

Zarządzanie konfiguracją

W rozdziale tym przedstawię różne metody zarządzania konfiguracją serwerów. Postaram się opisać poglądowo różne metody, jak również przedstawić zalety i wady poszczególnych z nim.

3.1 Ręczna konfiguracja każdego serwera za pomocą SSH

3.1.1 Opis

Ręczna konfiguracja serwerów stos osuwana jest głównie tam, gdzie administrator ma pod swoją opieką jeden bądź kilka serwerów. W takim przypadku zmiana konfiguracji na serwerze jest prosta i nie zajmuje dużej ilości czasu.
Konfiguracja taka nie wymaga od administratora żadnej wiedzy wykraczającej poza obszar konfigurowanego systemu oraz usług, a wprowadzane zmiany widoczne są od razu po wprowadzeniu. Ten sposób konfiguracji spotykany jest czasem w większych systemach informatycznych. Dzieje się tak zwykle w jednostkach szybko rozwijających się, gdzie nastąpił szybki wzrost liczby serwerów i nie opracowano jeszcze metoda automatyzacji konfiguracji.

Do konfiguracji ręcznej nie potrzeba żadnego dodatkowego oprogramowania ani po stronie maszyn konfigurowanych, ani maszyny z której następuje konfiguracja. Na maszynie z której następuje konfiguracja musi być dostępny klient SSH, który jest instalowany domyślnie we wszystkich dystrybucjach systemów GNU/Linux, a na maszynach konfigurowanych musi być zainstalowany i uruchomiony serwer SSH - jest on domyślnie zainstalowany w większości dystrybucji serwerowych GNU/Linux i w części dystrybucji przeznaczonych na komputery domowe.

Wadą takiej metody jest również sytuacja, w której tylko jedna osoba, bądź mała grupa osób, zna konfigurację poszczególnych serwerów oraz usług. W przypadku opuszczenia przez daną osobę zespołu, pozostali członkowie muszą, analizując pliki konfiguracyjne, zrozumieć zamysł osoby to tworzącej.
Kolejną wadą, jest brak możliwości powielenia konfiguracji. W przypadku gdy zaistnieje po-

trzeba skonfigurowania bliźniaczego serwera, jako serwera zapasowego, należy każdą usługę skonfigurować od nowa na wzór serwera pierwotnego. Również wprowadzane zmiany należy uwzględniać na wszystkich serwerach. Może to w prosty sposób prowadzić do błędów i rozbieżności konfiguracji.

3.1.2 Zalety i wady

Zalety:

- prostota
- używanie tylko domyślnych komponentów systemu
- szybkość wprowadzanych zmian
- informacja zwrotna czy usługa została uruchomiona poprawnie

Wady:

- **brak skalowalności**
- różnice między poszczególnymi serwerami
- trudność powielania
- wiedza o konfiguracji zależna od jednego pracownika

3.1.3 Przykład

Listing 3.1: konfiguracja ręczna przez SSH

```
admin@master:~$ ssh admin@conf
admin's password:
admin@conf:~$ vim /etc/http/vhosts.conf/test.pl.conf
admin@conf:~$ apachectl -t
Syntax OK
admin@conf:~$ apachectl graceful
admin@conf:~$ exit
```

3.1.4 CSSH

Istnieje narzędzie CSSH (*Cluster SSH*) które wychodzi na przeciw osobą chcącym konfigurować kilka serwerów jednocześnie poprzez SSH. Narzędzie to potrafi otworzyć wiele sesji SSH równolegle - każda sesja w osobnym terminalu. Głównym interface-em programu, jest małe okno wejścia, które przechwytując wpisywany do niego tekst, przesyła go do wszystkich otwartych sesji.

Zmniejsza to prawdopodobieństwo rozbieżności w konfiguracji, jak również przyśpiesza proces, ponieważ tekst jest wpisywany do wszystkich sesji jednocześnie i nie ma potrzeby wielokrotnego wpisywania tej samej konfiguracji na wielu maszynach.
Aplikacja umożliwia również przełączenie się w dowolnej chwili na konkretny terminal i interakcję tylko z jednym serwerem, np: w celu zdiagnozowania problemu występującego tylko na tej jednej maszynie.

3.2 Fabric

3.2.1 Opis

Jest aplikacją napisaną w języku Python, służącą głównie do wykonywania poleceń powłoki na zdalnym serwerze. Aplikacja pozwala na zdefiniowanie kolejności w jakiej mają zostać poszczególne polecenia, jak również udostępnia kilka funkcji sprawdzających, np: czy plik istnieje, bądź kopiowanie plików na lub z serwera.
Sprawdza się wszędzie tam, gdzie chcemy wykonać konkretne operacje na zdalnym systemie niezależnie od aktualnego stanu tego systemu, bądź z niewielkim wpływem obecnych czynników. Zastosowanie fabrica można porównać do CSSH, z tą różnicą, że operacje nie są wpisywane przez administratora podczas sesji, a zdefiniowane wcześniej w pliku, co w znacznym stopniu ułatwia powtarzalność wykonywania zdefiniowanych operacji. Pozwala również w prosty sposób rozdzielić zdefiniowane zadania na poszczególne grupy serwerów na których należy je wykonać. Typowe zastosowania:

- restart nietypowych usług nie posiadających jeszcze odpowiednich skryptów sysvinit
- rekonfiguracja projektów na zdalnych serwerach po wysłaniu zmian przez system kontroli wersji
- przeszukiwanie logów poszczególnych serwerów

3.2.2 Zalety i wady

Zalety:

- łatwość instalacji - repozytoria dystrybucji oraz pythonowe
- równoległe wykonywanie operacji
- łatwość konfiguracji
- powtarzalność wykonywania
- skalowalność

- niewymagana instalacja oprogramowania na zdalnych maszynach

Wady:

- ograniczone możliwości decyzji na podstawie aktualnej konfiguracji
- wykonywanie tylko poleceń powłoki

3.2.3 Przykład

Listing 3.2: fabfile.py

```
from fabric.api import *
from fabric.contrib.files import exists, contains
env.warn_only = True
env.disable_known_hosts = True
env.user = 'root'
env.hosts  = [
              '192.168.0.10',
              '192.168.0.11',
              '192.168.0.12',
              '192.168.0.13',
              '192.168.0.14',
              '192.168.0.15',
              ]

def show_problem():
    if exists('/var/problem'):
        run('cat␣/var/problem')

def fix():
    if contains('/var/problem', 'podmontowany'):
        run ('umount␣/mnt/autologs')
    if exists('/var/problem'):
        run('./scripts/autologs.sh')
```

przykład działania powyższego skryptu:

Listing 3.3: użycie fabric

```
admin@master:~$ fab -P -z 5 show_problem -I
Initial value for env.password:
[192.168.0.10] Executing task 'show_problem'
[192.168.0.11] Executing task 'show_problem'
[192.168.0.12] Executing task 'show_problem'
[192.168.0.13] Executing task 'show_problem'
[192.168.0.14] Executing task 'show_problem'
[192.168.0.15] Executing task 'show_problem'
```

```
[192.168.0.12] run: cat /var/problem
[192.168.0.12] out: zasob byl podmontowany
[192.168.0.12] out:
```

Aplikacja została uruchomiona z parametrami:

-P równolegle wykonywanie zadań

-z 5 uruchomienie pięciu równoległych połączeń

-I zapytanie o hasło do serwerów (używane gdy niedostępne logowanie po kluczach SSH)

show_problem nazwa zadania zdefiniowana w pliku `fabfile.py`

Fabric wykonuje połączenia do hostów zdefiniowanych w zmiennej `env.hosts` w liczbie pięciu połączeń równoległych. W przypadku nie podania parametru `-z`, aplikacja wykona liczbę równoległych połączeń równą liczbie zdefiniowanych hostów dla danego zadania.
Po połączeniu się do zdalnego hosta, następuje sprawdzenie czy istnieje plik `/var/problem`. W przypadku wykrycia istnienia takiego pliku, zostaje wywołane polecenie powłoki `cat`. W wyniku wykonywania widzimy, ze plik `/var/problem` istniał tylko na serwerze o adresie IP `192.168.0.12` i zawierał tekst *zasob byl podmontowany*.

3.3 Puppet

3.3.1 Opis

Puppet jest jednym z najbardziej rozpowszechnionych systemów do zarządzania konfiguracją. U podstaw ideologii działania puppeta stoi definicja oczekiwanego stanu serwera. Administrator definiuje oczekiwany stan systemu, np: istnienie użytkownika o podanych parametrach, bądź istnienie pliku o zadanej zawartości, a puppet dąży do uzyskania takiego stanu - stworzy użytkownika lub plik taki aby spełniał zadane wymagania.
Puppet został napisany w języku Ruby, co wpływa na składnię manifestów przez niego wykorzystywanych. Manifest jest opisem żądanego stanu danego obiektu (użytkownik, plik, zamontowany zasób). Manifesty są zapisywane w plikach `*.pp`.
Puppet działa w trybie klient-serwer. Serwer posiada zapisane manifesty dla wszystkich maszyn, dlatego w celu rekonfiguracji wielu maszyn, wystarczy zmiana jedynie w centralnym punkcie - serwerze *master*.
Istnieją trzy możliwości wykonywania manifestów.

bezpośrednio na maszynie poprzez jawne wywołanie manifestu na danej maszynie, następuje jego sparsowanie oraz zastosowanie zawartych w nim deklaracji do lokalnej maszyny

cykliczne pobieranie danych przez agenta ponieważ puppet w zamyśle ma działać w trybie *pull*, dlatego jest to jego domyślny tryb pracy. Agent, działający na maszynach klienckich

- maszynach których stan ma być kontrolowany przez puppeta - w cyklicznych odstępach czasu pobiera z serwera *master* żądaną konfiguracje. Nie pobiera on bezpośrednio manifestów zapisanych przed administratora, lecz skompilowaną ich wersję specjalne dla tej maszyny.
  Połączenia do *master*-a są szyfrowane poprzez SSL co zwiększa bezpieczeństwo przesyłanych danych wrażliwych zawartych w manifestach.

wymuszone uruchomienie agenta tryb ten działa podobnie do cyklicznego pobierania danych, z tą różnicą, ze użytkownik jawnie zmusza agenta do poprania danych z serwera *master* natychmiast, zamiast czekać aż agent pobierze te dane samoistnie.

3.3.2 Zalety i wady

Zalety:

- popularność
- duże zaplecze *community*
- dojrzałość projektu

Wady:

- potrzeba instalacji oprogramowania na maszynach klienckich
- działanie w trybie *pull*
- skomplikowana instalacja

3.4 CFEngine

3.4.1 Opis

CFEngine jest jednym z najstarszych systemów do zarządzania konfiguracją. Powstał z myślą o systemach w których nie jest zapewniona dostateczna jakoś łącza, np: łodzie podwodne.

Z racji swojego wieku, architektura CFEngine, w odpowiedzi na zmieniające się potrzeby systemów, ulegała zmianom. Aktualnie rozwijana jest trzecia generacja CFEngine. Kolejne generacje starały się upraszczać składnie konfiguracji oraz zwiększać możliwości oferowane przez oprogramowanie.

CFEngine opiera się na "Teorii obietnic". Jest to teoria stojąca u podstaw wszystkich systemów zarządzania konfiguracją. W celu opisu teorii obietnic por. z 3.3.1.

CFEngine, podobnie jak Puppet działa w trybie *pull*, czyli agent pobiera dane z centralnego serwera. W przeciwieństwie do Puppeta, CFEngine zapisuje pobrane *polityki* na lokalnej maszynie, dzięki czemu w przypadku braku łącza do maszyny centralnej, jest w stanie kontrolować i ewentualnie korygować konfigurację maszyny w stanie offline.

3.4.2 Zalety i Wady

Zalety:

- dojrzałość projektu
- odporność na przerwy w działaniu łącza
- bardzo duża skalowalność

Wady:

- trudna konfiguracja
- trudna instalacja

3.5 Ansible

3.5.1 Opis

Ansible jest również narzędziem do zarządzania konfiguracja serwerów. Został napisany w języku Python i w przeciwieństwie do poprzedników nie wymaga instalacji żadnego oprogramowania na maszynach klienckich. Wymaga jedynie, aby na maszynach które będą miały być obsługiwane przez Ansible, był zainstalowany serwer SSH oraz interpreter języka Python. Obie te rzeczy są instalowany domyślnie przez znaczna większość dystrybucji. Zalecane jest również skonfigurowanie logowania przy użyciu kluczy SSH, jednak wpływa to tylko na bezpieczeństwo i wygodę użytkowania.

3.5.1.1 Tryb aktywny i pasywny

Inną cechą odróżniającą Ansible od jego alternatyw jest kierunek działania. Ansible jest system działającym w trybie aktywnym, natomiast Puppet, Chef bądź CFEngine działają pasywnie. Znaczy to, że działanie Ansible jest wymuszane przez administratora poprzez wywołanie odpowiedniego *playbooka*, w przeciwieństwie do pozostałych, gdzie demon działający na serwerze klienckim odpytuje serwer z konfiguracją w celu pobrania aktualnych polityk. Ansible tutaj daje administratorowi większe pole do działania, ponieważ, po wykonaniu *playbook*-a dostaje on raport, jakie kroki zostały podjęte, które polityki były spełnione a które nie, oraz czy jakieś

akcje się nie powiodły.
Pozwala on również w prosty sposób na konfigurację działania w trybie quasi-pasywnym poprzez zastosowanie np: *cron*-a do cyklicznego wykonywania *playbook*-a. W efekcie Ansible daje możliwość pracy w trybie aktywnym jak i pasywnym.

3.5.1.2 Instalacja

Istnieje kilka metod instalacji Ansible

- ze źródeł
  Jest to najprostsza metoda instalacji. Ponieważ Ansible jest napisane w języku Python, nie wymaga on kompilacji ani ingerencji w system.

  Listing 3.4: instalacja ze źródeł

  ```
  admin@machine:~$ git clone git://github.com/ansible/ansible.git
  admin@machine:~$ cd ./ansible
  admin@machine:~/ansible $ source ./hacking/env-setup
  ```

  Metoda ta wymaga jednak aby w systemie zainstalowane były biblioteki Pythonowe:

  – paramiko
  – PyYAML
  – jinja2
  – httplib2

  Wykonanie powyższego kodu powoduje przełączenie się na wirtualne środowisko Pythona przygotowane przez developerów Ansible.
  Wirtualne środowisko zostanie opisane w kolejnym podrozdziale.

- przez repozytorium
  Ansible jest obecne w repozytorium praktycznie każdej dystrybucji. Instalacja zależna jest od konkretnej dystrybucji.
  Ta metoda może powodować problemy z używaniem Ansible w wirtualnym środowisku Pythona ponieważ narzędzie instalowane jest globalnie, natomiast środowisko wirtualne często tworzone jest bez dostępu do globalnych bibliotek.

- przez PIP
  Jest to zalecana metoda instalacji, ponieważ łączy w sobie prostotę procesu z elastycznością. Wykorzystuje on repozytorium bibliotek Pythonowych - `pip`.
  Instalacja Ansible poprzez `pip` wymaga wykonania polecenia:

Listing 3.5: instalacja poprzez PIP

```
admin@machine:~ $ pip install ansible
```

spowoduje ono ściągnięcie najnowszej wersji Ansible jak również zależnych pakietów. Instalacja odbędzie się do katalogów zdefiniowanych w zmiennych środowiskowych. Domyślnie są do główne katalogi `/usr` jednak mogą one zostać nadpisane przez użycie wirtualnego środowiska.

