Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Rok akademicki 2014/2015

Dokumentacja projektu RSO

Grupa C (godzina 16.00)
Domagała Bartosz, Kornata Jarosław, Modzelewski Jędrzej, Marcin Stepnowski

Usługa bezpiecznej niezawodnej dystrybucji przetworzonej chronionej informacji

Prowadzący projekt:
dr inż. Tomasz Jordan Kruk

Spis treści

1. Opis projektu

1.1. Treść zadania

Zaprojektować i zaimplementować usługę udostępniania danych statystycznych na temat studentów uczelni wyższej, przy założeniu bezpiecznej i niezawodnej dystrybucji danych wrażliwych - poufnych informacji o studentach.

Usługa powinna:

— mieć dokładnie jeden ten sam tekstowy plik konfiguracyjny dla części serwerowej i klientów usługi,

— składać się po stronie serwerowej z dwóch warstw: (1) wewnętrznej wytwarzającej, przechowującej i przesyłającej przetworzoną wrażliwą informację, (2) warstwy zewnętrznej do udostępniania przetworzonej informacji oprogramowaniu klienckiemu usługi,

— oprogramowanie realizujące część serwerową powinno być współbieżne,

— składać się po stronie klienckiej z prostych narzędzi opatulających zaproponowane API protokołu komunikacyjnego,

— zapewniać odporność na uszkodzenia węzła, stąd każda z warstw powinna być uruchomiona na co najmniej dwóch węzłach,

— zapewniać realizację usługi w trakcie awarii w czasie obsługi,

— obsługiwać scenariusz próby ponownego wpięcia się przez węzeł dowolnej z warstw, z którym wcześniej utracono łączność, a zarazem być odporną na próbę zastąpienia nieosiągalnego sieciowo węzła (np. w wyniku ataku DoS, odmowa usługi) węzłem wrogim do tego nieuprawnionym,

— na bieżąco i transparentnie dla oprogramowania klienckiego zarządzać dostępnymi zasobami lokalizacyjnymi,

— zapewniać zadany (większy niż 1 ale dopuszczalny konfiguracją mniejszy niż liczba serwerów danych) poziom redundancji - z obsługą uzupełniania kopii na innych serwerach w przypadku awarii włącznie,

— cała część sewerowa usługi powinna być uruchamiana i zamykana jednokrotnym wywołaniem skryptu na jednym węźle serwera z jednym argumentem <start|stop|status> (wykorzystanie nieinteraktywnych metod automatycznego uwierzytelniania węzłów w SSH).

1.2. Ogólny opis rozwiązania

Nasze rozwiązanie to rozproszony system do rejestracji na przedmioty studentów uczelni wyższej. Warstwą wewnętrzną przechowujące dane wrażliwe, w tym przypadku będą węzły rejestrujące studentów, ideowo umieszczone w dziekanatach instytutów i obsługiwane przez osoby tam pracujące. Warstwa pośrednia przechowuje informacje o studentach, jednak bez informacji wrażliwej. Posiada jedynie dane o istnieniu studenta zapisanego na przedmioty, bez imienia, nazwiska i daty urodzenia. Owe informacje są przetwarzane na żądanie aplikacji klienckiej, dostępnej dla studentów, którzy chcą dowiedzieć się, ile osób zapisanych jest na dany przedmiot.

1.2.1. Dane wrażliwe a kwestie prawne

W związku z ochroną danych wrażliwych zapoznaliśmy się z Ustawą o ochronie danych osobowych z dnia 29 sierpnia 1997 r. Pomimo, że generowane przez nas w ramach projektu losowe dane, nie zawierają faktycznych wrażliwych informacji, przystosowaliśmy nasz system do obowiązującego w Polsce prawa, pod uwagę biorąc przede wszystkim poniższe kwestie prawne z Ustawy o ochronie danych osobowych:

— Art. 2 *„Ustawę stosuje się do przetwarzania danych osobowych (...) w systemach informatycznych, także w przypadku przetwarzania danych poza zbiorem danych."*s

— Art. 6 *„W rozumieniu ustawy za dane osobowe uważa się wszelkie informacje dotyczące zidentyfikowanej lub możliwej do zidentyfikowania osoby fizycznej."*

— Art. 6 *„Informacji nie uważa się za umożliwiającą określenie tożsamości osoby, jeżeli wymagałoby to nadmiernych kosztów, czasu lub działań."*

— Informacje o Generalnym Inspektorze w Art. 14 i Art. 16.

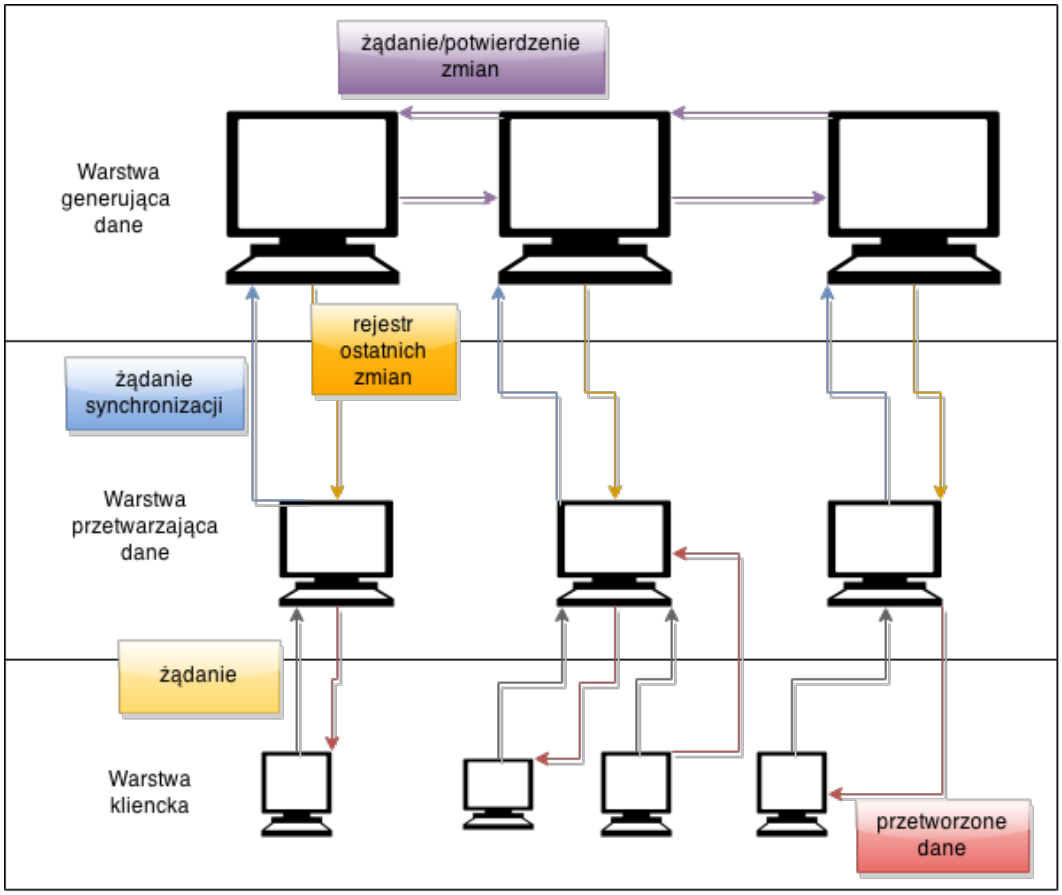
— Informacje na temat ochrony danych w Art. 26 oraz Art. 26a.

— Informacje o uprawnieniach administratora w Art. 31.

— Informacje o zabezpieczeniu danych osobowych z rozdziału 5.

1.3. Projekt architektury

Wewnętrzna warstwa serwera wytwarza i przechowuje informacje na temat studentów uczelni wyższej - stworzone na podstawie danych wprowadzonych przez pracowników dziekanatu. Następnie owa warstwa przesyła informacje do warstwy zewnętrznej serwera (warstwy przetwarzającej) świadczącej usługi udostępniania informacji przetworzonej rozwiązaniom klienckim. Każda z warstw komunikować się będzie z użyciem protokołu TCP/IP przesyłając wiadomości zserializowane za pomocą Google Protocols Buffer.



Rysunek 1.1. Diagram ilustrujący komunikację pomiędzy poszczególnymi warstwami

1.3.1. Warstwa wewnętrzna serwera

Warstwa stworzona w technologii token ring. Węzły połączone są w pierścień i przesyłają sobie kolejno token zawierający informacje o potencjalnych zmianach. Tylko węzeł posiadający token może wykonywać operację aktualizacji danych. Każda zmiana musi być zatwierdzona przez pozostałe węzły. Zapewnia to synchronizację danych oraz bieżące sprawdzenie działania pozostałych węzłów.

Dane wrażliwe przechowywane są w postaci rekordów zawierających:

— ID studenta

— imię studenta

— nazwisko studenta

— datę urodzenia

— listę przedmiotów na jakie jest zapisany

Opis szczegółowy

W momencie podłączania węzeł wewnętrzny wysyła wiadomość z chęcią zalogowania się do sieci do jednego z innych węzłów z listy znajdującej się w pliku konfiguracyjnym. Robi tak aż do skutku, tzn. dopóki nie dostanie wiadomości od któregokolwiek innych węzłów z informacją o włączeniu się do sieci. Jeśli jednak mimo wszystkich prób nie dostanie wiadomości - uznaje, że jest pierwszym węzłem w sieci. Jest to rozwiązanie czasochłonne, niemniej będzie ono miało miejsce bardzo rzadko, ze względu na ogólnie przyjętą zasadę, że warstwa wewnętrzna działa nieprzerwanie i wymaga przynajmniej 3 aktywnych węzłów do poprawnego działania.

Węzły w tej warstwie wysyłają sobie kolejno token po okręgu, tzn. kiedy węzeł dostaje token wprowadza odpowiednią aktualizację (nie zawsze!) oraz przesyła go dalej). Token zawiera w sobie informację o stanie, w którym znajduje się sieć, a także obraz jej topologii (następujące po sobie węzły). Dzięki temu można zapewnić stały stan sieci dla wszystkich węzłów. Z każdą kolejną operacją (zatwierdzoną przez wszystkie węzły) stan sieci zwiększa się o jeden. Zasady są następujące:

— Jeśli stan jest większy od posiadanego przez węzeł, następuje zapis nowego stan i przesłanie do następnego węzła.

— Jeśli stan jest równy posiadanemu i nie jest on ustawiony przez ten węzeł, to może on wykonać swoją operację (zwiększając stan o 1).

— Jeśli stan jest równy posiadanemu i jest on ustawiony przez ten węzeł, oznacza to, że ostatnia operacja wygenerowana przez ten węzeł została zaakceptowana przez sieć.

— Jeśli stan jest mniejszy od posiadanego, token jest ignorowany.

Jeśli token jest oznaczony jako „wolny” (przypadek 2 z powyższej listy) dany węzeł może użyć go do wprowadzenia nowych danych, nowego węzła do sieci bądź odpytania o aktualność danych. Następnie węzeł czeka na powrót tokena z odpowiedzią i po jej uzyskaniu podaje go dalej. Dzięki temu, że węzeł nie może użyć tokena od razu po poprzednim zadaniu, które wygenerował, eliminujemy problem zagłodzenia innych węzłów. Ponadto poprzez ruch tokena po pierścieniu możemy w łatwy sposób „ustalać” dane decyzje z innymi węzłami oraz sprawdzać ich aktywność.

W przypadku zerwania połączenia z następnikiem wykonywane są kolejne operacje:

— Usuwanie następnika z topologii sieci w stanie, który aktualnie węzeł posiada.

— Zwiększenie stanu o 2.

— Połączenie się ze swoim nowym następnikiem.

