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

1. Projekt

1.1. Harmonogram

Poniżej przedstawiony został harmonogram projektu. Diagram Gantta projektu został dołączony jako dodatek C.

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 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

2. Wymagania

2.1. Wymagania funkcjonalne

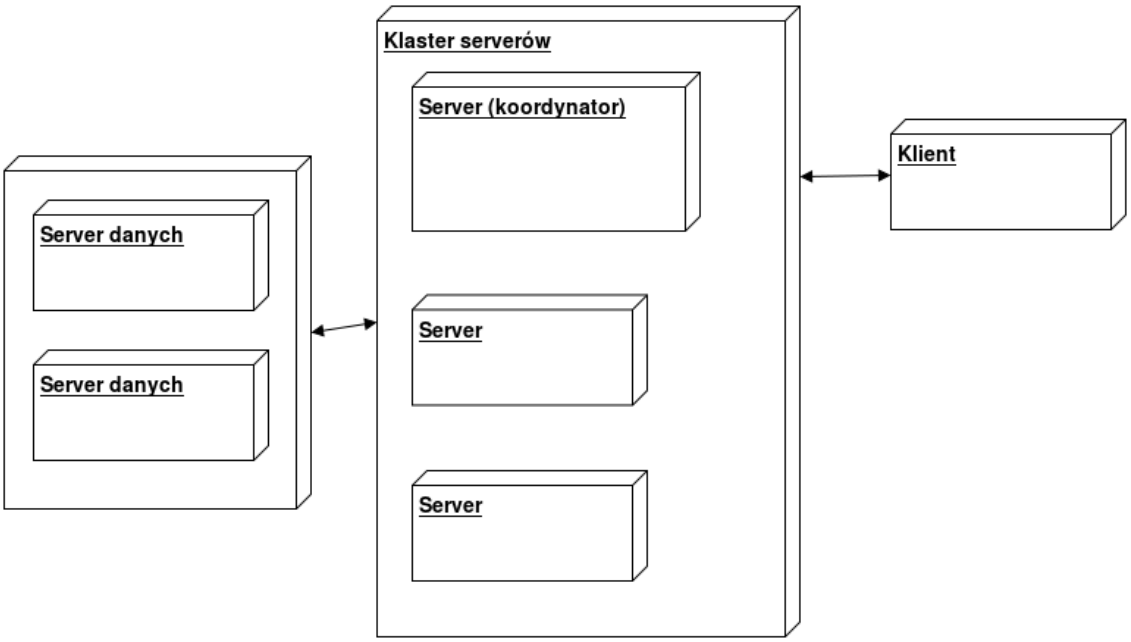
— Przechowywanie wrażliwych danych medycznych
— Udostępnianie danych odpowiednim osobom w zależności od uprawnień
— Prosta, konsolowa aplikacja kliencka wysyłająca zapytania do serwera i wyświetlająca wyniki

2.2. Wymagania niefunkcjonalne

— Wspólny testowy plik konfiguracyjny dla części serwerowej i klienckiej
— Współbieżne oprogramowanie realizujące część serwerową
— Odporność na uszkodzenie węzła
— Realizacja usługi w trakcie awarii w czasie obsługi
— Możliwość ponownego wpięcia węzła, z którym utracono łączność
— Odporność na próbę wpięcia wrogiego, nieuprawnionego węzła
— Zarządzanie zasobami transparentne dla oprogramowania klienckiego
— Większy niż 1 poziom redundancji danych
— Uruchamianie i zamykanie części serwerowej jednokrotnym wywolaniem skryptu na każdym węźle
— Środowisko Linux Ubuntu 14.04 LTS