3.5.1.3 Wirtualne środowisko Pythonowe

Wirtualne środowisko jest narzędziem pozwalającym na stworzenie odizolowanego od bibliotek systemowych środowiska uruchomieniowego dla aplikacji pythonowych.
Nowe środowisko tworzone jest przy pomocy polecenia `virtualenv`. Tworzy ono strukturę katalogów potrzebną interpreterowi Pythona do działania. Poniżej znajduje się przykład takiej struktury:

Listing 3.6: struktura wirtualnego środowiska

```
admin@machine:~ $ tree -L 3 -d default3
default3/
|- bin
|- lib
|  |- python3.4
|  |- collections -> /usr/lib/python3.4/collections
|  |- distutils
|  |- encodings -> /usr/lib/python3.4/encodings
|  |- importlib -> /usr/lib/python3.4/importlib
|  |- lib-dynload -> /usr/lib/python3.4/lib-dynload
|  |- plat-x86_64-linux-gnu -> /usr/lib/python3.4/plat-x86_64-linux-gnu
|  |- __pycache__
|  |- site-packages
|- share
|  |- man
|  |- man1

14 directories
```

Poleceniem `source <plik_aktywacji>` aktywujemy wirtualne środowisko. Powoduje to nadpisanie domyślnych ścieżek przeszukiwania z domyślnych systemowych na lokalne w wirtualnym środowisku. Następnie należy uruchamiać interpreter Pythona nie podając bezpośredniej ścieżki do niego (np: `/usr/bin/python2`) lecz poprzez zmodyfikowane środowisko: `/usr/bin/env python`. Całość pozwala na tworzenie wirtualnych środowisk ze specyficznymi wersjami Pythona, różnymi niż domyślny interpreter w systemie jak również instalacja odpowiednich bibliotek dla konkretnego projektu a nie dla całego systemu. Używanie wirtualnego środowiska nie wy-

maga również posiadania konta administratora. Wirtualne środowisko zwiększa również przenośność projektów. Istnieje możliwość wyeksportowania do pliku tekstowego przy pomocy `PIP`-a listy zainstalowanych wraz z ich wersjami. Pozwala on również za zaimportowanie na nowym środowisku, dokładnie tych samych bibliotek, to tworzy dokładną kopię środowiska źródłowego oraz ułatwia migrację aplikacji pomiędzy maszynami.

3.5.2 Struktura

Struktura Ansible jest bardzo prosta i skupia się na trzech podstawowych elementach. Są nimi:

inventory
: jest to plik zawierający listę hostów które mają być zarządzane.

moduły
: Ansible używa modułów w celu wykonywania konkretnych operacji. Pozwala na pisanie własnych modułów.

playbooki
: pliki zawierające całościowy opis stanu jaki ma zostać osiągnięty na konfigurowanych hostach.

3.5.2.1 Inventory

Plik *inventory* zawiera listę wszystkich hostów które mogą znajdować się pod kontrolą Ansible. W każdej linijce pliku znajduje się definicja jednego hosta. Format definicji hosta wygląda następująco:

```
<hostname> [klucz1=wartosc1]...
```

`hostname` jest nazwą wyświetlaną przez Ansible podczas generowania raportów, jak również nazwą po której będzie probował się łączyć do serwera. Jeżeli `hostname` nie jest rozwiązywane przez używany serwer DNS, należy użyć specjalnej opcji, aby powiedzieć Ansible pod jakim adresem znajduje się dany serwer. Poniżej znajduje się lista kilku najczęściej wykorzystywanych opcji. Pełna lista znajduje się w dokumentacji projektu Ansible.

ansible_ssh_host
: określa adres IP pod którym znajduje się dany serwer.

ansible_ssh_port
: określa port który ma zostać wykorzystany przy połączeniu. Przydatne gdy serwer SSH działa na niestandardowym porcie.

ansible_ssh_private_key_file
: określa położenie klucza prywatnego używanego podczas połączenia. Użyteczne gdy nie chcemy używać domyślnego klucza, bądź jeżeli któryś z serwerów ma inna bazę zaakceptowanych kluczy

Dodatkowo, można zdefiniować swoje własne zmienne, które można następnie wykorzystać przy ustalaniu parametrów modułów bądź w szablonach. Jak zostanie pokazane w dalszej w kolejnych sekcjach, definiowanie parametrów w pliku `inventory` nie jest zalecane. Zalecanym sposobem definiowania zmiennych jest używanie `host_vars` co zostanie przedstawione w dalszej części.

Dopuszczalne jest również definiowanie hostów poprzez użycie zakresów zarówno liczbowych jak i znakowych. Dla przykładu, dopuszczalne jest zdefiniowanie dwudziestu serwerów o nazwach `node01`, `node02` aż do `node20` poprzez poniższą definicję:

```
node[01:50]
```

bądź `hostA` do `hostE`:

```
host[A:F]
```

Istnieje również możliwość łączenia kilku hostów w grupy i późniejsze definiowanie zachowań w odniesieniu do grupy a nie każdego serwera osobo. Grupa serwerów tworzona jest poprzez podanie w nawiasach kwadratowych nazwy grupy, po czym pod nią następuje standardowe listowanie serwerów. Wszystkie serwery zdefiniowane po nazwie grupy a przed deklaracją następnej, należą do grupy pierwszej. Dla przykładu:

Listing 3.7: inventory

```
[centos]
node[1:6]

[http]
node2
node3

[haproxy]
node1

[dc1:children]
http
haproxy
```

tworzy trzy grupy hostów. Pierwsza grupa `centos` zawiera sześć węzłów i widzimy tutaj definiowanie hostów poprzez zakres. Następna grupa `http` zawiera dwa hosty. Oraz ostatnia grupa jest jednoelementowa.
Można zauważyć, że jeden serwer może być członkiem więcej niż jednej grupy.
Została również zdefiniowana grupa `dc1` której członkami są serwery należące do grup `http` oraz `haproxy`. Widzimy więc, że można tworzyć również grupy składające się z innych grup.

Możliwe jest, chociaż również nie zalecane, zdefiniowanie zmiennych dla całej grupy. Zmienne takie definiuje się w następujący sposób:

```
[grupa1]
node1
node2
[grupa1:vars]
zmienna1=wartosc1
zmienna2=wartosc2
```

Jednak, podobnie jak w przypadku zmiennych ustawianych dla hostów, istnieje mechanizm `group_vars` i jest on zalecanym mechanizmem ustawiania zmiennych dla grup.

Ostatnią interesującą rzeczą dotyczącą pliku `inventory` jest fakt, że plik ten nie musi być plikiem tekstowym, a może być skryptem wykonywalnym. Ansible jest w stanie wykonać taki skrypt i jeśli zwrócona treść jest poprawnym formatem `inventory` potraktuje to wyjście jako `inventory`.

3.5.2.2 Moduły

Ansible wyposażony jest w dużą gamę gotowych modułów. Moduły w Ansible są to skrypty napisane w języku Python oraz zwykle wykonują jedną konkretną rzecz do której zostały stworzone. I tak na przykład mamy moduły:

users
: służy do zarządzania użytkownikami w systemie. Pozwala on na tworzenie, usuwanie, oraz modyfikowanie wszelkim parametrów użytkowników kont, takich jak np: katalog domowy, hasło czy domyślna powłoka, ale również pozwala na automatyczne wygenerowanie klucza ssh dla użytkownika podczas tworzenia konta.

git
: służy do zarządzania repozytoriami `git`-a. Pozwala na klonowanie oraz aktualizacje repozytorium `git`-a. Daje możliwość wyboru gałęzi bądź *commit*-a na który ma być *zdeployowana* aplikacja bądź wybór konkretnego pliku z kluczem ssh który zostanie wykorzystany do połączenia.

apt/yum/pip/portage/...
: zestaw modułów pozwalających na zarządzanie oprogramowaniem na serwerze. Wspólną i najważniejszą opcją dla wszystkich modułów z tego grupy jest opcja `state`. Może ona przyjmować co najmniej trzy wartości `present/absent/latest`. Oznaczają one:

  present
  : jeżeli pakiet nie jest zainstalowany w systemie, to go zainstaluj. Jeżeli nie podano wersji, instalowana jest najnowsza. Natomiast pakiet już jest w systemie to moduł zwraca komunikat "OK".

absent

zasada odwrotna co przy stanie *present*. Jeżeli pakiet jest zainstalowany, to zostanie on usunięty. A jeżeli nie było danego pakietu zainstalowanego w systemie, to moduł nie zrobi nic.

latest

Jest to stan bardziej skomplikowany niż dwa poprzednie, ponieważ w przypadku gdy pakiet nie jest zainstalowany, następuje jego instalacja do wersji najnowszej. Natomiast, jeżeli pakiet jest już zainstalowany, sprawdzane jest, czy zainstalowana wersja jest najnowszą dostępną w repozytorium. Jeżeli tak nie jest, to następuje aktualizacja pakietu do wersji najnowszej.
Są to jedne z niewielu domyślnych modułów których trzeba używać jawnie w zależności od dystrybucji systemu GNU/Linux na serwerze.

service

moduł zarządzający uruchamianiem usługami. Pozwala na zdefiniowanie poprzez parametr `enable` czy usługa powinna być uruchamiana przy starcie systemu. Drugim ważnym parametrem jest opcja `state` który może przyjmować następujące wartości:

started

upewnia się, że usługa jest uruchomiona. Jeżeli tak nie jest, uruchamia usługę.

stoped

upewnia się, że usługa jest zatrzymana. Jeżeli tak nie jest, zatrzymuje ją.

restarted

przeprowadza procedurę restartu usługi niezależnie od jej aktualnego stanu.

reloaded

przeładowuje daną usługę

powyżej zostało wymienionych tylko kilka z całej bogatej gamy modułów, jak również zostały one opisane tylko w najczęściej używanym zakresie. Pełnej listy modułów oraz ich parametrów należy szukać w dokumentacji

Playbook-i zostaną opisane w osobnej sekcji.

3.5.3 Użytkowanie

3.5.3.1 Tryb pracy Ad-Hoc

Ansible pozwala na wywołanie konkretnego modułu z konkretnymi parametrami. W przeciwieństwie do opisywanych w kolejnej sekcji *playbook*-ów, tryb Ad-Hoc przydaje się do szybkich jednorazowych operacji takich jak restart systemu bądź sprawdzenie aktualnych ustawień serwerów DNS na maszynach.

Listing 3.8: ansbile ad-hoc

```
(env)mgr@mgr0:~/SZZ$ ansible all -m shell -a "cat /etc/resolv.conf|grep nameserver"
mgr7 | FAILED => SSH encountered an unknown error during the connection. We recommend
        you re-run the command using -vvvv, which will enable SSH debugging output
        to help diagnose the issue
mgr9 | FAILED => SSH encountered an unknown error during the connection. We recommend
        you re-run the command using -vvvv, which will enable SSH debugging output
        to help diagnose the issue
mgr6 | FAILED => SSH encountered an unknown error during the connection. We recommend
        you re-run the command using -vvvv, which will enable SSH debugging output
        to help diagnose the issue
mgr5 | FAILED => SSH encountered an unknown error during the connection. We recommend
        you re-run the command using -vvvv, which will enable SSH debugging output
        to help diagnose the issue
mgr8 | FAILED => SSH encountered an unknown error during the connection. We recommend
        you re-run the command using -vvvv, which will enable SSH debugging output
        to help diagnose the issue
mgr4 | success | rc=0 >>
nameserver 10.13.0.1

mgr2 | success | rc=0 >>
nameserver 10.13.0.1

mgr3 | success | rc=0 >>
nameserver 10.13.0.1

mgr1 | success | rc=0 >>
nameserver 10.13.0.1

mgr0 | success | rc=0 >>
nameserver 8.8.8.8
```

na powyższym przykładzie widzimy wywołanie *ad-hoc* polecenia `shell` który wywołuje powłokę na zdalnej maszynie. Widzimy ze wywoływane jest to dla grupy `all` która oznacza, ze należy wywołać polecenie dla wszystkich hostów zdefiniowanych w pliku `inventory`. Parametrem podanym do modułu było polecenie `cat /etc/resolv.conf|grep nameserver` które wypisuje adresy serwerów DNS używanych przez serwer. Na wyjściu widzimy, ze dla serwerół `mgr0-4` otrzymaliśmy linijki pliku `resolv.conf` zawierające adresy, natomiast dla pozostałych serwerów otrzymaliśmy informację, ze nie udało się połączyć z nimi. W tym przypadku było to spowodowane tym, ze nie zostały one włączone.

Inny, bardzo częstym zastosowaniem trybu *ad-hoc* jest wgrywanie pliku na serwer zdalny. Należy zaznaczyć, ze z poniższego przykładu, celem zwiększenia czytelności, zostały usunięte komunikaty o błędach połączeń do niewłączonych maszyn, jak również usunięte zostały powtarzające się komunikaty o udanym wykonaniu polecenia na pozostałych hostach. Należy również

zaznaczyć, ze hosty zostały wcześniej tak przygotowane, aby każdy zwrócił inny komunikat.

Listing 3.9: ansible ah-hoc output

```
mgr3 | FAILED => failed to parse: {"msg": "Could not replace file:
        /root/.ansible/tmp/ansible-tmp-1410485097.3-207744126329446/source
        to /tmp/test2.txt: [Errno 1] Operation not permitted", "failed": true}
        Exception OSError: (2, 'No such file or directory',
        '/tmp/.ansible_tmpsQm1butest2.txt')
        in <bound method _TemporaryFileWrapper.__del__ of <closed file '<fdopen>',
        mode 'w+b' at 0x7f2bf55931e0>> ignored

mgr1 | success >> {
    "changed": true,
    "dest": "/tmp/test2.txt",
    "gid": 0,
    "group": "root",
    "md5sum": "fe60965969111c5638b51944a7575887",
    "mode": "0644",
    "owner": "root",
    "size": 13,
    "src": "/root/.ansible/tmp/ansible-tmp-1410485098.33-180775292256028/source",
    "state": "file",
    "uid": 0
}

mgr2 | success >> {
    "changed": false,
    "dest": "/tmp/test2.txt",
    "gid": 0,
    "group": "root",
    "md5sum": "fe60965969111c5638b51944a7575887",
    "mode": "0644",
    "owner": "root",
    "path": "/tmp/test2.txt",
    "size": 13,
    "state": "file",
    "uid": 0
}
```

Na powyższym przykładzie widzimy trzy możliwe stany wywołania polecenia:

changed=true

Oznacza, ze polecenie zakończyło się sukcesem oraz że podmiot operacji uległ zmianie. W tym przypadku oznacza to, ze plik został wgrany na serwer i zmienił on stan. Zmianę stanu należy rozumieć jako utworzenie nowe pliku, zmianę jego treści, bądź któregoś z parametrów takich jak właściciel, prawa dostępu itp.

changed=false

Oznacza, że przeprowadzona operacja nie wprowadziła żadnych zmian do aktualnego stanu systemu. Jest to pożądany stan przy używaniu *playbook*-ów, co zostanie opisane w następnej sekcji.

failed

Oznacza, że nie udało się wykonać polecenia. Często komunikat `failed` niesie ze sobą opis błędu. Bądź zdefiniowany przez autora modułu, bądź odpowiedź systemu operacyjnego.

Ostatnim wartym wspomnienia modułem jest moduł `setup`. Uruchamiany jest poprzez polecenie:

```
ansible <host> -m setup
```

moduł ten służy do tzw. zbierania faktów. Czy audytu systemu pod kątem informacji o nim. Niestety, wyjście tego polecenia posiada ponad 200 linijek, dlatego nie zostanie ono tutaj załączone. Moduł ten dostarcza informacji m.in o:

- adresach IP maszyny
- architekturze
- wersji jądra
- dokładnych konfiguracji interface-ów sieciowych
- informacji o dyskach twardych: podziale na partycje, sektorach przypadających na partycje, rozmiarze sektora
- dystrybucji systemu
- konfiguracji sprzętowej

oraz wielu innych nie wspomnianych powyżej.

