Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Rok akademicki 2016/2017

Tomasz Rydzewski
Magdalena Malenda
Joanna Ohradka
Włodzimierz Szewczyk
Piotr Kuciński
Dominik Giżyński
Michał Herman

Projekt RSO

Spis treści

1. Projekt

1.1. Harmonogram

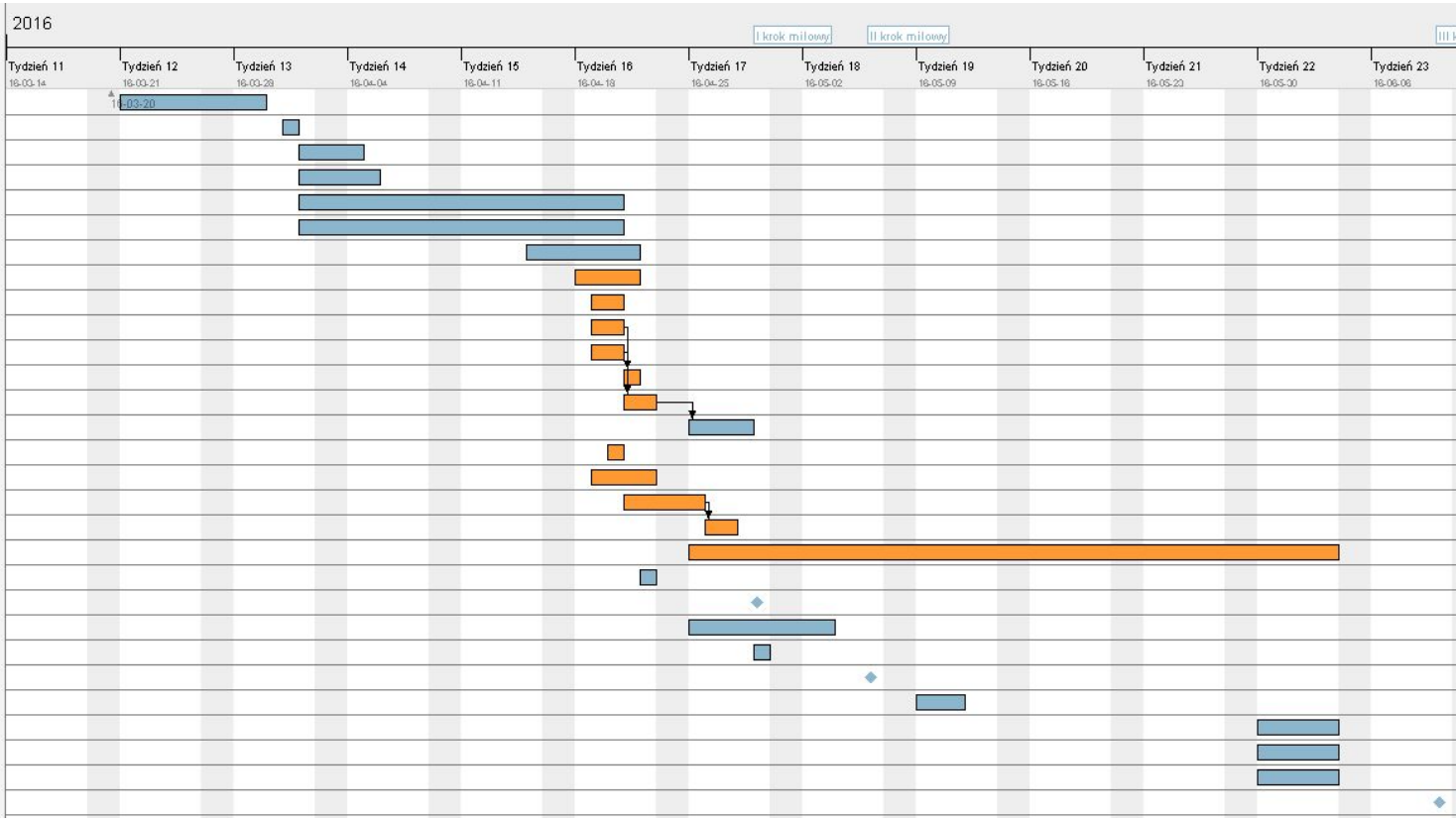
Poniżej przedstawiony został harmonogram projektu wraz z diagramem Gantta.

Nazwa zadania	**Data rozpoczęcia**	**Data zakończenia**
Organizacja struktury projektu	16-03-21	16-03-29
Pierwsze spotkanie projektowe	16-03-31	16-03-31
Wystartowanie repozytorium Git	16-04-01	16-04-04
Ustalenia funkcjonalne i niefunkcjonalne	16-04-01	16-04-05
Utworzenie koncepcji architektury systemu	16-04-01	16-04-20
Utworzenie koncepcji wykorzystania narzędzia docker	16-04-01	16-04-20
Utworzenie dokumentu dobrych praktyk	16-04-15	16-04-21
Utworzenie tutoriala docker	16-04-18	16-04-21
Dodanie rozwiązania docker do repozytorium	16-04-19	16-04-20
Doprecyzowanie wymagań przedmiotu	16-04-19	16-04-20
Wybranie algorytmów	16-04-19	16-04-20
Ustalenie prezentacji na drugi krok milowy	16-04-21	16-04-21
Modułowy podział projektu	16-04-21	16-04-22
Szczegłowa koncepcja rozwiązania	16-04-25	16-04-28
Ustawienie środowiska programistycznego	16-04-20	16-04-20
Utworzenie pierwszych scenariuszy testowych (do prezentacji pierwszego kroku milowego)	16-04-19	16-04-22
Ustalenie wykorzystywanej bazy danych	16-04-21	16-04-25
Postawienie bazy danych	16-04-26	16-04-27
Część programistyczna	16-04-25	16-06-03
Zebranie dokumentacji na pierwszy krok	16-04-22	16-04-22
I krok milowy	16-04-29	16-04-29

Utworzenie wszystkich scenariuszy testowych	16-04-25	16-05-03
Odłożenie bruncha na II krok milowy	16-04-29	16-04-29
II krok milowy	16-05-06	16-05-06
Ustalenie prezentacji na trzeci krok milowy	16-05-09	16-05-11
Testy bezpieczeństwa	16-05-30	16-06-03
Testy fukcjonalne	16-05-30	16-06-03
Testy awaryjności	16-05-30	16-06-03
III krok milowy	16-06-10	16-06-10

Tablica 1.1: Harmonogram projektu

Nazwa zadania	Data rozpoczęcia	Data zakończenia
Organizacja struktury projektu	16-03-21	16-03-29
Pierwsze spotkanie projektowe	16-03-31	16-03-31
Wystartowanie repozytorium Git	16-04-01	16-04-04
Ustalenia funkcjonalne i niefunkcjonalne	16-04-01	16-04-05
Utworzenie koncepcji architektury systemu	16-04-01	16-04-20
Utworzenie koncepcji wykorzystania narz...	16-04-01	16-04-20
Utworzenie dokumentu dobrych praktyk	16-04-15	16-04-21
Utworzenie tutoriala docker	16-04-18	16-04-21
Dodanie rozwiązania docker do repozytor...	16-04-19	16-04-20
Doprecyzowanie wymagań przedmiotu	16-04-19	16-04-20
Wybranie algorytmów	16-04-19	16-04-20
Ustalenie prezentacji na drugi krok milowy	16-04-21	16-04-21
Modułowy podział projektu	16-04-21	16-04-22
Szczegłowa koncepcja rozwiązania	16-04-25	16-04-28
Ustawienie środowiska programistycznego	16-04-20	16-04-20
Utworzenie pierwszych scenariuszy testo...	16-04-19	16-04-22
Ustalenie wykorzystywanej bazy danych	16-04-21	16-04-25
Postawienie bazy danych	16-04-26	16-04-27
Część programistyczna	16-04-25	16-06-03
Zebranie dokumentacji na pierwszy krok	16-04-22	16-04-22
I krok milowy	16-04-29	16-04-29
Utworzenie wszystkich scenariuszy testow...	16-04-25	16-05-03
Odłożenie bruncha na II krok milowy	16-04-29	16-04-29
II krok milowy	16-05-06	16-05-06
Ustalenie prezentacji na trzeci krok milowy	16-05-09	16-05-11
Testy bezpieczeństwa	16-05-30	16-06-03
Testy fukcjonalne	16-05-30	16-06-03
Testy awaryjności	16-05-30	16-06-03
III krok milowy	16-06-10	16-06-10



2. Wymagania

2.1. Wymagania funkcjonalne

Identyfikator wymagania	**Opis wymagania**
FR1	System umożliwia wyświetlenie listy dostępnych do pobrania wyników badań medycznych
FR2	System umożliwia pobranie zanonimizowaych wyników badań medycznych
FR3	System umożliwia pobranie statystyk wykonanych badań medycznych

