POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Inżynieria systemów informatycznych (INS)

PRACA DYPLOMOWA MAGISTERSKA

Zastosowanie mechanizmów kryptograficznych przy projektowaniu wieloplatformowych aplikacji rozproszonych

Use of cryptographic mechanisms for designing distributed cross-platform applications

autor: Michał Lange 154313

Opiekun pracy:
dr inż. Robert Wójcik, W4/I06

OCENA PRACY:

WROCŁAW 2016

Spis treści

Spis rysunków

Spis listingów

1 Wstęp

Słowo „kryptografia” pochodzi z języka greckiego i może być przetłumaczone jako „ukryte pismo”. Kryptografia początkowo opierała się na tajności sposobu przekazywania informacji oraz na steganograficznych sposobach zapisywania informacji w tekstach lub grafikach w taki sposób, aby nawet osoba przewożąca wiadomości nie wiedziała, że przewozi jakąś tajną wiadomość. Obecnie kryptografia zajmuje się przekształceniem wiadomości w taki sposób, aby stała się niemożliwa do odczytania bez użycia specjalnego, tajnego klucza [10].

Przekształcona w powyższy sposób wiadomość nazywana jest szyfrogramem lub krypotogramem, a cały proces nazywany jest szyfrowaniem.

W dzisiejszych czasach wiadomości wysyłane, przetwarzane i odczytywane są przy pomocy komputerów lub innych urządzeń elektronicznych. Kryptografia w kontekście dnia dzisiejszego nie zajmuje się zatem tylko szyfrowaniem tekstów, ale może być również i jest stosowania do szyfwowania wszelakich informacji przetwarzanych przez systemy komputerowe, takich jak: teksty, obrazy, klipy wideo, muzyka. Dane w systemach komputerowych przechowywane są w postaci cyfrowej i każde takie dane mogą być zaszyfrowane.

Współczesną kryptografie można podzielić na:

- symetryczną – do szyfrowania i deszyfrowania używany jest ten sam klucz,
- asymetryczną – do szyfrowania i deszyfrowania używane są różne klucze.

Kryptografia nie zajmuje się jednak tylko szyfrowaniem. Poza szyfrowaniem, czyli poufnością metody krypograficzne mogą zapewnić:

- integralność - niezmienność danych w czasie procesu,
- uwierzytelnianie - autentyczność, pewność co do pochodzenia danych,
- niezaprzeczalność - nadawca nie może się wyprzeć przesłania komunikatu o ustalonej treści.

Powyższe cechy mogą być dostarczone przy pomocy podpisu cyfrowego. Podpis cyfrowy to dodadkowa informacja dołączona do danych służaca do weryfikacji ich źródła oraz zapewniająca integralność, autentyczność i niezaprzeczalność. Podpisy cyfrowe korzystają z kryptografii asymetrycznej - tworzona jest para kluczy:

- klucz prywatny - służący do podpisywania wiadomości,
- klucz publiczny - służący do weryfikacji wiadomości.

Ponadto domeną kryptografii są takie zagadnienia jak:

- dowód z wiedzą zerową,

- głosowanie elektroniczne,
- współdzielenie tajemnic,
- obliczenia wielopodmiotowe.

Kolejnym zagadnieniem będącym przedmiotem tej pracy są „systemy rozproszone". Ze względu na różnorodność rozważanych aspektów nie istnieje jednolita definicja systemu rozproszonego. Definicja ta ewoluowała wraz z rozwojem komputerów oraz sieci komputerowych. Za system rozproszony uważa się powszechnie zbiór niezależnych urządzeń technicznych (np. komputerów) połączonych w jedną, spójną logiczną całość (np. przy pomocy sieci komputerowej).

Cechy systemów rozproszonych to[11]:

- Ukrycie przed użytkownikami systemu:
  - różnic pomiędzy poszczególnymi komputerami,
  - sposobów komunikowania się komputerów,
  - wewnętrznej organizacji systemu rozproszonego.
- Jednolity i spójny interfejs dla użytkownika - niezależnie od czasu i miejsca interakcji.

Systemy rozproszone są tworzone ze względu na istotne potencjalne zalety tych systemów. Efektywne zagospodarowanie tych zalet może sprawić, że będą one bardziej atrakcyjne dla użytkownika końcowego niż systemy scentralizowane[11].

Następnym elementem występującym w temacie, a także będącym nieodłączym elementem tego opracowania jest „wieloplatformowość". Wieloplatformowość jest to cecha systemu lub aplikacji, który działa na więcej niż jednej platformie. Platforma to kombinacja sprzętu i oprogramowania, na której uruchamiana jest aplikacja lub system. Platforma sprzętowa może odnosić się do architektury procesora lub architektury komputera. Platforma systemowa odnosi się do systemu operacyjnego lub maszyny wirtualnej, może być także kombinacją obydwu. Użytkownik stosuje określony program po to, aby zrealizować konkretne zadanie. Tak długo, dopółki funkcjonowanie aplikacji nie zależy od platformy, w której się wykonuje, dla użytkownika jest całkowicie obojętne na jakiej platformie jest uruchamiana. Platforma sprzętowa i programowa stanowią jedynie środowisko dla uruchamiania systemów i aplikacji, same w sobie bez oprogramowania są bezużyteczne.

1.1 Cel pracy

Celem pracy jest zaprojektowanie, zaimplementowanie oraz wdrożenie systemu komputerowego umożliwiającego bezpieczną komunikację użytkowników oraz cyfrowe podpisywanie plików, a także weryfikację cyfrowo podpisanych plików. Dla uzyskania większego bezpieczeństwa generowanie klucza dla podpisu cyfrowego oraz podpisywanie plików będzie rozproszone, a więc odbywać się ono będzie w kilku miejscach systemu.

Wynikiem pracy będzie system rozproszony, którego użytkownicy końcowi będą mogli z korzystać z niego pracując na różnych platformach sprzętowych i programowych. Ponadto system będzie zaprojektowany również w taki sposób, aby przy możliwie małych zmianach lub bez nich mógł być uruchamiany na różnych platformach. W pracy zostaną przeprowadzone testy aplikacji oraz zastosowanych mechanizmów kryptograficznych.

1.2 Zakres pracy

Pierwszym elementem pracy jest zaprojektowanie i zaimplementowanie aplikacji działającej na serwerze, która funkcjonować będzie jako punkt dostępowy dla użytkowników systemu czyli klientów. Kolejnym elementem będzie mediator, będzie to część systemu, która będzie uczestniczyć przy generowaniu klucza dla podpisu cyfrowego oraz będzie brała udział przy podpisywaniu plików. Klienci chcący korzystać z systemu będą musieli posiadać aplikacje kliencką, która będzie kolejnym elementem systemu. Administratorzy będą posiadali specjalną wersję klienta, za pomocą której będzie można dodawać, edytować lub usuwać użytkowników systemu.

2 Analiza wymagań i założenia projektowe

W niniejszym rozdziale zawarto analizę wymagań funkcjonalnych i niefunkcjonalnych oraz założenia stawiane przed systemem.

2.1 Koncepcja działania systemu

W skład systemu wchodzą następujące komponenty:

- `serwer dostępowy`,
- `mediator`,
- `klient`,
- `administrator klient`.

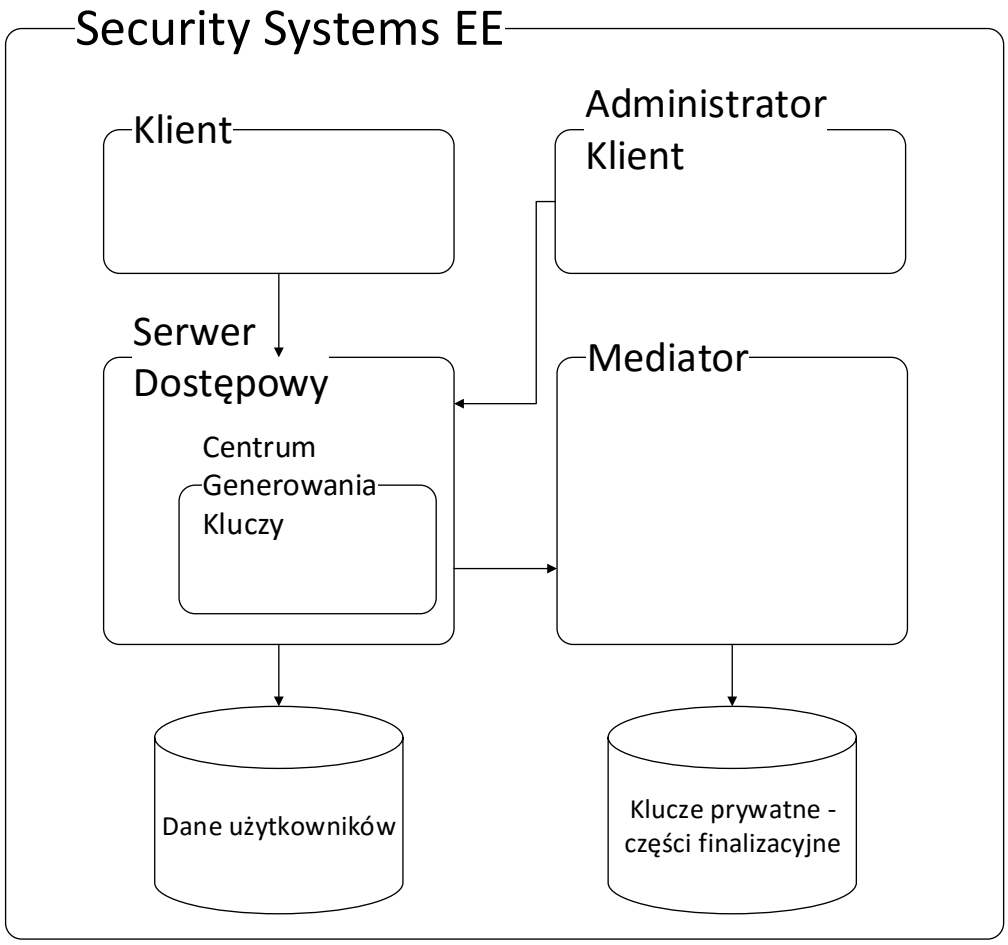
Głównym węzłem systemu jest `serwer dostępowy`. Serwer ten przechowuje informacje o zarejstrowanych użytkownikach systemu w bazie danych oraz implementuje logikę biznesową. Serwer ten jest punktem dostępowym, do którego przy pomocy aplikacji klienckiej łączą się użytkownicy.