3.5.3.2 Playbook

Głównym celem używania Ansible, nie jest jednokrotne wywoływanie poleceń opisane w poprzednim przykładzie, lecz utrzymywanie stanu serwera w konkretnej konfiguracji. Do opisu pożądanego stanu, używane są tzw. *playbook*-i. Definiują one stan w jakim ma się znaleźć system po ich wykonaniu. I tak na przykład opisem stanu może być zdefiniowanie, że usłucha `apache2` ma być uruchomiona bądź, że pakiet `postfix` ma być zainstalowany w wersji $2.10.2-1$. Wtedy, przy każdym wywołaniu *playbook*-a, Ansible będzie sprawdzał czy te kryteria są spełnione, i w przypadku gdy któreś nie zostanie, ansible spróbuje doprowadzić system do stanu kiedy kryterium będzie spełnione.

Playbook-i wykorzystują do pracy moduły. Te same moduły których możemy używać w trybie *ad-hoc*.

Domyślnie, przed wykonaniem *playbook*-a, następuje zebranie faktów o hostach na których mają zostać wykonane operacje. Zbieranie faktów odbywa się przy pomocy modułu `setup` który został opisany w poprzedniej sekcji. Dane które zostaną zebrane mogę posłużyć zarówno do użycia ich w szablonach konfiguracji, jak również do podejmowania decyzji jakie operacje należy wykonać dla hosta. Jedną z najczęstszych decyzji które są podejmowane na podstawie faktów, jest podział systemów na rodziny systemów operacyjnych. Dla przykładu, systemy z rodziny *Debian* używają menadżera pakietów `apt` natomiast rodzina *Red Hat* `yum`-3.

Playbook-i oraz inne pliki wykorzystywane przez Ansible są zapisywane w formacie *Yaml*. Po dokładną specyfikację formatu *Yaml* odsyłam do dokumentacji.

Poniżej znajduje się przykładowy *playbook*, na podstawie którego zostaną opisane najważniejsze jego elementy.

Listing 3.10: example_playbook.yml

```
---
- hosts: www
  vars:
    zmienna: wartosc
  tasks:
  - name: instalacja serwera HTTP
    yum: name=httpd state=latest
  - name: uruchomienie apache
    service: name=httpd state=started
  - name: automatyczny start przy uruchomieniu
    service: name=httpd enabled=true

- host: db
  tasks:
  - name: wgranie configuracji
    template:
      src=templates/mysql.conf.jinja2
      dest=/etc/my.conf
    notify:
    - restart mysql
  handlers:
  - name: restart mysql
    service: name=mysql state=restarted
```

zwyczajowo, pliki `yaml` zaczynają się się od trzech znaków myślnika.
Następnie następuje określenie grupy hostów na których należy wykonać aktualnego *playbook*-a. W tym przypadku, jest to grupa `wwww`. Należy tutaj zaznaczyć, że taka grupa musi zostać

zdefiniowana w pliku `inventory`
W następnej kolejności możemy zdefiniować dodatkowe zmienne. Zmienne ustawione w *playbook*-u mają wyższy priorytet niż te zdefiniowane w `host_vars`, dlatego istnieje możliwość stworzenia *playbook*-a dla środowiska developerskiego nadpisującego parametry używane na środowisku produkcyjnym.
Kolejną, najważniejsza sekcja, jest zdefiniowane operacji które ma wykonać *playbook*. Składa się ona z nazwy, która jest nieobowiązkowa i służy jedynie informacji co w danej chwili robi *playbook* oraz z modułu z parametrami. Moduły używane w *playbook*-u, są tymi samymi modułami które były używane w trybie *ad-hoc*.
Istnieje możliwość zdefiniowania opcjonalnej sekcji `notify` w definicji zadania. Jej zadaniem jest wykonanie jakiejś operacji tylko w przypadku gdy efektem wykonania zadania będzie stan `changed`. Na załączonym przykładzie widzimy, że na zdalny serwer jest wgrywany plik konfiguracyjne do dazy MySQL. Plik jest tworzony na podstawie szablonu przy użyciu silnika *jinja2*. Opcja *notify* sprawia, że jeżeli nowy plik konfiguracyjny różni się od obecnego, następuje ponowne uruchomienie bazy danych. Dzięki temu nie ma potrzeby restartować bazy danych przy każdym uruchomieniu *playbook*-a oraz w przypadku zmiany konfiguracji, efekty są zauważalne od razu.

Podczas, gdy w trybie *ad-hoc* wykonanie jakieś modułu miało na celu wykonanie jakiejś operacji oraz zmiana stanu serwera, tak w przypadku *playbook*-ów, dąży się aby wywołanie zwracało jak najwięcej stanów `ok` oraz jak najmniej *changed*. Stan `ok` oznacza, że stan serwera jest zgodny z naszymi założeniami i nie ma potrzeby podejmować żadnych kroków. Stan `changed` pojawia się w przypadku zmian w konfiguracji jakiś usług, bądź w przypadku ingerencji manualnej administratora lub nieoczekiwanego zachowania aplikacji, np: wystąpienia błędu powodującego wyłączenie się aplikacj.
Uruchomienie *playbook*-a następuje poprzez polecenie `ansible-playbook` oraz podanie jako parametru nazwy pliku zawierającego konfigurację żądanego *playbook*-a. Dodatkowo, do wywołania można dodać parametry sterujące wykonywaniem *playbook*-a. M.in. liczbę równoległych połączeń, wskazać plik `inventory` bądź ograniczenie wykonywanych *task*-ów tylko do tych oznaczonym określonymi *tag*-ami.

3.5.3.3 Role

Jak łatwo zauważyć, w poprzednim przykładzie wykonaliśmy cztery operacje angażujące dwie grupy serwerów a plik zawierał 23 linijki kodu. Można w prosty sposób estymować, że rzeczywisty *playbook* będzie tych linii zawierał kilkaset bądź więcej, co sprawi ze jego czytanie i utrzymywanie stanie się bardzo trudne, jeśli nie niemożliwe. Dlatego wprowadzono mechanizm ról. Jest to zalecana forma definicji stanu poszczególnych typów serwerów. Role pozwalają na separacje zadań, zmiennych, plików, szablonów i innych. Pozwala to również na migrowanie definicji ról pomiędzy systemami.

Definicje ról znajdują się w katalogu `roles`, w którym znajdują się podkatalogi z nazwie roli. W każdym katalogu z rolą, może znajdować się od jednego do sześciu katalogów odpowiadających za poszczególne elementy roli. Poniżej przedstawione jest przykładowe drzewo zawierające dwie role.

Listing 3.11: struktura roli w ansible

```
roles
|- role1
|  |- files
|  |- templates
|  |- tasks
|  |- handlers
|  |- vars
|  |- meta
|- role2
|  |- templates
|  |- tasks
|  |- handlers
```

Ansible, podczas ładowania roli, sprawdza po kolei czy istnieją powyższe katalogi. Jeżeli katalog istnieje, sprawdzane jest, czy istnieje w nim plik `main.yml` (nie dotyczy to katalogów `files` oraz `templates`) i jeżeli istnieje, jego zawartość zostaje załączona do *playbook*-a. Następuje wywołanie dyrektywy `include`, która pobiera zawartość pliku i wstawia w miejsce w którym została wywołana.
Znaczenie poszczególnych katalogów jest następujące:

`files`

w katalogu tym powinny znaleźć się wszystkie pliki które dana rola będzie kopiowała na serwer zdalny. Umieszczenie plików w tym katalogu daje możliwość podania do modułu kopiującego jedynie nazwy pliku źródłowego zamiast pełnej ścieżki dostępu. Np:

```
- name: kopiowanie issue.net
- copy: src=issue.net dest=/etc/issue.net
```

zamiast

```
- name: kopiowanie issue.net
- copy: src=/home/admin/ansible/all_files/issue.net dest=/etc/issue.net
```

trzeba mieć na uwadze, że niejeden administrator, zamiast zrobić kopię pliku w katalogu z rolą, utworzy katalog zbiorczy ze wszystkimi plikami. W znacznym stopniu może to utrudnić migrację definicji takiej roli, ponieważ, oprócz katalogu z jej definicją, należy przenieść wszystkie pliki jej dotyczące ze wspólnego katalogu. Użycie katalogu `files` wewnątrz katalogu roli, rozwiązuje ten problem, ponieważ, przekopiowanie katalogu z rolą, zapewnia pełną kompatybilność niezależnie od systemu.

`templates`

w katalogu tym, znajdują się wszystkie definicje szablonów których może używać rola. Tego katalogu tyczą się wszystkie zastrzeżenia dotyczące katalogu `files`

Ansible korzysta z silnika *jinja2*, który jest wzorowany na silniku szablonów wykorzystywanym we frameworku Django.
Szablony pozwalają na wykorzystywanie w obie wartości zmiennych. Zmienne te można można podać w pliku *inventory* co zostało opisane wcześniej w sekcji `vars` w *playbook*-u, bądź w plikach przygotowanych do tego celu.
Pierwszym miejscem gdzie można umieszczać zmienne są katalogi `host_vars` oraz `group_vars` które należy utworzyć w katalogu głównym *playbook*-a. Wewnątrz nich tworzy się pliki odpowiednio: z nazwami hostów bądź z nazwami grup. w przypadku `host_vars` host o nazwie zgodniej z nazwą pliku, będzie zawierał zmienne w nim zdefiniowane, natomiast w przypadku `group_vars`, wszystkie hosty należące do grupy zgodnej z nazwą pliku, będą miały zdefiniowane zmienne umieszczone w pliku.
Przykładowy plik `host_vars` definiujący port SSH po którym Ansible powinien łączyć się do hosta:

Listing 3.12: temida.yml

```
---
ansible_ssh_port: 24401
```

Natomiast poniższy przykład definiuje nazwę pakietu programu Apache2 oraz lokalizacje konfiguracji *vhost*-ów dla systemu rodziny Debian.

Listing 3.13: debian.yml

```
---
apache: apache2
apache_vhosts: /etc/apache2/sites-enabled
haproxy_conf: /etc/haproxy/haproxy.cfg
```

Drugim jest katalog `vars` który zostanie wyjaśniony poniżej.

`tasks`

jest to katalog w którym znajdują się definicje zadań które należy wykonać dla danej roli. Jest to jedyny wymagany katalog w definicji roli.
W katalogu tym powinien znaleźć się plik `main.yml`. Po odnalezieniu pliku `main.yml`, następuje jego załączenie do *playbook*-a. Operacja którą można przestawić w następujący sposób:

```
tasks:
 - include: tasks/main.yml
```

co jest równoznaczne z wpisaniem zawartości tego pliku do sekcji *tasks* jednak pozwala na zachowanie czytelności kodu.
Poniżej został przedstawiony przykładowy plik `tasks/main.yml`

Listing 3.14: tasks/main.yml

```
- name: Instalacja libselinux-python
  yum: name=libselinux-python state=latest
- name: Dodawanie repo epel
  copy: src=epel.repo dest=/etc/yum.repos.d/epel.repo owner=root mode=0644
- name: Dodawanie klucza repo epel
  copy: src=RPM-GPG-KEY-EPEL-6 dest=/etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6
- name: Wlaczanie repo epel
  ini_file: dest=/etc/yum.repos.d/epel.repo section=epel option=enabled value=1
- name: Instalacja epel-release
  yum: name=epel-release state=latest
```

widzimy, że powyższa rola upewnia się ze w systemie jest zainstalowana najnowsza wersja pakietu `libselinux-python`. Następnie moduł `copy` kopiuje konfiguracje dodatkowych repozytoriów na serwer zdalny. Pliki te znajdują się w katalogu `files`. Następnie widzimy użycie modułu `ini_file` który jest w stanie edytować pliki konfiguracyjne. Po wskazaniu mu nazwy pliku oraz sekcji, następuje upewnienie się ze opcji `enabled` ma ustawioną wrtość 1. W przypadku gdyby wartość byłą inna, bądź nie istniała, nastąpi jej dopisanie, tak aby stan końcowy był zgodny z założonym.

`handlers`

w katalogu tym również umieszcza się plik `main.yml` w którym umieszcza się funkcje obsługujące wystąpienie statusu `changed`.
Stosują się do niego wszystkie zastrzeżenia wymienione w sekcji o katalogu `tasks` Przykładem takiego pliku jest ten zamieszczony poniżej:

Listing 3.15: handlers/main.yml

```
- name: restart_http
  service: name={{apache}} state=restarted
```

widzimy tutaj definicję akcji o nazwie *restart_http*. Używa ona modułu `service`, który, jak zostało wspomniane wcześniej, obsługuje uruchamianie procesów. Jako nazwę widzimy zmienną o nazwie `apache`. Jest to zmienna zdefiniowana poprzednio w pliku `group_vars` i oznacza nazwę usługi Apache pod systemem Debian.

`vars`

znajdują się w nim definicje zmiennych dotyczących roli. Podobnie jak w przypadku poprzednich katalogów, następuje wyszukanie pliku `main.yml` oraz załadowanie go do sekcji `vars` w *playbook*-u.

`meta`

w tym katalogu znajdują się informacje o zależnościach pomiędzy rolami. Katalog zachowuje się jak `tasks`.

Dla przykładu, tworząc role `apache2`, `mysql`, `php` oraz `lamp`, mamy możliwość w katalogu `meta` roli `lamp` zdefiniować zależności:

Listing 3.16: meta/main.yml

```
---
dependencies:
- { role: apache2 }
- { role: mysql }
- { role: php }
```

czego efektem będzie, po przypisaniu do hosta roli `lamp`, przed wykonaniem zadań z tej roli, zastosowanie ról `apache2`, `mysql`, `php` na tym serwerze.

Powoduje to zmniejszenie rozmiaru *playbook*-a, jak również tworzenie *playbook*-ów w duchu "co zrobić" a nie "jak zrobić". Definiujemy ze dane hosty mają mieć strukturę *LAMP* bez podawania szczegółów jak należy to zrobić. Jest to niejako stworzenie dodatkowej warstwy abstrakcji pomiędzy wysoko ideowymi *playbook*-ami a nisko ideowymi rolami.

Rozdział 4

Testowanie metod klastrowania

4.1 Środowisko testowe

Wszystkie rozwiązania testowane będą przy użyciu następującego środowiska testowego:

Maszyna fizyczna:

- CPU: `Intel(R) Core(TM)2 Quad CPU Q9400 @ 2.66GHz` posiadający wsparcie dla wirtualizacji (VT-x)
- Pamięć RAM: `8G DDR2`
- OS: `Gentoo Linux 64bit, kernel 3.18.1`
- Platforma wirtualizacyjna: `KVM` (host)

Maszyna wirtualna na potrzeby kontenerów:

- CPU: Mapowany z maszyny fizycznej. Przydział 3 rdzeni
- Pamięć RAM: `2G`
- OS: `Ubuntu Linux 64 bit, kernel 3.13.0-24-generic`
- Platforma wirtualizacyjna: `KVM` (guest), `LXC` (host)

Kontenery `LXC` do celów testowania aplikacji:

- OS: Ubuntu linux. Jądra współdzielone z maszyną hostującą.
- Ustawienia `cgroups`: `lxc.cgroup.cpu.cfs_quota_us = 30000`

Maszyna wirtualna na potrzeby `LVS`:

- CPU: Mapowany z maszyny fizycznej. Przydział 1 rdzeń
- OS: `Ubuntu Linux 64 bit, kernel 3.13.0-24-generic`
- Pamięć RAM: `192M`

Aplikacje działające w `userspace`, tj. `apache, nginx, haproxy, php-fpm`, zostają uruchamiane w dedykowanych kontenerach `LXC`. Usługi działające w warstwie jądra, tj. `LVS` zostają uruchomione na dedykowanej maszynie wirtualnej przy użyciu `KVM`.