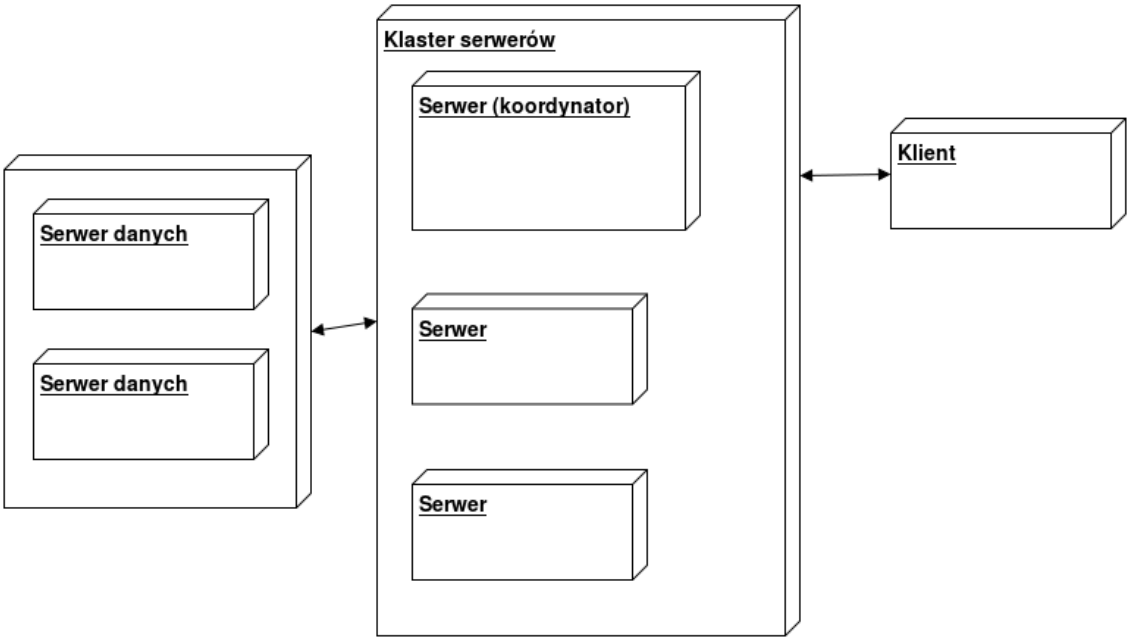
2.2. Wymagania niefunkcjonalne

Identyfikator wymagania	**Opis wymagania**
NFR1	Współbieżne oprogramowanie realizujące część serwerową
NFR2	Uszkodzenie węzła, nie powoduje zatrzymania pracy systemu
NFR3	Realizacja usługi w trakcie awarii w czasie obsługi
NFR4	Możliwość ponownego wpięcia węzła, z którym utracono łączność
NFR5	Odporność na próbę wpięcia wrogiego, nieuprawnionego węzła
NFR6	Zarządzanie zasobami transparentnie dla oprogramowania klienckiego
NFR7	Zapewnić poziom redundancji danych równy 2
NFR8	Maksymalna pojemność przechowywanych wszystkich danych równa 1GB
NFR9	Dane o pacjentach oraz badaniach przechowywane w relacyjnej bazie danych PostrgreSQL
NFR10	Wyniki badań medycznych przechowywane w plikach formatu .xml lub .bmp na serwerze danych
NFR11	Uruchamianie i zamykanie części serwerowej jednokrotnym wywołaniem skryptu dowolnym węźle
NFR12	Liczba jednocześnie obsłużonych użytkowników równa 100
NFR13	System uruchamiany w środowisku Linux Ubuntu 14.04 LTS

NFR14	Dwa serwery danych, każdy posiada procesor z minimum czterema wątkami sprzętowymi oraz dyskiem twardym o pojemności min. 20 GB
NFR15	Klaster trzech serwerów (nie wliczając serwerów danych), każdy posiada procesor z minimum dwoma wątkami sprzętowymi

3. Architektura i opis systemu

3.1. Opis ogólny architektury systemu

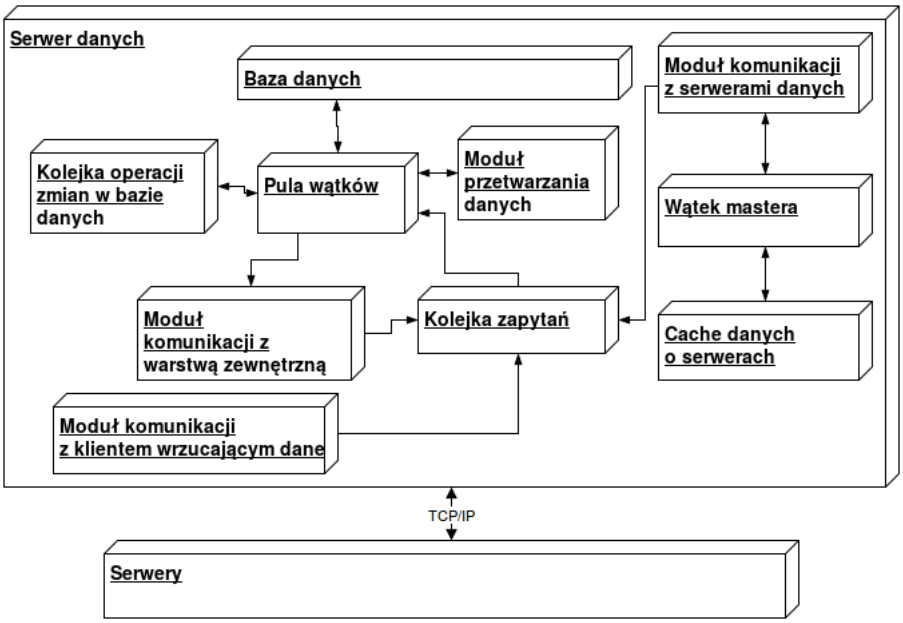


Rysunek 3.1. *Schemat architektury systemu*

System po stronie serwerowej składa się z dwóch warstw: wewnętrznej i zewnętrznej. Warstwa wewnętrzna przechowuje wrażliwe informacje medyczne oraz je przetwarza. Przetwarzanie polega na anonimizacji wyników badań medycznych oraz tworzeniu statystyk. Warstwa zewnętrzna zajmuje się dystrybucją danych dla klientów. Część kliencka jest prostą aplikacją dostępową korzystającą z API serwerów warstwy zewnętrznej. W projekcie założono, że do realizacji systemu wykorzystane zostaną dwa serwery danych i trzy serwery warstwy zewnętrznej. Ogólny schemat systemu został przedstawiony na rysunku 3.1.

3.2. Serwer danych

Serwer danych ma za zadnie przechowywać informacje w bazie danych, przetwarzać informacje, udostępniać przetworzone informacje, udostępniać API do bezpośredniego wstawienia informacji oraz komunikować się z innymi serwerami danych w celu realizacji redundancji danych, zachowania spójności danych i współbieżnego realizowania zapytań. Schemat serwera danych przedstawiono na rysunku 3.2.



Rysunek 3.2. *Schemat serwera bazy danych*

3.2.1. Redundancja danych

Aby zapewnić redundancję danych zastosowano prostą replikację całego serwera. W systemie poziom redundancji jest równy liczbie serwerów danych. W projekcie będzie on równy 2.

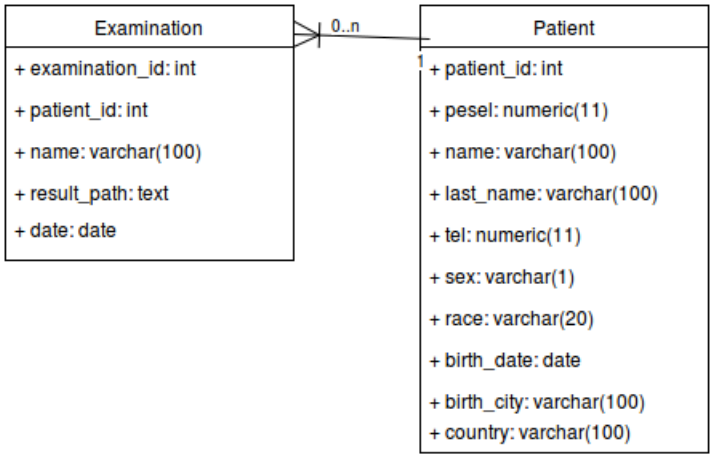
3.2.2. Protokół spójności

Zrealizowany zostanie model spójności sekwencyjnej. To oznacza, że dozwolony jest każdy przeplot operacji czytania i pisania ale wszystkie procesy, na wszystkich maszynach oglądają ten sam przeplot operacji. Zostanie on zapewniony m.in. przez rozsyłanie każdej zmiany w bazie do drugiego serwera. Zapis może odbywać się równolegle na dwóch serwerach. Może zdarzyć się sytuacja, w której na skutek modyfikacji tego samego rekordu w bazie jednocześnie na dwóch serwerach, dwie różne zmiany zostaną rozpropagowane w tym samym czasie i pojawi się niespójność. Aby temu zapobiec zastosowano algorytm Lamporta dostrajania lokalnych zegarów. Każdy komunikat otrzymuje znacznik czasu, zgodny z lokalnym zegarem. Jeżeli po nadejściu komunikatu zegar odbiorcy pokazuje wartość wcześniejszą, niż czas w komunikacie to odbiorca przesuwa swój zegar w przód, tak aby wskazywał wartość czasu o 1 większą niż czas nadania. Komunikaty są umieszczane w lokalnej kolejce zgodnie z jego znacznikiem czasu. Zakłada się również, że nadawca komunikatu jest również jego odbiorcą. Ta metoda zapewnia, że wszystkie procesy będą miały ta samą kopię kolejki zapasowej.

Gdy po awarii, jeden z serwerów danych połączy się ponownie, musi zaktualizować swoje dane. W tym celu można zapisywać wszystkie zmiany w bazie danych. Nowo doleczony serwer sprawdziłby jaką ostatnią operację wykonywał, a następnie wysłał do mastera prośbę o odesłanie wszystkich późniejszych operacji. Tego

rozwiązania nie będzie w opisywanym systemie ze względu krotki czas realizacji projektu.

3.2.3. Przechowywanie danych



Rysunek 3.3. *Schemat bazy danych*

Dane są przechowywane w bazie danych PostgreSQL, a wyniki badań medycznych w plikach, których ścieżki są zapisane w bazie danych. Schemat bazy danych przedstawiono na 3.3. Na potrzeby projektu przyjęto, że format przechowywanych wyników badań to .xml dla wyników w formie tekstowej lub .bmp dla wyników w formie zdjęć.

3.2.4. Przetwarzanie danych

Przetwarzanie danych obejmuje anonimizację wyników danych medycznych i tworzenie statystyk. Anonimizacja danych w formacie .xml polega na usunięciu nazwiska pacjenta z pliku. Danych w formacie .bmp nie trzeba anonimizować, ponieważ samo zdjęcie nie wskazuje na konkretnego pacjenta. Dostępne statystyki badań będą pokazywały liczbę wykonanych badań w zależności od czasu, kraju, płci, wieku itp. Przykładowy plik .xml z wynikiem badań:

```
<examination>
    <patient>
        <first_name>Jan</first_name>
        <last_name>Kowalski</last_name>
    </patient>
    <examination_type>morfologia</examination_type>
    <result>
    </result>
</examination>
```

3.2.5. Komunikacja

Komunikacja z serwerami warstwy zewnętrznej i serwerami danych odbywa się przez TCP/IP. W komunikacji pośredniczy moduł komunikacyjny, który oczekuje

na zapytania. W serwerze danych znajdują się trzy moduły komunikacyjne (dla serwerów warstwy zewnętrznej, dla serwerów danych i pomocniczy do wstawiania danych). Każdy nasłuchuje na oddzielnym porcie. Numery portów znajdują się w pliku konfiguracyjnym.