— Przesłanie stanu do następnika (generacja tokena).

W sytuacji awarii węzeł automatycznie generuje token, a poprzedni staje się nieaktualny i wkrótce zostanie usunięty z sieci. Cecha generowania tokena powoduje, że zostanie on zawsze odtworzony w sieci niezależnie od tego czy awarii ulegnie węzeł, który go posiadał czy dowolny inny.

Należy pamiętać, że w trakcie zapisywania stanu trzeba sprawdzić, czy nie zmienił się nasz następnik w topologii sieci. Jest to szczególnie istotne w przypadku jednoczesnej awarii wielu węzłów, gdy wiele węzłów wygeneruje nowe tokeny. Jeden z nich zdominuje sieć, a drugi będzie zmuszony do ponownego sprawdzenia braku połączenia ze swoim dawnym następnikiem i ponownego wymuszenia zmiany topologii sieci. Takie rozwiązania zwiększa czas przywracania sieci do działania przy awarii wielu węzłów jednocześnie, niemniej zapewnia spójność jej stanu i istnienie w stanie ustalonym tylko jednego ważnego tokena.

Ponadto każdy węzeł zapisuje na bieżąco (w pliku na dysku) swój stan i w razie awarii i ponownego włączenia węzła do sieci może kontynuować pracę od momentu, na którym skończył.

1.3.2. Warstwa zewnętrzna serwera (warstwa przetwarzająca)

Warstwa przeznaczona do komunikacji z aplikacją kliencką. Na żądanie klienta przetwarza wcześniej otrzymane od warstwy wewnętrznej dane i wynik przetwarzania zwraca klientowi. Moduł ten synchronizuje dane okresowo (częstość jest ustalana w pliku konfiguracyjnym), tak aby administrator sam był wstanie dostosować interwał do aktualnych potrzeb).

Dane o studentach przechowywane są w poniższej postaci:

— ID studenta

— listę przedmiotów na jakie jest zapisany

Opis szczegółowy

Serwer zewnętrzny w momencie logowania wysyła Heartbeata do wszystkich serwerów z listy ustalonej w pliku konfiguracyjnym. Węzły, które mu odpowiedzą serwer uznaje za aktywne w danej chwili. Dzięki temu zyskuje obraz aktywnych węzłów w warstwie. Od tej pory co ustalony interwał wysyła wiadomość do każdego z tych serwerów. Jeśli otrzyma TestRequesta od węzła uznawanego do tej pory za nieaktywny, dodaje go do listy aktywnych oraz zaczyna wysyłać mu Heartbeaty. W momencie, gdy czas oczekiwania na Heartbeat przekracza ustalony interwał wysyłany jest TestRequest wymuszający odpowiedź Heartbeatem. Gdy i to zawiedzie węzeł uznany jest za nieaktywny.

W określonych w pliku konfiguracyjnym interwałach czasu serwer zewnętrzny pobiera dane od serwera wewnętrznego, zapisując je w swojej bazie w bezpiecznej formie (bez danych wrażliwych). Jeśli dany węzeł wewnętrzny nie odpowie - próbuje z kolejnym, itd.

W Heartbeatach przesyłana jest informacja o ilości podłączonych w danej chwili klientów.

Każdy węzeł utrzymuje dane na temat podłączonych do niego klientów oraz klientów podłączonych do pozostałych aktywnych węzłów. Dzięki temu można zapewnić równomierne obciążenie między węzłami serwera. Jeśli dany serwer dostanie wiadomość o chęci zalogowania się nowego klienta sprawdza czy inny serwer nie ma mniej klientów od niego (o więcej niż 2). Jeśli tak - przełącza klienta na inny serwer. Algorytm ten został szerzej opisany w opisie szczegółowym aplikacji klienckiej.

Nie jest potrzebne stałe sprawdzanie aktywności serwerów z warstwy wewnętrznej, ze względu na możliwe długie interwały czasowe między synchronizacją (np. jeden dzień). W przypadku częstszych synchronizacji (np. co minutę) sprawdzanie aktywności wewnętrznego węzła będzie i tak sprawdzane na bieżąco. W trakcie synchronizacji nie jest pobierana cała baza, a jedynie zmiany od ostatniej zgodności. Dokładnie cały algorytm został opisany w rozdziale „Sposób synchronizacji danych między warstwą zewnętrzną a wewnętrzną".

W momencie otrzymania wiadomości od klienta z danym żądaniem wykonywane są wymagane obliczenia i wysyłana odpowiedź.

1.3.3. Aplikacja kliencka

Warstwa przeznaczona dla klienta to z założenia lekka i intuicyjna aplikacja konsolowa. Moduł ten bezpośrednio łączyć będzie się tylko z modułem zewnętrznym serwera. Użytkownik, za pomocą tej aplikacji będzie mógł dowiedzieć się ile osób w danym momencie jest zapisany na wybranie przez niego przedmiot. Aplikacja będzie działała synchronicznie, tzn. po wysłaniu żądania będzie się zawieszała w oczekiwaniu na odpowiedź.

Opis szczegółowy

Każda aplikacja kliencka w momencie uruchomienia stara się połączyć do jednego z serwerów zewnętrznych. Lista serwerów jest z góry ustalona i znajduje się pliku konfiguracyjnym. Klient próbuje zestawić połączenie najpierw z pierwszym serwerem z listy, czekając 10 sekund na odpowiedź. W razie jej braku próbuje tego z kolejnym serwerem, itd. Jeśli klientowi uda się połączyć serwerem, jednak ten „stwierdzi", że obsługuje zbyt dużą ilość klientów w stosunku do pozostałych węzłów, wtedy wysyła on wiadomość zwrotną z informacją, do którego serwera klient ma się podłączyć. Następuje, więc ponowna próba połączenia, tym razem z polecanym węzłem. Cała operacja jest „ukryta" przed użytkownikiem aplikacji klienckiej, tzn. nie wie on, do którego serwera się podłącza.

Od momentu zestawienia połączenia klient może wysyłać żądania do serwera. Po wysłaniu żądania czeka na odpowiedź. Po jej otrzymaniu może wysłać kolejne żądanie. Ponadto co ustalony interwał klient wysyła Hearbeat do serwera dając znać, że ciągle jest podłączony. Analogicznie serwer informuje klienta, że działa poprawnie.

W razie awarii bądź braku odpowiedzi od serwera wysyłany jest TestRequest wymuszający wysłanie Heartbeata. Jeśli mimo to serwer nie odpowiada następuje próba zestawienia połączenia z kolejnym serwerem z listy.

1.3.4. Sposób synchronizacji danych między warstwami

W momencie rozpoczęcia synchronizacji serwer zewnętrzny nie wprowadza od razu pobranych danych do bazy. Przechowuje je lokalnie dopóki nie otrzyma całości danych. Dopiero wtedy wprowadza je do bazy (wszystkie jako jeden INSERT). Zapewnia to albo wprowadzanie danych w całości albo w ogóle.

Jeśli w trakcie synchronizacji nastąpi błąd serwera zewnętrznego a potem jego ponowne podłączenie cała synchronizacja rozpoczyna się od nowa (należy przypomnieć, że pobierane są ostatnie zmiany, a nie cała baza). Podobnie w przypadku awarii serwera wewnętrznego - serwer zewnętrzny łączy się z kolejnym węzłem z listy i z nim rozpoczyna synchronizację. Jeśli poprzedni serwer będzie ponownie aktywny (ale serwer zewnętrzny zdąży rozpocząć synchronizację z innym węzłem) to informacje przez niego wysłane są ignorowane. Mimo narzutu związanego z duplikacją wysyłanych danych jest to najbezpieczniejsze rozwiązanie w związku z tym, że dane mogą być już nieaktualne.

1.4. Wymagania i założenia

1.4.1. Wymagania funkcjonalne

ID	Wymagania	Powody decyzji
WF1	**Plik konfiguracyjny** Każdy z węzłów danej warstwy będzie posiadała plik konfiguracyjny o tej samej strukturze. W zależności od typu dane z pliku zostaną odpowiednio skonfigurowane. Plik będzie zawierał: ID węzła, listę serwerów (zależnie od typu - zewnętrznych lub wewnętrznych), dodatkowe dane konfiguracyjne (np. odpowiadających za redundancję danych, okresową synchronizację, itd.)	Takie rozwiązanie jest wymagane w treści zadania. Ponadto ujednolica i ułatwia konfigurację projektu bez potrzeby ponownej kompilacji.
WF2	**Zmiana jednego rekordu na raz** Jeśli dany węzeł chce dodać/usunąć/zmodyfikować dane o studencie musi posiadać token, który przesyła do kolejnych serwerów na końcu otrzymując potwierdzenie, że każdy z węzłów wprowadził żądaną zmianę.	Ze względu na to, że w sieci warstwy wewnętrznej serwera istnieje jeden token jest sensowne przyjąć założenie, że na raz można wprowadzić jedną modyfikację w bazie. Ponadto pozwala to utrzymać nieduży rozmiar tokena, co sprzyja jego szybszemu przesyłaniu między węzłami. Nawet w przypadku wielu żądań (od każdego węzła jedno) powinno to działać na tyle szybko, że osoby wprowadzające dane nie odczują opóźnienia. Dodatkowo wprowadzanie pojedynczych informacji pozwala na szybsze wykrycie awarii węzła serwera oraz zaoszczędza więcej czasu, niż w przypadku wpisywania dużej ilości danych na raz, a dopiero potem sprawdzania czy węzeł działa poprawnie.

ID	Wymagania	Powody decyzji
WF3	**Okresowa synchronizacja** Synchronizacja z warstwą wewnętrzną następuje okresowo, w danych odstępach czasu, sterowanych za pomocą pliku konfiguracyjnego. Jest ona wykonywana na rządanie węzłów warstwy zewnętrznej. Synchronizacja polega na pobraniu jedynie zmienionych rekordów bazy, a nie wszystkich. Ostatnia funkcjonalność zostanie rozwiązana za pomocą, tzw. „znaczników czasu" *(ang. timestamp)*.	Synchronizacja o konkretnych porach pozwala na odpowiednie rozporządzanie pracą systemu. W przypadku rzadkiej modyfikacji danych przez warstwę wewnętrzną serwera synchronizacja może być wykonywana, np. raz na dzień. Z drugiej strony można ustawić synchronizację co 1 sekundę, co spowoduje wrażenie ciągłej synchronizacji (oczywiście nie biorąc pod uwagę potencjalnych opóźnień). Takie rozwiązanie pozwala administratorowi na odpowiednią optymalizację działania systemu. Dzięki aktualizacji jedynie konkretnych danych, a nie bazy w całości jest to stosunkowo szybkie.
WF4	**Redundancja danych (warstwa wewnętrzna serwera)** W zależności od zmiennej ustawionej w pliku konfiguracyjnym warstwa wewnętrzna może posiadać pełną redundancję bądź nie. Przyjmuje się, że minimalna liczba serwerów posiadających w pełni aktualne dane to 3.	To założenie służy spełnieniu warunków zadania. Możliwość modyfikacji poziomu redundancji pozwala dopasować system do ilości węzłów w warstwie wewnętrznej serwera i do danego zapotrzebowania.
WF5	**Udostępnianie usług** Warstwa zewnętrzna obsługuje żądania klientów realizując usługę zwracania ilości studentów zapisanych na dany przedmiot. Serwer nasłuchuje czekając na żądania, a następnie (w ramach możliwości) je spełnia.	To założenie służy spełnieniu warunków zadania.
WF6	**Obliczanie danych statystycznych** Serwery zewnętrzne, zwane też przetwarzającymi, otrzymują dane od warstwy wewnętrznej, jednak zapisują z nich jedynie dane niewrażliwe używając ich do wyliczania ilości studentów zapisanych na dany przedmiot. Do tej operacji pełne informacje o studentach nie są potrzebne.	To założenie służy spełnieniu warunków zadania.