4.2 Wybór serwera WWW

W rozdziale tym zostanie przedstawione zestawienie kilku testów wydajnościowych dwóch serwerów WWW.

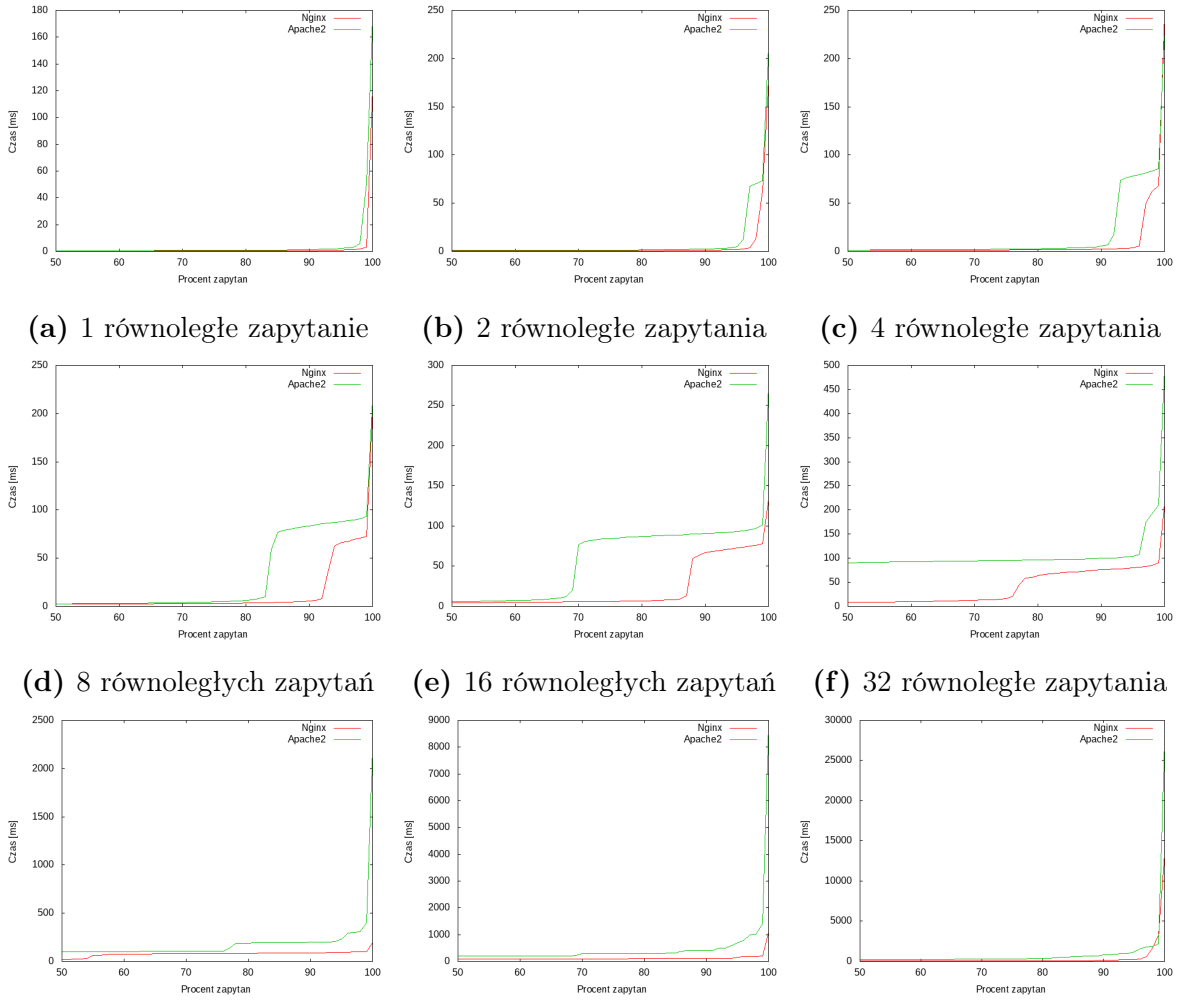
- Apache2
- Nginx

Przetestowane zostanie serwrowanie plików statycznych oraz treści dynamicznych PHP.
Wszystkie testy zostały przeprowadzone z wykorzystaniem 10 000 połączeń.
Wszystkie czasy zostały podane w milisekundach.

4.2.1 Pliki statyczne

Testy plików statycznych przeprowadzone zostaną przy użyciu dwóch plików HTML. Jeden o rozmiarze 10 bajtów, drugi o rozmiarze 100 kilobajtów.

Wykresy na rys. 4.1 przedstawiają czasy obsłużenia zapytań o plik HTML o rozmiarze 10 bajtów, natomiast wykresy na rys. 4.2 czasy zapytań o plik o rozmiarze 100 kilobajtów.
Można zauważyć, że wyniki dla małych plików statycznych są zbliżone zarówno dla Apache jak i Nginx z lekką przewagą dla Nginx. Największą przewagę Nginx-a widać przy średnim i dużym obciążeniu.

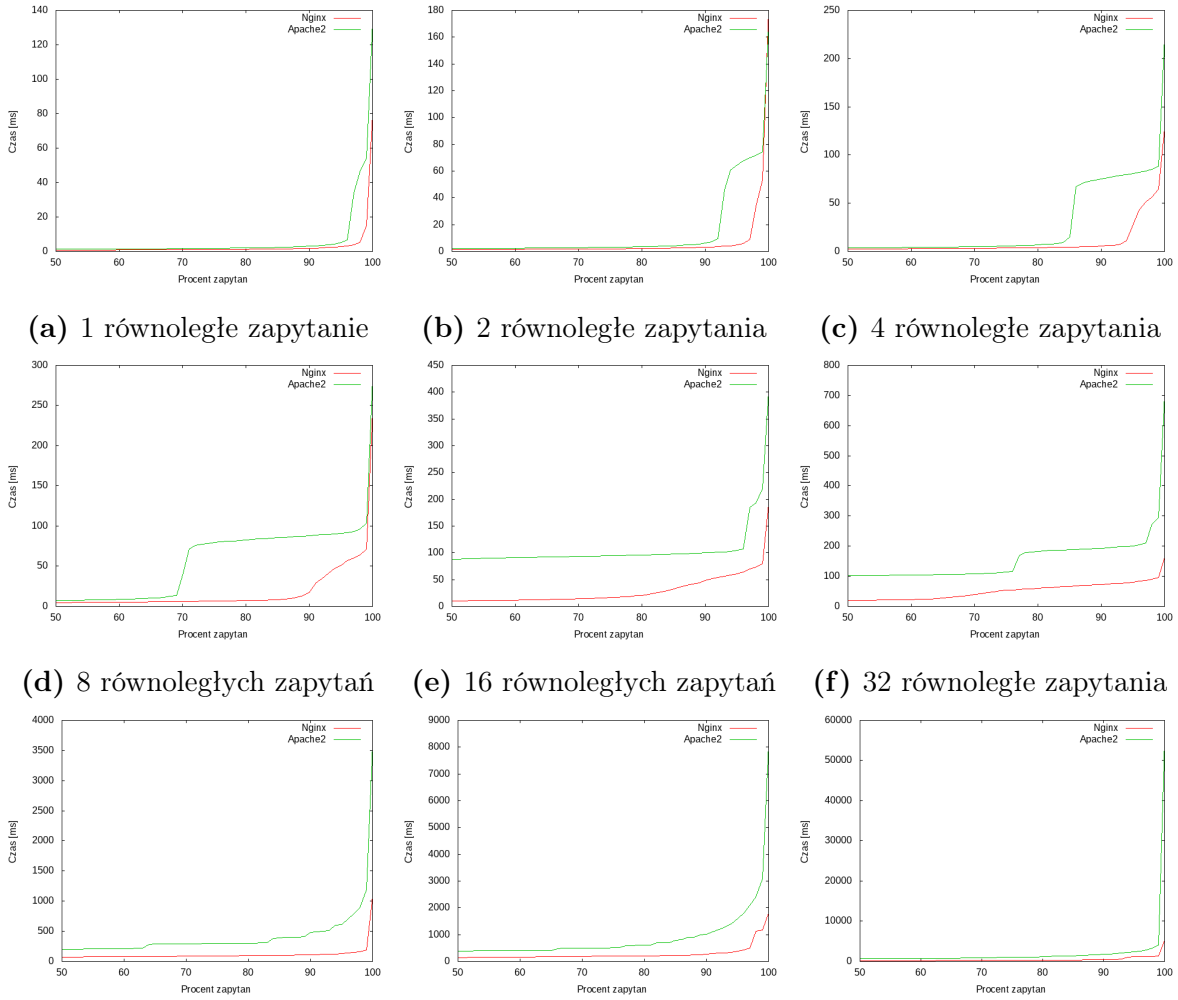


(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.1: Zapytania o mały plik statyczny



(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.2: Zapytania o duży plik statyczny

4.2.2 Treść dynamiczna

Testy treści dynamicznej przeprowadzane są przy użyciu konfiguracji Nginx + php-fpm oraz Apache + php-fpm. Konfiguracja Apache + mod_php została odrzucona, ponieważ wymaga umieszczenia serwera WWW oraz serwera PHP na jednym serwerze, co uniemożliwia użycie wielu serwerów PHP dla jednego serwera WWW.
Do testów zostały wykorzystane dwa bliźniacze skrypty obliczające liczby ciągu Fibonacciego. Jeden ze skryptów został przedstawiony na listingu 4.1. Obliczane są wyrazy: piąty — dla skryptu wykonującego się szybko, oraz piętnasty — dla skryptu wykonującego się dłużej.
Wykorzystany został model obliczanie wartości rekurencyjny, ponieważ w przeciwieństwie do iteracyjnego wymaga większej mocy obliczeniowej. Jest to pożądane aby czas obsługi zapytania obejmował czas wykonywania skryptu, a nie jedynie obsługi sesji HTTP oraz transferu danych (zostało to przetestowane przy wykorzystaniu *szybkiego skryptu*).

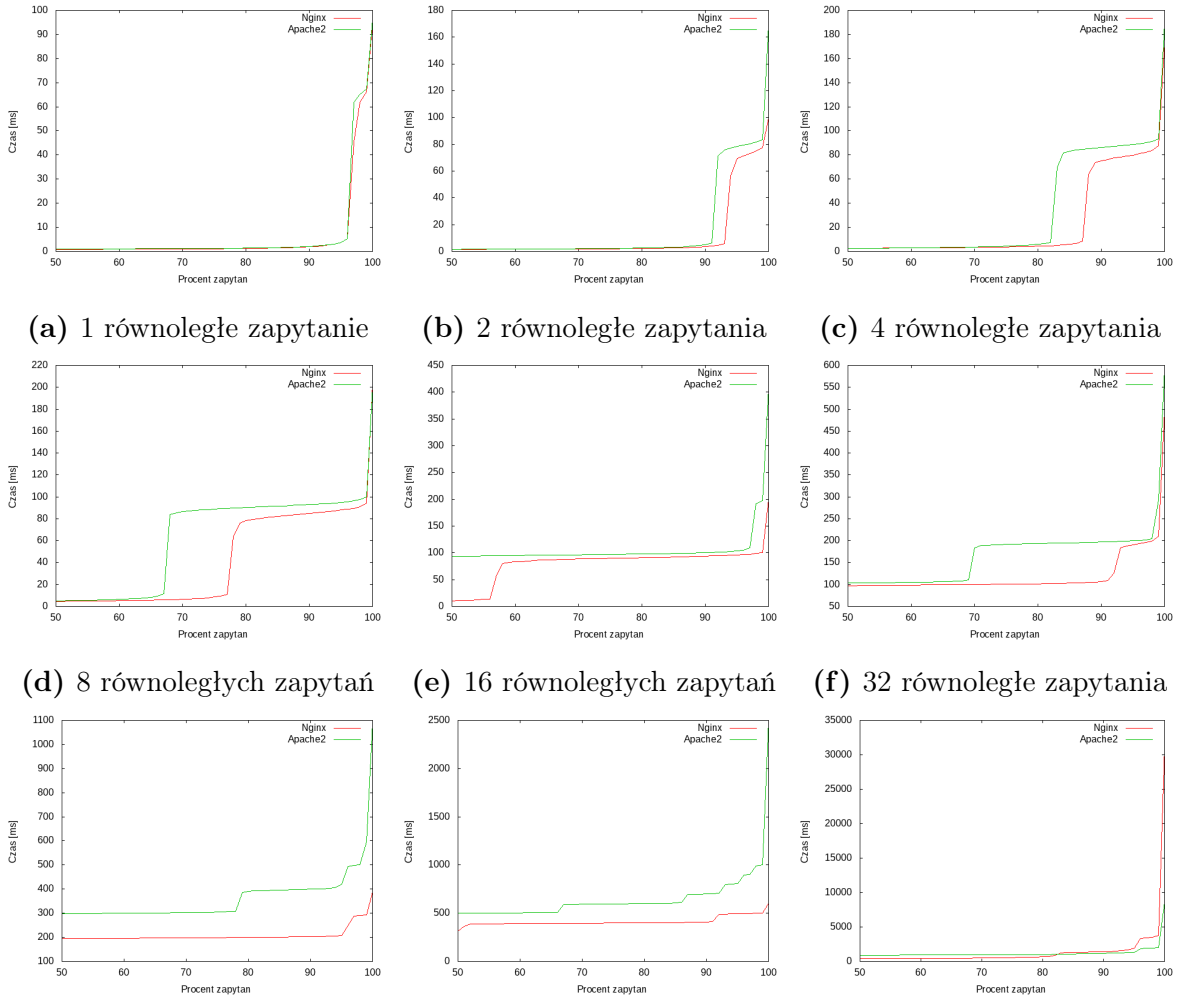
Listing 4.1: fib.php

```
<?php
function fib($n)
{
        if($n==1)
                return 1;
        if($n==2)
                return 1;
        return fib($n-1) + fib($n-2);
}
echo fib(15);
?>
```

Na wykresach 4.3 oraz 4.4 obrazujących czasy obsługi zapytań do skryptów PHP, można zauważyć że różnice pomiędzy Apache a Nginx są mniejsze niż dla plików statycznych. Wynika to z faktu, ze obsługą zapytań w obu przypadkach zajmuje się PHP-fpm, natomiast serwer WWW odpowiedzialny jest jedynie za przekazywanie zapytań do *backendu*.

4.2.3 Podsumowanie

Jak wykazały testy, Nginx daje krótsze czasy odpowiedzi we wszystkich testowanych sytuacjach, dlatego został wybrany jako podstawowy serwer wykorzystywany w przedstawionym projekcie.