3. Architektura

3.1. Opis ogólny architektury systemu



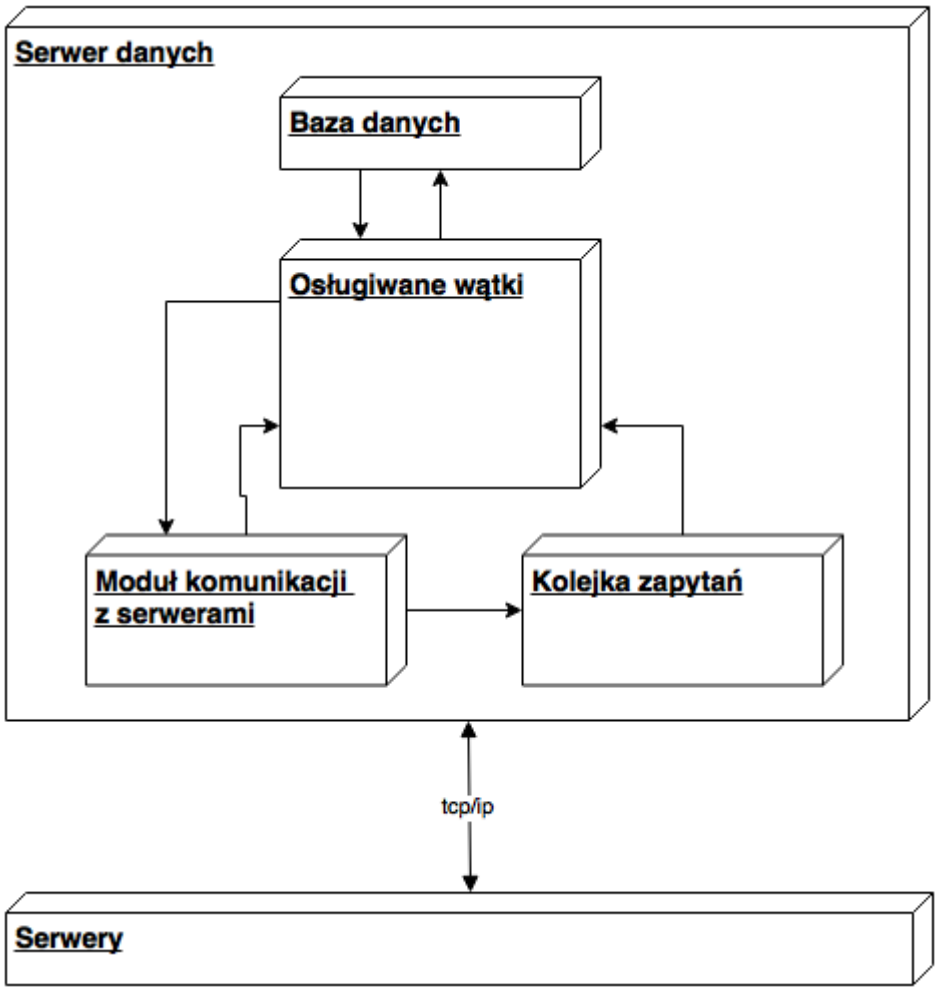
Rysunek 3.1. *Schemat architektury systemu*

System po stronie serwerowej składa się z dwóch warstw. Wewnętrznej, która przechowuje wrażliwe informacje medyczne i zewnętrznej udostępniającej informację oprogramowaniu klienckiemu. Część wewnętrzną stanowi system rozproszonej bazy danych, a część zewnętrzną klaster serwerów.

3.2. Baza danych

Zastosowany zostanie mechanizm prostej pojedynczej replikacji serwerów danych. Węzły łączą się z wybraną bazą, zapis realizowany jest na dwóch bazach danych jednocześnie. Przewidziane są dwa serwery danych, co daje redundancję równą 2.