ID	Wymagania	Powody decyzji
WF7	**Dostęp do usługi pobrania danych statystycznych** Aplikacja kliencka po połączeniu z serwerem zewnętrznym może pobrać informacje na temat ilości studentów zapisanych na dany przedmiot. Użytkownik otrzymuje jedynie nazwę przedmiotu i liczbę zapisanych studentów, bez dostępu do danych wrażliwych. Może być wysyłane jedno żądanie na raz od danego użytkownika. Nie ma funkcji agregacji wiele żądań w jedno.	Jest to dobry przykład funkcjonalności, która zabezpiecza nam dane wrażliwe przed zwyczajnym użytkownikiem, udostępniając jedynie informacje, które mogą być publiczne. Agregacja bądź wysyłanie wielu żądań na raz nie jest konieczne do sprawdzenia poprawnego działania systemu i nie zmienia w żaden sposób wizji jego funkcjonowania.

1.4.2. Wymagania niefunkcjonalne

ID	Wymaganie	Priorytet
WNF1	**Czas między awariami** Średni czas między awariami systemu powinien wynosić przynajmniej trzy miesiące, przy czym awarię definiujemy jako ciągłą niedostępność usługi przez co najmniej 10 minut bądź konieczność wykorzystania kopii zapasowej	bardzo wysoki
WNF2	**Spójność danych po awarii** Odzyskane dane nie mogą być starsze niż 6 godzin przed awarią.	bardzo wysoki
WNF4	**Ochrona danych** System powinien gwarantować ochronę danych osobowych studentów.	bardzo wysoki
WNF5	**Czas naprawy** Maksymalny czas niesprawności systemu po awarii nie może przekraczać 2 godzin.	bardzo wysoki
WNF6	**Czas odpowiedzi klientowi** Średni czas odpowiedzi klientowi na zapytanie, nie powinien trwać więcej niż 2 sekundy.	średni
WNF7	**Maksymalny napływ studentów** Maksymalny przypływ nowych studentów do rejestracji nie powinien być większy niż 10 na sekundę.	średni
WNF8	**Średni napływ studentów** Średni przypływ nowych studentów do rejestracji nie powinien być większy niż 5 na sekundę.	średni
WNF9	**Dostępność** Średni czas wyłączenia systemu w roku nie powinien przekraczać dwóch dni roboczych.	wysoki
WNF10	**Ergonomia** Użytkowanie systemu powinno być intuicyjne i przejrzyste dla użytkownika.	wysoki
WNF11	**Praca po awarii węzła** Awaria pojedyńczego węzła nie powinna powodować awarii całej sieci (przy założeniu, że zachowana zostaje minimalna ilość węzłów).	bardzo wysoki
WNF12	**Minimalna ilość węzłów do pracy warstwy wewnętrznej** Warstwa wewnętrzna serwera przechowująca dane powinna poprawnie działać przy minimalnej liczbie węzłów trzy. Jeśli liczba węzłów po awarii spadnie poniżej tej liczby, poprawne działanie nie jest gwarantowane.	bardzo wysoki
WNF13	**Minimalna ilość węzłów do pracy warstwy zewnętrznej** Warstwa zewnętrzna serwera przetwarzające dane powinna poprawnie działać jeśli działa przynajmniej jeden węzeł tej warstwy.	bardzo wysoki
WNF14	**Pomoc** System powinien posiadać dobrze przygotowaną pomoc dla użytkownika.	niski

1.4.3. Dodatkowe założenia

ID	Wymagania	Powody decyzji
ZO1	**Protokół TCP/IP** Komunikacja między warstwami (wewnętrzna serwera - zewnętrzna serwera jak i zewnętrzna serwera - klienci) odbywa się za pomocą protokołów TCP/IP, wykorzystując Google Protocol Buffer (więcej w podrozdziale 4.2).	Protokół został przez nas wybrany ze względu na elastyczność dla różnych konfiguracji serwerów, tzn. bez względu na to czy warstwy serwera będą działały w obrębie jednego węzła czy nie protokół pozwoli na poprawną komunikację. W przypadku działania obu warstw serwera na jednym komputerze czy nawet w jednym procesie wydajność będzie mniejsza (w porównaniu np. do pamięci współdzielonej), jednak ze względu na ograniczony czas projektu i mniejsze skomplikowanie zostało wybrane to rozwiązanie.
ZO2	**Proste komendy serwera** Warstwy serwera będą uruchamiane i zamykane jednokrotnym wywołaniem skryptu na danym węźle serwera z jednym argumentem: start, stop oraz status, które będą kolejno: uruchamiać usługi serwera, wyłączać usługi serwera oraz wyświetlać status serwera.	Takie rozwiązanie jest wymagane w treści zadania. Ponadto pozwoli na łatwą obsługę i testowanie części serwerowej systemu.
ZO3	**Technologia token ring (warstwa wewnętrzna serwera)** Serwery warstwy wewnętrznej połączone są ze sobą w pierścień (każdy ma z góry ustalonego poprzednika i następnika w pliku konfiguracyjnym). Jeśli dany następnik nie odpowiada - brany jest następny adres węzła z listy. W ten sposób bez względu na ilość węzłów mamy zawsze do czynienia z taką samą strukturą (w szczególności - mamy tylko jeden węzeł). Każdy z węzłów wysyła swojemu następnikowi token zawierający informacje odnośnie potencjalnych zmian.	Taka architektura warstwy wewnętrznej pozwala na bieżące sprawdzanie stanu kolejnych węzłów. Jeśli token gdzieś się „zagubi” oznacza to, że dany węzeł uległ awarii. Ponadto taka architektura pozwala na szybką i łatwą synchronizację danych.

ID	Wymagania	Powody decyzji
ZO4	**Pełna redundacja danych (warstwa zewnętrzna serwera)** Każdy z węzłów zewnętrznych ma pełne i aktualne dane. Są one synchronizowane na bieżąco.	Natychmiastowa synchronizacja danych i pełna redundancja pozwala uniknąć wielu problemów w trakcie działania serwera, między innymi: dodatkowej synchronizacji po awarii jednego z węzłów, która jest dość problematyczna i skomplikowana. Rozwiązanie to zostało zatwierdzone przez Prowadzącego.
ZO5	**Wysyłanie „heartbeatów" (warstwa zewnętrzna serwera/aplikacja kliencka)** Węzły zewnętrznej warstwy będą sprawdzały swoją dostępność za pomocą tzw. „uderzeń serca" *(ang. heartbeat)* wysyłanych w danych odstępach czasu (każdy z serwerów musi mieć ustalony ten sam interwał). Dzięki temu będą miały wiedzę na temat aktywności pozostałych serwerów. Brak „heartbeata" od danego serwera po upływie zadanego czasu będzie równoważny z jego brakiem dostępności. To samo tyczy się połączenia klient - serwer zewnętrzny.	Takie rozwiązanie pozwala na bieżąco monitorować stan poszczególnych węzłów warstwy zewnętrznej oraz ilość podłączonych klientów.
ZO6	**Brak funkcji logowania użytkownika** Klient łączy się z systemem identyfikując się jedynie swoim unikatowym ID, które wpisane jest pliku konfiguracyjnym.	Decyzja została zatwierdzona przez Prowadzącego. Wynika ona z braku konieczności stworzenia tej funkcjonalności (w związku z minimalnym wpływem na główny cel projektu) oraz na ograniczony czas i mniejszą ilość osób w grupie projektowej niż przewidywana.
ZO7	**Brak interfejsu graficznego (aplikacja kliencka)** -	Decyzja została zatwierdzona przez Prowadzącego. Wynika ona z braku konieczności stworzenia tej funkcjonalności (w związku z minimalnym wpływem na główny cel projektu) oraz na ograniczony czas i mniejszą ilość osób w grupie projektowej niż przewidywana.
ZO8	**Komunikacja synchroniczna (aplikacja kliencka)** Aplikacja kliencka będzie się komunikowała z serwerem w sposób asynchroniczny.	Ze względu na ograniczoną funkcjonalność aplikacji klienckiej, sprowadzoną jedynie do wysyłania żądań i uzyskiwania odpowiedzi, ten sposób komunikacji wydaje się najbardziej trafny. Jedyna dodatkowa funkcjonalność klienta to wysyłanie co określony interwał czasu Heartbeatów dających znać, że klient jest ciągle podłączony.

ID	Wymagania	Powody decyzji
ZO9	**Wysyłanie „heartbeatu" (aplikacja kliencka)** Pomimo początkowych założeń, że aplikacja kliencka nie będzie na bieżąco sprawdzała stanu węzła do którego jest podłączona, postanowiliśmy wprowadzić do systemu tą funkcjonalność.	Celem wprowadzenia tego mechanizmu jest przede wszystkim zrealizowanie algorytmu równomiernego rozłożenia klientów na węzłach serwera. Do tego potrzebna jest znajomość ilości aktywnych klientów, a mechanizm heartbeatów idealnie się do tego nadaje.
ZO10	**Brak interaktywności (aplikacja kliencka)** Aplikacja kliencka będzie ograniczona jedynie do wywoływania określonych funkcji, według określonego z góry scenariusza.	W związku z ograniczonym czasem i głównym celem projektu interaktywna aplikacja kliencka nie jest koniecznością. Funkcjonalność systemu można poprawnie przetestować inicjując konkretne scenariusze (między innymi symulacje awarii czy błędów systemu). Decyzja ta została zatwierdzona przez Prowadzącego.

1.5. Struktura systemu

1.5.1. Struktura tokena

```
enum TokenType {
    NONE = 0;
    UPDATE = 1;
    CHECK = 2;
    ENTRY = 3;
}

message Person {
    required string uuid = 1;
    optional string name = 2;
    optional string surname = 3;
    optional int32 birthDate = 4;
    required int64 timestamp = 5;
}

message PersonSubject{
    required string uuid = 1;
    required string UUIDPerson = 2;
    required string UUIDSubject = 3;
    required int64 timestamp = 4;
}

message Subject{
    required string uuid = 1;
    required string name = 2;
    required int64 timestamp = 3;
}

message Token{
    required TokenType tokenType = 1;
    optional int32 serverId = 2;
    repeated int32 nodeIds = 3;
```

```
33      optional int64 timespamp = 4;
34  }
```