Użytkownik końcowy przy dostępie do systemu korzysta z aplikacji `klient`, która oferuje graficzny interfejs użytkownika. Aplikacja kliencka komunikuje się bezpośrednio jedynie z serwerem dostępowym. Aplikacja `Klient` umożliwia wymianę wiadomości z innymi użytkownikami systemu, a także podpisywanie cyfrowe plików oraz weryfikację podpisów cyfrowych.

`Mediator` jest to osobny serwer, a zarazem węzeł systemu, który bierze udział podczas generowania kluczy dla podpisu cyfrowego, a także podczas procesu podpisywania pliku. `Mediator` posiada własną bazę danych, w której przechowuje swoje części kluczy prywatnych użytkowników.

`Administrator Klient` to aplikacja, dzięki której można dodawać, edytować i usuwać użytkowników systemu. Komunikuje się ona bezpośrednio z `serwerem dostępowym`.

2.1.1 Komponenty systemu



Rysunek 1: Kierunek wywoływania komponetów systemu.

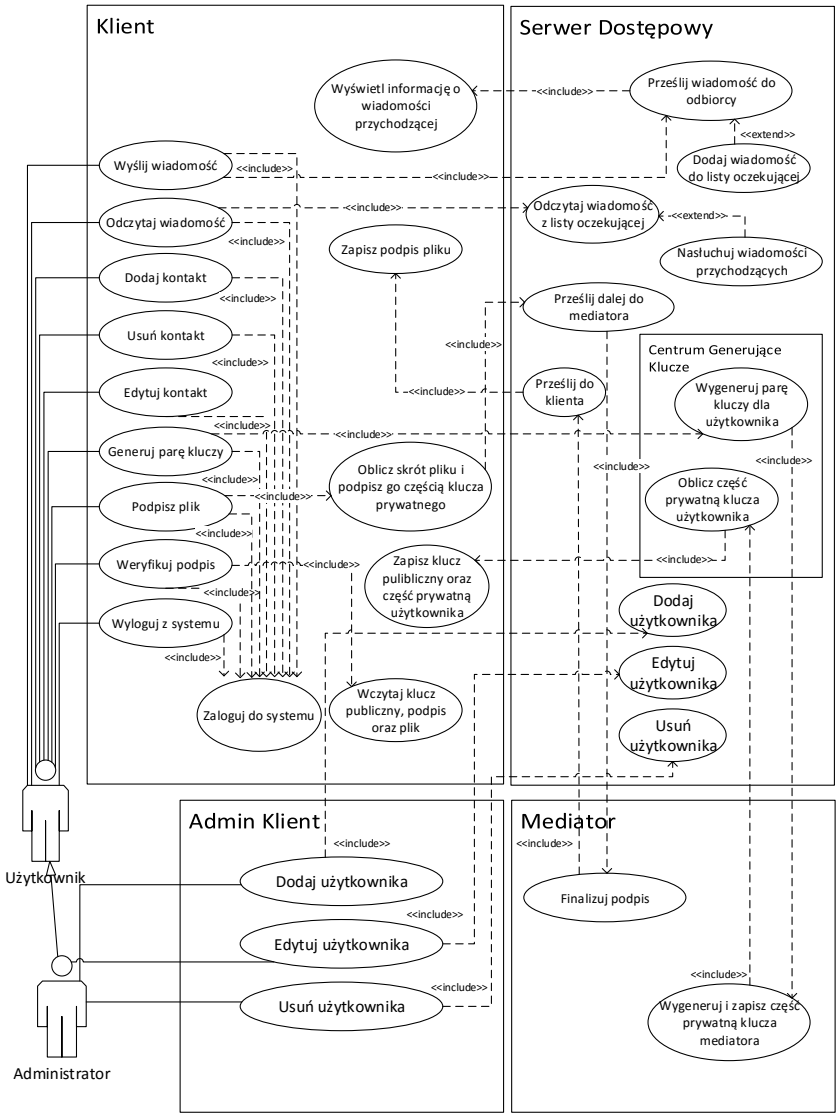
2.2 Wymagania funkcjonalne

Wymagania funkcjonalne dla systemu to:

- Możliwość dostępu do systemu jedynie po pomyślnym zalogowaniu.
- Oprócz nazwy użytkownika i hasła, system przydziela każdemu użytkownikowi specjalny klucz serwisowy, który używany jest przy komunikacji pomiędzy serwerem dostępowym a aplikacją kliencką oraz pomiędzy serwerem dostępowym a mediatorem oraz służy jako dodadkowy element walidacji.
- Wymiana wiadomości tekstowych z innymi użytkownikami w trybie rzeczywistym poprzez sieć internet.
- Dostępna lista kontaktów z możliwością modyfikacji, czyli:
  - dodawanie nowego kontaktu,
  - edytowanie kontaktu,
  - usuwanie kontaktu.
- Przechowywanie listy użytkowników lokalnie w pliku XML.
- Przechowywanie konfiguracji klienta w lokalnym pliku konfiguracyjnym.
- Powiadomienie o otrzymaniu nowej wiadomości od innego użytkownika systemu.
- Generowanie kluczy dla podpisu cyfrowego:
- Generowanie kluczy wymaga ponownej autoryzacji (podanie loginu i hasła)
- Podczas generowania klucz prywatny dzielony jest na dwie części:
  - część prywatna użytkownika - przechowywaną przez aplikacje kliencką,
  - część prywatna mediatora - przechowywaną w mediatorze.
- Cyfrowe podpisywanie plików.
- Podpisywanie pliku odbywa się dwuetapowo. Pierwszy etap odbywa się w aplikacji klienckiej, za pomocą części prywatnej klucza użytkownika. Częściowy podpis finalizowany jest przez mediatora za pomocą części prywatnej klucza mediatora.
- Weryfikacja podpisanego pliku.

- Weryfikacja odbywa się po stronie `aplikacji klienckiej`.
- Osobna aplikacja `klient do administracji` do zarządzania systemem posiadająca następujące funkcjonalności:
  - dodawanie `użytkownika systemu`,
  - edytowanie `użytkownika systemu`,
  - kasowanie `użytkownika systemu`.

2.2.1 Diagram przypadków użycia



Rysunek 2: Diagram przypadków użycia.

2.2.2 Scenariusze wybrancyh przypadków użycia

W rozdziale tym przedstawione zostały wybrane scenariusze przypadków użycia.
Scenariusz do przypadku użycia: ***Generuj parę kluczy***

Warunki początkowe

1. System jest poprawnie skonfigurowany i działa prawidłowo.
2. `Użytkownik` ma uruchomionego `klienta` i jest zalogowany do systemu oraz posiada swój własny unikalny `klucz serwisowy`.

Główny ciąg zdażeń

1. `Użytkownik` przechodzi do zakładki `konfiguracja`.
2. `Użytkownik` klika w przycisk `zarządaj nową parę kluczy`.
3. `Użytkownik` podaje prawidłowy `login` i `hasło`, a następnie klika w przycisk `generuj klucze`.
4. Zostaje wysłane żądanie do `serwera dostępowego` o wygenerowanie nowej `pary kluczy`.
5. `Para kluczy` zostaje wygenerowana. `Klucz prywatny` zostaje podzielony pomiędzy `mediatorem` i `użytkownikiem`.
6. Zostaje wyświetlona informacja o `kluczu` oraz jego poprawnym wygenerowaniu.

Scenariusz alternatywny

1. `Użytkownik` podaje nieprawidłowy `login` lub `hasło`.
2. Po wciśnięciu przycisku `generuj klucze` zostaje wyświetlony komunikat o błędzie.