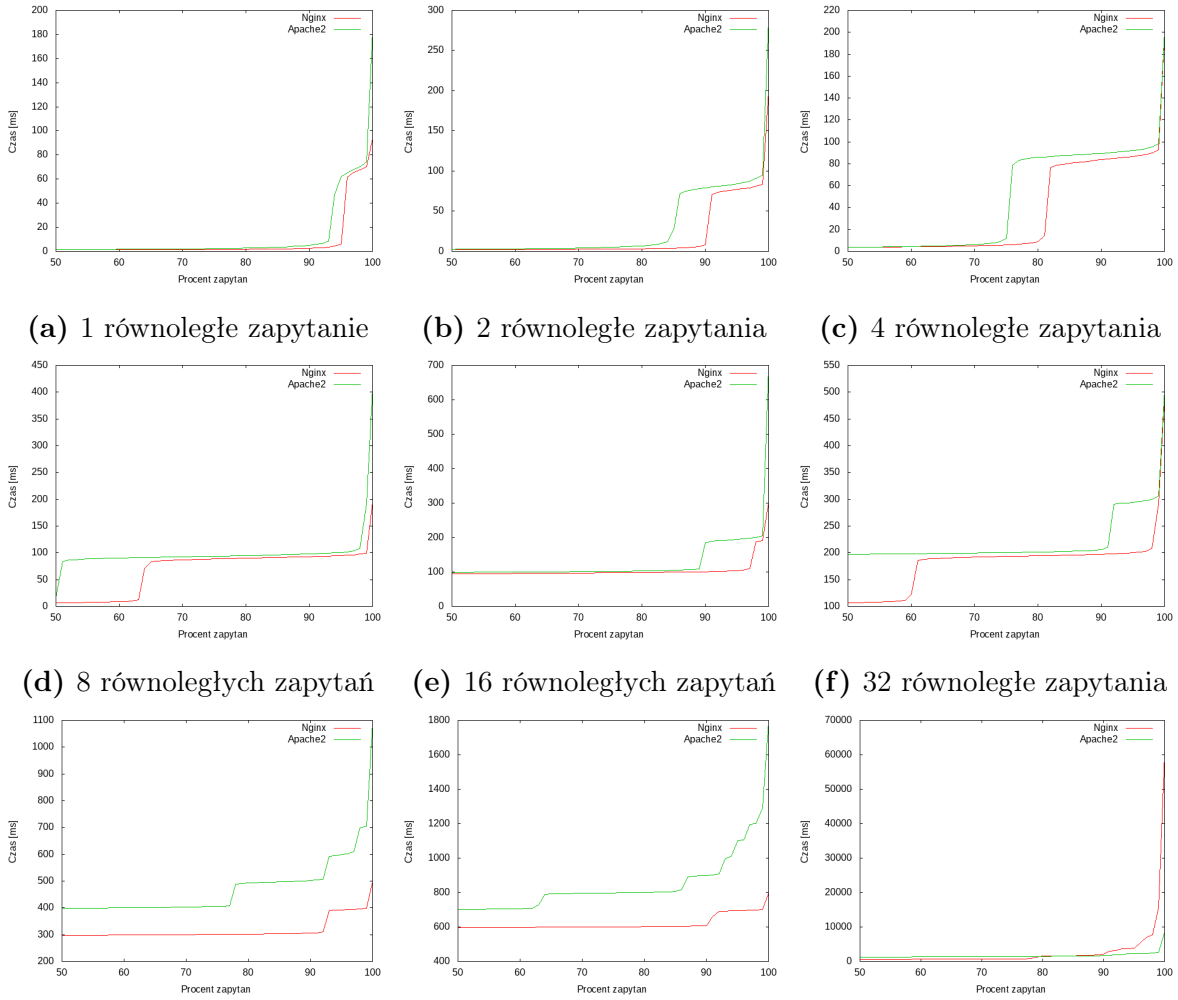


(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.3: Zapytania o szybki skrypt PHP



(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.4: Zapytania o wolny skrypt PHP

4.3 DNS round robin

4.3.1 Uwagi wstępne

Jak zostało wspomniane w rozdziale 2.1, DNS round robin nie jest metodą *load balancingu* a jedynie wstępną metodą dystrybucji ruchu. Pociąga to za sobą pewne konsekwencje. Przeciętna aplikacja sieciowa, np: Chrome bądź links, dokonuje rozwiązania nazwy przy pierwszym odwołaniu się do danego adresu a następnie w celu zwiększenia wydajności, *cachuje* jej adres. Efektem tego takiego zachowania jest nawiązywanie późniejszych połączeń do tego samego adresu IP. Aplikacja *Apache Benchmark* która jest wykorzystywana do testowania rozwiązań w poniższej pracy również dokonywała jednorazowego rozwiązania nazwy i łączenia się do jednego adresu, co uniemożliwiało przeprowadzenie testów wydajnościowych *DNS round robin* przy użyciu tego narzędzia. Została zgłoszona poprawka[4] poprawiająca tą niedogodność. Testy z wykorzystaniem tej poprawki symulują łączenie się dużej liczby niezależnych użytkowników.

Przy testowanie *DNS round robin* nie zostanie przeprowadzona tak dokładna analiza jak przy wyborze serwera WWW, ponieważ metoda ta nie wpływa na działanie serwera WWW a jedynie na dystrybucję ruchu.

W testach tych, w przeciwieństwie do wyboru serwera WWW, użyto opcji `Keep Alive` powodującej utrzymywanie połączenia i wykonywanie kolejnych bez ponownego nawiązywania połączeń TCP. W przypadku użycia poprawki [4] każde połączenie jest tworzone do serwera wynikającego z technologi *DNS round robin* a następnie jest ono utrzymywane aż do zakończenia programu. Liczba połączeń zależy od parametru `concurrent`

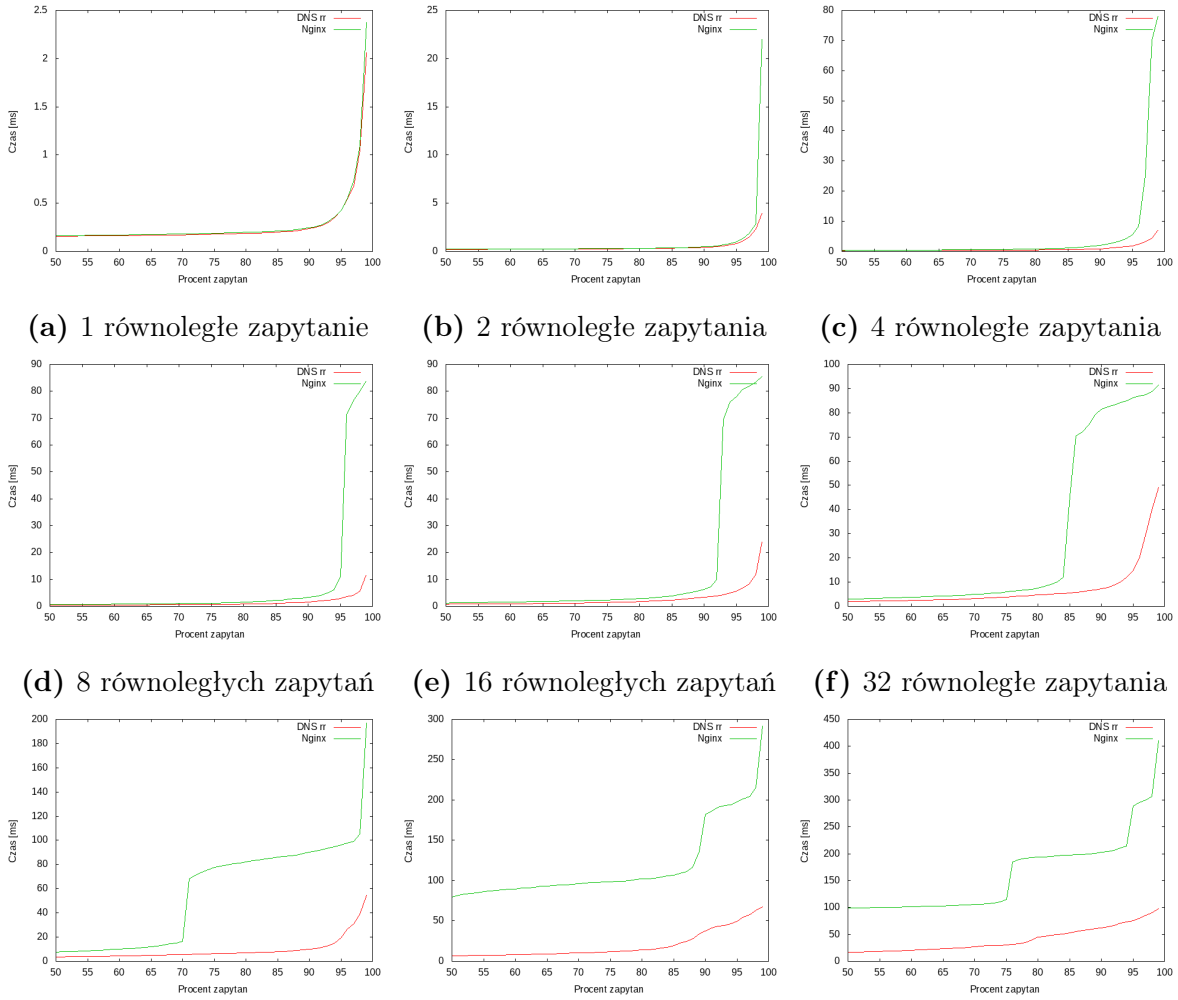
4.3.2 Testy wydajnościowe

Testowanie rozwiązania *DNS round robin* zostanie przedstawione na odpytywaniu jedynie pliku statycznego ponieważ testy dla pozostałych treści będą analogiczne do tych przedstawionych w rozdziale 4.2.
Do testów został wykorzystany *DNS round robin* przedstawiony na listingu 4.2.

Listing 4.2: DNS round robin

```
(default3.2)[nie 15/03/01 13:40 CET][pts/32][x86_64/linux-gnu/3.18.1-gentoo][5.0.7]
<torgiren@redraptor:~/SZZ/tekst/lst>
zsh/6 13647  (git)-[tekst]-% dig rr.mgr.fabrykowski.pl +short
10.13.0.14
10.13.0.11
10.13.0.12
10.13.0.13
```

Zawiera on cztery serwery z identyczną konfiguracją.
Wykresy 4.5 przedstawiają czasy obsługi zapytań w przypadku pojedynczego serwera WWW



(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.5: Dystrybuja ruchu w oparciu o DNS RR

oraz grupy serwerów z dystrybucją ruchu poprzez *DNS round robin*. Na wykresach został pominięty czas najdłuższego połączenia z powodu małego wkładu informacyjnego oraz dużego stopnia zaciemniania obrazu. Tabela 4.1 przedstawia liczbę obsłużonych zapytań na sekundę.

4.3.3 Podsumowanie

Jak można zaobserwować na wykresach, użycie dystrybucji ruchy zmniejsza czasy oczekiwania na przetworzenie zapytań.

Dane przedstawione w tabeli 4.1 pokazują, iż użycie czterech serwerów zamiast jednego daje w większości przypadków przyrost ok 300%, co jest wartością oczekiwaną, ponieważ przy użyciu trzech dodatkowych serwerów oczekujemy trzykrotnego przyrostu wydajności.

W przypadku jednego równoległego połączenia oczekiwanym przyrostem były przyrost 0% ponieważ wykonywane jest jedno połączenie, jednak specyfika serwera Nginx jest taka, że po przyjęciu stu połączeń w trybie `Keep-Alive` zamyka on połączenie. Następuje wtedy nawiązanie kolejnego, które zostaje przekierowane do kolejnego serwera z puli *DNS round robin*. Jest

Liczba połączeń	Nginx	DNS RR	Przyrost
1	2571	4301	+67%
2	2001	5458	+172%
4	1353	6264	+362%
8	1704	7582	+344%
16	2097	8720	+315%
32	2133	7484	+250%
64	1968	10617	+439%
128	1945	10194	+424%
256	2190	9821	+348%

Tabela 4.1: Obsłużonych zapytań na sekundę

to przyczyną zwiększonego przyrostu dla jednego i dwóch równoległych połączeń.
Przy liczbie połączeń cztery i więcej powyższy efekt nie ma znaczenia, ponieważ w obsłudze zapytań biorą udział wszystkie serwery.

4.4 LVS

W tej sekcji przetestowany zostanie LVS w trybie *direct routing*, ponieważ jest on najrozsądniejszym trybem. Testom poddany zostanie narzut wynikający z użycia LVS, jak również przetestowane zostaną rozwiązanie bliższe rzeczywistości, czyli zawierające kilka *real server*-ów. Ponadto, sprawdzone zostanie zachowanie LVS w warunkach nieidealnych, tj. w przypadku awarii jednego z *real server*-ów oraz w przypadku wysycenia łącza.

4.4.1 Uwagi dotyczące urządzeń sieciowych

Ponieważ w omawianym rozwiązaniu występuje jeden węzeł do którego nawiązywane są połączenia, parametry łącza są w tym przypadku (w przeciwieństwie do DNS round robin gdzie klient łączy się do różnych serwerów) istotne.

Maszyna z *directorem*

- Wirtualizowany sterownik karty sieciowej: rtl8139
- Szybkość karty podawana przez ethtool: 100Mb/s

Maszyna z LXC

- Wirtualizowany sterownik karty sieciowej: virtio
- Szybkość karty podawana przez ethtool: Brak

Maszyna z Nginx

- Wirtualizowany sterownik karty sieciowej: wirtualny interface hosta
- Szybkość karty podawana przez ethtool: 10000Mb/s

Na uwagę zasługują tutaj parametry maszyny będącej hostem dla LXC. Użycie sterownika `virtio` pozwala maszynie wirtualizowanej komunikować się z maszyną hostującą bez emulacji konkretnych sterowników sieciowych, lecz już na poziomie jądra. Dlatego nie jest możliwe określenie prędkości takiego interface-u, ponieważ zasada jego działania jest inna niż rzeczywistych kart sieciowych.
Podobnie sytuacja wygląda dla kontenerów LXC. Tam komunikacja następuje poprzez wirtualne interface-y na jeden maszynie, dlatego prędkość podana przez `ethtool` wynosi 10Gb/s.

W przypadku testowania zachowania rozwiązania na wysycenie łącza, zastosowany zostanie duży plik HTML o rozmiarze ok 1MB. Pozwoli to zasymulować sytuacje w której żądany ruch będzie większy niż możliwości łącza serwera.

4.4.2 Narzut własny LVS

Aby sprawdzić jakie opóźnienie generuje mechanizm LVS, skonfigurowany zostanie jeden serwer *director* oraz jeden *real server*. Porównując czasy odpowiedzi serwera WWW odpytywanego bezpośrednio oraz poprzez LVS, otrzymamy jaki wpływ na działanie ma LVS. Żądany będzie mały plik statyczny.