1.5.2. Struktura wiadomości

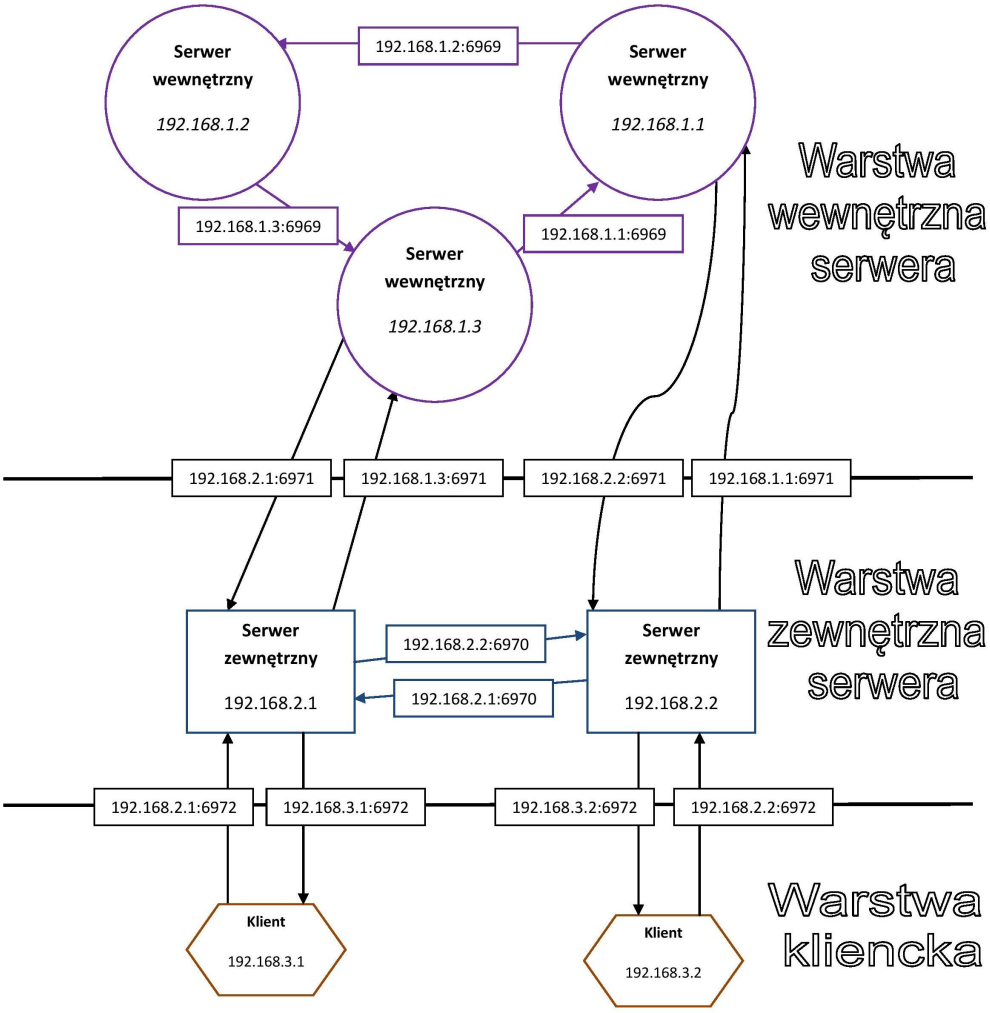
```
1   message EntityState{
2       repeated Person students = 1;
3       repeated PersonSubject personSubjects = 2;
4       repeated Subject subjects = 3;
5   }
6
7   message MiddlewareRequest{
8       required int32 nodeId= 4;
9       required int64 timestamp = 1;
10  }
11
12  message MiddlewareResponse{
13      required EntityState changes = 1;
14  }
15
16  enum MiddlewareMessageType{
17      Request = 0;
18      Response = 1;
19      Redirect = 2;
20      Heartbeat = 3;
21  }
22
23  message MiddlewareMessage{
24      required int32 nodeId= 2;
25      optional string subjectName = 3;
26      optional int32 registeredStudents = 4;
27  }
28
29  message MiddlewareHeartbeat{
30      required int32 serverId = 1;
31      optional int32 connectedClients = 2;
32      required MiddlewareMessageType messageType = 3;
33  }
34
35  message RSOMessage{
36      optional Token token = 1;
37      optional MiddlewareMessage middlewareMessage = 2;
38      optional MiddlewareRequest middlewareRequest = 3;
39      optional MiddlewareResponse middlewareResponse = 4;
40      optional MiddlewareHeartbeat middlewareHeartbeat = 5;
41  }
```

1.5.3. Zastosowanie struktur tokena i wiadomości

1.5.4. Komunikacja i adresacja

Poszczególne części systemu będą się ze sobą komunikowały na z góry określonych portach. Wysokie numery portów zostały wybrane ze względu na mniejsze prawdopodobieństwo trafienia na port używany przez inną, powszechnie używaną, aplikację (Rysunek 1.3).

Warstwa wewnętrzna (w obrębie warstwy)
Wysyłanie i odbieranie danych - na porcie 6969



Rysunek 1.2. Schemat komunikacji systemu

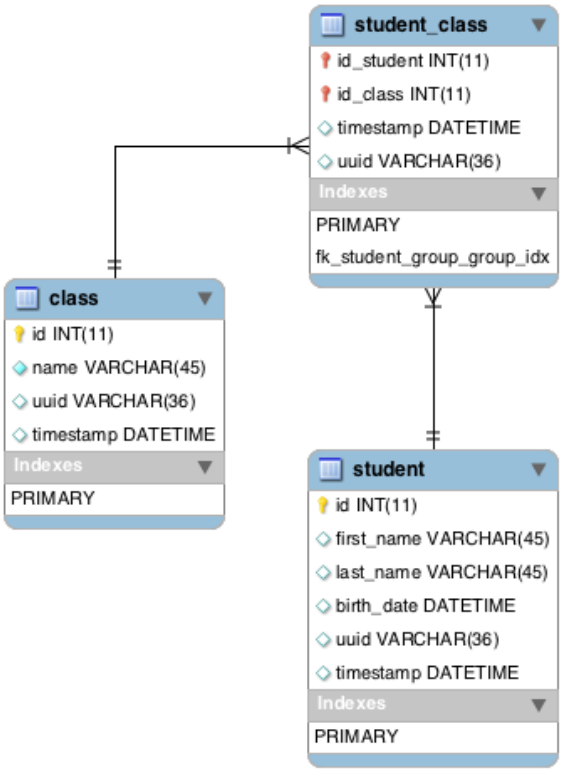
Warstwa zewnętrzna (w obrębie warstwy)
Wysyłanie i odbieranie danych - na porcie 6970

Komunikacja między warstwą zewnętrzną a wewnętrzną
Wysyłanie i odbieranie danych - na porcie 6971

Komunikacja między warstwą zewnętrzną a klientem
Wysyłanie i odbieranie danych - na porcie 6972

1.5.5. Struktura bazy danych

W trakcie uruchamiania serwer wewnętrzny, w celu poprawnego działania, musi posiadać gotową listę przedmiotów na jakie mogą być zapisani studenci. Natomiast serwer zewnętrzny nie powinien posiadać na ten temat żadnych informacji -



Rysunek 1.3. Schemat bazy danych

wszystkie je uzyska dopiero w momencie synchronizacji z serwerem wewnętrznym. Mechanizm ten zapewnia większe bezpieczeństwo danych i sprawia, że system jest mniej podatny na błędy.

1.5.6. Struktura pliku konfiguracyjnego

```
# RSO — konfiguracja

# Typ wezla.
# Mozliwe wartosci: server, middleware, client
rso.type = middleware

# Lista adresow IP wezlow middleware
# Wartosci oddzielone przecinkami
rso.addresses.middleware = 192.168.1.25, 8.8.8.8, 192.168.1.1

# Lista adresow IP wezlow servera
# Wartosci oddzielone przecinkami
rso.addresses.server = 192.168.1.108, 192.168.1.108

# Port na ktorym dziala serwer wewnetrzny/zewnetrzny
# Wewnetrzny: 6969; Zewnetrzny :6970
rso.port.external = 6969

# Port na ktorym dziala serwer wewnetrzny/wewnetrzny: 6971;
rso.port.internal = 6971

# Port na ktorym laczy sie klient: 6972

```

```
24 rso.port.client = 6972
25
26 # Baza (nie dotyczy klienta)
27
28 jdbc.driverClassName=com.mysql.jdbc.Driver
29 jdbc.url=jdbc:mysql://localhost:3306/RSO
30 jdbc.username=root
31 jdbc.password=
32
33 init-db=false
34
35 # Czas timeoutu (w sekundach)
36 # Wewntrzny - oczekiwanie na token; Zewntrzny - oczekiwanie na Heartbeat
37 timeout = 30
```

1.6. Potencjalne problematyczne scenariusze

— **Podłączenie nowego serwera warstwy wewnętrznej**

1. Wysłanie wiadomości logującej do następnika z listy serwerów zewnętrznych (i oczekiwanie 5 sekund na odpowiedź), a następnie, w przypadku braku odpowiedzi, wysłanie wiadomości do kolejnego serwera z listy
2. Jeśli serwer odpowie:
   a) Serwer, któremu węzeł wysłał wiadomość, wysyła przy najbliższej okazji aktualizacje topologii aktualnej sieci, uwzględniając najnowszy węzeł
   b) Nowy węzeł staje się częścią sieci
3. Jeśli żaden serwer nie odpowie węzeł uznaje, że jest pierwszy i działa (do czasu pojawienia się nowego węzła) w pojedynkę
4. Jeśli dwa serwery na raz będą chciały się ze sobą połączyć (tzn. wyślą do siebie wiadomość logującą) to priorytet w tworzeniu sieci ma węzeł o niższym numerze porządkowym znajdującym się w pliku konfiguracyjnym (każdy z serwerów ma identyczną listę serwerów)

— **Podłączenie nowego serwera warstwy zewnętrznej i sprawdzanie aktywności pozostałych węzłów**

1. Próba połączenia z pozostałymi serwerami zewnętrznymi.
2. W razie udanego połączenia - serwer uznany za aktywny.
3. Od tej pory do każdego aktywnego serwera co określony interwał czasu wysyłany jest Heartbeat (chyba, że została wysłana jakakolwiek inna wiadomość).
4. Analogicznie co ten sam interwał serwer oczekuje jakiejkolwiek wiadomości od aktywnych węzłów. Jeśli ich nie otrzyma - wysyła wiadomość TestRequest, żeby wymusić wysłanie Heartbeata od danego serwera. Jeśli serwer odpowie - nadal uznawany jest za aktywny - jeśli nie, uznawany jest od tego momentu za nieaktywny.

— **Awaria węzłów warstwy wewnętrznej serwera**

— Wszystkie operacje zapisujemy na dysku (oczekujące oraz wykonywane) w pliku, lokalnie dla każdego węzła. Dzięki temu po ponownym podłączeniu

węzła (po chwilowej awarii), ma on stan sprzed awarii i może poprawnie kontynuować pracę. W przypadku dłuższej nieaktywności czy przy całkowitej awarii węzła, system działa zgodnie z algorytmem opisanym w sekcji „Warstwa wewnętrzna serwera".

— **Awaria węzłów warstwy zewnętrznej serwera**

Węzeł serwera może ulec awarii w sytuacji:

1. Braku wykonywania konkretnych zadań

   W takim wypadku po rozłączeniu na dłużej niż określony interwał uznany jest przez pozostałe węzły za nieaktywny. Jeśli po krótkiej chwili połączy się ponownie - zostanie uznany za aktywny i cała warstwa będzie działała poprawnie. To samo tyczy się komunikacji serwera z klientami.

   W przypadku awarii więcej niż jednego węzła na raz - algorytm nadal działa poprawnie (aż do awarii wszystkich serwerów).

2. W trakcie wykonywania zapytania klienta

   Jeśli węzeł ulegnie awarii w trakcie wykonywania zapytania po określonym interwale klient próbuje się połączyć z innym serwerem (ponawiając proces logowania), jeśli to się powiedzie - ponawia zapytanie. Algorytm ten może mieć dość długi czas realizacji w porównaniu do wykonania samego zapytania (np. jeśli kolejny serwer, zgodnie z algorytmem wyrównującym obciążenie klientami serwerów zewnętrznych, zaproponuje serwer nieaktywny przez klienta) ale zapewnia odpowiednią synchronizację i stan całego systemu.

   Jeśli serwer połączy się ponownie i wyśle odpowiedź do klienta - ten uzna ją za błąd i zgłosi to serwerowi. W tym momencie serwer ten wykreśla ze swojej listy danego klienta i dopisze do listy danego serwera (podanego w wiadomości). Następnie informacja ta zostanie spropagowana do innych serwerów (w Heartbeatach). Dzięki temu informacje lokalne na temat serwerów w warstwie będą poprawne.

   Algorytm ten działa także w przypadku awarii większej liczby serwerów, lecz ma też w związku z tym dłuższy czas realizacji.