Scenariusz alternatywny 2

1. Któryś z węzłów systemu: `mediator` lub `serwer dostępowy` są niedostępne.
2. Po wciśnięciu przycisku `generuj klucze` zostaje wyświetlony komunikat o błędzie.

Scenariusz do przypadku użycia: *Podpisz plik*

Warunki początkowe 1. System jest poprawnie skonfigurowany i działa prawidłowo. 2. `Użytkownik` ma uruchomionego `klienta` i jest zalogowany do systemu oraz posiada swój własny unikalny `klucz serwisowy`. 3. `Użytkownik` wygenerował `parę kluczy`, posiada swoją część prywatną, a druga część znajduje się na `mediatorze`.
Główny ciąg zdażeń 1. `Użytkownik` przechodzi do zakładki `podpis cyfrowy`. 2. `Użytkownik` klika w przycisk `wczytaj plik`, a następnie wskasuje plik, który ma zostać podpisany. 3. `Użytkownik` klika w przycisk `podpisz plik`. 4. `Aplikacja kliencka` składa częściowy podpis. 5. Zostaje wysłane żądanie do `serwera dostępowego` o żądanie podpisu. 6. Zwrócony zostaje sfinalizowany podpis pliku 7. `Użytkownik` dostaje informację o podpisie cyfrowym i ma możliwość jego zapisania.
Scenariusz alternatywny 1. `Klucz prywatny użytkownika` nie jest kompatybilny z `częścią prywatną klucza mediatora`. 2. Po wciśnięciu przycisku `podpisz plik` zostaje wyświetlony komunikat o błędzie.
Scenariusz alternatywny 2 1. Któryś z węzłów systemu: `mediator` lub `serwer dostępowy` są niedostępne. 2. Po wciśnięciu przycisku `podpisz plik` zostaje wyświetlony komunikat o błędzie.

Scenariusz do przypadku użycia: *Wyślij wiadomość*

Warunki początkowe
1. System jest poprawnie skonfigurowany i działa prawidłowo. 2. `Użytkownik` posiada na `liście kontaktów` `kontakt` do którego chce wysłać `wiadomość`. 3. `Użytkownik` ma uruchomionego `klienta`, jest zalogowany do systemu oraz posiada swój własny unikalny `klucz serwisowy`.
Główny ciąg zdażeń
1. `Użytkownik` przechodzi do zakładki `komunikator`. 2. `Użytkownik` klika w `kontakt`, do którego chce wysłać `wiadomość`, a następnie w `oknie wiadomości` wpisuje `wiadomość`. 3. `Użytkownik` klika w przycisk `wyślij wiadomość`. 4. `Wiadomość` zostaje wysłana do odbiorcy.
Scenariusz alternatywny
1. Któryś z węzłów systemu: `mediator` lub `serwer dostępowy` są niedostępne. 2. Po wciśnięciu przycisku `podpisz plik` zostaje wyświetlony komunikat o błędzie.

2.3 Wymagania niefunkcjonalne

Projektowany system musi spełniać równierz szereg wymagań niefunkcjonalnych, które zostały przedstawione w kolejnych podpunktach.

2.3.1 Bezpieczeństwo

Kluczowym wymaganiem niefunkcjonalnym projektowanego systemu jest bezpieczeństwo.

- Komunikacja pomiędzy `aplikacją kliencką`, a `serwerem dostępowym` powinna odbywać się w bezpiecznym kanale np HTTPS.
- `Mediator` nie powinien być dostępny z sieci internet, ale powinien być umiejscowiony w sieci wewnętrznej za firewallem i być dostępny dla `serwera dostępowego`.
- `Serwer dostępowy` powinien zostać umiejscowiony w strefie zdemilitaryzowanej DMZ sieci lokalnej i być dostępny z sieci internet oraz mieć dostęp do `mediatora`.

2.3.2 Wykorzystywane środowiska, technologie i narzędzia

Z systemu powinni móc korzystać użytkownicy niezależnie od systemu operacyjnego, na którym pracują. Sam system powinien przy znikomych zmianach lub bez nich być uruchomialny na innej platformie oraz współpracować z różnymi bazami danych.

- `Aplikacja kliencka` będzie napisana w technologii `Java SE 1.8`. Maszyna wirtualna Java działa na systemach `Windows`, `Linux`, `Mac OS` oraz na wielu platformach sprzętowych.
- `Serwer dostępowy` oraz `mediator` będą napisane w technologii `Java EE 7`. Aplikacje `Java EE` można uruchamiać na wielu platformach serwerowych, które pracują na różnych platformach sprzętowych i systemowych.
- Baza danych dla `serwera dostępowego` i `mediatora` powinna być zdefiniowana w serwerze aplikacji.
- Do budowania wszystkich projektów używane będzie narzędzie `Maven 3.3`.
- `Serwer dostępowy` oraz `mediator` domyślnie będą uruchamiane i testowane na serwerze aplikacji `Glassfish 4.1` oraz z bazą danych `MySql 5.7`.
- Komunikacja między elementami systemu będzie odbywać się za pomocą web serwisów typu `REST`.

- Wszystkie elementy sytemu będą napisane przy pomocy zinegrowanego środowiska programistycznego `Eclipse JEE Mars 4.5.1`.

2.3.3 Dostępność

- Możliwość korzystania z systemu 7 dni w tygodniu, 24h na dobę z wyjątkiem przerwy na zarządzanie i konserwację (średnio 5h w miesiącu).
- Możliwosc korzystania z systemu z sieci internet.

2.3.4 Inne wymagania

- Legalne funkcjonowanie systemu bez zakupu licencji na dodadkowe oprogramowanie.
- System napisany modułowo, umożliwiający łątwą modyfikację funkcjonalności, rozszerzalność oraz ponowne użycie kodu.
- `Użytkownik` powinien logować się do systemu jednocześnie tylko z jednej lokalizacji.
- `Aplikacja kliencka` posiada intuicyjny i estetyczny interfejs graficzny.

3 Analiza zastosowanych metod i technologii

W tym rozdziale omówione zastostały główne pojęcia, technologie stosowane w systemie oraz metody wykorzystywane podczas jego tworzenia. W ostatnim podrozdziale przedstawiony został model matematyczny podpisu cyfrowego, na podstawie którego utworzona została implementacja w systemie.

3.1 Aplikacja Enterprise

Nie ma jednoznacznej definicji opisującej aplikację typu enterprise, istnieje jednak wiele cech i właściwości, które charakteryzują tego typu oprogramowanie. Są to między innymi:

- skalowalność – to właściwość, która charakteryzuje system przygotowany na wzrost liczby użytkowników zarówno pod względem architektonicznym (rozszerzalność kodu) jak i wydajnościowym(np. load balancing),
- podział na warstwy, w dobrze zaprojektowanym systemie można wyróżnić:
  - warstwę klienta,
  - warstwę prezentacji,
  - warstwę biznesową,
  - warstwę integracji,
  - warstwę zasobów;
- bezpieczeństwo - system powinien być odporny na ataki z zewnątrz oraz spełniać aktualnie przyjęte normy bezpieczeństwa,
- modularność - elementy systemu powinny być pogrupowane w moduły,
- oddzielenie komponentów technicznych oraz logiki biznesowej,
- wysoce skompikowane systemy implementowane i wdrażane przez wiele lat,
- odporność na awarie - uzyskiwaną np. poprzez redundancję krytycznych elementów systemu,
- systemy rozproszone,
- łatwość odzyskiwania po awarii - uzyskiwaną np. poprzez automatyczne i regularne tworzenie kopii zapasowych aktualnego stanu aplikacji.

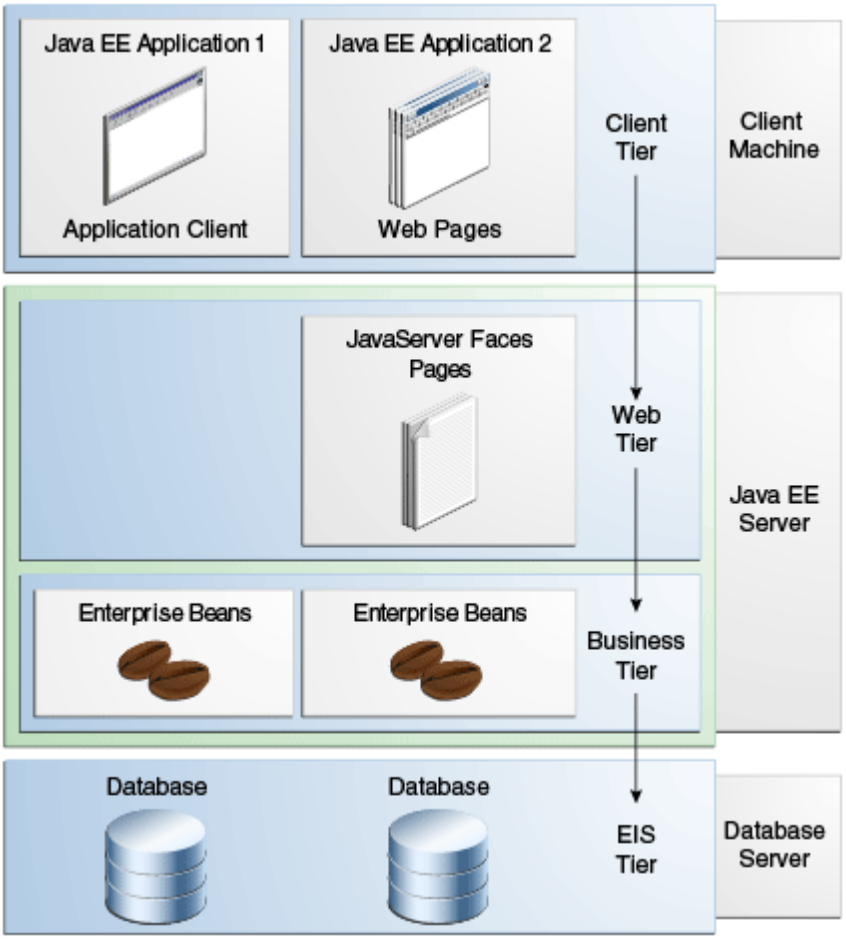
3.1.1 Technologie Java Enterprise Edition