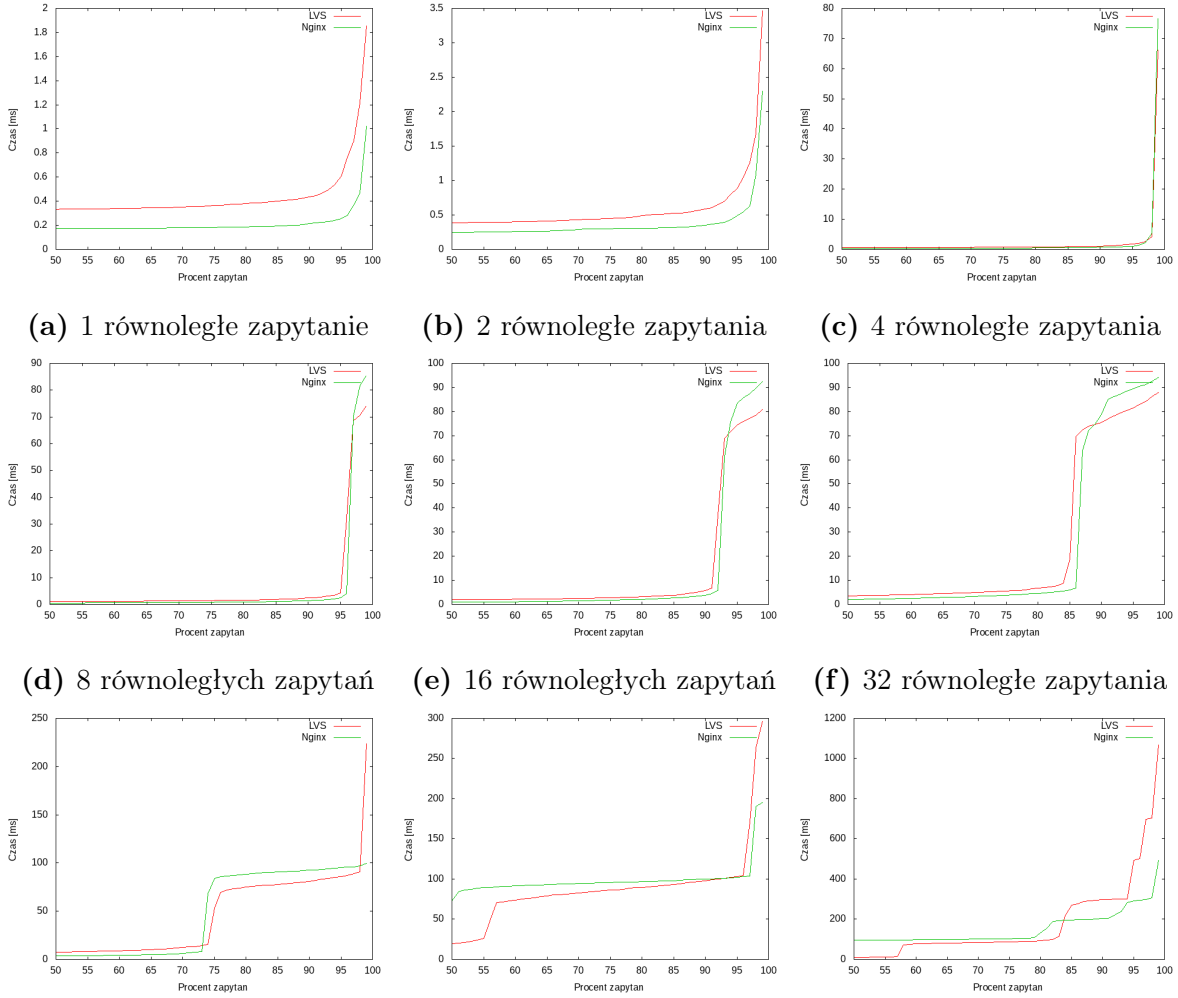
Jak widać na rysunku 4.6, czasy odpowiedzi serwera bezpośrednio oraz poprzez LVS nie odbiegają zbytnio od siebie. Różnica na wykresie jest widoczna jedynie dla jednego konkurencyjnego zapytania, jednak wartość tych czasów (0.2ms i 0.4ms) są na tyle małe, ze ta różnica może zostać pominięta.

4.4.3 Wiele real serverów

Rozważmy teraz przypadek, w którym mamy cztery takie same serwery WWW. Zostały skonfigurowane 3 usługi LVS. Tą konfiguracje przedstawia listing 4.3.

Listing 4.3: LVS z wieloma serwerami

```
root@lvs:/home/mgr# ipvsadm -L -n
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.13.0.101:80 rr
  -> 10.13.0.11:80                Route   1      0          0
```

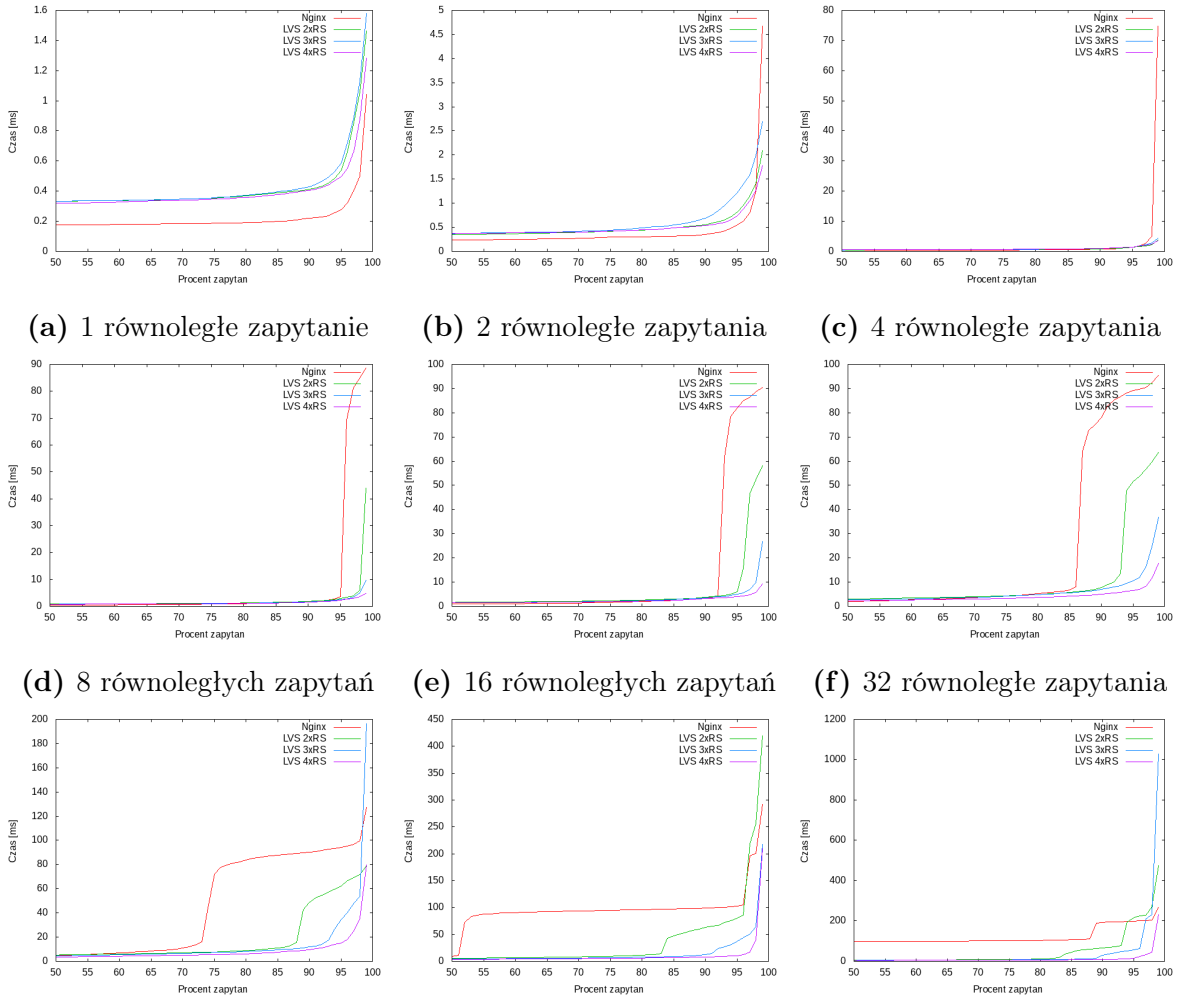


(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.6: Narzut własny LVS



(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.7: Czasy odpowiedzi LVS w zależności od ilości serverów

```
  -> 10.13.0.12:80                Route   1      0          0
TCP  10.13.0.102:80 rr
  -> 10.13.0.11:80                Route   1      0          0
  -> 10.13.0.12:80                Route   1      0          0
  -> 10.13.0.13:80                Route   1      0          0
TCP  10.13.0.103:80 rr
  -> 10.13.0.11:80                Route   1      0          0
  -> 10.13.0.12:80                Route   1      0          0
  -> 10.13.0.13:80                Route   1      0          0
  -> 10.13.0.14:80                Route   1      0          0
```

Widzimy, ze każda z usług posiada inną liczę serwerów oraz posiada taki sam algorytm balansowania. Każdy z serwerów ma wagę 1 oraz każdy jest wykorzystywany w trybie *direct routing*.

Wykresy 4.7 przedstawiają czasy odpowiedzi w zależności o ilości *real server*-ów. Wymaga on trochę dokładniejszej analizy niż poprzednie wykresy. Dla jednego równoległego zapytania, widzimy ze czasy każdej z usług LVS są gorsze niż bezpośrednie zapytania do server WWW. Dzieje się tak dlatego, ze mając jedno równoczesne zapytanie, wykorzystywany jest tylko jeden

serwer na raz, czyli jest to sytuacja tożsama z rozdziałem 4.4.2.
W przypadku 2 połączeń, wykresy zaczynają się zbiegać, ponieważ narzut LVS zaczyna być kompensowany poprzez dystrybucję zapytań.
Począwszy od czterech zapytań, zauważamy że połączenia kierowane do LVS są obsługiwane szybciej. Najbardziej jest to widoczne przy dużym obciążeniu serwera, tj. 128 i 256 równoległych zapytań.

4.4.4 Odporność na błędy

W rozdziale tym, omówione zostanie obsługiwanie błędów przez LVS.

Obsługa wysokiej wydajności

Niestety narzędzie wykorzystywane w tej pracy do testowania wydajności rozwiązań, posiada bardzo ubogą obsługę błędów, dlatego nie zostaną zamieszczone żadne wykresy.
Ponieważ LVS jest narzędziem nastawionym na wydajność i mały rozmiar, nie wspiera natywnie sprawdzania dostępności usług na *real server*-ach. W przypadku niedostępności usługi, pakiet zostaje przekazany do *real server*-a, który natomiast odpowiada błędem. Mogą to być błędy `connection refused, timeout, file not found` bądź wiele innych. Taki też błąd zostaje przekazany do klienta.
Sytuacja taka wynika z kilku czynników. Jednym z nich jest tryb działania LVS:

LVS DR/TUN
: W tym trybie odpowiedź jest kierowana bezpośrednio do klienta. W związku z tym, *director* nie jest świadomy błędnej odpowiedzi *real server*-a, dlatego nie może ponowić zapytania do innego, działającego.

LVS NAT
: W tym przypadku, ruch powracający przechodzi ponownie przez *directora* i istnieje techniczna możliwość wykrywania błędów w odpowiedzi. Jednak, aby zachować homogeniczność zachowania w każdym trybie, nie jest to wykonywane.

Kolejną przyczyną takiego zachowania LVS jest warstwa w której on pracuje. Jego głównym zadaniem jest przekazywanie pakietów i ingerencja w nie jedynie na warstwie drugiej a inspekcja jedynie do warstwy czwartej. Tak niskie działanie usługi w znacznym stopniu ograniczyło by możliwości monitorowania usług na *real server*-ach do sprawdzania możliwości połączenia na dany port (co w wielu przypadkach wystarcza).
Ponadto, LVS nastawiony jest na wydajność i dążenie, aby jego kod był jak najmniejszy i najszybszy. Dodanie do niego kodu obsługującego różne scenariusze sprawdzania dostępności usługi obniżyłyby tą wydajność.

Wszystkie powyższe powody sprawiają, że aby dodać do LVS wsparcie dla wysokiej dostępności, należy użyć dodatkowych narzędzi. Bądź po stronie LVS, które sprawdzają dostępność usług (na dowolnym poziomie) i ewentualnie *wypinają* z LVS niedostępne serwery, bądź rzadziej, po stronie *real server*-ów zastosowanie `floating IP`, nadające innej, działającej maszynie adres IP maszyny która uległa awarii.

Zachowanie w przypadku awarii

Jak zostało wspomniane w poprzednim paragrafie, *real server* odpowiada bezpośrednio do klienta, dlatego w przypadku awarii jednego z serwerów, zapytania kierowane na niego zostaną odrzucone. Spowoduje to zwrócenie błędu do klienta. Ten może to odczuć na kilka sposobów, np w przypadku stron WWW, jeżeli zapytanie o plik `index.html` zostanie przekazane do niedziałającego serwera, klient otrzyma w przeglądarce informację `connection refused`. Po odświeżeniu strony, zapytanie powinno trafić na inną maszynę, która zwróci treść strony.
To prowadzi do kolejnej sytuacji, w której zapytanie o `index.html` zakończyło się sukcesem, natomiast zapytania o kolejne pliki statyczne będą trafiać na niedziałającą maszynę. Spowoduje to niewczytanie się obrazów, arkuszy stylów bądź skryptów `java script`. Jest to sytuacja bardziej irytująca dla użytkownika, ponieważ zostaje mu przedstawiona część treści na stronie, lecz zwykle w sposób niepozwalający na jej komfortowe przeglądanie.
Trzecią sytuacją która może wystąpić - będącą chyba najgroźniejszą pod kątem logistycznym - jest poprawne załadowanie się całej treści strony, jednak zapytania dynamicznie generowane przez np: `Ajax`-a trafiają na niedziałający serwer i nie są przetwarzane. Zwykle klient nie jest informowany o niepowodzeniu przy takim zapytaniu, bądź jest informowany o niepowodzeniu bez podawania przyczyny. Jest to szczególnie niebezpieczne przy wykonywaniu ważnych operacji typu składanie wniosków bądź robienie przelewów w banku.

Wysycenie łącza

Do testowanie wysycenia łącza, został wybrany plik o rozmiarze 1MB. Niestety przy badanych maszynach nie udało się osiągnąć pełniej mocy łącza, a jedynie ruch na poziomie 900KB/s. Sytuacja taka spowodowana jest utylizacją pracy procesora na poziomie 95%.
Tak wysokie zużycie CPU wynika z potrzeby przekazania wszystkich pakietów `ACK` wracających od klienta.

Powyższy test pokazuje, iż LVS jest odporny na problem wysycenia łącza i jego wydajność ograniczają inne parametry maszyny.
W przeprowadzanym teście, przybliżona wartość wydajności wynosi ok 320 zapytań na sekundę.

4.4.5 Podsumowanie

Wykonane testy pokazują, iż *Linux Virtual Server* jest technologią bardzo wydajną. Jej narzut własny jest bliski zeru, natomiast przy większej ilości serwerów następuje bardzo efektywny rozdział zapytań.
Niestety LVS nie posiada żadnych metod wykrywania awarii *real server*-ów, dlatego w tym celu należy użyć dodatkowych narzędzi. Tabela 4.2 potwierdza przedstawione wnioski. W przypad-

Liczba połączeń	Nginx	LVS 4xRS	Przyrost
1	2697.30	2722.17	0%
2	2148.13	4610.76	114%
4	2182.04	5619.42	157%
8	1934.86	8103.03	318%
16	2182.62	8940.06	309%
32	2333.41	10542.87	351%
64	2278.52	9411.09	313%
128	2390.89	10907.20	356%
256	2648.01	9499.84	258%

Tabela 4.2: Obsłużonych zapytań na sekundę

ku jednego połączenia przyrost wynosi 0, natomiast dla połączeń 8 i więcej, przyrost oscyluje około wartości 300, co jest wartością oczekiwaną.