3. W trakcie integracji z serwerem wewnętrznym

— **Kwestia równomiernego podziału klientów między serwery**

1. Wysłanie przez klienta wiadomości logowania do serwera
2. Serwer porównuje swoją ilość klientów do ilości klientów innych węzłów.
3. Jeśli serwer otrzymujący wiadomość po dodaniu kolejnego klienta NIE będzie miał przynajmniej o 2 więcej klientów od któregokolwiek innego serwera - loguje klienta, a algorytm zostaje zakończony.
4. Jeśli serwer otrzymujący wiadomość po dodaniu kolejnego klienta miałby przynajmniej o 2 więcej klientów od któregokolwiek innego serwera - wysyła klientowi informację z adresem serwera aktywnego o najmniejszej ilości podłączonych klientów. Klient ponawia logowanie do podanego serwera.

Algorytm zapewnia równomierne rozłożenie klientów pomiędzy aktywne serwery co wpływa pozytywnie na działanie i czas obsługi żądań przez system.

— **Kwestia rozmieszczenia warstw serwera w obrębie jednego komputera bądź wielu odrębnych**

Cały system jak i każda jego warstwa działa za pośrednictwem protokołu TCP/IP. Oznacza to, że poszczególne węzły różnych warstw mogą działać lokalnie, na jednym komputerze i poprawnie komunikować się ze sobą (przy założeniu poprawnych adresów). Problem pojawia się dopiero gdy mamy węzły tej samej warstwy na jednym komputerze - prowadzi to do problemu z adresowaniem (ten sam adres dla dwóch różnych węzłów). W związku z tym na jednym komputerze dopuszczalne są jedynie węzły różnych warstw (w ogólności - jeden klient, jeden serwer zewnętrzny oraz jeden serwer wewnętrzny).

Zastosowanie takiego mechanizmu nie jest najbardziej optymalne (w związku z możliwością użycia pamięci współdzielonej, dostęp do której jest szybszy niż protokół TPC/IP oraz ze względu na możliwe połączenie „krzyżowe", np. między czterema węzłami serwerów znajdujących się na dwóch komputerach), jednak zapewnia większą uniwersalność i łatwość samego rozwiązania.

1.7. Ograniczenia i możliwe problemy

— **Brak odporności na ataki typu DDOs**

System nie jest odporny na ataki typu DDOs w związku z brakiem czasu i skupieniu się na wymaganiach o wyższym priorytecie.

— **Długi czas algorytmów synchronizacji w trakcie awarii** (nawet do parudziesięciu sekund w sytuacjach kryzysowych)

Czas trwania tej synchronizacji jest efektem zastosowanych przez nas rozwiązań. Najwyższym priorytetem w trakcie projektowania i implementacji systemu było bezpieczeństwo danych. Nawet kosztem dłuższego czasu oczekiwania użytkownika na działanie systemu po awarii.

— **Po jednej instancji węzła danej warstwy na komputerze**

Ze względu na komunikację TCP/IP na jednym komputerze może być jedynie jedna instancja klienta, serwera zewnętrznego bądź serwera wewnętrznego. Wyjątkiem jest uruchamianie instancji na przykład w maszynach wirtualnych przez co mogą mieć one różne IP.

1.8. Problemy w trakcie tworzenia projektu

— **Błędy bizantyjskie**

W trakcie tworzenia i debugowania projektu często występowały trudne do znalezienia błędy, objawiające się w najmniej oczekiwanych momentach. Ciężko było je znaleźć, ze względu na to, że pojawiały się losowo i czasami (mimo ich obecności) program działał pozornie poprawnie. Część z nich została znaleziona dopiero na końcowych etapach pracy.

— **Komunikacja między trzema warstwami jednocześnie**

Z początku ciężko było zestawić połączenie między trzema warstwami (klienty, serwery zewnętrzne oraz serwery wewnętrzne), tak, żeby każdy z węzłów w danej warstwie działał stabilnie. Te próby pozwalały wykryć błędy, które wcześniej się nie pojawiały (w przypadku łączenia dwóch warstw na raz).

— **Zbyt mała ilość osób w projekcie**

Mimo wielu ograniczeń i zmniejszenia ilości założeń w projekcie brakowało w grupie jednej, dwóch dodatkowych osób. Było to choćby związane z kwestią samego pisania kodu źródłowego (który pisały tylko trzy osoby). Tym sposobem odpowiedzialność za jedną warstwę spadała w całości na jedną osobę. Mimo późniejszej wspólnej pracy nad całością do końca doskwierał brak osób.

1.9. Organizacja środowiska programistycznego

1.9.1. Język programowania Java

Zdecydowaliśmy się napisać kod źródłowy systemu w Javie. Wybór ten wynika z wielu zalet tego języka programowania. Funkcje sieciowe są dostępne w podstawowych bibliotekach. Wliczone w to są TCP/IP, UDP/IP, wydajne nieblokujące I/O, HTTP, wsparcie REST, XML, JSON oraz SSL.

Kolejną zaletą jest duży zbiór struktur danych (również w podstawowych bibliotekach). Budowa rozproszonego systemu wymaga wielu, często skomplikowanych i różnorodnych, struktur, które Java posiada. Ważna jest tu także kwestia wątków i synchronizacji między nimi. Na tym polu Java również dostarcza odpowiednie funkcje, struktury danych oraz narzędzie realizujące współdzielony dostęp. Ponadto bez problemu współpracuje z Google Protocol Buffers i jest multiplatformowa.

1.9.2. IntelliJ IDE

Jest to środowisko programistyczne dla Javy. Owe środowisko wyposażono w bardzo bogatą paletę narzędzi pozwalających na komfortowe tworzenie oraz edycję kodu. Mnogość narzędzi oraz znajomość środowiska przez zespół ostatecznie zaważyła na decyzji o wykorzystaniu IntelliJ.

2. Testy

2.1. Plan testów

Testy mają za zadanie sprawdzenie poprawności działania systemu, nie tylko w stanie ustabilizowanym, ale także w przypadkach awarii. Pomagają ustalić czy system zachowuje się zgodnie z przewidywaniami w konkretnych sytuacjach i przy danych scenariuszach. Test uznawany jest za zaliczony w momencie zachowania się systemu (w danej symulowanej sytuacji) w przewidywany przez nas sposób.

2.1.1. Test protokołu

1. Sprawdzenie poprawności przesyłanej ramki danych z węzła wewnętrznego do zewnętrznego oraz z zewnętrznego do wewnętrznego.
2. Sprawdzenie poprawności przesyłanej ramki danych z węzła zewnętrznego do klienta oraz od klienta do węzła zewnętrznego.

Testy obejmują sprawdzenie poprawności serializacji oraz deserializacji danych przesyłanych między elementami systemu. Sprawdzone również zostanie połączenie między tymi elementami.

1. zbudowanie wiadomości z danymi z serwera wewnętrznego/zewnętrznego do serwera zewnętrznego/wewnętrznego RSOMessage.
2. wysłanie tablicy bajtów do serwera zewnętrznego
3. sprawdzenie poprawności przesłanych danych

2.1.2. Test bazy danych

1. Sprawdzenie połączenia z bazą danych.
2. Sprawdzenie poprawności działania algorytmu aktualizacji danych systemu.
   Test obejmuje sprawdzenie algorytmu wykorzystującego Token Ring do aktualizacji danych. Weryfikacji zostanie podany również konfigurowalny czas posiadania tokena przez węzeł.
3. Sprawdzenie redundancji danych.

2.1.3. Test przetwarzania danych

Sprawdzenie czy węzeł zewnętrzny poprawnie przetwarza dane wrażliwe na dane gotowe do wysłania klientowi. Wysyłane dane powinny zawierać numer identyfikacyjny oraz listę przedmiotów.

2.1.4. Test algorytmu Heartbeatów

Sprawdzenie czy węzły zewnętrzne poprawnie komunikują się między sobą oraz czy poprawnie interpretowane są braki komunikacji.

Sprawdzenie działania algorytmu dla interwału czasowego równego: 5, 30 oraz 60 sekund:

1. dodanie trzech węzłów zewnętrznych
2. wysłanie wiadomości Heartbeat
3. oczekiwanie całego interwału na Heartbeat zwrotny
4. (Przypadek 1) Heartbeat przyszedł, zerowanie timera
5. (Przypadek 2) Heartbeat nie przyszedł, wysłanie TestRequesta
6. (Przypadek 2.1) Heartbeat przyszedł, zerowanie timera
7. (Przypadek 2.2) Heartbeat nie przyszedł, uznanie węzła za nieaktywny

2.1.5. Testy połączeniowe węzłów

Służą sprawdzeniu czy w przypadkach awaryjnych system zachowa się przewidywalnie i zastosuje odpowiednie operacje służące zachowaniu stanu ustalonego oraz odpowiedniej synchronizacji.

1. Test podłączenia węzła wewnętrznego
   a) Brak innych węzłów
   b) Dołączanie do istniejącej struktury pierścienia
2. Test podłączenia węzła zewnętrznego
   a) Brak innych węzłów
   b) Dołączanie do istniejącej struktury pierścienia
3. Test awarii węzła wewnętrznego
   a) Węzeł rozłącza się permanentnie
   b) Węzeł rozłącza się, a po chwili podłącza ponownie
   c) Awaria więcej niż jednego węzła na raz
4. Test awarii węzła zewnętrznego
   a) Węzeł rozłącza się permanentnie
      i. Węzeł bezczynny
      ii. Węzeł w trakcie synchronizacji
      iii. Węzeł w trakcie obsługi żądania
   b) Węzeł rozłącza się, a po chwili podłącza ponownie
      i. Węzeł bezczynny
      ii. Węzeł w trakcie synchronizacji
      iii. Węzeł w trakcie obsługi żądania

c) Awaria więcej niż jednego węzła na raz

2.1.6. Testy bezpieczeństwa

Sprawdzają czy system posiada dostatecznie wysoki poziom bezpieczeństwa przy danych scenariuszach.

1. Test podłączenia „obcego" węzła zewnętrznego
   a) całkowicie nowy węzeł
   b) węzeł „podszywający" się pod inny, rozłączony
2. Test podłączenia obcego węzła wewnętrznego
   a) całkowicie nowy węzeł
   b) węzeł „podszywający" się pod inny, rozłączony

3. Organizacja projektu

3.1. Podział zadań

Kierownik projektu, Dokumentalista - Domagała Bartosz

Odpowiedzialny za planowanie, realizację oraz zamykanie projektu. Ma zapewnić osiągnięcie założonych celów projektu i wytworzenie oprogramowania spełniającego określone wymagania jakościowe. Ponadto odpowiedzialny za całą dokumentację projektu, nadzorujący jej tworzenie i ostateczną formę.

Handlowiec, Specjalista ds. Docker'a, Programista - Kornata Jarosław

Odpowiedzialny za prezentację projektu przed prowadzącym. Ponadto główny specjalista rozwiązania Docker, posiadający na ten temat największą wiedzę, przekazywaną w trakcie projektu pozostałym osobom. Bierze czynny udział w tworzeniu kodu źródłowego programu oraz sumiennie dokumentuje swoją pracę.

Tester, Programista - Jędrzej Modzelewski

Odpowiedzialny za wszelkie testy związane z oprogramowaniem - projektujący je, programujący oraz egzekwujący. Bierze czynny udział w tworzeniu kodu źródłowego programu oraz sumiennie dokumentuje swoją pracę.

Architekt, Osoba odp. za repozytorium, Programista - Stepnowski Marcin

Odpowiedzialny za stworzenie i dbanie o odpowiednie wdrożenie architektury projektu oraz dba o zgodność tworzonych rozwiązań informatycznych z obowiązującymi standardami, wzorcami i strategią. Ponadto odpowiedzialny za założenie, zarządzanie oraz tworzenie kopii zapasowych repozytorium projektu. Bierze czynny udział w tworzeniu kodu źródłowego programu oraz sumiennie dokumentuje swoją pracę.