Java EE lub inaczej Java Enterprise Edition jest to platforma programistyczna języka Java, służąca do pisania aplikacji biznesowych. Java EE posiada wszystkie elementy oraz zalety platformy Java SE, dzięki czemu jest w pełni przenośna, może być implementowana oraz wdrażana na dowolnym systemie operacyjnym posiadający wirtualną maszynę Javy (JVM). Jave EE rozszerza wersję SE o zbiór komponentów przeznaczonych do tworzeniaw aplikacjach klasy enterprise. Komponenent Javy EE jest niezależną, funkcjonalną jednostką oprogramowania wbudowywaną w aplikację JEE (wraz z powiązanymi z nim klasami i plikami) i komunikującą się z innymi komponentami[2]. Java EE 7 wprowadziła szereg adnotacji, dzięki którym zbędne staje się przechowywanie dużej części konfiguracji w plikach XML.

W specyfikacji Javy EE zdefiniowano następujące komponenty[2]:

- aplikacje klienckie i aplety są komponentami uruchamianymi po stronie klienta,
- komponenty technologii Java Servlet, JavaServer Faces oraz JavaServer Pages (JSP) są komponentami webowymi działającymi na serwerze,
- komponenty Enterprise JavaBeans (zwane także ziarnami EJB) będące komponentami biznesowymi działającymi na serwerze.

Warstwy aplikacji zostały przedstawione na rysunku 3.



Rysunek 3: Wielowarstwowa aplikacja w technologii Java EE[13].

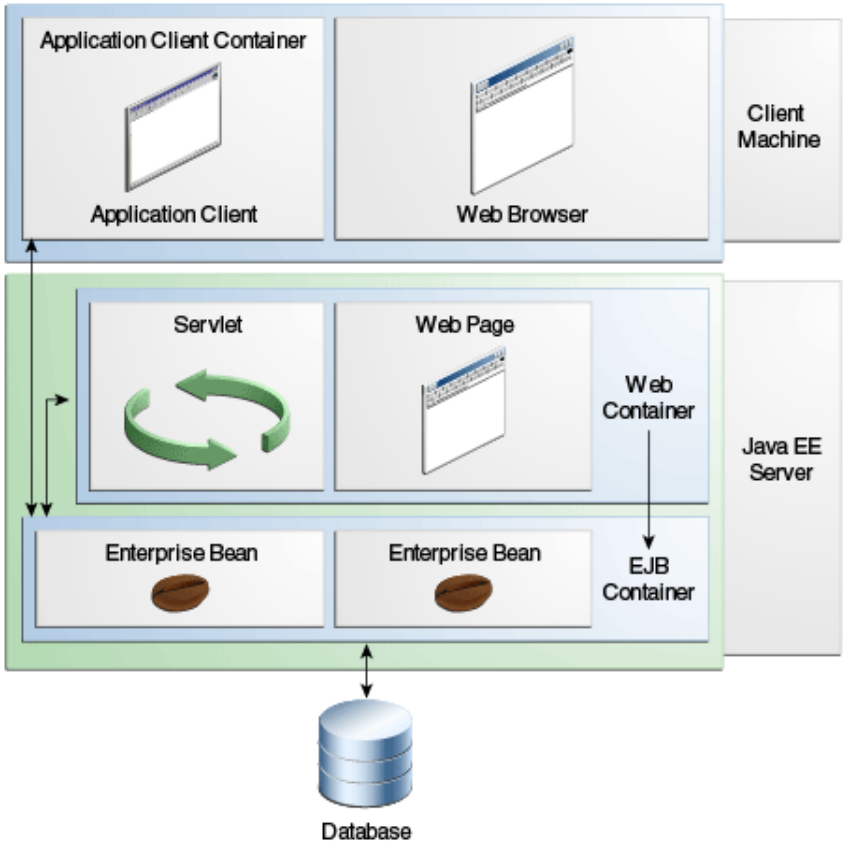
3.1.2 Serwer i kontenery Java EE

Aplikacja Java EE uruchamiana jest na serwerze Javy EE. Serwery aplikacji Javy EE to np:

- Apache Geronimo,
- GlassFish,
- JBoss Application Server,
- ObjectWeb JOnAS,
- IBM WebSphere Application Server,
- Oracle WebLogic Server,
- Oracle AS/OC4J (nierozwijany, porzucony na rzecz WebLogic Server),
- SAP Web Application Server.

Serwer Javy EE zawierea kontenery przeznaczone do przechowywania oraz uruchamiania poszczególnych części aplikacji. Na rysunku 4 przedstawiony został serwer Javy EE oraz osadzone na nim kontenery.

- Serwer Javy EE - środowisko uruchomieniowe dla aplikacji JEE. Serwer Javy EE dostarcza kontenery EJB oraz kontenery webowe.
- Kontener Enterprise JavaBeans(EJB) - zarządza wykonywaniem ziaren EJB w aplikacjach JEE. Komponenty EJB i ich kontener działają na serwerze Javy EE.
- Kontener webowy - zarządza działaniem stron internetowych, serwletów i niektórych komponentów EJB aplikacji JEE. Komponenty webowe i ich kontener działają na serwerze Javy EE
- Kontener aplikacji klienckich - zarządza działaniem komponentów aplikacji klienckiej. Aplikacje klienckie i ich kontener działają po stronie klienta.



Rysunek 4: Serwer i kontenery Javy EE[13].

3.1.3 Enterprise JavaBeans

Komponenty EJB (Enterprise JavaBeans) zwane tekże ziarnami EJB, są fragmentami kodu zawierającymi pola oraz metody implementujące część logiki biznesowej. Komponenty EJB umożliwiają tworzenie i wdrażanie rozproszonych aplikacji na bazie komponentów, które oferują skalowalność, możliwośc przetwarzania transakcyjnego i bezpieczeństwo. Za zarządzanie instancją komponentu w czasie wykonywania odpowiada kontener. Klient uzyskuje dostęp do komponentu za pośrednictwem kontenera, w którym dany komponent został wdrożony. Klient może też działać po stronie serwera i mieć na przykład postać komponentu zarządzalnego, komponentu CDI lub serwletu[1].

Istnieją trzy rodzaje komponentów EJB:

- komponenty sesyjne, dzielą się na:

  - stanowe - każdy komponent przechowuje unikalny stan, np. stan komunikacji z klientem;

  - bezstanowe - komponenty te wbrew nazwie też posiadają swój unikalny stan, jednak nie może być on powiązany bezpośrednio z żądaniem, na które może zostać zwrócony inny komponent tego samego typu,

  - singletony - unikalne komponenty na skalę środowiska uruchomieniowego, przechowujące np stan aplikacji;

- komponenty sterowane komunikatami - komponenty wykorzystywane podczas komunikacji w technologii JMS;

- komponenty encyjne - komponenty przechowujące encje danych, od wersji EJB 3.0 oznaczone są jako przestarzałe, a do przechowywania encji zaleca się stosowanie klaz z adnotacją @Entity.

3.1.4 Mapowanie obiektowo-relacyjne

Mapowanie obiektowo-relacyjne (ORM) jest to sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych (lub inny element systemu) o relacyjnym charakterze. Implementacja takiego odwzorowania stosowana jest m.in. w przypadku, gdy tworzony system oparty jest na podejściu obiektowym, a system bazy danych operuje na relacjach.

JavaPersistence API (JPA) jest standardem ORM dla języka Java. Z punktu widzenia programisty jest to możliwość operowania na obiektach - zwanych encjami - oraz zapisywania wyników operacji do relacyjnej bazy danych za pomocą obiektu EntityManager. Sposób w jaki obiekty i ich połączenia przekładają się na elementy bazy danych są definiowane za pomocą adnotacji lub dokumentów XML. Poza standardowym zestawem operacji udostępnianych przez obiekt EntityManager standard JPA definiuje język zapytań JPA Query Language podobny do SQL.[15]. Encje w JPA oznaczane są adnotacją @Entity.