3.2.6. Przetwarzanie żądań

Każde zapytanie jest obsługiwane w oddzielnym wątku. Serwer korzysta z puli wątków. ich liczba jest określona w pliku konfiguracyjnym. Powinna być równa liczbie wątków sprzętowych, co zapewni efektywne wykorzystanie procesora. Zapytania przychodzące z warstwy zewnętrznej lub wewnętrznej są kolejkowane w kolejności przyjścia lub odrzucane, gdy zapytań będzie za dużo (kolejka FIFO). Maksymalny rozmiar kolejki jest określony w pliku konfiguracyjnym. Żądania dotyczące modyfikacji są dodatkowo kolejkowane zgodnie z ich znacznikami czasu. Wątki realizują na zmianę, raz zapytania z kolejki modyfikacji, raz z kolejki zapytań.

3.2.7. Szyfrowanie wiadomości

Komunikacja jest szyfrowana przy pomocy RSA. Każdy z serwerów dostanie wygenerowany klucz publiczny oraz prywatny. Klucze publiczne wszystkich serwerów oraz serwerów danych są zapisane w pliku konfiguracyjnym. Wysłanie wiadomości będzie wymagało zaszyfrowania jej kluczem publicznym odbiorcy i tylko on będzie mógł ją odszyfrować swoim kluczem prywatnym. Jest to konieczne aby zachować bezpieczeństwo przesyłania wrażliwych danych między serwerami danych. Zapewni również to, że niepowołany węzeł w warstwie wewnętrznej lub zewnętrznej, jeżeli dostanie informację, nie będzie w stanie jej odszyfrować.

3.2.8. Koordynator warstwy wewnętrznej

Jeden z serwerów danych będzie koordynatorem. Koordynator kontroluje pracę pozostałych serwerów, sprawdza, czy wszystkie są dostępne, tworzy i rozsyła do wszystkich węzłów tablicę aktywności serwerów danych. Sprawdzanie obecności serwerów odbywa się poprzez cykliczne odpytywanie. Okres odpytywania jest zawarty w pliku konfiguracyjnym. Stan warstwy wewnętrznej jest wysyłany tylko wtedy gdy się zmieni, np. jeden z serwerów ulegnie awarii. Informacja ta jest zapisywana w pamięci RAM w przeznaczonej do tego klasie.

3.2.9. Algorytm elekcji

Gdy master ulegnie awarii jego rolę przejmuje inny serwer. W projekcie przewidziano dwa serwery danych, więc serwer który zauważy brak mastera powinien od razu przejąć jego rolę. Jednakże, rozwiązanie powinno być skalowalne więc do elekcji zastosowany zostanie algorytm Tyrana. Został on szczegółowo opisany w następnym podrozdziale dotyczącym serwerów warstwy zewnętrznej.

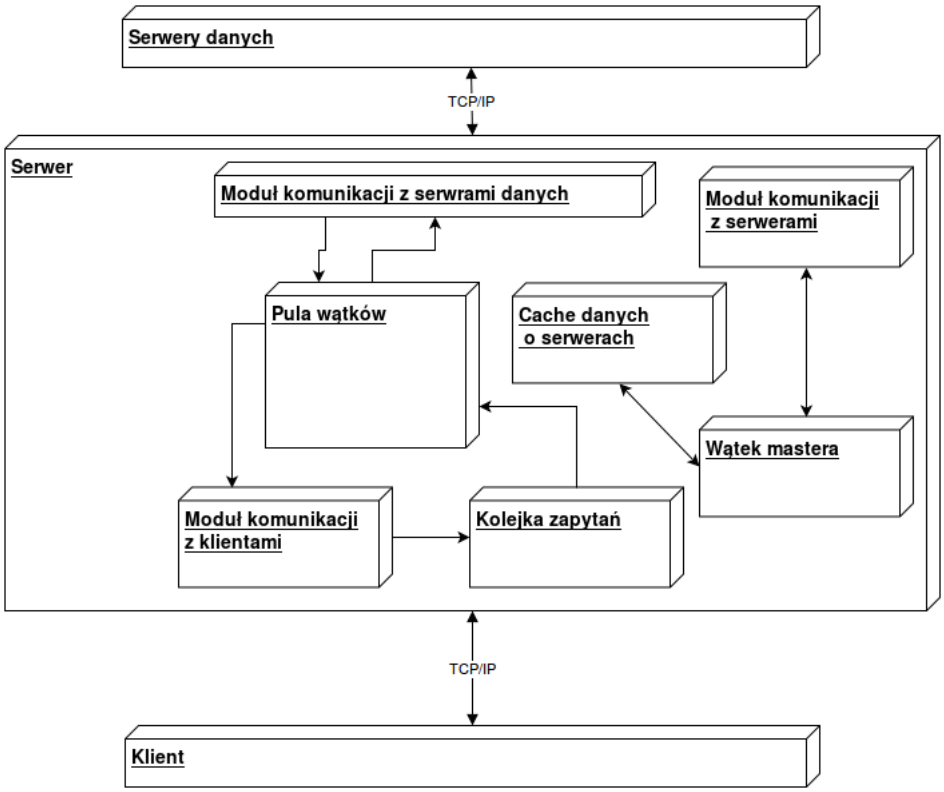
3.2.10. Bezpośrednie wstawianie danych

Dodatkowo na potrzeby projektu serwery danych udostępnią API do bezpośredniego zapisu danych.

3.2.11. Dziennik operacji

Prowadzony jest dziennik operacji wykonywanych na serwerze danych. Ma on formę pliku tekstowego, w którym zapisywane są przeprowadzane operacje na danych, informacje o błędach, utracie węzła, dołączeniu węzła oraz zmianie koordynatora.

3.3. Serwer warstwy zewnętrznej



Rysunek 3.4. *Schemat serwera aplikacyjnego*

Serwery warstwy zewnętrznej przekazują zapytania od klienta do serwera danych i przesyłają już przetworzone dane klientowi. Ich rolą jest również komunikowanie się z innymi serwerami warstwy zewnętrznej w celu współbieżnej realizacji żądań i realizacji odporności systemu na awarie węzłów. Schemat serwera warstwy zewnętrznej przedstawia rysunek 3.4.

3.3.1. Komunikacja

Komunikacja z klientem i serwerami danych odbywa się przez TCP/IP. W komunikacji pośredniczą moduły komunikacyjne, który oczekują na zapytania. W serwerze danych znajdują się trzy moduły komunikacyjne (dla serwerów warstwy zewnętrznej, dla serwerów danych i klienta). Każdy nasłuchuje na oddzielnym porcie. Numery portów znajdują się w pliku konfiguracyjnym.

Serwer może wysyłać zapytania do dowolnego serwera danych. Jeżeli któryś z nich będzie niedostępny próbuje połączyć się z kolejnym z listy zawartej w pliku konfiguracyjnym.

3.3.2. Szyfrowanie danych

Komunikacja między serwerami warstwy zewnętrznej oraz między serwerem warstwy zewnętrznej, a serwerem danych jest szyfrowana przy użyciu RSA. Dzięki temu niepowołane węzeł nie będzie w stanie rozszyfrować żadnych informacji. Komunikacja z klientem nie jest szyfrowana, nie jest to konieczne, gdyż przesyłane w tym obszarze informacje nie są już wrażliwe.

3.3.3. Przetwarzanie żądań

Serwer korzysta z puli wątków. ich liczba jest określona w pliku konfiguracyjnym i powinna być równa liczbie wątków sprzętowych. Po odebraniu żądania od klienta lub innego węzła, wstawiane jest ono do kolejki zapytań (kolejka FIFO). Długość kolejki jest odczytywana z pliku konfiguracyjnego.

3.3.4. Koordynator

Jeden z serwerów jest koordynatorem. Koordynator kontroluje pracę pozostałych serwerów, sprawdza, czy wszystkie są dostępne. Tworzy i rozsyła do wszystkich węzłów tablicę aktywności serwerów. Sprawdzanie obecności serwerów odbywa się poprzez cykliczne odpytywanie. Odpytywany jest też jeden z serwerów danych o aktualny stan warstwy wewnętrznej. Okres odpytywania jest zawarty w pliku konfiguracyjnym. Stan całego systemu jest wysyłany tylko wtedy gdy się zmieni, np. jeden z serwerów ulegnie awarii. Informacja ta jest zapisywana w pamięci RAM w przeznaczonej do tego klasie.

3.3.5. Algorytm elekcji

Gdy master ulegnie awarii jego rolę przejmuje inny z serwerów. W projekcie zastosowano algorytm Tyrana. Wszystkie serwery znają liczbę i adresy pozostałych. Każdy z serwerów ma przyporządkowany numer. Masterem staje się ten o najniższym numerze. Pierwszy serwer, który zauważy, że nie ma mastera, czeka losowy czas i wysyła komunikat **ELECTION** do wszystkich węzłów o niższym numerze, jeżeli nikt nie odpowie to serwer staje się nowym koordynatorem i wysyła do wszystkich pozostałych węzłów informujący komunikat **COORDINATOR**. Jeżeli któryś z węzłów o niższym numerze odpowie to on przejmuje kontrolę. Brak koordynatora będzie zauważony, gdy węzeł nie zostanie odpytany w odpowiednim czasie. Dodanie losowego czasu po zauważeniu braku mastera zapobiegnie sytuacji, gdy kilka serwerów jednocześnie wyśle wiadomość **ELECTION**.

3.3.6. Awaria węzła

Awaria węzła zostanie wykryta, podczas cyklicznego odpytywania przez koordynatora. Nieaktywny serwer nie odpowie, więc koordynator zaktualizuje stan systemu i roześle do pozostałych węzłów. Gdy serwer będzie znów dostępny, przy najbliższym odpytywaniu, zostanie zauważony przez koordynatora, który ponownie zaktualizuje stan systemu.

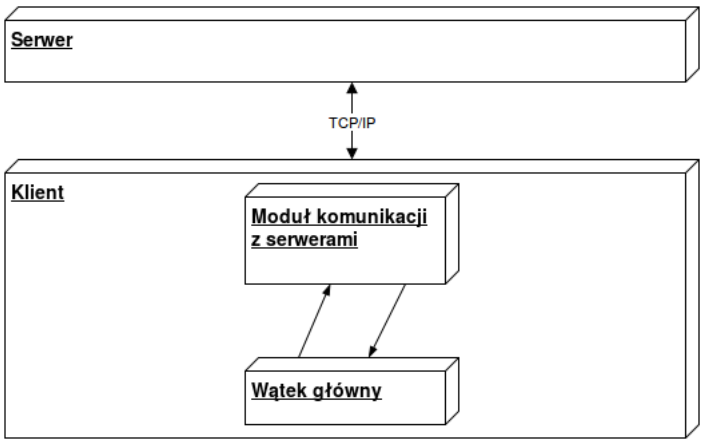
3.3.7. Przerwanie połączenia między węzłami

Może pojawić się sytuacja, gdy przerwane zostanie połączenie między dwoma poprawnie pracującymi serwerami. Wtedy każdy z nich przeprowadzi elekcję oddzielnie. Powstaną dwa klastry serwerów, ale wszystko będzie poprawnie pracowało. Gdy połączenie wróci, w warstwie zewnętrznej będzie dwóch koordynatorów. W celu rozwiązania tego problemu, pierwszy koordynator, który zostanie odpytany przez drugiego wyłącza u siebie funkcję koordynatora.