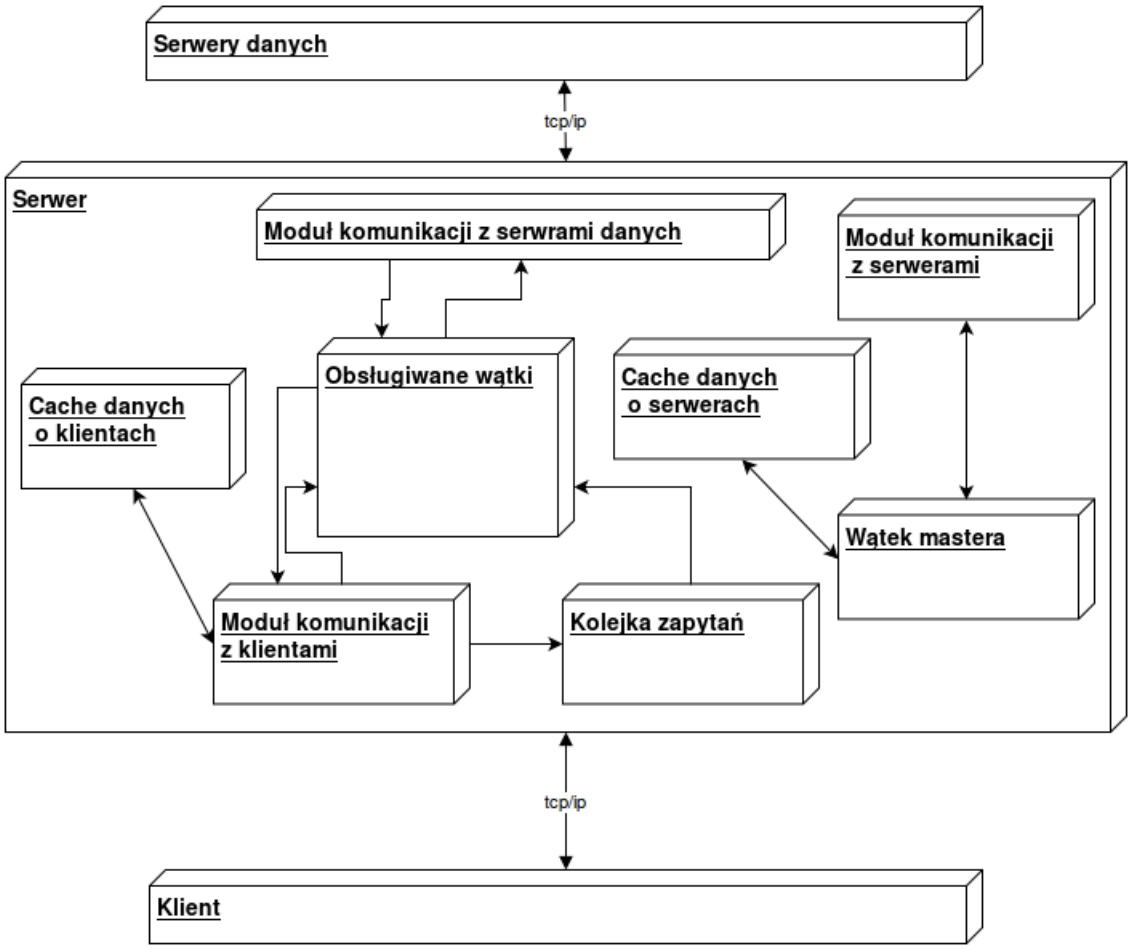
Komunikacja z serwerami odbywa się przez tcp/ip. W komunikacji pośredniczy moduł komunikacyjny, który oczekuje na zapytania. Każde zapytanie jest obsługiwane w oddzielnym wątku. Zapytania są kolejkowane lub odrzucane, gdy zapytań będzie za dużo. Komunikacja jest szyfrowana przy pomocy RSA.