3.2 Usługi sieciowe typu REST

REST to akronim od representational state transfer. REST jest usługą sieciową zaimplementowaną na bazie protokołu HTTP. REST to też architektura, która określa model i zasady umożliwiające budowanie rozwiązań, pozwalających na bardzo dużą skalowalność, wydajność i łatwość modyfikacji. W architekturze REST dane i funkcjonalność reprezentowane są jako zasób. Każdy zasób reprezentowany jest przez oddzielny URL. REST oprócz zasobów reprezentowanych przez Uri to również kwestia reprezentacji - dany zasób może być reprezentowany jako wynik typu XML, Json, Text, Html, Image oraz wiele innych typów. Komunikacja REST jest bezstanowa. Usługi sieciowe (web serwisy) typu REST nazywane są też serwisami RESTful.

Usługi REST korzystają zazwyczaj z czterech metod. Metody te to:

- Get – używana przy dostępie do zasobu,
- Put – używana przy tworzeniu zasobu oraz przy innych operacjach,
- Update – używana przy aktualizacji zasobu,
- Delete – używana do kasowania zasobu.

3.2.1 Interfejs JAX-RS oraz implementacja Jersey

JAX-RS (od ang. Java API for RESTful web services) to API języka Java, które powstało w celu ułatwienia tworzenia aplikacji opartych o architekturę REST. Od wersji 2.0 JAX-RS definiuje interfejs sterowany adnotacjami. Adnotacje JAX-RS są adnotacjami środowiska uruchomieniowego Java EE. Po uruchomieniu w kontenerze Java EE w środowisku uruchomieniowy zostają wygenerowane klasy pomocnicze oraz artefakty niezbędne do działania web serwisu, następnie usługa jest automatycznie konfigurowana, po czyn zostaje udostępniona[2].

JAX-RS oferuje między innymi następujące adnotacje:

- @Path – jest względną ścieżką URI wskazującą miejsce hostowania klasy Javy,
- @GET – metody oznaczone tym desygnatorem będą obsługiwały żądania GET,
- @POST – metody oznaczone tym desygnatorem będą obsługiwały żądania POST,
- @PUT – metody oznaczone tym desygnatorem będą obsługiwały żądania PUT,

- @DELETE - metody oznaczone tym desygnatorem będą obsługiwały żądania DELETE,
- @HEAD – metody oznaczone tym desygnatorem będą obsługiwały żądania HEAD,
- @PathParam – adnotacja ta służy do wyekstrahowania parametru z URI. nazwy parametrów odpowiadają nazwom zmiennych określonych we wzorcu URI,
- @QueryParam – adnotacja ta służy do wyekstrahowania parametru z URI, parametry zapytania są ekstrahowane z parametrów zapytania adresu URI;
- @Consumes – używana do określenia przetwarzanych przez zasób typów MIME,
- @Produces – używana do określenia zwracanych przez zasób tymów MIME
- @Provider – adnotacją tą oznacza się wszystkie elementy, które mogą być potrzebne w środowisku uruchomieniowym.

Istnieje szereg dalszych adnotacji np: @MatrixParam, @HeaderParam, @CookieParam, @FormParam, @DefaultValue, @Context.

Jersey jest to referencyjna implementacja JAX-RS w Javie. Jersey to framework o otwartym źródle, oferujący również szereg funkcjonalności wykraczających poza standardowe API JAX-RS np. implementację klienta REST.

3.2.2 Asynchroniczność i Long Pooling

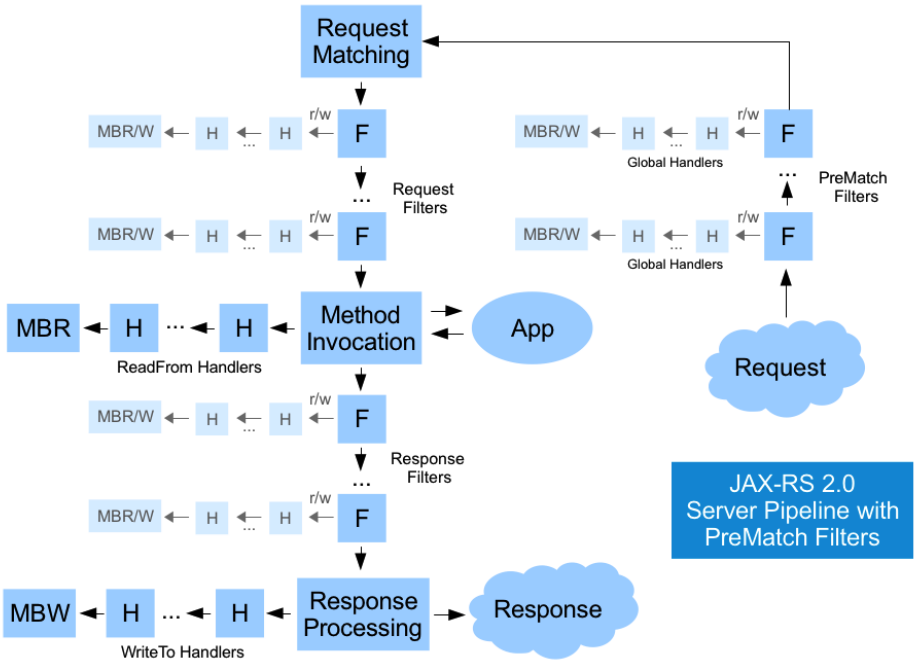
Typowe żądania typu REST przetwarzane są w trybie synchronicznym tzn. klient wysyła żądanie, serwer je przetwarza i zwraca odpowiedź. Żądania synchroniczne stosuje się wtedy gdy reprezentowany przez URI zasób może być odrazu zwrócony. W przypadku, gdy zasób nie jest od razu dostępny lub jego przygotowanie trwa zbyt długo żądanie synchroniczne zakończy się niepowodzeniem. W takim przypadku należy stosować żądania asynchroniczne. Przetwarzanie asynchroniczne polega na tym, że serwer nie zwraca odrazu żądania, lecz przekazuje je do innego wątku, gdzie odpowiedź na żądanie jest przygotowywana. Gdy zasób zostanie utworzony jest on zwracany do klienta. Istnieje technika przetwarzania asynchronicznego, która nazywa się Long Pooling. Polega ona na tym, że klient wysyła żądanie do serwera. W przypadku gdy zasób nie jest dostępny lub nie może zostać od razu utworzony to żądanie przechodzi w tryb oczekujący i zwrócone zostanie dopiero wtedy, gdy zasób będzie dostępny. Technikę taką stosuje się np. w komunikatorach. Klient wysyła żądanie, które zwrócone zostanie dopiero wtedy, gdy inny klient wyśle do niego wiadomość.

3.2.3 Filtry

Interfejs JAX-RS oferuje szereg filtrów. Filtry mogą być stosowane np. w celu modyfikacji, odrzucenia, czy przekierowania żądania. Przy pomocy filtrów można np. uwierzytelnić nadawcę żądania. Filtrowanie w JAX-RS odbywa się potokowo, tzn. można zdefiniować kilka filtrów oraz ich kolejność. Filtry w JAX-RS dzielą się na trzy grupy:

- PreMatch Filters – są to filtry które stosowane są do żądania jescze przed dopasowaniem do danego zasobu.
- Request Filters – filtry te stosowane są po filtrach PreMatch oraz po tym jak żądanie zostało dopasowane do zasobu, ale przed wykonaniem metody zasobu.
- Response Filters – są to filtry, które stosowane są do odpowiedzi na żądanie. Filtry te stosowane są po zakończeniu metody zasobu, ale przed odesłaniem zasobu do klienta. Mogą operować na danych zwróconych przez zasób.

Przetwarzanie potokowe filtrów w JAX-RS 2.0 zostało przedstawione na rysunku 5.



Rysunek 5: Przetwarzanie potokowe filtrów w JAX-RS 2.0[14]

3.3 Podpis cyfrowy

Podpis cyfrowy to matematyczny sposób sprawdzenia autentyczności danych elektronicznych. Zweryfikowany podpis cyfrowy oznacza, że dane pochodzą od właściwego nadawcy, który nie może zaprzeczyć faktowi ich podpisania oraz że dane od momentu podpisu nie zostały zmienione.

Podpis cyfrowy zapewnia następujące funckcje bezpieczeństwa:

- autentyczności pochodzenia, która daje pewność co do autorstwa danych,
- niezaprzeczalności, która utrudnia wyparcie się autorstwa lub znajomości treści danych,
- integralności, która pozwala wykryć nieautoryzowane modyfikacje danych po ich podpisaniu.

Dwa dominujące standardy podpisu cyfrowego to RSA oraz DSA.

W niniejszym opracowaniu rozpatrywany jest standard RSA. Jego nazwa pochodzi od pierwszych liter nazwisk jego twórców: Rivest, Shamir i Adleman. RSA oparty jest o kryptografię asymetryczną. Bezpieczeństwo szyfrowania opiera się na trudności faktoryzacji dużych liczb złożonych. W podpisie cyfrowym opartym o algorytm RSA stosuje się klucz prywatny do podpisywania oraz klucz publiczy do weryfikacji podpisu.