3.3.8. Dziennik operacji

Prowadzony jest dziennik operacji wykonywanych na serwerze danych. Ma on formę pliku tekstowego, w którym zapisywane są informacje o błędach, utracie węzła, dołączeniu węzła i zmianie koordynatora.

3.4. Aplikacja kliencka



Rysunek 3.5. *Schemat aplikacji klienckiej*

Aplikacja kliencka wysyła polecenia od użytkownika do serwerów warstwy zewnętrznej i wyświetla ich wynik lub zapisuje pobrane dane. Schemat aplikacji klienckiej przedstawiono na rysunku 3.5.

3.4.1. Komunikacja

Komunikacja między klientem, a serwerem warstwy zewnętrznej odbywa się przez TCP/IP. Początkowo w pliku konfiguracyjnym klienta jest adres jednego serwera. Aplikacja próbuje nawiązać z nim połączenie. Jeżeli to połączenie zakończy się niepowodzeniem aplikacja kończy działanie. W przeciwnym wypadku klient pobiera informację o dostępnych serwerach. Uzupełnia tą informacją plik konfiguracyjny. Dzięki temu przy następnym uruchomieniu, gdy któryś z serwerów będzie niedostępny, nastąpi próba połączenia z innym.

3.4.2. Szyfrowanie

Komunikacja od serwera do klienta nie jest szyfrowana. Nie jest to konieczne, ponieważ przekazywane dane nie wymagają już ochrony.

3.4.3. Funkcjonalność

Aplikacja umożliwia dostęp do funkcji oferowanych przez API serwera, co oznacza wyświetlenie wyniku zapytania lub pobranie pliku z wynikiem danych medycznych. Dostępne zapytania są opisane w rozdziale dotyczącym protokołu komunikacyjnego. Jeżeli jakieś żądanie się nie powiedzie mimo dostępności serwera (z powodu przeciążenia), następuje ponowna próba wysłania żądania ale do innego serwera. Aplikacja kliencka jest aplikacją konsolową, nieinteraktywną, co oznacza, że wykonywane jest jedno zapytanie, zwracany jest wynik i następuje zakończenie działania.

3.5. Plik konfiguracyjny

W pliku konfiguracyjnym klienta na początku znajduje się adres i port jednego z serwerów. W pliku konfiguracyjnym serwerów znajdują się adresy, porty i nazwy serwerów, adresy serwerów baz danych, liczba wątków w puli, liczba zapytań w kolejce, okres odpytywania węzłów przez serwer główny, klucze publiczne serwerów. Plik konfiguracyjny zostanie zapisany w formacie .ini. Każdy serwer ma unikalny numer i typ. Typ może mieć wartość 'srv' dla serwerów warstwy zewnętrznej lub 'db' dla serwerów warstwy wewnętrznej.

Przykładowy plik konfiguracyjny dla klienta:

```
[settings]
servers_number = 1

[serwer1]
ip=192.168.1.130
port=4000
key=AAAAB3NzaC1yc2EAAAABiwAAAQEAklOUpkDHrfHY17SbrmTipNLTGK9Tjom/BWDSUGPl+nafzl
HDTYW7hdi4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3Pbv7kOdJ/MtyBl
WXFCR+Hao3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XAt3FaoJoAsncM1Q9x5+3
V0Ww68/eiFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/EnmZ+AW4OZPnTPi89ZPmVMLuay
rD2cE86Z/il8b+gw3r3+1nKatmikjn2so1d01QraTlMqVSsbxNrRFi9wrf+M7Q==
```

Wydruk 3.1. Plik konfiguracyjny klienta

Przykładowy plik konfiguracyjny dla serwera lub serwera danych:

```
[settings]
threads_num = 4 ; liczba watkow puli
queue_size = 20 ; rozmiar kolejki
servers_number = 3 ; liczba serwerow warstwy zewnetrznej
servers_DB_number = 2 ; liczba serwerow warstwy wewnetrznej


[master_settings]
interval = 60 ; okres odpytywania wezlow w sekundach

[serwer1]
type=srv ; typ serwera
```

```
ip=192.168.1.130 ; adres ip serwera
portExt=2000 ; port do komunikacji z serwerami warstwy zewnetrznej
portDB=3000 ; port do komunikacji z serwerami danych
portClient=4000 ; port do komunikacji z klientami
key=AAAAB3NzaC1yc2EAAAABiwAAAQEAklOUpkDHrfHY17SbrmTipNLTGK9Tjom/BWDSUGPl+nafzl
HDTYW7hdi4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3Pbv7kOdJ/MtyBl
WXFCR+Hao3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XAt3FaoJoAsncM1Q9x5+3
V0Ww68/eiFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/EnmZ+AW4OZPnTPi89ZPmVMLuay
rD2cE86Z/i18b+gw3r3+1nKatmikjn2so1d01QraTlMqVSsbxNrRFi9wrf+M7Q==
; klucz publiczny

[serwer2]
ip=192.168.1.131
type=srv
portExt=2000
portDB=3000
portClient=4000
key=AAAAB3NzaC1yc2EAAAABiwAAAQEAklOUpkDHrfHY17SbrmTipNLTGK9Tjom/BWDSUGPl+nafzl
HDTYW7hdi4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3Pbv7kOdJ/MtyBl
WXFCR+Hao3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XAt3FaoJoAsncM1Q9x5+3
V0Ww68/eiFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/EnmZ+AW4OZPnTPi89ZPmVMLuay
rD2cE86Z/i18b+gw3r3+1nKatmikjn2so1d01QraTlMqVSsbxNrRFi9wrf+M7Q==

[serwer3]
ip=192.168.1.132
type=srv
portExt=2000
portDB=3000
portClient=4000
key=AAAAB3NzaC1yc2EAAAABiwAAAQEAklOUpkDHrfHY17SbrmTipNLTGK9Tjom/BWDSUGPl+nafzl
HDTYW7hdi4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3Pbv7kOdJ/MtyBl
WXFCR+Hao3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XAt3FaoJoAsncM1Q9x5+3
V0Ww68/eiFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/EnmZ+AW4OZPnTPi89ZPmVMLuay
rD2cE86Z/i18b+gw3r3+1nKatmikjn2so1d01QraTlMqVSsbxNrRFi9wrf+M7Q==

[serwer4]
ip=192.168.1.132
type=db
portDB=2000
portExt=3000
portClient = 4000
key=AAAAB3NzaC1yc2EAAAABiwAAAQEAklOUpkDHrfHY17SbrmTipNLTGK9Tjom/BWDSUGPl+nafzl
HDTYW7hdi4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3Pbv7kOdJ/MtyBl
WXFCR+Hao3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XAt3FaoJoAsncM1Q9x5+3
V0Ww68/eiFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/EnmZ+AW4OZPnTPi89ZPmVMLuay
rD2cE86Z/i18b+gw3r3+1nKatmikjn2so1d01QraTlMqVSsbxNrRFi9wrf+M7Q==

[serwer5]
ip=192.168.1.133
type=db
portDB=2000
portExt=3000
portClient = 4000
key=AAAAB3NzaC1yc2EAAAABiwAAAQEAklOUpkDHrfHY17SbrmTipNLTGK9Tjom/BWDSUGPl+nafzl
HDTYW7hdi4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3Pbv7kOdJ/MtyBl
WXFCR+Hao3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XAt3FaoJoAsncM1Q9x5+3
V0Ww68/eiFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/EnmZ+AW4OZPnTPi89ZPmVMLuay
rD2cE86Z/i18b+gw3r3+1nKatmikjn2so1d01QraTlMqVSsbxNrRFi9wrf+M7Q==
```

Wydruk 3.2. Plik konfiguracyjny serwera

3.6. Protokół komunikacyjny

Poniżej przedstawiony został protokół komunikacyjny zaprojektowany i użyty w ramach projektu.

3.6.1. Format ramek

Każda ramka zaczyna się znakiem '(' a kończy znakiem ')'. Pola w ramce oddzielone są znakiem '.' (przecinek). Niektóre ramki zawierają zagnieżdżone zestawy pól, zestawy takie są ograniczone przez znaki " oraz ", a wartości oddzielone przez ','. Nazwy pól dla czytelności dokumentu zapisywane będą w nawiasach ostrych, np. **<kraj>** (w prawdziwej ramce zapisywane są bez nawiasów). Zapis **<...>** oznacza że odpowiednie pola mogą się powtarzać.

3.6.2. Komunikacja klient-serwer warstwy zewnętrznej

Pobranie listy dostępnych wyników

(GET_AVAILABLE_RESULTS,*)	→	
	←	(RESULTS, <liczba wyników>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <...>)

Przykładowe zapytanie:

```
1 (GET_AVAILABLE_RESULTS,*)
```

Przykładowa odpowiedź:

```
1 (RESULTS,2,1001,Badanie prostaty,Polska,M,biala,61,
1458,Pomiar cisnienia oka,Niemcy,K,biala,16)
```

Pobranie listy dostępnych wyników spełniających kryteria

(GET_AVAILABLE_RESULTS, <nazwa badania>, <kraj>, <płeć>, <rasa>, <minimalny wiek>, <maksymalny wiek>)	→	

	←	(RESULTS, <liczba wyników>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <...>)

Płeć należy podać jako K lub M. Aby nie precyzować któregoś z pól należy podać %. Można również wstawić % jako część wzorca, np. Po% będzie dopasowane zarówno do Polska jak i Portugalia.

Przykładowe zapytanie:

```
(GET_AVAILABLE_RESULTS,*,Polska,M,*,10,11)
```

Przykładowa odpowiedź:

```
(RESULTS,2,1234,Badanie ogolne,Polska,M,biala,10,
3214,Pobieranie krwi,Polska,M,biala,11)
```

Pobranie pliku przypisanego do badania

(GET_RESULT, <id badania>)	→	
	←	(RESULT, <nazwa pliku>, <rozmiar>, <dane>)

Rozmiar podany jest w bajtach po odkodowaniu. Dane przesłane są w postaci zakodowanej w Base64.