4.5 Haproxy

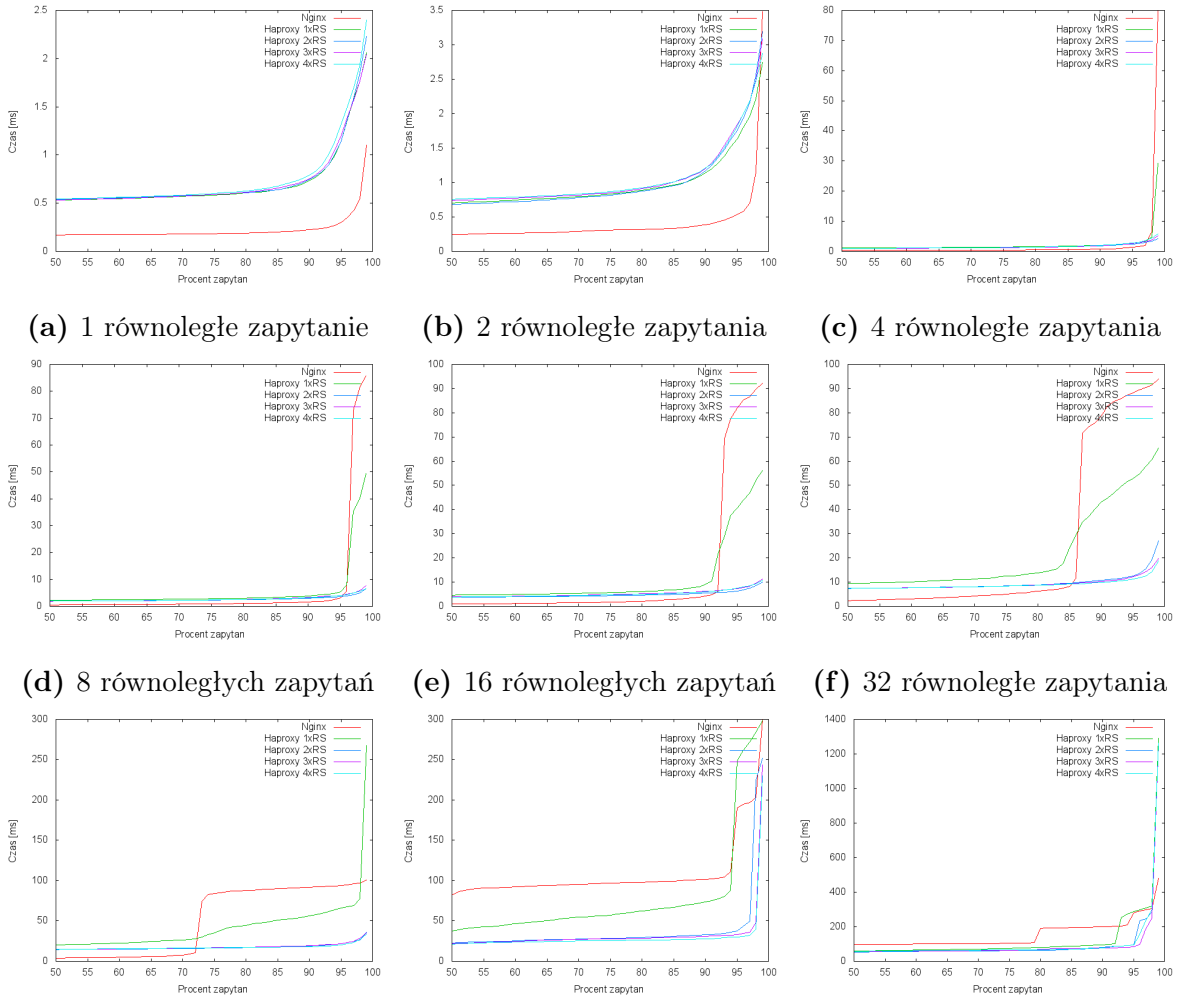
Rozdział ten skupi się na przetestowaniu wydajności aplikacji Haproxy, oraz jej odporności na awarie serwerów *backend*-owych.

4.5.1 Uwagi dotyczące urządzeń sieciowych

W celu zachowania porównywalnego środowiska co LVS, zostaną zastosowane wytyczne z rozdziału 4.4.1.

4.5.1.1 Wydajność

Rozdział ten skupi się na przetestowaniu wydajności Haproxy. Testowane zostanie bezpośrednie połączenie do Nginx-a oraz połączenia czterech usług Haproxy, zawierające odpowied-



(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.8: Czasy odpowiedzi Haproxy w zależności od ilości serwerów

nia, jeden, dwa, trzy i cztery serwery *backend*-owe.

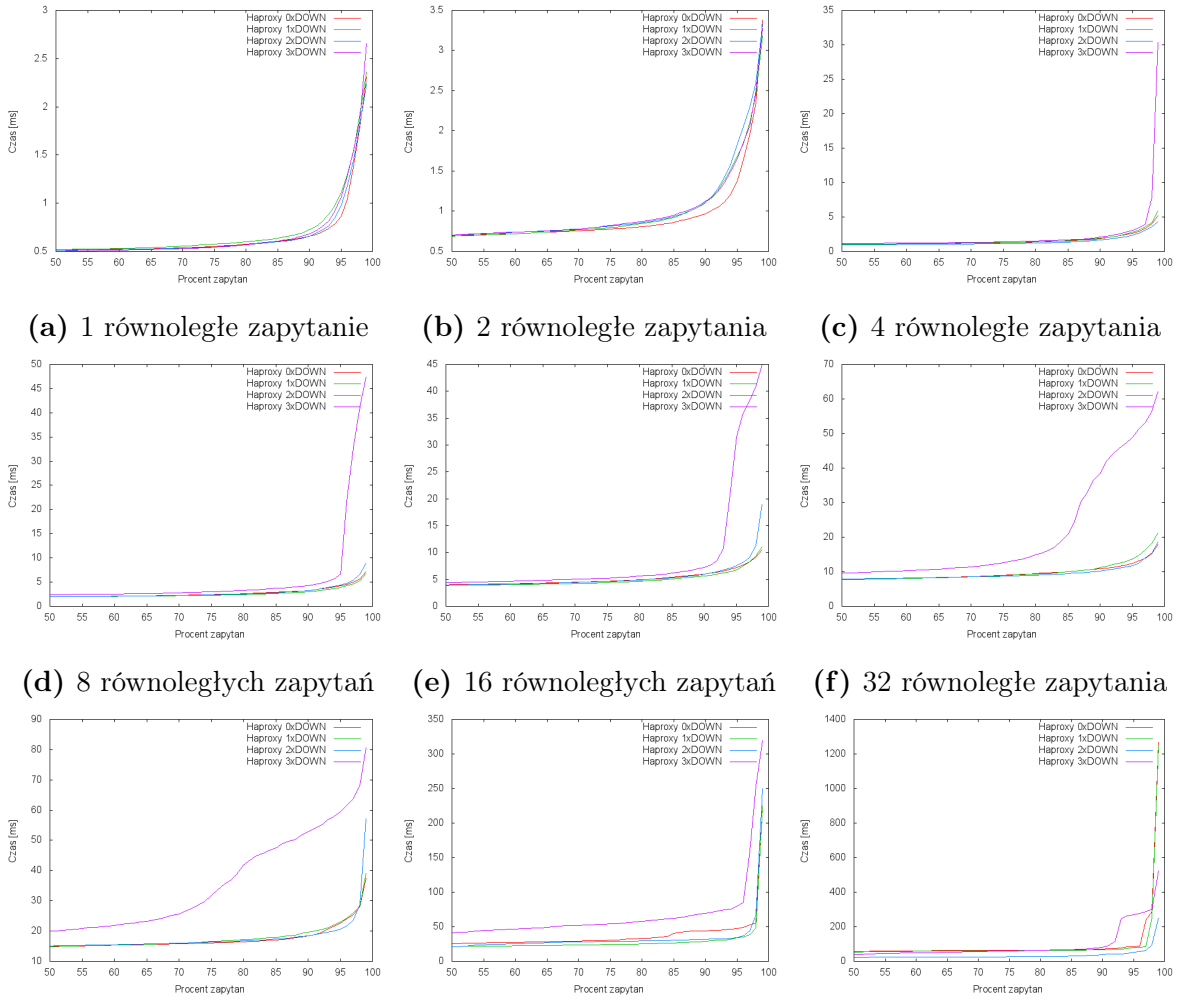
Zapytania, podobnie jak w LVS będą kierowane o mały plik statyczny. Wykres **??** przedstawia czasy odpowiedzi Haproxy oraz Nginxa w zależności od ilości równoległych zapytań.

Jak można zauważyć, dla niewielkiej liczby równoległych zapytać, czasy odpowiedzi Haproxy są zbliżone do natywnego Nginx-a. Wzrost wydajności widać dopiero przy większej ilości zapytań, tj. od około 64 równoległych zapytań.

4.5.2 Odporność na awarie

4.5.2.1 Awaria serwerów *backendowych*

Ponieważ Haproxy został zaprojektowany z myślą o wysokiej dostępności, posiada on natywną obsługę wykrywania awarii serwerów. W przypadku wykrycia awarii, dany serwer zostaje oznaczony jako `DOWN` i nie zostają do niego przekazywane zapytania. Zmniejsza to wydajność całego klastra, jednak zapytania użytkownika zostają przekazane do działających maszyn, dzięki czemu, klient zawsze otrzymuje odpowiedź.



(a) 1 równoległe zapytanie **(b)** 2 równoległe zapytania **(c)** 4 równoległe zapytania

(d) 8 równoległych zapytań **(e)** 16 równoległych zapytań **(f)** 32 równoległe zapytania

(g) 64 równoległe zapytania **(h)** 128 równoległe zapytania **(i)** 256 równoległych zapytań

Rysunek 4.9: Czasy odpowiedzi Haproxy w zależności od ilości serwerów i awarii węzłów

Aby zobrazować zachowanie Haproxy na awarie maszyn, zostaną przeprowadzone testy wydajnościowe z wszystkimi czterema serwerami działającymi, a następnie te same testy zostaną powtórzone dla wyłączonych usług Nginx na jednym, dwóch oraz na trzech maszynach.
Testy dla wyłączonych czterech maszyn nie zostaną przeprowadzone, ponieważ z powodu braku jakiekolwiek serwera *backend*-owego, klient nigdy nie otrzyma odpowiedzi. Wykresy przedstawione na rysunku 4.9 pokazują, że awaria węzłów nie powoduje błędnych odpowiedzi do klienta. Poprzednie spostrzeżenie jest prawdziwe pomijając drobną anomalię dla 32 i 64 połączeń, gdzie Haproxy z trzeba niedziałającymi serwerami daje trochę gorsze odpowiedzi.

Powyższe testy zapytań były przeprowadzane dopiero po ustaleniu przez Haproxy, niedostępności usługi po jej wyłączeniu.

Awaria węzła w trakcie obsługi zapytań

Poprzedni rozdział wykazał, iż po ustaleniu się stanu `DOWN` dla serwerów *backend*-owych, Haproxy nie powoduje żadnych błędów w odpowiedziach do klienta.

Kolejnym testem dla wysokiej dostępności jest zachowanie się Haproxy w przypadku awarii serwera dla połączeń obsługiwanych po awarii serwera a przed wykryciem tego faktu przez system.
Haproxy sprawdza dostępność usługi co jedną sekundę, natomiast po trzeciej nieudanej próbie połączenia, uznaje usługę za niedostępną. W efekcie, powstaje trzy sekundowe *okno* w którym klient może uzyskać odpowiedź o niedostępności usługi.
Test polegał na ciągłym wysyłaniu 128 równoległych zapytań do serwera. W tym samym czasie nastąpiło wyłączenie usługi Nginx na jednym z serwerów. W trakcie trzy sekundowego *okienka*, zostało zwróconych od 50 do 150 błędów (w zależności od uruchomienia testu).
Cały test trwał 14 sekund. W jego trakcie zostało wykonanych 30000 zapytań z czego ok 100 zakończyło się niepowodzeniem. W mojej opinii jest to bardzo wysoki poziom dostępności usługi.

Wysycenie łącza

Warunki testowanie wysycenia łącza dla Haproxy są analogiczne do tych stosowanych w LVS (rozdział 4.4.4).
W tym przypadku procesor podobnie został obciążony na 100%. Wykorzystanie łącza utrzymywało się na poziomie 10MB/s, natomiast liczba obłużonych zapytań na sekundę wynosiła około 20.

4.5.3 Podsumowanie

Jak wykazały testy, Haproxy daje bardzo duży poziom wysokiej dostępności. Można zauważyć bardzo krótki czas wykrywania niedostępności usługi, oraz bardzo dobrą reakcje minimalizującą ilość błędów zwracanych do klienta.
Tabela 4.3 przedstawia przyrost wydajności względem pojedynczego serwera WWW . Można zauważyć, że dla jednego połączenia mamy przyrost ujemny. Wartości przyrostów dla większej liczby połączeń również nie są zbyt wysokie. Sytuacja ta wynika z faktu zapotrzebowania na dużą moc obliczeniową związaną z *Nat*-owaniem pakietów oraz przeliczaniem adresów.

4.6 Porównanie LVS oraz Haproxy

Jak wykazały testy, oba omawiane systemy służą do balansowania połączeń pomiędzy realne serwery obsługujące zapytania. Zadaniem administratora jest ocena i wybór odpowiedniego narzędzia do odpowiedniej sytuacji.
Dla dużego ruchu odpowiedniejszy wydaje się LVS, gdyż jest dużo bardziej wydajny, jednak nie wspiera natywnie wysokiej dostępności. Administrator jest odpowiedzialny za dobranie innego narzędzia dodającego obsługę awarii serwerów, aby klient nie otrzymywał błędnych komunika-

Liczba połączeń	Nginx	Haproxy 4xRS	Przyrost
1	2741.46	1521.45	-44%
2	2142.82	2207.01	2%
4	2006.00	2940.17	46%
8	2133.27	3563.32	67%
16	2139.60	3532.11	65%
32	2226.88	4067.57	82%
64	2335.50	3930.86	68%
128	2183.21	4189.07	91%
256	2394.04	2875.26	20%

Tabela 4.3: Przyrost obsłużonych zapytań przez Haproxy

tów zwrotnych.
Natomiast przy obsłudze mniejszego ruchu, Haproxy zyskuje dzięki dobremu systemowi wykrywania awarii, co zapewnia ciągłą dostępność usługi dla klienta.

Warto zwrócić uwagę, ze dla jednego i dwóch równoległych połączeń, zarówno Haproxy jak i LVS mają bardzo dobre czasy odpowiedzi. Różnice zaczynają się dopiero dla większego obciążenia, co zostało wyjaśnione w poprzednim paragrafie.

Dodatkowo, można zauważyć różnicę w wykorzystaniu łącza. W testowanym środowisku, ani Haproxy ani LVS nie wysyciły łącza, gdyż procesor był elementem limitującym możliwości obu metod, jednak różnica wykorzystania łącza przy obecnych parametrach, pozwala wysnuć teorię, że przy wykorzystaniu mocniejszych maszyn w środowisku produkcyjnym, różnica ta może mieć znaczenie, gdy ich możliwości zaczną być limitowane przez łącze.

4.7 Nietestowane rozwiązania

W tym rozdziale nie zostały poruszone wszystkie kwestie związane z wysoką dostępnością. Oto najważniejsze pozycje które nie zostały przetestowane.

- możliwości Nginx-a do rozdzielania połączeń. Zostały pominięte ponieważ możliwość wykrywania niedostępności usług jest dostępna jedynie w płatnej subskrypcji. Z powodu mnogości darmowych rozwiązań open source-owych, płatne moduły nie były testowane.
- heartbeat/keepalied/ldirector/...- szereg narzędzi dodających funkcjonalność wysokiej dostępności do LVS.

- corosync/...- narzędzia do zapewniania wysokiej dostępności dla urządzeń brzegowych. Pozwalają ustawić tzw. *floating IP*. W przypadku awarii jednej maszyny (np: *director*-a LVS), druga, zapasowa, wykrywa ten fakt i ustawia odpowiednie adresy na swoim interface-ie oraz konfiguruje odpowiednie usługi, tak, aby móc przejąć funkcje maszyny która uległa awarii.