3.2. Planowany harmonogram pracy

Spotkanie I - koniec marca

— Spotkanie organizacyjne

— Ustalenie ról w projekcie i podział zadań

— Rozmowa na temat wizji projektu

— Umówienie się co do organizacji pracy i spotkań projektowych

Spotkanie II - 1 tydzień kwietnia

— Nauka i zrozumienie zasad działania Docker'a

— Wspólne sprawdzenie testowych konfiguracji Dockera

— Ustalenie wstępnej architektury systemu

— Ustalenie wymagań projektu

— Ustalenie języka w jakim projekt ma być tworzony

— Rozmowa na temat dodatkowych narzędzi do realizacji projektu

Spotkanie III - 2 tydzień kwietnia

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Rozważanie na temat potencjalnych wad systemu i możliwych sytuacji awaryjnych

— Ostateczna definicja wymagań funkcjonalnych i niefunkcjonalnych programu

Spotkanie IV - 3 tydzień kwietnia

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Upewnienie się o poprawności dotychczasowych założeń, wymagań

— Wprowadzenie poprawek do dokumentacji

— Opracowanie ostatecznej dokumentacji na prezentację etapu 1

Prezentacja etau I - 23 kwietnia

Spotkanie VI - 4 tydzień kwietnia

— Wprowadzenie poprawek i sugestii Prowadzącego po prezentacji etapu I

— Podział pracy przy pisaniu kodu źródłowego programu

— Ustalenie i doprecyzowanie funkcjonalności na prezentację etapu II

— Rozpoczęcie pisania kodu źródłowego

— Dyskusja na temat testów systemu i ich definicja

— Szczegółowe doprecyzowanie rozwiązania

Spotkanie VII - 4 tydzień kwietnia

— Wykonanie czynności i kwestii, które nie zostały zakończone bądź poruszone podczas spotkania VI

— Praca nad kodem źródłowym programu i poprawa błędów

— Testy oprogramowania, przygotowanie do prezentacji funkcjonalności

Spotkanie VIII - 1 tydzień maja

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Zdefiniowanie pełnego planu testów i upewnienie się o jego poprawności

— Ostateczne poprawki do dokumentacji dla etapu II

— Ostateczne przygotowanie prezentacji funkcjonalności systemu dla etapu II

Prezentacja etau II - 7 maja

Spotkanie IX - 2 tydzień maja

— Wprowadzenie poprawek i sugestii Prowadzącego po prezentacji etapu II

— Praca nad kodem źródłowym programu i poprawa błędów

— Ustalenie pozostałych zadań do zrobienia

Spotkanie X - 3 tydzień maja

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Dyskusja na temat dalszego planu tworzenia kodu źródłowego systemu

— Praca nad kodem źródłowym programu i poprawa błędów

Spotkanie XI - 4 tydzień maja

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Praca nad kodem źródłowym programu i poprawa błędów

— Opracowanie wstępnej prezentacji końcowej

Spotkanie XII - 1 tydzień czerwca

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Praca nad kodem źródłowym programu i poprawa błędów

— Poprawki do prezentacji końcowej

Spotkanie XIII - 1 tydzień czerwca

— Spotkanie awaryjne, pozwalajace na wykonanie zaległych czy niedokończonych zadań z poprzednich spotkań

Spotkanie XIV - 2 tydzień czerwca

— Sprawdzenie i podsumowanie dotychczasowej pracy

— Ostateczne poprawki do dokumentacji dla etapu III

— Ostateczne przygotowanie prezentacji funkcjonalności systemu dla etapu III

Spotkanie XV - 2 tydzień czerwca

— Spotkanie awaryjne, pozwalajace na wykonanie zaległych czy niedokończonych zadań z poprzednich spotkań

Prezentacja etapu III - 11 czerwca

Spotkanie XVI - 11 czerwca

— Świętowanie zaliczenia projektu przy piwie

3.3. Sprawozdania ze spotkań

Sprawozdania z konkretnych spotkań zawierają listę ustaleń oraz wykonanych czynności w związku z projektem. Poza spotkaniami, każdy z uczestników projektu pracował nad przydzielonymi zadaniami indywidualnie, we własnym zakresie, trzymając się konkretnych terminów.

3.3.1. Spotkanie I - 31 marca 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Spotkanie organizacyjne, którego celem był podział ról w projekcie oraz przydzielenie zadań, a także wstępne omówienie samej wizji projektu.

Szczegółowy opis wykonanej pracy

— Kierownik projektu ustalił podział ról zgodnie z umiejętnościami i preferencjami osób z grupy projektowej:
  — Domagała Bartosz - Kierownik, Dokumentalista
  — Kornata Jarosław - Handlowiec, Specjalista ds. Docker'a, Programista
  — Modzelewski Jędrzej - Tester, Programista
  — Stepnowski Marcin - Architekt, Osoba odpowiedzialna za repozytorium, Programista

— Ustalony został termin cotygodniowych spotkań projektowych

— Omówiona została treść zadania, upewniono się, że każdy z uczestników projektu je rozumie

— Prowadzone były rozmowy na temat wizji projektu każdego z uczestników, dobre pomysły spisywane i komentowane

— Zastanowiono się nad sprecyzowaniem treści zadania

— Każdy z uczestników projektu otrzymał zadania do wykonania:
  — Domagała Bartosz - stworzenie szkieletu dokumentacji za pomocą LateXa, zapisanie sprawozdania z odbytego spotkania
  — Kornata Jarosław - dowiedzenie się jak najwięcej o rozwiązaniu Docker
  — Modzelewski Jędrzej - pomyślenie o narzędziach, bibliotekach i językach programowania jakich można użyć w projekcie
  — Stepnowski Marcin - stworzenie repozytorium Git oraz zastanowienie się nad architekturą projektu

3.3.2. Spotkanie II - 8 kwietnia 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Spotkanie podsumowujące ostatnio wykonane zadania oraz występujące problemy. Ponadto podczas niego ustalono wstępną treść zadania i omówiono, a także przetestowano rozwiązanie Docker.

Szczegółowy opis wykonanej pracy

— Przekazanie przez specjalistę ds. Dockera informacji o tym rozwiązaniu pozostałym uczestnikom

— Wspólne sprawdzenie testowych konfiguracji Dockera

— Wstępny projekt architektury projektu zaproponowany przez Architekta

— Zastanowienie się nad wymaganiami niefunkcjonalnymi projektu

— Ustalenie języka w jakim stworzony zostanie projekt - Java

— Rozmowa na temat przydatnych i potrzebnych bibliotek w projekcie

— Każdy z uczestników projektu otrzymał zadania do wykonania:

  — Domagała Bartosz - zapis sprawozdania ze spotkania

  — Kornata Jarosław - zapis informacji o rozwiązaniu Docker oraz opis przeprowadzonych testów jego konfiguracji

  — Modzelewski Jędrzej - zapis informacji o platformie, języku i bibliotekach używanych w projekcie

  — Stepnowski Marcin - zapis informacji o projekcie architektury

3.3.3. Spotkanie X1 - 14 kwietnia 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *nieobecny*

— Modzelewski Jędrzej - *nieobecny*

— Stepnowski Marcin - *obecny*

Pierwsze spotkanie z Prowadzącym projekt. Informacje ze spotkania:

— Zatwierdzenie przez Prowadzącego tematu

— Zgoda Prowadzącego na brak funkcji logowania w aplikacji klienckiej, ze względu na główny cel projektu, ograniczony czas i mniejszą liczbę osób w grupie niż przewidziana

— Zgoda Prowadzącego na brak graficznego interfejsu użytkownika, ze względu na główny cel projektu, ograniczony czas i mniejszą liczbę osób w grupie niż przewidziana

— Zgoda Prowadzącego na brak interaktywnej aplikacji klienckiej, ze względu na główny cel projektu, ograniczony czas i mniejszą liczbę osób w grupie niż przewidziana

— Doprecyzowanie szczegółów w związku z 3-warstwową naturą systemu

— Uzyskanie odpowiedzi na pytania odnośnie architektury

3.3.4. Spotkanie III - 14 kwietnia 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Spotkanie podsumowujące ostatnio wykonane zadania oraz występujące problemy. Zastanowiono się podczas niego nad dotychczasowym wyborem rozwiązań, sprecyzowano treść zadania, ustalono wymagania funkcjonalne i niefunkcjonalne projektu.

Szczegółowy opis wykonanej pracy

— Dyskusja na temat założeń systemu i jego wstępnej architektury

— Wprowadzenie poprawek do projektu na podstawie wniosków z dyskusji

— Wspólne ustalenie dodatkowych założeń, przesłanek do ich egzekwowania i konsekwencji z tym związanych

— Definicja wymagań funkcjonalnych i niefunkcjonalnych programu

— Rozmowa na temat wad systemu, potencjalnych zagrożeń, spisanie pomysłów

— Każdy z uczestników projektu otrzymał zadania do wykonania:

  — Domagała Bartosz - zapis sprawozdania ze spotkania, poprawienie błędów w dokumentacji oraz edycja szkieletu stylu, upewnienie się, że naniesione przez resztę grupy poprawki są prawidłowe

  — Kornata Jarosław - wprowadzenie poprawek do opisu przykładowej konfiguracji Dockera

  — Modzelewski Jędrzej - wprowadzenie poprawek do opisu języka oraz IDE w jakim tworzony będzie projekt

  — Stepnowski Marcin - rozwinięcie i wyjaśnienie decyzji projektowych dotyczących architektury podjętych w trakcie spotkania oraz zapisanie ich w dokumentacji

3.3.5. Spotkanie X2 - 21 kwietnia 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *nieobecny*

— Modzelewski Jędrzej - *nieobecny*

— Stepnowski Marcin - *obecny*

Drugie spotkanie z Prowadzącym projekt. Informacje ze spotkania:

— Zgoda Prowadzącego na pełną redundancję w warstwie wewnętrznej serwera (tzn. każdy z węzłów ma posiadać wszystkie aktualne dane) w związku z ograniczonym czasem i mniejszą liczbą osób w grupie projektowej niż przewidziana

— Zdobycie informacji odnośnie warstw serwera, które mogą być na tym samym węźle jak i na dwóch niezależnych węzłach

— Zgoda Prowadzącego na użycie protokołu TCP/IP do komunikacji między warstwami serwera bez względu na ich lokalizację - w związku z ograniczonym czasem i mniejszym skomplikowaniem zagadnienia

3.3.6. Spotkanie IV (TeamSpeak) - 22 kwietnia 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *nieobecny*

— Stepnowski Marcin - *obecny*

Spotkanie za pośrednictwem programu TeamSpeak mające na celu podsumowanie dotychczasowej pracy oraz wprowadzenie ostatecznych poprawek do dokumentacji przed prezentacją etapu I projektu.

Szczegółowy opis wykonanej pracy

— Poprawienie błędów w dokumentacji

— Dopisanie brakujących rzeczy do dokumentacji (głównie - kwestii potencjalnych problemów z systemem)

— Ostateczna edycja i poprawki szablonu projektu

— Podsumowanie dotychczasowej pracy, umówienie się na kolejne spotkanie w celu zaczęcia pracy nad kodem źródłowym systemu

— Rozmowa i ustalenia na temat prezentacji etapu I

3.3.7. Spotkanie V - 27 kwietnia 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny (TeamSpeak)*

Pierwsze spotkanie po oddaniu etapu I projektu. Omówienie niezbędnych poprawek do dotychczasowej dokumentacji oraz opracowanie planu dalszej pracy związanej z kodem źródłowym programu i prezentacją.