3.3.1 Podpis cyfrowy z mediatorem

Podpis elektroniczny z mediatorem to podpis cyfrowy, do którego złożenia potrzebna jest dodadkowa zaufana strona zwana mediatorem. Klucz prywatny dzielony jest na część użytkownika i część mediatora. Mediator bierze też udział przy generowaniu klucza prywatnego.

Generowanie klucza składa się z następujących faz:

- utworzenie przez dilera (generatora kluczy) pary kluczy, klucza prywatnego i publicznego,
- wysłanie do mediatora klucza publicznego oraz identyfikatora klucza,
- wygenerowanie przez mediator z modułu klucza publicznego i identyfikatora klucza swojej części klucza prywatnego i odesłania go do dilera kluczy,
- obliczenie przez dilera kluczy części prywatnej użytkownika na podstawie części klucza otrzymanego od mediatora i wygenerowanego w pierwszym punkcie klucza prywatnego

Złożenie podpisu cyfrowego przy z pomocą mediatora składa się z dwóch faz:

- utworzenie częściowego podpisu po stronie użytkownika, a następnie wysłanie częściowego podpisu oraz podpisywanych danych do mediatora,
- podpisanie częściowego podpisu (finalizacja podpisu) przez stronę zaufaną, czyli mediatora, a następnie odesłanie podpisu do użytkownika.

Weryfikacja podpisu odbywa identyczne jak w przypadku podpisu cyfrowego opartego o standardowy algorytm RSA w całości po stronie użytkowniika.

3.3.2 Model matematyczny i weryfikowanie podpisu

`Użytkownik` posiadający w systemie swój własny niepowtarzalny identyfikator `ID` oznaczony zostanie jako - `u`.

Dla każdego `u` może zostać wygenerowana `para kluczy`:

- (d_u, N_u) – `klucz prywatny`,
- (e_u, N_u) – `klucz publiczny`,

gdzie:

- (d_u) – `eksponenta klucza prywatnego`,
- (e_u) – `eksponenta klucza publicznego`,
- (N_u) – `moduł klucza prywatnego` i `klucza publicznego`.

`klucz prywatny` dzielony jest na dwie części:

- (d_{K_u}, N_u) – `część prywatna klucza` strony finalizującej podpis `mediatora`,
- (d_{M_u}, N_u) – `część prywatna klucza` strony podpisującej `użytkownika`,

gdzie:

- d_{K_u} – `część prywatna eksponenty` strony podpisującej `użykownika`,
- d_{M_u} – `część prywatna eksponenty` strony finalizującej podpis `mediatora`.

Eksponenty kluczy po podziale muszą spełniać następujący warunek:

- $d_u \equiv d_{K_u} + d_{M_u} \pmod{\varphi(N_u)}$,

gdzie:

- φ – funkcja nosząca nazwisko Eulera przypisuje każdej liczbie naturalnej liczbę liczb względnie z nią pierwszych nie większych od niej samej.

`Plik` – `p` podpisany cyfrowo przez `użytkownika` - `u` ma postać:

- $(p, h(p)^{d_u} \pmod{N_u})$,

gdzie:

- $h(p)$ – to skrót kryptograficzny `pliku` - `p` lub inaczej funkcja haszująca `pliku` - `p`,
- $h(p)^{d_u} \pmod{N_u}$ – to skrót `pliku` - `p` podniesiony do potęgi d_u modulo N_u
- $h(p)^{d_u} \pmod{N_u} \equiv h(p)^{d_{K_u}} \pmod{N_u} \times h(p)^{d_{M_u}} \pmod{N_u}$.

Aby zweryfikować `cyfrowo podpisany plik` w postaci $(p, h(p)^{d_u} \pmod{N_u})$, należy dysponować `kluczem publicznym` (e_u, N_u) `użytkownika` - `u`. W przypadku gdy do danego `klucza publicznego` dostarczony jest `certyfikat` - `c` ma on postać:

- (e_u, N_u, c_u).

Certyfikat c_u ma postać:

- $(u, e_u, N_u, h(u, e_u, N_u)^{d_c} \pmod{N_c}, e_c, N_c)$,

gdzie:

- $h(u, e_u, N_u)^{d_c} \pmod{N_c}$ to skrót `klucza publicznego użytkownika` oraz `identyfikatora użytkownika` podniesiony do potęgi `eksponenty klucza prywatnego certyfikatu` modulo `moduł klucza publicznego certyfikatu` N_c,

`Klucz publiczny` i `klucz prywatny` `certyfikatu` - `c` przechowywany jest w `urzędzie certyfikującym`:

- U_c.

Weryfikacja `podpisu` następuje w dwóch etapach. Pierwszy etap to sprawdzenie autentyczności `certyfikatu`. Następuje to poprzez obliczenie skrótu z pierwszych trzech wartości `certyfikatu` (u, e_u, N_u). Otrzymany w ten sposób skrót $h(u, e_u, N_u)$ porównywany jest z wartością $(h(u, e_u, N_u)^{d_c})^{e_c} \pmod{N_c}$, która jest czwartą częścią `certyfikatu` podniesioną do potęgi e_c modulo N_c. Jeżeli zachodzi równość:

- $h(u, e_u, N_u) = (h(u, e_u, N_u)^{d_c})^{e_c} \pmod{N_c}$.

Oznacza to, że certyfikat pomyślnie przeszedł próbę autoryzacji. Autentyczność certyfikatu implikuje autentyczność klucza publicznego użytkownika - u. Umożliwia to weryfikację podpisu samego pliku:

- $(u, p, h(p)^{d_u} \pmod{N_u})$.

Weryfikacja pliku - p następuje poprzez obliczenie $h(p)$, a następnie porównanie tej wartości z $(h(p)^{d_u})^{e_u} \pmod{N_u}$, gdzie $(h(p)^{d_u})^{e_u} \pmod{N_u}$ to $h(p)^{d_u} \pmod{N_u}$ podniesione do potęgi eksponenty e_u autoryzowanego klucza publicznego (e_u, N_u) modulo N_u. Jeżeli zachodzi równość:

- $h(p) = (h(p)^{d_u})^{e_u} \pmod{N_u}$,

oznacza to, że proces weryfikacji zakończył się pomyślnie. Jeżeli równość ta nie zachodzi może oznaczać to że:

- plik - p od momentu podpisu został zmieniony,
- plik - p został podpisany przy pomocy innego klucza prywatnego,
- klucz publiczny nie należy do użytkownika, który podpisał plik.

Plik może być równierz zweryfikowany bez certyfikatu klucza publicznego.

3.3.3 Generowanie i podział kluczy

Użytkownik - u chcący podpisać plik - p musi posiadać parę kluczy, która składa się z klucza prywatnego biorącego udział przy składaniu podpisu, oraz z klucza publicznego, który jest niezbędny do weryfikacji podpisu. Za generowanie kluczy odpowiedzialne jest centrum generowania kluczy lub inaczej generator kluczy. Generowanie kluczy przebiega następująco:

- Wybierz losowo dwie duże liczby pierwsze p i q przy uwzględnieniu następujących warunków:

  - liczby powinny mieć zbliżoną długość w bitach, ale jednocześnie powinny być od siebie odległe wartościami;

  - długość bitowa wartości $p \times q$ powinna być równa lub zbliżona do długości bitowej obliczanego klucza.

- Oblicz $N = p \times q$. N jest modułem klucza publicznego i klucza prywatnego.
- Oblicz $\varphi(N) = \varphi(p) \times \varphi(q) = (p-1) \times (q-1) = N - (p+q-1)$.

- Wybierz losowo $e \in \{1,2,3,...,\varphi(N)\}$ względnie pierwsze z $\varphi(N)$ tzn. największy wspólny dzielnik $NWD(e,\varphi(N)) = 1$. Składnik e jest `eksponentą klucza publicznego`.

  Zaleca się aby wartość e miała krótką długość bitową oraz małą wagę hamminga, tzn. sumę wartości różnych od 0 w danym alfabecie, w tym przypadku $alfabet = \{0,1\}$ (rozpatrywany jest zapis bitowy liczby). Typową stosowaną wartością jest np. $e = 2^{16} + 1 = 65,537$.

- Oblicz eksponentę klucza prywatnego $d \in \{1,2,3,...,\varphi(N)\}$ taką że:

  - wartość $d \equiv e^{-1} \pmod{\varphi(N)}$, lub inaczej $d \times e = 1 \pmod{\varphi(N)}$, tzn. wartość d jest odwrotnością modularną e.

- Wartości p, q oraz $\varphi(N)$ powinny zostać niejawne, ponieważ mogą one służyć do obliczenia d.

Z tak obliczonych wartości tworzy się następnie:

- (d,N) – `klucz prywatny`,
- (e,N) – `klucz publiczny`.