Przykładowe zapytanie:

```
(GET_RESULT,1234)
```

Przykładowa odpowiedź:

```
(RESULT,badanie.xml,357,
 MS4gQ2VydHlmaWthdCBCMg0KVHJ6ZWJhIHphcMWCYWNpxIcgbmEga29udG8gMTAgemV0YSw
gYSBwb3RlbSBuYSBtYWlsYSBhLmNoYWJyb3dza2FAc2pvLnB3LmVkdS5wbCB3eXPFgmHEhyB
wb3R3aWVyZHplbmllIHByemVsZXd1IChhbGJvIG5pZSB3eXN5xYJhxIcgYWxlIG1pZcSHIHB
yenkgc29iaWUgcHJ6eSBvZGJpZXJhbml1KS4gUGFwaWVyIGRhasSFIHcgIER6aWFsZSBkcy4
gU3R1ZGVudMOzdyBTSk8gKHBvay4gNDE5LCBHLkcuKSB3IGdvZHouIDk6MDAtMTM6MDAu)
```

Pobranie statystyk

(GET_STATISTICS, <nazwa badania>, <data od>, <data do>, <kraj>, <płeć>, <rasa>, <minimalny wiek>, <maksymalny wiek>, <pole grupowania 1>, <pole grupowania 2>, <...>)	→	

	←	(STATISTICS, <liczba wyników>, <zestaw grupowania>, <liczba badań>, <zestaw grupowania>, <liczba badań>, <...>)

Pola <nazwa badania>, <data od>, <data do>, <kraj>, <płeć>, <rasa> są opcjonalne, można je pominąć podając *. Dla nazwy badania i kraju działa znak % zastępujący dowolny ciąg znaków (np. "Rentgen %" będzie dopasowany zarówno do "Rentgen ręki" jak i do "Rentgen kolana". Pola grupowania są opcjonalne, ich podanie spowoduje pogrupowanie liczby badań. W przeciwnym przypadku zapytanie zwróci tylko jeden wynik.

Na przykład dla ramki:

```
1 (GET_STATISTICS,Rentgen %,01-01-2010,*,*,M,*,*,*,{kraj})
```

generowane jest zapytanie zbliżone do:

```
1 select count(zestaw) from badania where nazwa_badania like 'Rentgen %'
  and data >= 01-01-2010 and plec = 'M' group by (kraj) as zestaw
```

a odpowiedź to np.

```
1 (STATISTICS,2,Polska,20,Czechy,74)
```

Pobranie listy aktywnych serwerów warstwy zewnętrznej

(ACTIVE_SERVERS)	→	
	←	(ACTIVE_SERVERS, <liczba serwerów>, <adres ip>, <port>, <adres ip>, <port>, <...>)

Przykładowe zapytanie:

```
1 (ACTIVE_SERVERS)
```

Przykładowa odpowiedź:

```
1 (ACTIVE_SERVERS,3,192.168.1.40,2000,10.0.5.72,3400,192.168.15.2,3400)
```

3.6.3. Komunikacja między serwerami warstwy zewnętrznej

Ramki tej warstwy są szyfrowane kluczem publicznym odbiorcy i kodowane w base64. Ich format to (zaszyfrowana treść). Szyfrowane są pełne ramki, aby po odszyfrowaniu nadal były ramkami poprawnymi, np. (ACTIVE_SERVERS) -> (KGRzZHNkc2RzZHNkZHNkc2RzZHNkc2RzZHNkc2RzKQ==). Poniżej opisywane będą ramki przed zaszyfrowaniem. **<nr nadawcy>** oznacza numer serwera, który nadaje konkretną ramkę.

Sprawdzenie czy serwer odpowiada

(<nr nadawcy>, STATUS)	→	
	←	(<nr nadawcy>, STATUS_OK)

Przykładowe zapytanie:

```
1 (2 ,STATUS)
```

Przykładowa odpowiedź:

```
1 (5 ,STATUS_OK)
```

Rozesłanie tablicy aktywności serwerów warstwy zewnętrznej

(<nr nadawcy>, ACTIVE_SERVERS_EXT, <nr serwera>, <nr serwera>, <...>)	→	

Ramka jest przesyłana przez koordynatora do każdego aktywnego serwera warstwy zewnętrznej w momencie gdy nastąpi jakakolwiek zmiana w tablicy aktywności serwerów.

Rozesłanie tablicy aktywności serwerów danych

(<nr nadawcy>, ACTIVE_SERVERS_DB, <nr serwera>, <nr serwera>, <...>)	→	

Ramka jest przesyłana przez koordynatora do każdego aktywnego serwera warstwy zewnętrznej w momencie gdy nastąpi jakakolwiek zmiana w tablicy aktywności serwerów. Tablica aktywności serwerów danych aktualizowana jest na żądanie.

Informacja o braku koordynatora / elekcja

(<nr nadawcy>, ELECTION)	→	

Ramka jest przesyłana przez każdy serwer który zauważy brak koordynatora do serwerów o numerze niższym niż własny.

Zatrzymanie elekcji dla konkretnego węzła

(<nr nadawcy>, ELECTION_STOP)	→	

Ramka jest przesyłana przez możliwych koordynatorów do węzłów o wyższym numerze.

Zakończenie elekcji (wybór nowego koordynatora)

(<nr nadawcy>, COORDINATOR)	→	

Ramka jest przesyłana przez nowego koordynatora do każdego serwera.

3.6.4. Komunikacja między serwerami danych

Sprawdzenie czy serwer odpowiada

(<nr nadawcy>, STATUS)	→	
	←	(<nr nadawcy>, STATUS_OK)

Przykładowe zapytanie:

```
1 (2,STATUS)
```

Przykładowa odpowiedź:

```
1 (5,STATUS_OK)
```

Rozesłanie tablicy aktywności serwerów danych

(<nr nadawcy>, ACTIVE_SERVERS_DB, <nr serwera>, <nr serwera>, <...>)	→	

Ramka jest przesyłana przez koordynatora do każdego aktywnego serwera danych w momencie gdy nastąpi jakakolwiek zmiana w tablicy aktywności serwerów.

Rozesłanie tablicy aktywności serwerów warstwy zewnętrznej

(<nr nadawcy>, ACTIVE_SERVERS_EXT, <nr serwera>, <nr serwera>, <...>)	→	

Ramka jest przesyłana przez koordynatora do każdego aktywnego serwera danych w momencie gdy nastąpi jakakolwiek zmiana w tablicy aktywności serwerów warstwy zewnętrznej. Tablica aktywności serwerów warstwy zewnętrznej aktualizowana jest na żądanie.

Informacja o braku koordynatora / elekcja

(<nr nadawcy>, ELECTION)	→	

Ramka jest przesyłana przez każdy serwer który zauważy brak koordynatora do serwerów o numerze niższym niż własny.

Zatrzymanie elekcji dla konkretnego węzła

(<nr nadawcy>, ELECTION_STOP)	→	

Ramka jest przesyłana przez możliwych koordynatorów do węzłów o wyższym numerze.

Zakończenie elekcji (wybór nowego koordynatora)

(<nr nadawcy>, COORDINATOR)	→	

Ramka jest przesyłana przez nowego koordynatora do każdego serwera.

Wykonanie masowego wrzutu danych do bazy

(<nr nadawcy>, UPLOAD, <czas>, <dane>)	→	
	←	(<nr nadawcy>, UPLOAD, OK)

<dane> to archiwum zip zakodowane w base64. Archiwum powinno zawierać plik upload.sql, który jest wykonywany na bazie danych oraz folder data, w którym zawarte są inne pliki, które mają być dostępne jako wyniki badań.

Wykonanie pojedynczego wstawienia do bazy danych

(<nr nadawcy>, INSERT, <czas>, <tabela>, <kolumna 1>, <kolumna 2>, <...>)	→	
	←	(<nr nadawcy>, INSERT, OK)

Wykonanie pojedynczego wstawienia pliku na serwer danych

(<nr nadawcy>, ATTACH, <czas>, <nazwa pliku>, <dane>)	→	
	←	(<nr nadawcy>, ATTACH, OK)

<dane> to plik zakodowany w base64.

Wykonanie pojedynczego usunięcia z bazy danych

(<nr nadawcy>, DELETE, <czas>, <tabela>, <id wiersza>)	→	
	←	(<nr nadawcy>, DELETE,OK)

Wykonanie pojedynczego usunięcia pliku z serwera danych

(<nr nadawcy>, UNLINK, <czas>, <nazwa pliku>)	→	

	←	(<nr nadawcy>, UNLINK,OK)

3.6.5. Komunikacja między serwerami warstwy zewnętrznej a serwerami danych

Wykonanie pojedynczego usunięcia pliku z serwera danych

(<nr nadawcy>, GET_ACTIVE_SERVERS_DB)	→	
	←	(<nr nadawcy>, ACTIVE_SERVERS_DB, <nr serwera>, <nr serwera>, <...>)

Koordynator warstwy zewnętrznej co jakiś czas odpytuje dowolny serwer danych

Pobranie listy dostępnych wyników

(<nr nadawcy>, GET_AVAILABLE_RESULTS, *)	→	
	←	(<nr nadawcy>, RESULTS, <liczba wyników>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <...>)

Pobranie listy dostępnych wyników spełniających kryteria

(<nr nadawcy>, GET_AVAILABLE_RESULTS, <nazwa badania>, <kraj>, <płeć>, <rasa>, <minimalny wiek>, <maksymalny wiek>)	→	
	←	(<nr nadawcy>, RESULTS, <liczba wyników>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <id badania>, <nazwa badania>, <kraj>, <płeć>, <rasa>, <wiek>, <...>)

Pobranie pliku przypisanego do badania

(<nr nadawcy>, GET_RESULT, <id badania>)	→	

	←	(<nr nadawcy>, RESULT, <nazwa pliku>, <rozmiar>, <dane>)

Pobranie statystyk

(<nr nadawcy>, GET_STATISTICS, <nazwa badania>, <data od>, <data do>, <kraj>, <płeć>, <rasa>, <minimalny wiek>, <maksymalny wiek>, <pole grupowania 1>, <pole grupowania 2>, <...>)	→	
	←	(<nr nadawcy>, STATISTICS, <liczba wyników>, <zestaw grupowania>, <liczba badań>, <zestaw grupowania>, <liczba badań>, <...>)

3.6.6. Komunikacja między serwerami danych a klientem edytującym dane

Wykonanie masowego wrzutu danych do bazy

(UPLOAD, <dane>)	→	
	←	(UPLOAD, OK)

<dane> to archiwum zip zakodowane w base64. Archiwum powinno zawierać plik upload.sql, który jest wykonywany na bazie danych oraz folder data, w którym zawarte są inne pliki, które mają być dostępne jako wyniki badań.