Rysunek 3.2. *Schemat serwera bazy danych*

3.3. Serwer

Serwer odbiera zapytania od klienta i pobiera dane z bazy. W tym celu nasłuchuje w oczekiwaniu na połączenie od klienta. Komunikacja klient-serwer odbywa się przez tcp/ip. Po odebraniu żądania , tworzony jest nowy wątek, który realizuje zapytanie, komunikuje się z bazą danych oraz formułuje odpowiedź. Serwer może obsłużyć ograniczoną liczbę klientów. Nadmiarowe zapytania może przekazać innym serwerom, a jeżeli wszystkie są zajęte to wstawia do kolejki oczekujących zapytań lub odsyła odpowiedni komunikat, jeżeli w kolejce nie ma miejsc. Zadaniem serwera jest też autoryzacja łączącego się klienta, sprawdzenie jego uprawnień. Autoryzacja i uwierzytelnienie odbywa się przez podanie loginu i hasła. Gdy serwer dostanie zapytanie od niezalogowanego klienta odsyła prośbę o login i hasło. Po autoryzacji klient jest zapisywany w tablicy aktywnych klientów w pamięci cache, następnie serwer generuje, zapisuje i wysyła token, który będzie służył do uwierzytelniania przy kolejnych żądaniach aż do momentu wylogowania lub gdy użytkownik przez określony czas będzie nieaktywny. Dane o aktywnych klientach nie są przechowywane na wszystkich węzłach, więc jeżeli jeden z nich ulegnie awarii



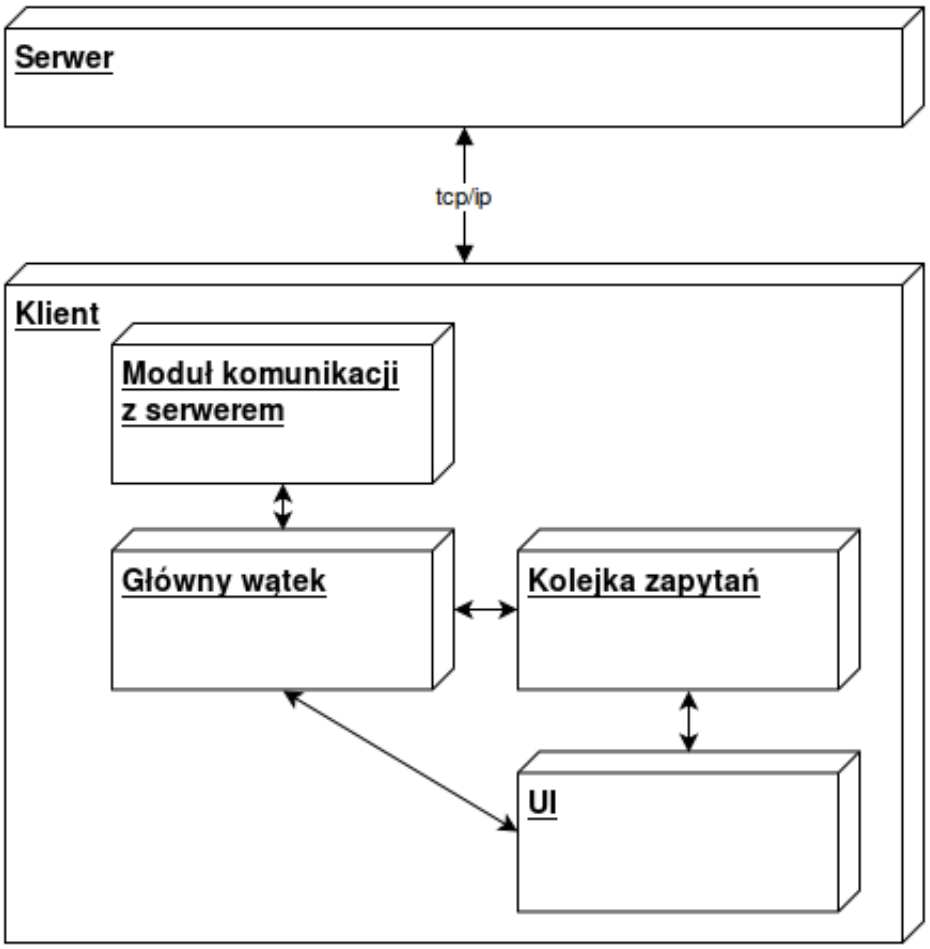
Rysunek 3.3. *Schemat serwera aplikacyjnego*