Rozdział 5

Opis projektu

5.1 Opis

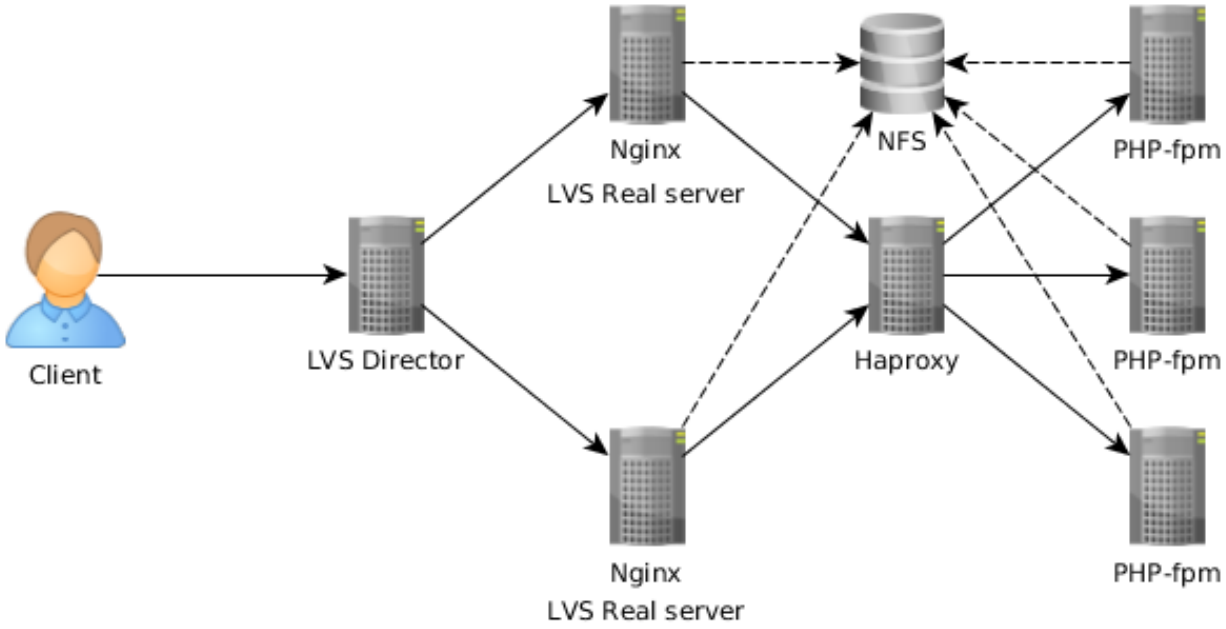
W poprzednich rozdziałach zostały przedstawione metody klastrowania oraz zarządzania konfiguracją, a następnie zostały one przetestowane. Została również wykazana zasadność ich stosowania.
Jednak, aby wdrożyć takie rozwiązania, potrzebna jest wiedza oraz czas pracy administratora.

System zautomatyzowanego zarządzania konfiguracją farmy serwerów aplikacji WWW (zwany dalej *SZZ*) ma za zadanie uprościć konfigurację klastra WWW, pozwalając zaoszczędzić czas i pieniądze. Opisywany system będzie mógł być obsługiwany przez osoby nie posiadające dogłębnej wiedzy z zakresu administracji systemami linux ani serwerami WWW. Wymagana jest jedynie podstawowa wiedza techniczna, którą posiada przeciętny programista.

Konfiguracja odbywa się poprzez edycję plików, dlatego obsługująca system powinna być w stanie obsługiwać połączenia `ssh` oraz edytor tekstowy.

5.2 Struktura

SZZ wykorzystuje różne metody klastrowania. Struktura systemu została przedstawiona na rys 5.1 Pierwszą warstwą klastrowania jest LVS (por. 2.4). Zapytanie trafiające na serwer obsługiwane są przez *direcotor*-a. Następnie przekazywane są do serwerów WWW, które analizując zapytania serwują treści statyczne ze wspólnego zasobu NFS. W przypadku zapytania o treści dynamiczne, zapytania przekazywane jest do warstwy trzeciej projektu, czyli usługi Haproxy, która przekazuje zadania do odpowiedniego serwera z usługą *PHP-fpm*. Po wygenerowaniu odpowiedzi, serwer roboczy zwraca odpowiedź do Haproxy, które przekazuje je do serwera WWW. Ten natomiast, będąc *real server*-em w klastrze LVS, odpowiada bezpośrednio klientowi.



Rysunek 5.1: Struktura SZZ

5.2.1 Warstwa zero - storage

Projekt posiada wspólny zasób dyskowy wystawiany poprzez protokół NFS. Metoda ta w znaczny sposób uprasza aktualizację aplikacji na wszystkich węzłach równocześnie - kosztem braku możliwości wykonywania tzw. *rolling update*. Ta technologi rozwiązuje również problem wgrywania plików na serwer oraz ich propagacji ponieważ każdy wgrywany plik trafia na wspólny zasób i jest od razu widziany przez pozostałe węzły.
Wydajność NFS jest zadowalająca przy wykorzystywaniu w obrębie jeden serwerowni i jednej sieci LAN. W przypadku chęci użycia rozproszenia systemu między kilkoma *datacenter* należy ze własnym zakresie obsłużyć synchronizację wgrywanych plików oraz aktualizacji.

5.2.2 Warstwa pierwsza - LVS

Jedynym wystawionym na świat serwerem jest *director*. Do niego trafiają wszystkie zapytania od klientów. Wykorzystana w projekcie konfiguracja używa *scheduler*-a opartego o algorytm *round robin*, czyli przekazuje zapytania na wszystkie serwery po kolei. Technologia LVS pozwala na posiadanie tylko jednego serwera typu *director*, ponieważ jego zadaniem jest jedynie przekazywanie zapytać do *real server*-ów. Ponadto, jak zostało omówione wcześniej, odpowiedzi do klienta wysyłane są bezpośrednio od *real server*-ów, bez udziału *director*-a co pozwala na obsługę nawet dużego ruchu.
Obecna konfiguracja nie posiada narzędzi do wykrywania niedostępności *director*-a bądź *real server*-ów, dlatego konfiguracja narzędzi typu *heartbeat* oraz technologi *floating IP* i/lub monitoringu stanu serwerów, leży po stronie użytkownika.

5.2.3 Warstwa druga - Nginx

Drugą warstwą jest warstwa serwerów WWW. Do nich trafiają zapytania przekazywane z pierwszej warstwy. Serwer WWW obsługujący wiele *Virtual Host*-ów, analizuje zapytanie pod kontem, czy żądana ścieżka jest istniejącym plikiem na dysku. Jeżeli plik istnieje, jest on serwowany klientowi. W przeciwnym wypadku, zapytanie zostaje przekazywane do haproxy.

5.2.4 Warstwa trzecia - Haproxy

Haproxy jest trzecią warstwą systemu. Przez tą warstwę przechodzą wszystkie zapytania o treści dynamiczne. Usługa tworzy osobny *frontend* oraz *backend* dla każdego projektu.
Haproxy posiada wbudowaną obsługę wykrywania, dlatego warstwa trzecia zapewnia pełna *HA*. Wysycenie łącza dla warstwy trzeciej nie powinna być problemem, ponieważ zapytania odbywają się jedynie po dane dynamiczne - zwykle tekstowe. Wszystkie zapytania o obrazy i inne treści statyczne zostają obsłużone warstwę wcześniej. System nie zapewnia wysokiej dostępności dla usługi haproxy. Administrator powinien skonfigurować monitoring aby móc taką awarię wykryć maksymalnie szybko i usunąć usterkę. W przypadku niemożliwości naprawy maszyny, system pozwala na skonfigurowanie nowej maszyny dla warstwy trzeciej oraz przekonfigurowanie w stosunkowo krótkim czasie.

5.2.5 Warstwa czwarta - PHP-fpm

Najniższą warstwą systemu jest warstwa robocza. PHP-fpm odpowiedzialny jest za generowanie treści dynamicznych. Podobnie jak serwer WWW, korzysta on ze współdzielonego zasobu dyskowego udostępnianego po NFS. Na jednej maszynie może być uruchomionych kilka aplikacji PHP-fpm.

5.3 Nazwa robocza: backend

Sekcja ta opisuje kroki jakie podejmuje system, aby skonfigurować klaster zgodnie z założeniami.

5.3.1 NFS

Aby skonfigurować serwer NFS, system instaluje potrzebne pakiety a następnie kopiuje plik konfiguracyjny na serwer. W następnej kolejności ustawia autostart server NFS oraz go uruchamia.
W drugiej kolejności, następuje instalacja *git*-a. Ostatnią wykonywaną rzeczą, jest *deploy* wszystkich aplikacji. *Deploy* wykonywany jest do aktualnej wersji w gałęzi *master*.

5.3.2 Director

Do skonfigurowania *Directora*, potrzebna jest instalacja pakietu `ipvsadm`, który dostarcza narzędzia do konfiguracji *Linux Virtual Server*. Konfiguracja *LVS* przeprowadzana jest poprzez użycie mechanizmu zapisu i odczytu aktualnej tablicy *LVS*. Tuż po instalacji, wykonywany jest zapis konfiguracji, w celu przeprowadzenia całej procedury zapisu tablicy do pliku. Następnie, generowany jest nowy plik konfiguracji na podstawie parametrów zadanych przez użytkownika. Plik ten jest wgrywany na serwer i podmienia poprzednio utworzony przy poleceniu zapisu. Następnie wykonywana jest procedura wczytywania tablicy z pliku do aktualnie działającej instancji. W efekcie, tablica wygenerowana przez system staje się aktualnie działającą. Następnie ustawia się autouruchamianie usługi `LVS`.
W drugiej kolejności, tworzony jest wirtualny interface sieciowy, oraz zostają mu przypisane adresy IP odpowiednie dla konkretnych projektów.
Każdy projekt nasłuchuje na dedykowanym sobie adresie IP. Daje to możliwość dedykowania konkretnych serwerów WWW dla projektów, zamiast przypisywać obsługę wszystkich serwerów dla każdego projektu.

5.3.3 Real server

Konfiguracja *real server*-ów jest zbliżona do *director*-a. Następuje stworzenie wirtualnego interface-u a następnie przypisanie mu odpowiednich adresów IP. Ważną różnicą w przypadku *real server*-ów jest zapewnienie, aby *real server*-y nie odpowiadały na zapytania `ARP`. Uzyskiwane jest to poprzez użycie `arptables`. System blokuje wszystkie pakiety typu *ARP response* i pochodzące z adresacji używanej przez *LVS* do nasłuchiwania przez projekty.

5.3.4 Server WWW

Serwer WWW instalowany jest na *real server*-rze. Następuje instalacja serwera `nginx` oraz ustawienie jego autouruchamiania.
W kolejnym kroku generowana jest konfiguracja *vhost*-ów. Dla każdej *hostgroup*-y z konfiguracji, zaczynającej się od *proj_* (założenie konfiguracji), tworzona jest osobna sekcja `server`. `Document root` jest ustawiany do odpowiedniego katalogu na zasobie sieciowym. Następnie konfigurowane jest *proxy* plików `php` do serwera z odpowiednio skonfigurowanym *haproxy*.
Obecna wersja oprogramowania nie wspiera przyjaznych linków, prowadzących do skryptów PHP a nie kończących się rozsrzerzeniem `.php`.

5.3.5 Server haproxy

Konfiguracja serwera haproxy polega, podobnie jak innych komponentów, na instalacji aplikacji, wgraniu konfiguracji oraz uruchomieniu usługi. System tworzy konfigurację dla haproxy

bazując na ustawieniach projektów użytkownika. Dla każdego zdefiniowanego w systemie projektu, `backend`, wykorzystujący algorytm *round robin*, a następnie umieszcza w nim wszystkie serwery typu `worker` odpowiedzialne za procesowany projekt. Następnie tworzy `frontend` nasłuchujący na porcie odpowiadającym *id* projetu oraz wykorzystujący odpowiedni `backend` Dodatkowo, *haproxy* wystawia na standardowym porcie 9000 statystyki aktualnie działającej instancji.

5.3.6 Serwer roboczy - `worker`

Serwery robocze pracują w oparciu o `PHP-fpm`, dlatego pierwszym krokiem konfiguracji jest upewnienie się, że jest on zainstalowany, a jeśli nie, to jego instalacja. Następnie wgrywana jest konfiguracja, która w typ przypadku zawiera konfiguracje *pool*-i dla każdego projektu obsługiwanego przez dany serwer roboczy. Każda pula nasługuje na porcie odpowiadającym `id` projektu.

5.4 Konfiguracja

5.4.1 Maszyny konfigurowane

Podstawowa konfiguracja maszyn będących w klastrze jest w gestii administratora oraz powinien posiadać system `Debian Linux` bądź `CentOS Linux`. Dodatkowo, muszą mieć uruchomioną usługę `SSH`. Zalecane jest również wgranie kluczy `RSA`. Nalezy również zapewnić, aby na maszynie był zainstalowany interpreter Pythona w wersji min. 2.6.

5.4.2 Maszyna konfigurująca

Podobnie jak w przypadku maszyn konfigurowanych, administrator musi zadbać, aby na maszynie był zainstalowany system GNU/Linux, Interpreter języka Python w wersji min. 2.6 oraz zalecane jest wygenerowanie pary kluczy `RSA`. Instalacja oprogramowanie Ansible, została opisane w rozdziale 3.5.1.2.

Podsumowanie

Niniejsza praca wykazała potrzebę stosowania klastrów WWW. Przy obecnym rozwoju internetu, ilość zapotrzebowania na dane jest daleko wykraczająca poza możliwości pojedynczych komputerów. Ponadto, niemożność dostarczenia klientowi żądanych danych jest równoznaczne z ponoszonymi przez firmę stratami finansowymi oraz wizerunkowymi.

Obecnie każdy popularny serwis internetowy wykorzystuje różnego rodzaju klastry. Poziom zaawansowanie konfiguracji w przypadku dużych firm wykracza daleko poza możliwości opisywanego systemu.

Ponadto, w niniejszej pracy zostały opisane tylko metody *software*owe, a zostały pominięte metody *hardware*owe. Wynika to z braku dostępu do profesjonalnego sprzętu.
Dodatkowo, z racji objętości pracy, pominiętę zostały niektóre, bardziej zaawansowane metody wykorzystywane przez duże korporacjie, np: zastosowanie modułu `GeoIP`, pozwalającego na serwowanie różnej treści klientowi w zależności od jego położenia geograficznego. Wykorzystywane jest to np: przy zwracaniu różnych adresów IP dla danej domeny. Klientowi zwraca się adres serwera w *datacenter* znajdującego się najbliżej, w celu zmiejszenia czasu odpowiedzi.
Nie został również poruszony temat tworzenia własnych modułów do `nginx`a pozwalających dostosować sposób dystrybucji ruchu do konkretnych warunków panujących w firmie.
Nie został również przetestowany serwer `lighthttp`. Wynika to z faktu dość specyficznej charakterystyki tego serwera, sprawiającej że potrafi w szybki sposób serwować pliki statyczne, lecz posiada mniejsze możliwości konfiguracyjne niż `apache` lub `nginx`.
Innym ważnym zaganieniem nieporuszonym w niniejszej pracy, jest użycie `keepalived` bądź innego narzędzia wspomagającego wykrywanie i wypinanie nieaktywnych węzłów z klastra `LVS`. Nieopisany również został mechanizm *floating IP*, powodujący na dynamiczne przepinanie publiczego adresu IP na zapasowy (*host swap*) węzeł - np: zapasowy `director`.

Jednak opisany system jest w stanie wystarzyć dla małej lub średniej strony. Wykorzystuje on najczęściej używane metody klastrowania i daje duże możliwości profilowania konfiguracji na potrzeby konkretnego rozwiązania.

Trzeba jednak pamiętać, że nawet najlepszy klaster wysokiej wydajności nie zastąpi optymalizacji aplikacji działającej

Bibliografia

[1] http://www.shorewall.net/images/Netfilter.png.

[2] http://nginx.org/en/docs/http/ngx_http_upstream_module.html.

[3] http://nginx.org/en/docs/http/ngx_http_fastcgi_module.html.

[4] https://bz.apache.org/bugzilla/show_bug.cgi?id=57616.

Spis rysunków

Listings

Spis tabel