Wykonanie pojedynczego wstawienia do bazy danych

(INSERT, <tabela>, <kolumna 1>, <kolumna 2>, <...>)	→	
	←	(INSERT, OK)

Wykonanie pojedynczego wstawienia pliku na serwer danych

(ATTACH, <nazwa pliku>, <dane>)	→	
	←	(ATTACH, OK)

<dane> to plik zakodowany w base64.

Wykonanie pojedynczego usunięcia z bazy danych

(DELETE, <tabela>, <id wiersza>)	→	
	←	(DELETE, OK)

Wykonanie pojedynczego usunięcia pliku z serwera danych

(UNLINK, <nazwa pliku>)	→	
	←	(UNLINK, OK)

3.6.7. Obsługa błędów

Każde zapytanie może się też nie udać, należy wtedy zwrócić błąd:
— (<nr nadawcy>, ERROR, <kod błędu>) w przypadku komunikacji serwer-serwer
— (ERROR, <kod błędu>) w przypadku komunikacji klient-serwer
Lista błędów:
— HOST_OVERLOADED – system obsługuje maksymalną liczbę użytkowników, nie jest możliwe zrealizowanie żądania
— ENCRYPTION_ERROR – węzeł nie może odszyfrować otrzymanej wiadomości
— DB_LOST_CONNECTION – utracone połączenie z bazą danych
— INVALID_REQUEST – błędna ramka

3.7. Klucze prywatne i publiczne

Klucz prywatny zostanie zapisany w pliku *~/.ssh/id_rsa*. Klucz publiczny zostanie zapisany w pliku *~/.ssh/id_rsa.pub*. Wygenerowaniem kluczy publicznych i prywatnych oraz zapisaniem ich do pliku konfiguracyjnego zajmuje się węzeł, z którego jest uruchamiana aplikacja. Do generowania kluczy wykorzystywany jest biblioteka OpenSSL.

Przykładowy klucz prywatny:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQCp2w+8HUdECo8V5yuKYrWJmUbLtD6nSyVifN543axXvNSFzQfW
NOGVkMsCo6W4hpl5eHv1p9Hqdcf/ZYQDWCK726u6hsZA81AblAOOXKaUaxvFC+ZK
RJf+MtUGnv0v7CrGoblm1mMC/OQI1JfSsYi68EpnaOLepTZw+GLTnusQgwIDAQAB
AoGBAKDuq3PikblH/9YS11AgwjwC++7ZcltzeZJdGTSPY1El2n6Dip9ML0hUjeSM
ROIWtac/nsNcJCnvOnUjK/c3NIAaGJcfRPiH/S0Ga6ROiDfFj2UXAmk/v4wRRUzr
5lsA0jgEt5qcq2Xr/JPQVGB4wUgL/yQK0dDhW0EdrJ707e3BAkEA1aIHbmcVfCP8
Y/uWuK0lvWxrIWfR5MlHhI8tD9lvkot2kyXiV+jB6/gktwk1QaFsy7dCXn7w03+k
xrjEGGN+kQJBAMuKf55lDtU9K2Js3YSStTZAXP+Hz7XpoLxmbWFyGvBx806WjgAD
624irwS+0tBxkERbRcisfb2cXmAx8earT9MCQDZuVCpjBWxd1t66qYpgQ29iAmG+
jBIY3qn9uOOC6RSTiCCx1FvFqDMxRFmGdRVFxeyZwsVE3qNksF0Zko0MPKECQCEe
oDV97DP2iCCz5je0R5hUUM2jo8DOC0GcyR+aGZgWcqjPBrwp5x08t43mHxeb4wW8
dFZ6+trnntO4TMxkA9ECQB+yCPgO1zisJWYuD46KISoesYhwHe5C1BQElQgi9bio
U39fFo88w1pok23a2CZBEXguSvCvexeB68OggdDXvy0=
-----END RSA PRIVATE KEY-----
```

Przykładowy klucz publiczny

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCp2w+8HUdECo8V5yuKYrWJmUbL
tD6nSyVifN543axXvNSFzQfWNOGVkMsCo6W4hpl5eHv1p9Hqdcf/ZYQDWCK726u6
hsZA81AblAOOXKaUaxvFC+ZKRJf+MtUGnv0v7CrGoblm1mMC/OQI1JfSsYi68Epn
aOLepTZw+GLTnusQgwIDAQAB
-----END PUBLIC KEY-----
```

3.8. Skrypt uruchamiający aplikację

Jest uruchamiany na dowolnym węźle, który generuje klucze publiczne i prywatne dla wszystkich serwerów oraz serwerów danych. Musi uzupełnić nimi plik konfiguracyjny. Następnie wysyła plik konfiguracyjny przez ssh pozostałym węzłom. Ostatni krok to uruchomienie aplikacji podając w argumentach numer węzła zgodny z danymi znajdującymi się w pliku.

4. Docker

4.1. Opis rozwiązania Docker

Docker jest narzędziem ułatwiającym proces tworzenia, dystrybucji i wdrażania oprogramowania. Pozwala on na umieszczenie aplikacji wraz z jej bezpośrednimi zależnościami w kontenerze i uruchomienie jej na dowolnej maszynie z systemem Linux. Aplikacje działające za pomocą Dockera są odizolowane od infrastruktury, dzięki czemu możliwe jest uruchomienie kilku niezależnych kontenerów z aplikacjami symulujących pracę w środowisku rozproszonym. Jednocześnie narzędzie to zapewnia niewielkie zużycie pamięci dzięki współdzieleniu warstw UFS obrazów (ang. Union File System) pomiędzy kontenerami. Współdzielenie jądra systemu pomiędzy kontenerami, a systemem gospodarza zapewnia natomiast krótkie czasy uruchomienia. Ze względu na te cechy korzystanie z aplikacji uruchomionej za pomocą Dockera jest niemal tak wydajne, jak działającej na systemie gospodarza.

Podstawowymi komponentami Dockera są:

— *Docker Engine* - platforma do tworzenia kontenerów na uruchamiane przez użytkownika aplikacje,

— *Docker Hub* – oficjalne repozytorium obrazów przechowujące obrazy udostępniane przez użytkowników

Docker działa w architekturze klient – serwer. Do komunikacji klienta z demonem wykorzystywany jest protokół HTTP i odbywa się ona poprzez gniazda (ang. sockets) lub RESTful API. Klient Dockera jest podstawowym interfejsem komunikacyjnym z Dockerem – przyjmuje komendy z określonego zestawu poleceń wpisywane przez użytkownika i przekazuje je do demona. Demon odpowiada za budowanie obrazów, uruchamianie i dystrybucję kontenerów. Klient i demon mogą działać na tym samym systemie lub ich funkcje mogą zosać rozdzielone pomiędzy różne hosty.

Podstawowymi pojęciami Dockera są:

— *obrazy* – komponent odpowiadający budowaniu,

— *rejestry* – komponent odpowiadający dystrybucji,

— *kontenery* – komponent odpowiadający uruchamianiu.

Obraz jest to szablon służący tylko do odczytu, stanowiący podstawę do utworzenia kontenera. Obraz może zawierać np. system operacyjny Ubuntu z serwerem Apache oraz zainstalowaną aplikacją webową, którą chcemy uruchomić. Za pomocą Dockera mamy możliwość budowania nowych obrazów, aktualizowania istniejących, pobierania obrazów stworzonych przez innych i udostępniania własnych. Każdy obraz składa się z warstw tworzących ujednolicony system plików (*ang. UFS*). Ta technologia powoduje, że obrazy Dockera są lekkie – wprowadzenie zmian w aplikacji nie wiąże się z przebudową całego obrazu, a jedynie z aktualizacją lub dodaniem danej warstwy.

Każdy obraz ma obraz bazowy (np. obraz systemu Ubuntu lub obraz utworzony przez użytkownika), na podstawie którego jest budowany za pomocą zestawu instrukcji. Każda instrukcja powoduje dodanie warstwy do naszego obrazu (taką instrukcją może być np. wywołanie komendy, dodanie pliku lub folderu, instalacja pakietu, utworzenie zmiennej środowiskowej). Polecenia tworzące obraz Dockera przechowywane są w pliku Dockerfile. Podczas budowania obrazu Docker odczytuje kod źródłowy zawarty w pliku Dockerfile, wykonuje zapisane instrukcje i zwraca końcowy obraz.

Przykładowe operacje na obrazie:

— Pobieranie obrazu:

```
docker pull {nazwa obrazu}
```

— Wyświetlanie lokalnie dostępnych obrazów:

```
docker images
```

— Wyświetlenie warstw składających się na obraz:

```
docker history {id lub nazwa obrazu}
```

— Usunięcie obrazu

```
docker  rmi {id lub nazwa obrazu}
```

Rejestry są miejscem gdzie można udostępniać i skąd można pobierać obrazy. Mogą one być zarówno publiczne, jak i prywatne. Publiczny rejestr Dockera stanowi Docker Hub zawierający bazę obrazów stworzonych przez użytkowników Dockera. Istnieje również lokalne repozytorium na maszynie użytkownika. Za pomocą klienta Dockera możliwe jest przeszukiwanie opublikowanych obrazów oraz pobieranie ich w celu utworzenia kontenera.

Kontener tworzony jest na podstawie obrazu, który zawiera informacje o tym, co przechowuje kontener, jaki proces ma zostać uruchomiony po jego utworzeniu oraz inne dane konfiguracyjne. Kontener składa się z zestawu ujednoliconych warstw tylko do odczytu, pochodzących z obrazu kontenera oraz z pojedynczej warstwy do odczytu i zapisu umożliwiającej działanie procesów uruchamianych w kontenerze. Na kontenerach można wykonywać podstawowe operacje: uruchomić, zatrzymać, przenieść i usunąć.

Przykładowe operacje na kontenerze

— Tworzenie i uruchamianie kontenera:

```
docker runl {nazwa lub id obrazu}
```

Uruchomienie tego polecenia z parametrem -d powoduje uruchomienie kontenera działającego w tle, natomiast parametr –rm=true powoduje, że kontener zostanie usunięty natychmiast po wykonaniu zadania.

— Wyświetlanie wszystkich kontenerów:

```
docker ps –a
```

— Zatrzymanie kontenera

```
docker stop {id lub nazwa kontenera}
```

— Usunięcie kontenera

```
1 docker rm {id lub nazwa kontenera}
```

Docker udostępnia również możliwość automatycznego budowania obrazu w synchronizacji z systemem kontroli wersji (GitHub lub Bitbucket). Dzięki umieszczeniu w repozytorium pliku *Dockerfile* oraz powiązaniu konta DockerHub I kontem w wybranym systemie kontroli wersji, po każdorazowej aktualizacji kodu w repozytorium budowany, na jego podstawie budowany jest aktualny obraz Dockera i zapisywany w repozytorium DockerHub.