Po obliczeniu `pary kluczy`, `klucz prywtny` dzielony jest na dwie części:

- (d_M,N) – `część prywatna klucza` strony finalizującej podpis `mediatora`,
- (d_K,N) – `część prywatna klucza` strony podpisującej `użytkownika`.

Klucze mają wspólny moduł, lecz różne eksponenty. Eksponenty te muszą spełniać następujący warunek:

- $d \equiv d_K + d_M \pmod{\varphi(N)}$.

Podział `klucz prywtnego` odbywa się w dwóch etapach. Najpierw generowana jest `część prywatna klucza` strony finalizującej podpis `mediatora`. Proces ten przebiega następująco:

- `Centrum generowania kluczy` wysyła do `mediatora` wygenerowany `klucz publiczny` oraz `identyfikator ID użytkownika - u`, dla którego zostały wygenerowane `klucze`.

- `Mediator` posiada swoją własną parę kluczy znaną tylko jemu. Swoim kluczem prywatnym (d_m,N_m) `mediator` podpisuje `ID użytkownika`. Podpis ten jest postaci $h(ID)^{d_m} \pmod{N_m}$.

  Podpis $h(ID)^{d_m} \pmod{N_m}$ służy jako ziarno dla `generatora liczb pseudolosowych`.

- `Generator liczb pseudolosowych` o ziarnie $h(ID)^{d_m} \pmod{N_m}$ losuje liczbę pseudolosową d_M, taką że:

  - długość bitowa liczby d_M to suma długości bitowej `modułu klucza publicznego` N przesłanego przez `Centrum generowania kluczy` oraz wartości `delta`, gdzie $delta \in \{80...128\}$.

  - Długość bitowa N może być obliczona jako $floor(\log_2(N)) + 1$.

Liczby (d_M, N) tworzą `część prywatna klucza` strony finalizującej podpis `mediatora`. Liczba d_M wygenerowana w `mediatorze` odsyłana jest następnie do `centrum generowania kluczy`, gdzie na jej podstawie obliczana jest `część prywatna klucza` strony podpisującej `użytkownika`. Odbywa się to w następujący sposób:

- $d_K = d - d_M \pmod{\varphi(N)}$.

Liczba d_K wraz z modułem N tworzą `część prywatna klucza` strony podpisującej `użytkownika` (d_K, N).

3.3.4 Podpis pliku

`Użytkownik - u` chący podpisać `plik - p` oblicza skrót pliku $h(p)$, a następnie podnosi go do potęgi d_{K_u} modulo N_u. Tak częściowo podpisany plik ma postać:

- $h(p)^{d_{K_u}} \pmod{N_u}$.

`Użytkownik - u` wysyła do mediatora skrót `pliku - p` $h(p)$ oraz częściowy podpis $h(p)^{d_{K_u}} \pmod{N_u}$.

Mediator przy pomocy swojej części klucza prywatnego podpisuje otrzymany skrót $h(p)$ otrzymując w ten sposób $h(p)^{d_u} \pmod{N_u}$, a następnie mnoży go z otrzymanym podpisem modulo N_u. Po takiej operacji podpis jest kompletny i ma postać:

- $h(p)^{d_{K_u}} \pmod{N_u} \times h(p)^{d_{M_u}} \pmod{N_u} = h(p)^{d_u} \pmod{N_u}$.

Podpis taki odsyłany jest do `użytkownika - u`, który może dołączyć go do `pliku - p`. Otrzymując podpisany plik w postaci:

- $(p, h(p)^{d_u} \pmod{N_u})$.

4 Projekt systemu

W niniejszym rozdziale został przedstawiony projekt systemu. W pierwszym podrozdziale przedstawiona została komunikacja oraz interakcje pomiędzy poszczególnymi elementami systemu. W kolejnych podrozdziałach przedstawione zostały projekty poszczególnych elementów systemu: `serwera dostępowego`, `mediatora`, `klienta` oraz `klienta administratorskiego`.

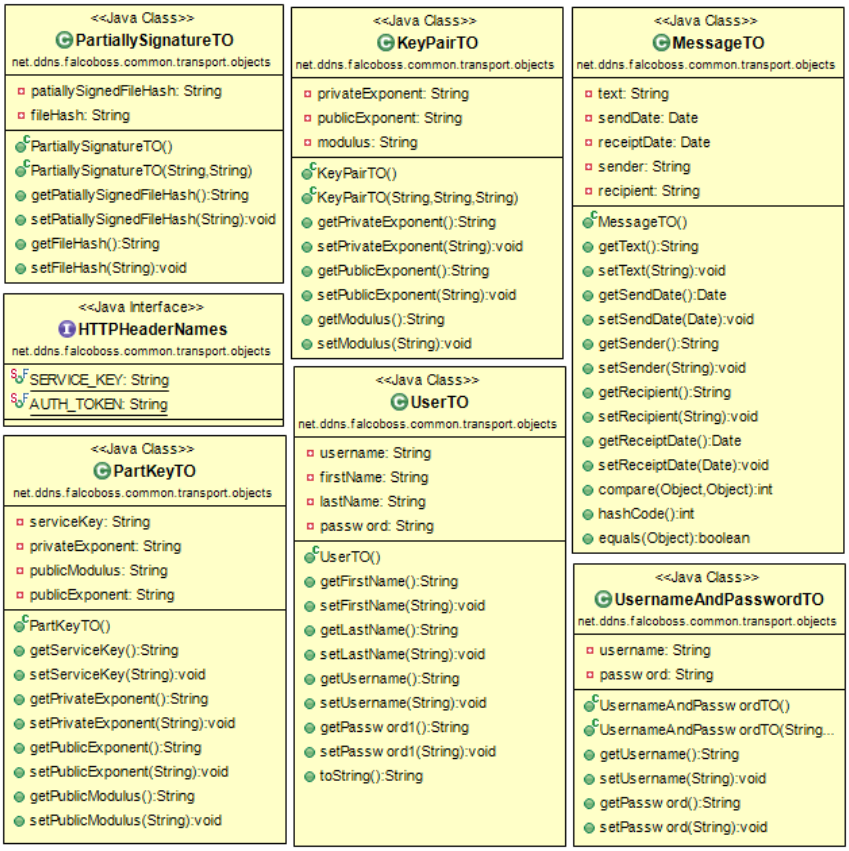
4.1 Użytkownicy systemu

`Użytkownik` w systemie może być przypisany do następujących `ról`.

- `Użytkownik standardowy` - posiada on standardową funkcjonalność, przedstawioną w diagramie przypadku użycia z wyjątkiem, z wyłączeniem działań administracyjnych, a więc zarządzaniem `użytkownikami` systemu.
- `Administrator` - posiada on funkcjonalność użytkownika standardowego, a dodadkowo możliwość zarządzania `użytkownikami` systemu. `Administrator` ma możliwość logowania się do systemu za pośrednictwem `aplikcji administratorskiej`.

4.2 Projekt komunikacji pomiędzy węzłami

W podrozdziale tym przedstawiony został przebieg komunikacji pomiędzy elementami systemu dla wybranych przypadków użycia. Komunikacja w systemie odbywa się za pośrednictwem web serwisów typu REST. W przypadku `logowania`, `wylogowywania` oraz `wysyłania wiadomości` są to zapytania synchroniczne. W przypadku `odbioru wiadomości`, `generowania klucza` oraz `podpisywania pliku` są to zapytania asynchroniczne typu Long Pooling. `Klient administratorski` używa zapytania asynchronicznego do pobrania listy użytkowników. W przypadku gdy jakiś węzeł system jest niedostępny `klient` systemu dostaje komunikat o błędzie. W przypadku łączenia się `klientów` z sieci Internet, komunikacja pomiędzy elementami systemu powinna być szyfrowana. Konfiguracja szyfrowania odbywa się na serwerze Javy EE. Zaleca się włączenie szyfrowania za pomocą protokołu TLS. Należy wtedy dostarczyć `certyfikat serwera` lub taki wygenerować. Komunikacja węzłami odbywa się za pomocą obiektów transportowych przedstawionych na rysunku 6. Obiekty przed wysłaniem serializowane są do formatu JSON i w takim formacie przesyłane. Następnie po odbiorze deserializowane z powrotem do obiektów transportowych.