Szczegółowy opis wykonanej pracy

— Poprawki w dokumentacji związane z sugestiami od Prowadzącego

— Opracowanie wstępnego planu testów

— Podział obowiązków przy pisaniu kodu źródłowego programu:

  - Kornata Jarosław - struktura wiadomości przesyłanych między warstwami (oraz klasy pomocnicze); obiekty i funkcje liczące dane statystyczne w warstwie zewnętrznej; klasy do komunikacji z bazą danych, program do generowania danych wrażliwych
  - Modzelewski Jędrzej - komunikacja wewnętrzna serwerów wewnętrznych, komunikacja między warstwami
  - Stepnowski Marcin - komunikacja wewnętrzna serwerów zewnętrznych, komunikacja między warstwami
- Opracowanie wstępnego scenariusza prezentacji etapu II projektu
- Ustalenie zastosowania wzorca czynnościowego wzorca projektowego przy komunikacji między warstwami zwanego łańcuchem zobowiązań

3.3.8. Spotkanie X3 - 28 kwietnia 2015

- Domagała Bartosz - *obecny*
- Kornata Jarosław - *nieobecny*
- Modzelewski Jędrzej - *nieobecny*
- Stepnowski Marcin - *obecny*

Trzecie spotkanie z Prowadzącym projekt. Informacje ze spotkania:

- Pytanie odnośnie prezentacji funkcjonalności podczas oddawania II etapu - zgoda Prowadzącego na pokazanie działającego systemu, ale bez zadbania o kwestie awarii systemu czy redundancję
- Pytanie odnośnie struktury testów i potwierdzenie przez Prowadzącego, że testy nie muszą być automatyczne, ale możliwe do ponownej symulacji
- Rozmowa na temat token ringu i redundancji danych w warstwie wewnętrznej

3.3.9. Spotkanie VI (TeamSpeak) - 28 kwietnia 2015

- Domagała Bartosz - *obecny*
- Kornata Jarosław - *obecny*
- Modzelewski Jędrzej - *obecny*
- Stepnowski Marcin - *obecny*

Spotkanie za pośrednictwem programu TeamSpeak mające na celu ustalenie struktury wiadomości jakie będą służyły do komunikacji w systemie oraz dalszy podział pracy.

Szczegółowy opis wykonanej pracy

- Ustalenie struktury wiadomości do komunikacji wewnątrz warstw i pomiędzy między warstwami systemu
- Skonfigurowanie środowiska projektowego i stworzenie szkieletu aplikacji

— Dyskuje oraz ustalenia odnośnie wspólnej pracy nad kodem źródłowym systemu

3.3.10. Spotkanie VII (TeamSpeak) - 4 maja 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *nieobecny*

— Stepnowski Marcin - *obecny*

Praca nad kodem programu oraz dokumentacją. Przygotowywanie projektu na prezentację etapu II.

Szczegółowy opis wykonanej pracy

— Praca programistów nad kodem, ustalenia dotyczące struktury klas i potrzebnych metod do komunikacji między warstwami

— Ustalenie portów na jakich komunikować się będą poszczególne części systemu

— Uzupełnienie dokumentacji o brakujące informacje

3.3.11. Spotkanie VIII - 5 maja 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Praca nad kodem programu oraz dokumentacją. Uszczegółowienie architektury w dokumentacji.

Szczegółowy opis wykonanej pracy

— Praca programistów nad kodem, praktyczne próby stworzenia komunikacji między warstwami

— Opracowanie API do bazy danych

— Stworzenie komunikacji wewnątrz warstw

— Uszczegółowienie opisu i architektury systemu w dokumentacji

— Stworzenie planu prezentacji etapu II

— Stworzenie wstępnego planu prezentacji etapu III

3.3.12. Spotkanie IX - 6 maja 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Końcowe prace nad kodem programu oraz dokumentacją przed prezentacją etapu II.

Szczegółowy opis wykonanej pracy

— Praca programistów nad kodem, końcowe testy prezentacji etapu II

— Dodanie informacji o ochronie danych osobowych

— Skończenie opisu pełnego planu testów

— Przeczytanie całej dotychczasowej dokumentacji i upewnienie się o jej poprawności

— Końcowe ustalenia na temat konkretnej prezentacji etapu II

3.3.13. Spotkanie X - 12 maja 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Podsumowanie po prezentacji 2 etapu projektu oraz podział dalszej pracy.

Szczegółowy opis wykonanej pracy

— Omówienie potencjalnych błędów w dotychczasowym kodzie projektu

— Podział pracy do końca przyszłego tygodnia

  — Domagała Bartosz - *Zapis w dokumentacji końcowego planu prezentacji i opisu sytuacji krytycznych w projekcie*

  — Kornata Jarosław - *Wykorzystanie Dockera przy właściwym projekcje*

  — Modzelewski Jędrzej - *Wewnętrzna komunikacja między węzłami warstwy zewnętrznej*

  — Stepnowski Marcin - *Wewnętrzna komunikacja między węzłami warstwy wewnętrznej*

— Omówienie wskazówek Prowadzącego z prezentacji etapu II

3.3.14. Spotkanie XI - 19 maja 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Praca nad kodem programu oraz dokumentacją. Poprawa błędów i sprawdzenie poprawności kodu źródłowego.

Szczegółowy opis wykonanej pracy

— Poprawienie błędów powodujących błędy połączenia między węzłami różnych warstw

— Poprawienie działania klasy TaskManagera

— Omówienie dalszego planu działania

— Upewnienie się o poprawności napisanego do tej pory kodu

— Praca programistów nad kodem

— Rozmowy i ustalenia na temat synchronizacji i sytuacji awaryjnych

3.3.15. Spotkanie XII - 28 maja 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Praca programistów nad kodem. Dopisanie szczegółów sytuacji awaryjnych do projektu i sposobów ich rozwiązywania. Testowanie kodu. Konfiguracja Dockera.

Szczegółowy opis wykonanej pracy

— Konfiguracja Dockera w celu utworzenia przenośnych kopii systemu

— Praca nad kodem źródłowym warstwy zewnętrznej i wewnętrznej serwera

— Praca nad komunikacją wewnętrzną między węzłami w danej warstwie

3.3.16. Spotkanie XIII - 5 czerwca 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *nieobecny*

Testy Dockera oraz przygotowanie i implementacja rozwiązań sytuacji wyjątkowych.

Szczegółowy opis wykonanej pracy

— Testy działania Dockera

— Praca nad kodem źródłowym warstwy zewnętrznej i wewnętrznej serwera

— Przygotowanie sytuacji awaryjnych do testowania

3.3.17. Spotkanie XIV - 9 czerwca 2015

— Domagała Bartosz - *nieobecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Ostateczne testy, poprawki w implementacji oraz ćwiczenie prezentacji.

Szczegółowy opis wykonanej pracy

— Sprawdzenie całej dokumentacji, wprowadzenie poprawek

— Testowanie sytuacji wyjątkowych

— Rozmowy i rozważania na temat prezentacji etapu III

— Testowanie implementacji, ostateczne poprawki błędów

— Sprawdzanie założeń z dokumentacji ze stanem faktycznym projektu

3.3.18. Spotkanie XV - 10 czerwca 2015

— Domagała Bartosz - *obecny*

— Kornata Jarosław - *obecny*

— Modzelewski Jędrzej - *obecny*

— Stepnowski Marcin - *obecny*

Ostatnie spotkanie skupione głównie na prezentacji projektu.

Szczegółowy opis wykonanej pracy

— Przygotowanie materiałów do prezentacji

— Przećwiczenie prezentacji

— Omówienie całokształtu pracy nad projektem

3.4. Podsumowanie pracy nad projektem

Odbyło się tyle samo spotkań co we wstępnym planie, z tą różnicą, że w nieznacznie innych terminach. Prace i zadania wykonywane w trakcie spotkań nieznacznie się różnią od założonych, co jest związane z trudnością w przewidywaniu konkretnych problemów, opóźnień czy błędów, które wystąpiły przy tworzeniu projektu.

Implementacja projektu w ostatecznej formie na dzień jego oddania nie jest idealnym odzwierciedleniem systemu zawartego w dokumentacji. Choć kod źródłowy był tworzony zgodnie z samą dokumentacją, krok po kroku, wystąpiło po drodze wiele problemów (nie związanych z samą architekturą), które ostatecznie uniemożliwiły w ograniczonym czasie i przy ograniczonych zasobach stworzyć sprawnie działający

projekt. W związku z tym zrezygnowaliśmy z części funkcjonalności na rzecz najważniejszych, naszym zdaniem, zagadnień ukazujących meritum tworzenia aplikacji rozproszonych.

3.5. Plan prezentacji etapu II

Prezentacja zostanie przeprowadzona na osobistych komputerach uczestników projektu. Na jednym komputerze zostanie uruchomiona instancja warstwy zewnętrznej, a na drugim wewnętrznej. Na trzecim natomiast zostanie uruchomiona aplikacja kliencka.

— zostanie pokazana komunikacja między poszczególnymi warstwami przy ustalonym scenariuszu

— zostanie pokazana podstawowa funkcjonalność programu - tzn. wysłanie żądania od klienta do serwera i zwrot przetworzonych danych

— aplikacja kliencka będzie miała minimalną funkcjonalność - tzn. z góry wpisane konkretne metody

— system nie będzie odporny na awarie, np. odłączanie danych węzłów

— system nie będzie obsługiwał w pełni dodawania nowych węzłów oraz kwestii utrzymywania połączenia (np. mechanizmu Heartbeatów)

3.6. Wstępny plan prezentacji etapu III

Prezentacja zostanie przeprowadzona na osobistych komputerach uczestników projektu. Na trzech komputerach zostaną uruchomione instancje warstwy zewnętrznej i wewnętrznej (na każdym po jednej z nich). Dwa serwery zewnętrzne będą się łączyły z serwerami wewnętrznymi na INNYCH komputerach. Jeden będzie łączył się lokalnie. Aplikacje klienckie zostaną uruchomione na każdym z komputerów oraz na dodatkowym - czwartym.

Przebieg prezentacji będzie automatyczny, program przy każdym kolejnym kroku będzie się zatrzymywał, aby pokazać konkretne wyniki.

3.6.1. Zaprezentowane scenariusze

1. Podłączenie wielu klientów i wykorzystanie algorytmu dzielenia obciążenia na wiele serwerów zewnętrznych
2. Pokazanie obsługi trzech żądań od różnych klientów
3. Działania zmiennej redundancji węzłów wewnętrznych serwera
4. Symulacja awarii serwera zewnętrznego w trakcie wykonywania konkretnego działania
5. Symulacja awarii serwera wewnętrznego w trakcie wykonywania konkretnego działania
6. Symulacja awarii serwera zewnętrznego w trakcie wykonywania konkretnego działania i jego ponowne podłączenie po chwili
7. Symulacja awarii serwera wewnętrznego w trakcie wykonywania konkretnego działania i jego ponowne podłączenie po chwili

3.7. Ostateczny plan prezentacji etapu III

1. Wstępne przedstawienie projektu
2. Zaprezentowanie zestawiania połączenia między warstwami (w określonym, pomyślnym i bezawaryjnym scenariuszu) przy użyciu 3 komputerów i 3 węzłów w każdej z warstw
   a) Zaprezentowanie logowania kolejnych węzłów w danej warstwie
   b) Zaprezentowanie działania algorytmu równomiernego rozłożenia klientów między serwery
3. Zaprezentowanie działającego systemu
4. Zaprezentowanie sytuacji awaryjnej
5. Podsumowanie, napotkane problemy, wnioski

4. Narzędzia i zewnętrzne biblioteki

4.1. Docker

4.1.1. Czym jest docker?

Docker jest narzędziem przeznaczonym do tworzenie przenośnych, wirtualnych kontenerów pozwalających na prostą i szybką replikację środowiska.