4.2. Raport z przykładowych uruchomień

4.2.1. Uruchomienie aplikacji *'Hello World'*

Aplikacja uruchamiana jest poleceniem

```
1 $ docker run ubuntu /bin/echo 'Hello world'
```

gdzie:
— *docker run* – uruchamia kontener
— *ubuntu* – obraz, na podstawie którego tworzony jest kontener (obraz systemu ubuntu)
— */bin/echo 'Hello world'* – polecenie, które ma zostać wywołane wewnątrz utworzonego kontenera

Odpowiedź Dockera na zadaną komendę:

```
1 Unable to find image 'ubuntu:latest' locally
  latest: Pulling from library/ubuntu
  759d6771041e: Pull complete
  8836b825667b: Pull complete
5 c2f5e51744e6: Pull complete
  a3ed95caeb02: Pull complete
  Digest: sha256:b4dbab2d8029edddfe494f42183de20b7e2e871a424ff16ffe7b15a31f102536
  Status: Downloaded newer image for ubuntu:latest
  Hello world
```

Docker przeszukuje dostępne lokalnie obrazy. Jeżeli nie znajduje danego obrazu lokalnie, wyszukuje i pobiera go z repozytorium Docker Hub. Następnie uruchamia kontener, wykonuje polecenie wyświetlenia napisu "Hello world" I zatrzymuje kontener.

Aplikację można uruchomić również w trybie interaktywnym, stosując flagi -i -t za pomocą polecenia

```
1 $ docker run -t -i ubuntu /bin/bash
```

gdzie flaga - t powoduje uruchomienie terminala w kontenerze, natomiast flaga - i pozwala wczytywać w kontenerze znaki wpisywane na klawiaturze. Odpowiedź Dockera:

```
1 root@879d9c1665ca:/# ls
  bin   dev  home  lib64  mnt  proc  run   srv  tmp  var boot  etc  lib   media
```

```
opt  root  sbin  sys  usr
root@879d9c1665ca:/# exit
exit
```

Innym sposobem uruchomienia aplikacji jest uruchomienie jej w trybie demona dzięki zastosowaniu flagi -d. Poniższe polecenie spowoduje wyświetlenie napisu 'Hello world' co 1 sekundę:

```
$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world;
sleep 1; done"
```

Sprawdzenie, czy kontener został uruchomiony:

```
$ docker ps
```

Odpowiedź Dockera:

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
c99a2503f886        ubuntu              ''/bin/sh -c 'while tr''   2 minutes ago
Up 2 minutes                            evil_morse
```

Sprawdzenie wyjścia kontenera:

```
$ docker logs evil_morse
```

Odpowiedź Dockera:

```
hello world
hello world
hello world ....
```

Zatrzymanie kontenera:

```
$ docker stop evil_morse
```

4.2.2. Uruchomienie aplikacji webowej

Aplikacja uruchamiana jest poleceniem:

```
$ docker run -d -P training/webapp python app.py
```

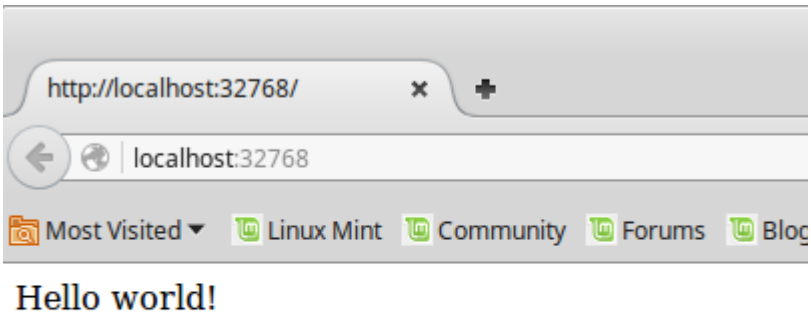
gdzie
— *docker run* – uruchamia kontener
— *-d* - flaga powodująca działanie w tle kontenera
— *-P* – flaga nakazująca Dockerowi odwzorowanie portów kontenera na porty na maszynie gospodarza
— *training/webapp* – obraz, na podstawie którego tworzony jest kontener
— *python app.py* – polecenie, które ma zostać wywołane wewnątrz utworzonego kontenera (uruchomienie aplikacji webowej)
Odpowiedź Dockera:

```
Unable to find image 'training/webapp:latest' locally
latest: Pulling from training/webapp
e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
10bbbc0fc0ff: Pull complete
fca59b508e9f: Pull complete
e7ae2541b15b: Pull complete
9dd97ef58ce9: Pull complete
a4c1b0cb7af7: Pull complete
Digest: sha256:06e9c1983bd6d5db5fba376ccd63bfa529e8d02f23d5079b8f74a616308fb11d
Status: Downloaded newer image for training/webapp:latest
426c053faa028c411e8ca32f21ec9907da7b442dd83a321f4e47697a8e7cfb7f
```

Obraz nie został znaleziony w lokalnym rejestrze, dlatego pobrano go z rejestru Docker Hub. Za pomocą polecenia *docker ps -l* można sprawdzić szczegółowe informacje dotyczące ostatnio uruchomionego kontnera.

```
 $ docker ps -l
CONTAINER ID   IMAGE              COMMAND             CREATED
STATUS              PORTS                     NAMES
426c053faa02   training/webapp    "python app.py"     13 minutes ago
Up 12 minutes       0.0.0.0:32768->5000/tcp   berserk_euler
```

Kolumna *ports* mówi nam o tym, że port 5000 w kontenerze jest odwzorowany na port 32768 na maszynie lokalnej. Działanie aplikacji możemy sprawdzić wyszukując w przeglądarce port 32768.



Rysunek 4.1. *Działająca aplikacja webowa*

4.2.3. Tworzenie własnego obrazu

Własny obraz można utworzyć na dwa sposoby:

— Poprzez aktualizację kontenera stworzonego na podstawie obrazu oraz zapisanie wprowadzonych zmian do obrazu
— Za pomocą pliku *Dockerfile*

W pierwszym przypadku mamy utworzony kontener na podstawie obrazu i wprowadziliśmy do niego nowe dane, np. zainstalowaliśmy program. Nowy obraz tworzymy za pomocą polecenia:

```
$ docker commit -m "Added program" -a "Author" bd625e9176a7 author/ubuntu:v2
```

gdzie
— *docker commit* – zapisuje stn kontenera w postaci obrazu
— *-m* – flaga pozwalająca zapisać co zostało zmienione w obrazie
— *-a* – flaga informująca o autorze wprowadzonych zmian
— *bd625e9176a7* – numer ID kontenera, na bazie którego tworzony jest nowy obraz
— *author/ubuntu:v2* – nazwa repozytorium oraz znacznik wersji utworzonego obrazu
Po wywołaniu tego polecenia moemy sprawdzić listę obrazów w lokalnym repozytorium:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
author/ubuntu       v2                  adbc62b82a11        19 seconds ago
725.1 MB
ubuntu              latest              b72889fa879c        11 days ago
188 MB
training/webapp     latest              6fae60ef3446        11 months ago
348.8 MB
```

Tworzenie obrazu z pliku *Dockerfile* przebiega następująco. Najpierw tworzymy katalog na nasz plik, a w nim plik *Dockerfile*:

```
$ mkdir dockerbuild
$ cd dockerbuild/
$ touch Dockerfile
```

Zawartość pliku może wyglądać następująco:

```
FROM ubuntu:14.04
MAINTAINER Author <author@example.com>
RUN apt-get update
RUN apt-get install build-essential
RUN apt-get install qt5-default
```

Plik *Dockerfile* składa się z instrukcji, których nzwy pisane są wielkimi literami poprzedzających polecenia: *INSTRUKCJA polecenie*. Wyjaśnienie instrukcji wykorzystanych w powyższym pliku:
— *FROM ubuntu:14.04* – ta komenda mówi o tym, że obraz bazuje na obrazie systemu ubuntu wersji 14.04
— *MAINTAINER Author <author@example.com>* - wskazanie autora obrazu
— *RUN ...* - instrukcja RUN nakazuje wykonać w kontenerze dane polecenia
Następnie budujemy obraz na podstawie pliku *Dockerfile* (przy wywołaniu tej komendy musimy znajdować się w katalogu, w którym jest nasz plik *Dockerfile*):

```
$ docker build -t author/ubuntu:v2 .
```

4.2.4. Korzystanie z Docker Hub

Utworzony obraz może zostac udostępniony w repozytorium Docker Hub, a następnie pobrany przez członków naszego zespołu. W tym celu należy się zalogować na swoje konto Docker Hub za pomoc klienta dockera:

```
1 $ docker login
```

Po podaniu danych logowania możemy udostępnić obraz:

```
1 $ docker push author/ubuntu:v2
```

Następnie może on zostać pobrany przez innch użytkowników:

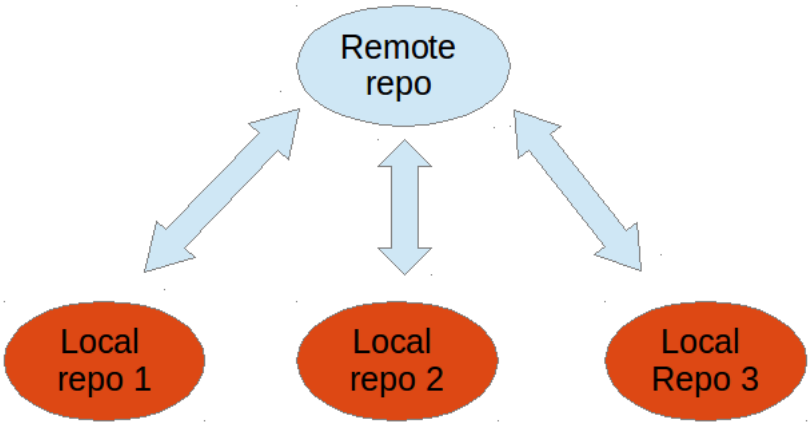
```
1 $ docker pull author/ubuntu:v2
```

Dodatek A. Wprowadzenie i instrukcja użytkowania systemu kontroli wersji Git

A.1. Wprowadzenie

Do wspomagania równoległego, rozgałęzionego procesu rozwoju projektu przez wielu programistów zdecydowano się wykorzystać rozproszony system kontroli wersji Git.