konieczna będzie ponowna autoryzacja i uwierzytelnienie na innym węźle. Jeden z serwerów jest koordynatorem. Koordynator kontroluje pracę pozostałych serwerów, sprawdza, czy wszystkie są dostępne, autoryzuje nowy węzeł, który chce się połączyć. Tworzy i rozsyła do wszystkich węzłów tablicę aktywności serwerów oraz serwerów danych. Sprawdzanie obecności serwerów odbywa się poprzez cykliczne odpytywanie. Gdy master ulegnie awarii jego rolę przejmuje inny z serwerów. 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 wysyła komunikat do wszystkich węzłów o niższym numerze, jeżeli nikt nie odpowie to serwer staje się nowym koordynatorem i wysyła do wszystkich pozostałych węzłów informujący komunikat. Jeżeli któryś z węzłów o niższym numerze odpowie to on przejmuje kontrolę. Brak koordynatora będzie zauważony, gdy węzeł nie zostanie odpytany w odpowiednim czasie.

3.3.1. Autoryzacja węzłów

Każdy węzeł posiada nazwę oraz numer. Aby dołączyć się do klastra serwerów musi wysłać masterowi numer oraz swoją nazwę zaszyfrowaną kluczem publicznym mastera. W odpowiedzi dostaje potwierdzenie i aktualny stan systemu, aby zaktualizować listę aktywnych serwerów. Klucze publiczne są zapisywane w plikach konfiguracyjnych serwera i klienta.

3.4. Aplikacja kliencka



Rysunek 3.4. *Schemat aplikacji klienckiej*

Zadaniem aplikacji klienckiej jest nawiązanie połączenia z jednym z serwerów. Adresy serwerów są określone w pliku konfiguracyjnym. Jeżeli połączenie z jednym z węzłów zakończy się niepowodzeniem, następuje próba połączenia się z kolejnymi. Aplikacja umożliwia zalogowanie się i dostęp do funkcji oferowanych przez API serwera. Jeżeli jakieś żądanie się nie powiedzie pomimo dostępności serwera, następuje ponowna próba, wysłania żądania ale do innego serwera. Klient musi aktualizować listę serwerów poprzez cykliczne odpytywanie.

3.5. Plik konfiguracyjny

W pliku konfiguracyjnym klienta znajdują się adresy, porty i nazwy serwerów oraz ich klucze publiczne. W pliku konfiguracyjnym serwerów znajdują się adresy, porty i nazwy serwerów, adresy serwerów baz danych, maksymalna liczba jednocześnie obsłużonych wątków, okres odpytywania węzłów przez serwer główny, czas trwania sesji z klientem, klucze publiczne serwerów. W pliku konfiguracyjnym serwerów danych znajdują się adresy serwerów danych oraz adresy węzłów z ich kluczami publicznymi, liczba maksymalnie obsłużonych wątków.

4. Docker

Dodatek A. Narzędzia

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

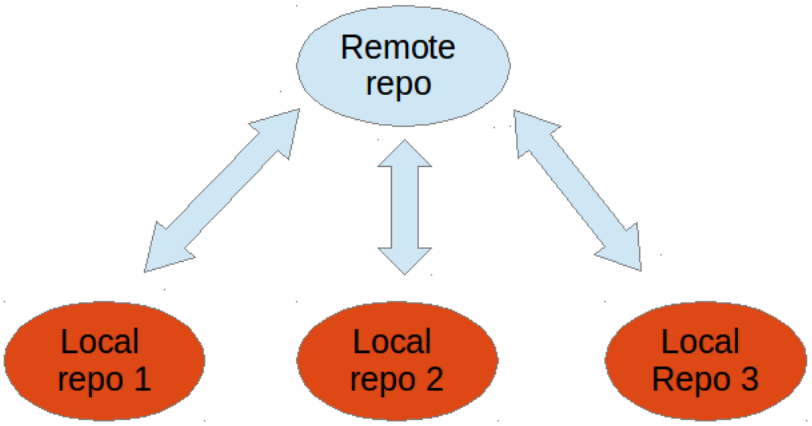
— C++
— Qt
— PostgreSQL

Dodatek B. Wprowadzenie i instrukcja użytkowania systemu kontroli wersji Git

B.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 B.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`

B.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

B.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.

B.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) (...)

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.

B.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 C. Diagram Gantta