Oprogramowanie Docker wprowadza standaryzację w środowisko uruchomieniowe. Generowane kontenery są spójne i takie same w różnych środowiskach. W chwili obecnej Docker wymaga jądra Linux do uruchomienia, jednak działa nie tylko w wersji natywnej, ale również poprzez wirtualizację specjalnych minimalistycznych dystrybucji Linux na systemach Mac OS X oraz Windows (na przykład w darmowym narzędziu do wirtualizacji – VirtualBox).

4.1.2. Spójność

Docker wprowadza spójność w środowisko deweloperskie. Często, kiedy nad projektem pracuje więcej niż jeden programista (co w dzisiejszych czasach jest normą) pojawia się problem z różnymi wersjami użytego oprogramowania. Zdarza się, że jedna wersja biblioteki zachowuje się inaczej od drugiej (na przykład z powodu błędu). Kontener z założenia posiada jedną, konkretną wersję każdej biblioteki koniecznej do uruchomienie aplikacji. Zwiększa to przewidywalność oprogramowania. Oczekujemy bowiem, że w tym samym środowisku, aplikacja będzie zachowywać się tak samo.

4.1.3. Cykl Dockera

1. Stworzenie kontenera wraz z wszelkimi narzędziami i bibliotekami koniecznymi do uruchomienia aplikacji.
2. Rozprowadzenie kontenera, wraz ze wszystkimi narzędziami i bibliotekami.
3. Uruchomienie identycznego kontenera na dowolnej liczbie węzłów.

Prowadzi to do znacznych ułatwień w tworzeniu oprogramowania. Ten sam kontener może zostać rozprowadzony pomiędzy deweloperami, testerami, serwerami ciągłej integracji i w końcu środowiskiem produkcyjnym.

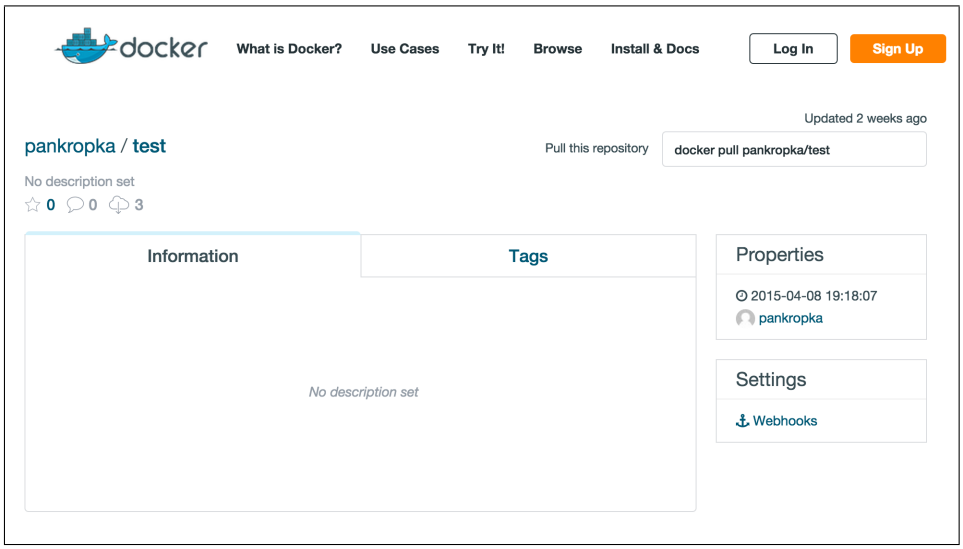
4.1.4. Centralne repozytorium

Każdy zarejestrowany użytkownik ma możliwość wgrywania własnych kontenerów do centralnego, ogólnie dostępnego repozytorium. Można tam znaleźć wiele różnych gotowych obrazów do pobrania. Są one przygotowane zarówno przez społeczność jak i przez twórców Dockera.

Centralne repozytorium pozwala użytkownikom na pobranie interesujących obrazów, szczególnie warte uwagi są repozytoria przygotowane pod konkretne rozwiązania (na przykład specjalnie pod serwer baz danych MySQL lub mongoDB).

Docker umożliwia również stworzenie prywatnych repozytoriów. Dzięki temu nie musimy upubliczniać prywatnych obrazów, ale jednocześnie możemy je rozprowadzać. Inną dostępną metodą jest zapis obrazu do pliku i przekazanie go w tradycyjny sposób.

4.1.5. Przeprowadzone testy



Rysunek 4.1. Repozytorium Docker

W ramach zapoznania się z platformą Docker wykonaliśmy następujące czynności:

— Zweryfikowaliśmy możliwość uruchomienia oprogramowania na systemach operacyjnych Ubuntu oraz Mac OS X.

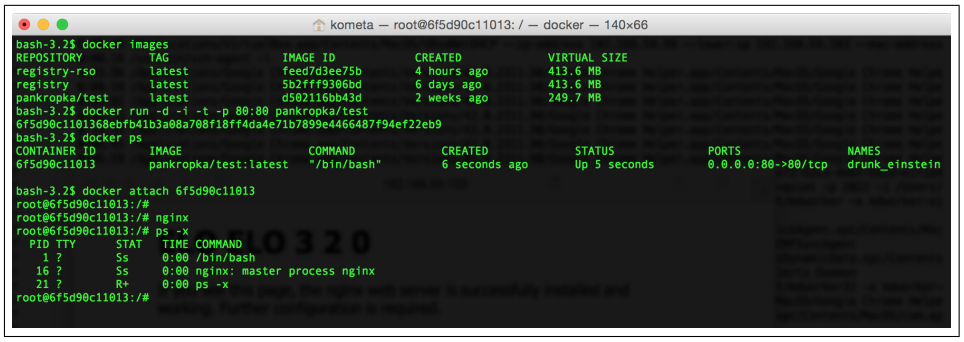
— Na systemie Max OS X skorzystaliśmy z programu Boot2Docker, tworzącego maszynę wirtualną z minimalistyczną dystrybucją Linux konieczną do uruchomienia aplikacji Docker.

— Pobraliśmy obraz Systemu operacyjnego Ubuntu, na którym zainstalowaliśmy programy nginx oraz vim.

— Przy pomocy programu vim, zmodyfikowaliśmy treść domyślnego pliku index.html, który dostępny jest na porcie 80 po uruchomieniu programu nginx.

— Zweryfikowaliśmy zmianę w przeglądarce internetowej.

Rysunek 4.2. Pobranie obrazu i weryfikacja zmian



Rysunek 4.3. Działa!

— Zapisaliśmy zmiany w kontenerze oraz wgraliśmy zmieniony obraz do centralnego repozytorium (Rysunek 4.1).

— Pobraliśmy obraz na innym komputerze i po uruchomieniu zweryfikowaliśmy zmiany wprowadzone do podstawowej dystrybucji Ubuntu (Rysunek 4.2).

— W pobranym obrazie zainstalowane były programy nginx oraz vim. Domyślny plik programu nginx był również zmodyfikowany (Rysunek 4.3).

4.1.6. Środowisko uruchomieniowe

Z racji na testowanie oprogramowania na różnych systemach operacyjnych przygotowany został obraz VirtualBox systemu Debian z zainstalowanym programem Docker.

System Debian posiada dwóch użytkowników:

— Nazwa: user hasło: 1234

— Nazwa: root hasło: 1234 (użytkownik z prawami administracyjnymi)

W przygotowanym obrazie dostępny jest skrypt powłoki Bash *rso* przyjmujący jeden z trzech parametrów:

— middleware

— server

— client

Skrypt ten ma za zadanie uruchomienie przygotowanego kontenera Docker. Zależnie od podanego argumentu dobierane są odpowiednie parametry przekierowania portów dla danego kontenera. Skrypt po dobraniu odpowiednich parametrów wywołuje następujące polecenie:

docker run -t -i $PORT $IMAGE $CMD $1

gdzie:

— $PORT to odpowiednie porty do przekierowania (na przykład *-p 6971:6971 -p 6969:6969*)

— $IMAGE to obraz do uruchomienia (*pankropka/rso:java*)

— $CMD to komenda do uruchomienia w kontenerze (*/bin/bash rso-run* uruchamiająca opisany niżej skrypt uruchomieniowy w kontenerze)

— $1 to argument wejściowy przekazywany do skryptu w kontenerze (*middleware-/server/client*, w przypadku nieznanego argumentu następuje wypisanie możliwych argumentów).

Skrypt powłoki Bash *rso-run* wywoływany w uruchomionym kontenerze ma za zadanie uruchomić serwer MySQL oraz program.

Możliwe jest również pominięcie użycia obrazu VirtualBox i bezpośrednie uruchomienie programu poprzez skrypt *rso* (który uruchomi kontener, a w nim skrypt *rso-run*).

4.2. Google Protocol Buffers

Jest to elastyczna i wydajna metoda serializacji strukturyzowanych danych do wykorzystania między innymi w protokołach komunikacyjnych oraz przechowywaniu danych. Określana jest struktura informacji która będzie serializowana poprzez zdefiniowanie typu wiadomości w pliku „.proto”. Każda taka wiadomość jest logicznym rekordem informacji zawierającym pary nazwa-wartość. Format takiej wiadomości jest prosty, każdy typ składa się z jednego lub więcej unikatowych, ponumerowanych pól z których każdy ma nazwę i typ wartości. Typem wartości pola mogą być liczby, zmienne logiczne, łańcuchy znaków oraz inne wiadomości PBM. Pozwala to na hierarchiczne ułożenie struktury. Każde pole może być oznaczone na trzy sposoby: opcjonalne, wymagane, powtarzające się z zachowaniem kolejności.

Kiedy wiadomości są określone należy włączyć kompilator protocol buffer dla języka aplikacji. Generuje on klasy dostępu do danych na podstawie pliku „.proto”. Tworzone są podstawowe funkcje dostępu do każdego pola oraz metody do serializowania/parsowania całej struktury. Dla Javy generowane są pliki „.java” z klasami dla każdego typu wiadomości oraz specjalna klasa „Builder” do tworzenia instancji klas wiadomości.

Można dodać nowe pola do wiadomości nie zaburzając kompatybilności wstecznej. Stare pliki binarne ignorują nowe pola podczas parsowania. W przypadku protokołów komunikacyjnych które wykorzystują protocol buffers jako format danych, można rozszerzyć protokół bez obaw o wpływ na istniejący kod.

Wszystkie wyżej wymienione cechy zaważyły przy wyborze tego narzędzia do tworzenia systemu.

4.3. MySQL

Aplikacja serwerowa będzie działać w oparciu o relacyjną bazę danych MySQL.

Baza MySQL jest popularnym systemem relacyjnych baz danych z dobrą dokumentacją oraz szeroką i aktywną społecznością użytkowników. Ponadto system ten jest powszechnie znany wśród wszystkich członków projektu. Dodatkowo MySQL dobrze współpracuje z wybranym językiem programowania Java. Stąd nasz wybór padł właśnie na MySQL.

5. Ważne zmiany w dokumentacji

5.1. Zmiany w stosunku do dokumentacji etapu I

5.04.2015r.

— Poprawa błędu związanego z wymaganiem WF4 oraz z założeniem ZO4 - zmiana dotyczy pomyłki związanej z **niepełną redundancją** - miała ona dotyczyć warstwy **wewnętrznej**.

— Zmiana w założeniu ZO8 - ze względu na funkcjonalność systemu i pozostałe założenia lepiej będzie zrobić asynchroniczną komunikację klienta z serwerem

5.2. Zmiany w stosunku do dokumentacji etapu II

28.05.2015r.

— Poprawa ZO9 związana z Heartbeatmi między klientem a warstwą zewnętrzną serwera - w związku z algorytmem równomiernego podziału klientów na węzły serwera system Heartbeatów jest obligatoryjny