Git jest obecnie najbardziej popularną implementacją rozproszonego systemu kontroli wersji. W przeciwieństwie do innych systemów kontroli wersji, Git nie zapamiętuje zmian między kolejnymi rewizjami, lecz kompletne obrazy. Każdy z użytkowników posiada lokalną kopię repozytorium na swoim własnym komputerze po uprzednim sklonowaniu repozytorium zewnętrznego (zdalnego). Pozwala to na pracę w trybie off-line i wprowadzanie zmian w wersji lokalnej projektu i efektywną pracę nad dużymi projektami.



Rysunek A.1. *Schemat połączeń między repozytoriami Git*

Następnie zmiany mogą być wymieniane między lokalnymi repozytoriami. Służy do tego repozytorium zewnętrzne (remote repository) działające na serwerze. Serwisem przechowującym rozwijany projekt jest GitHub (`https://github.com/`), który udostępnia darmowy hosting open source.

Repozytorium projektu jest dostępne pod adresem: `https://github.com/Gonz8/RSO-16L`

A.2. Przygotowanie do pracy i pierwsze pobranie zawartości repozytorium

Na samym początku wymagane jest posiadanie klienta Git zainstalowanego w swoim systemie. Wszelkie istotne informacje dotyczące korzystania z Git możemy uzyskać wpisując w terminalu:

```
$ git help
```

Następnie musimy określić własną nazwę oraz adres e-mail w systemie Git:

```
$ git config --global user.name ''Your Name''
$ git config --global user.email ''address@example.com''
```

Aby zaimportować repozytorium ze wspomnianego serwera należy wykonać polecenie:

```
$ git clone https://github.com/Gonz8/RSO-16L
```

Wszystkie pliki zostaną sklonowane do nowo utworzonego katalogu, z poziomu którego należy utworzyć lokalne repozytorium:

```
$ git init
```

Utworzone w ten sposób repozytorium jest już powiązane ze zdalną wersją (origin). Możemy to sprawdzić przy użyciu polecenia:

```
$ git remote -v
```

A.3. Użytkowanie

Po zaimportowaniu projektu oraz utworzeniu lokalnej kopii repozytorium można rozpocząć pracę z danymi. Aby sprawdzić status dokonanych zmian należy użyć w katalogu z kopią roboczą następującego polecenia:

```
$ git status
```

Dodanie nowego pliku, kilku plików lub katalogu do kopii roboczej wykonywane jest przy użyciu komendy:

```
$ git add <filename>
$ git add *
```

Aby zatwierdzić wszelkie dokonane zmiany w lokalnym repozytorium należy użyć polecenia:

```
$ git commit -a
```

a następnie podać treść/opis poczynionych zmian. W celu przechwycenia najnowszych zmian z serwera wykonujemy polecenie:

```
$ git fetch origin
```

natomiast, aby przechwycić zmiany z serwera i dodatkowo dołączyć je do własnego katalogu roboczego wykonujemy polecenie:

```
$ git pull
```

Wysłanie zmian poczynionych w wersji lokalnej do zdalnego repozytorium realizowane jest dzięki komendzie:

```
$ git push origin master
$ git push origin <branchname>
```

Git pozwala również na tworzenie, usuwanie i przełączanie się między gałęziami projektu, do wykonania tych operacji służą następujące polecenia:

```
$ git checkout –b <branchname>
$ git branch –d <branchname>
$ git checkout <branchname>
```

Wyświetlenie listy wszystkich gałęzi dostępnych w repozytorium możliwe jest poprzez komendę:

```
$ git branch
```

Natomiast w celu dołączenia innej gałęzi do obecnie aktywnej należy wykonać polecenie:

```
$ git merge <branchname>
```

Ostatnim również istotnym poleceniem jest wyświetlenie historii logów/commit'ów:

```
$ git log
```

Dodatkowo po zainstalowaniu pakietu ***gitk*** można wyświetlać graficzną prezentację historii zmian projektu.

A.4. Dodatkowe narzędzia

Mimo faktu, że Git jest rozproszonym systemem kontroli wersji zdecydowano się na wykonywanie dodatkowej i okresowej kopii zapasowej projektu (tzw. backup). Kopia bezpieczeństwa jest jednym z elementów utrzymania repozytorium i zabezpiecza przed utratą danych (np. awarii może ulec komputer głównego programisty, przez co istnieje ryzyko utraty lokalnej kopii repozytorium). Rozwiązaniem tego zagadnienia będzie okresowe wywoływanie napisanego skryptu, który aktualizuje zawartość sklonowanego repozytorium znajdującego się również w folderze powiązanym z naszym kontem Dropbox. Pozwala to w prosty sposób przechowywać kopię zawartości repozytorium na innym serwerze zewnętrznym (poza GitHub) i zabezpieczyć przed utratą danych.

Serwis GitHub świadczy szereg dodatkowych usług wspomagających rozwój projektu programistycznego. Zdecydowano się wykorzystać usługę Wiki, gdzie w łatwy sposób można wprowadzać i modyfikować istotne treści w kontekście projektu. Jest to miejsce, gdzie można w szybki i wygodny sposób odnaleźć uporządkowane informacje. Jednym z odnośników w tej usłudze jest zakładka "Dobre praktyki

kodowania", gdzie znajdują się wszystkie wspólnie ustalone przez programistów zasady pisania kodu pozwalające utrzymać przejrzystość i spójność kodu.

Ponadto zdecydowano się uruchomić usługę śledzenia zgłoszeń (tzw. *issue tracking*) na potrzeby wspierania testowania oprogramowania oraz zgłaszania napotkanych błędów i tworzenia dla nich poprawek.

A.5. Przydatne informacje

Instrukcja została opracowana na postawie materiałów znalezionych w sieci. Więcej informacji dotyczących użytkowania systemu kontroli wersji Git można znaleźć na stronach:

— `https://git-scm.com/docs/gittutorial`
— `http://www.vogella.com/tutorials/Git/article.html#git`

Dodatek B. Spotkania zespołu

Poniżej znajduje się lista spotkań zespołu (z datami i ustaleniami):

B.1. Spotkanie zespołu nr 1

Data spotkania: 31 marca 2016
Ustalenia:

— Przedstawienie oczekiwań wobec ról w projekcie
— Interpretacja i doprecyzowanie tematu projektu
— Wybór danych medycznych jako danych przetwarzanych w ramach projektu
— Przydział zadań na najbliższy czas:
  - Dominik Giżyński - założenie repozytorium, instrukcja korzystania oraz dokument dobrych praktyk obowiązujących w projekcie
  - Piotr Kuciński - doprecyzowanie wymagań przedmiotu, przegląd algorytmów do wykorzystania w projekcie
  - Włodzimierz Szewczyk - doprecyzowanie wymagań przedmiotu, przegląd algorytmów do wykorzystania w projekcie
  - Michał Herman - szkielet dokumentacji, wybór bazy danych używanej w projekcie
  - Magda Malenda - architektura systemu, podział systemu na moduły, specyfikacja wymagań, wybór bazy danych używanej w projekcie
  - Joanna Ohradka - koncepcja wykorzystania narzędzia Docker
  - Tomasz Rydzewski - harmonogram projektu, specyfikacja wymagań

B.2. Spotkanie zespołu nr 2

Data spotkania: 19 kwietnia 2016
Ustalenia:

— Porzucenie pomysłu klastrowej bazy danych
— Definiowanie zadań na drugi etap
— Spisanie koncepcji architektury, wymagań i zastosowania Dockera (do 23 kwietnia)
— Zebranie i ujednolicenie dokumentacji (do 27 kwietnia)

B.3. Spotkanie zespołu nr 3

Data spotkania: 10 maja 2016
Ustalenia:

— Omówienie poprawek do etapu pierwszego

— Cotygodniowe spotkania (termin wybrany później ankietą)
— W przypadku potrzeby częstszych spotkań w mniejszym gronie - telekonferencje
— Ustalenie rodzaju przetwarzania danych : statystyka i anonimizacja
— Przeniesienie komunikacji w ramach zadań na GitHub
— Przydział zadań na drugi etap:
  - Dominik Giżyński - prototyp aplikacji
  - Piotr Kuciński - plan testów
  - Włodzimierz Szewczyk - uruchamianie / zatrzymywanie aplikacji, scenariusz końcowej demonstracji projektu
  - Michał Herman - zebranie poprawionych dokumentacji pierwszego etapu, zredagowanie ustaleń ze spotkań zespołu, wyszukanie aktów prawnych dotyczących ochrony danych osobowych, wygenerowanie przykładowych danych do bazy
  - Magda Malenda - szczegółowy opis rozwiązania, protokół komunikacyjny
  - Joanna Ohradka - protokół spójności, szkolenia docker

B.4. Spotkanie zespołu nr 4

Data spotkania: 17 maja 2016
Ustalenia:

— Omówienie postępu prac
— Harmonogram na następne dwa tygodnie (do końca drugiego etapu):
  - Interfejs protokołu komunikacyjnego (do 20 maja) - osoby odpowiedzialne : Piotr Kuciński i Magda Malenda
  - Poprawiona i uzupełniona dokumentacja oraz raport z postępu prac (do 26 maja) - Michał Herman
  - Połączenie poszczególnych modułów (24 maja) - Dominik Giżyński i Włodzimierz Szewczyk
— Potrzeba środowiska wirtualnego (przygotowanie maszyny wirtualnej - Piotr Kuciński)

B.5. Spotkanie zespołu nr 5

Data spotkania: 17 maja 2016
Ustalenia:

— Dwie wersje klienta (jedna pobierająca - komunikuje się z serwerami warstwy zewnętrznej, druga dodająca/usuwająca dane z bazy - komunikuje się bezpośrednio z serwerami warstwy wewnętrznej).
— Przydział kolejnych zadań:
  - Piotr Kuciński - klient
  - Joanna Ohradka - protokół spójności
  - Magda Malenda - węzeł warstwy wewnętrznej (bez statystyki danych)
  - Włodzimierz Szewczyk - prezentacja końcowa
  - Dominik Giżyński - węzeł warstwy zewnętrznej
  - Michał Herman - zasilenie bazy danych przykładowymi danymi, statystyka
— Potrzeba środowiska wirtualnego (przygotowanie maszyny wirtualnej - Piotr Kuciński)

Dodatek C. Narzędzia

Poniżej znajduje się lista narzędzi i technologii użytych do realizacji projektu.

— C++
— Qt
— PostgreSQL
— GanttProject
— LaTeX
— Git
— OpenSSL