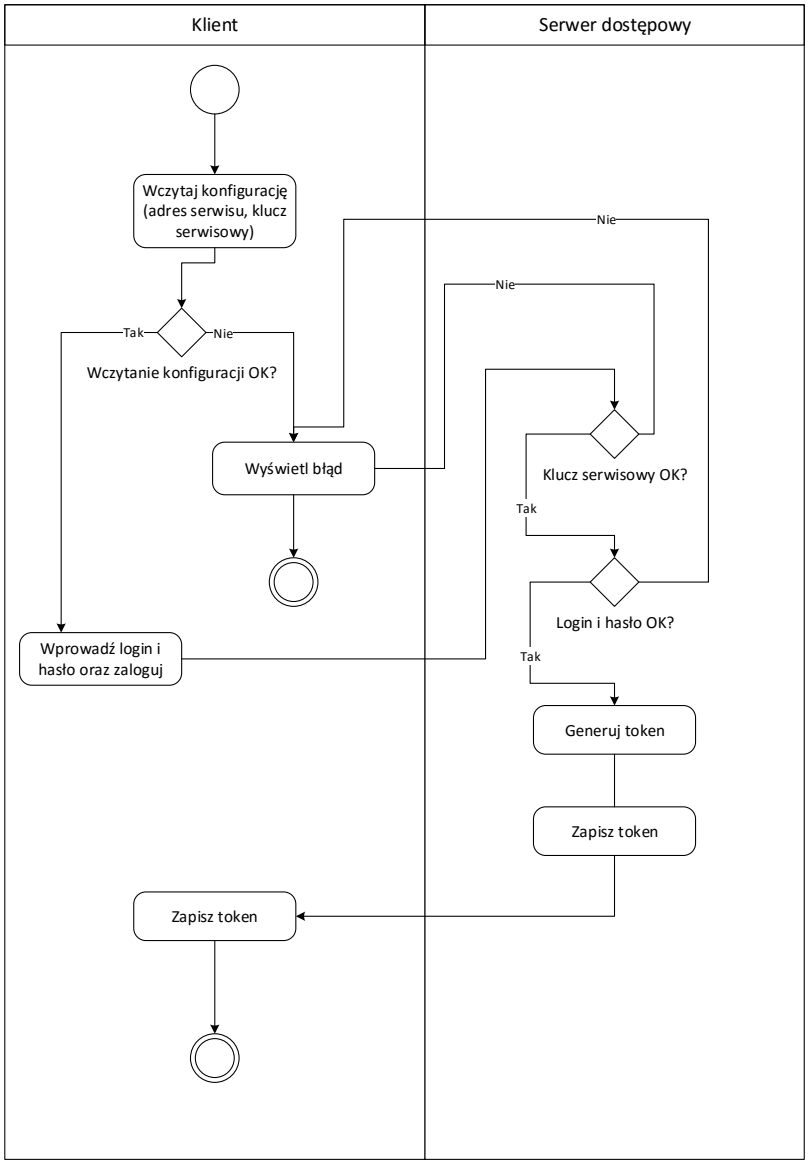


Rysunek 6: Diagram klas - obiekty transportowe.

4.2.1 Logowanie

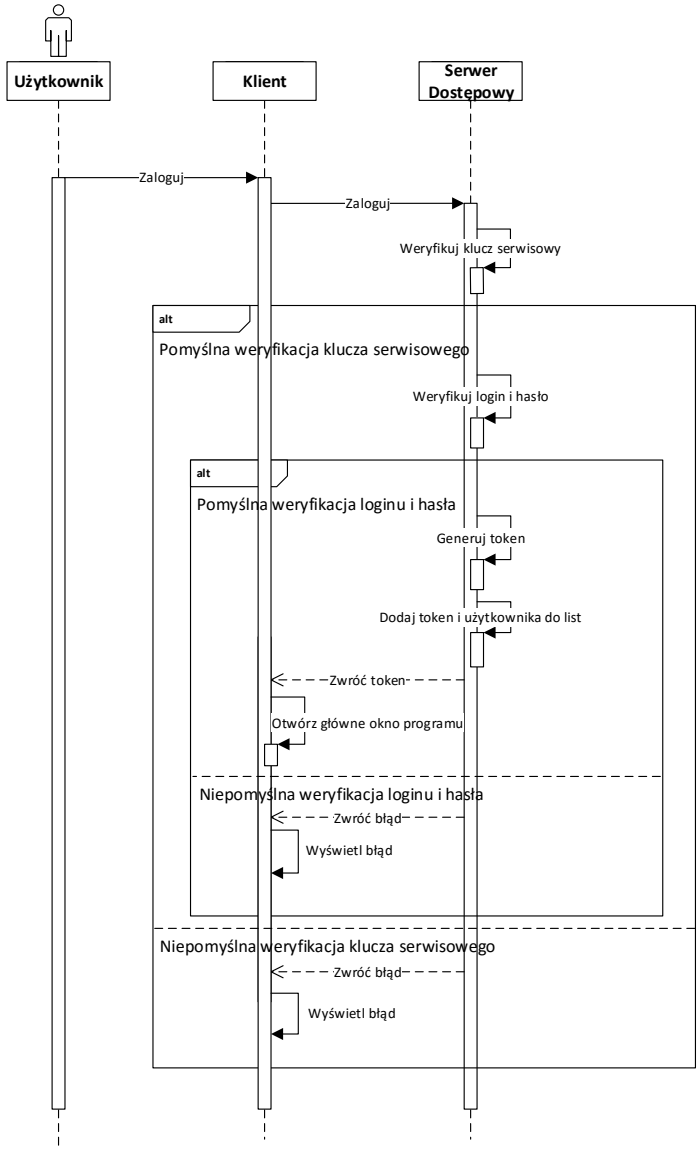
Logowanie to proces, który inicjuje klient systemu. Do zalogowania się do systemu niezbęda jest znajomość loginu, hasła oraz klucza serwisowego. Wszystkie te elementy dostarczane są przez administratora po założeniu konta użytkownika w systemie. Klient wysyła do serwera dostępowego żądanie zalogowania, a wraz z nim użytkownika, hasło i klucz serwisowy, następnie po pomyślnym zautoryzowaniu użytkownika na serwerze dostępowym generowany jest unikalny token. Token dodawany jest do listy tokenów, a następnie zwracany jest do klienta. Po otrzymaniu tokena w odpowiedzi REST klient zapisuje go. Od tego momentu klient jest zalogowany, a każde następne żądanie, aż do wylogowania odbywa się przy pomocy klucza serwisowego oraz tokena bez udziału loginu i hasła. Przy żadaniu wylogowania z systemu serwer dostępowy usuwa z token z listy tokenów. Diagramy na rysunkach 7 i 8 przedstawiją proces logowania.

Diagram aktywności dla logowania



Rysunek 7: Logowanie do systemu - diagram aktywności.

Diagram sekwencji dla logowania

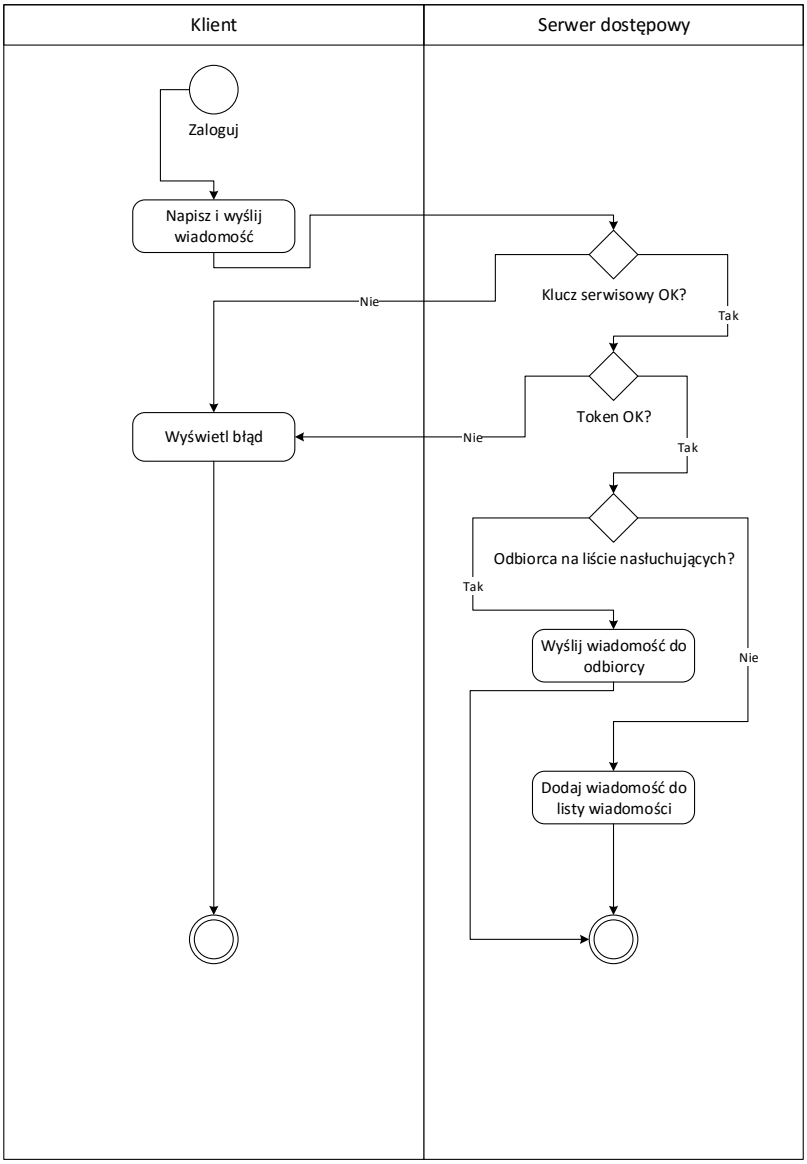


Rysunek 8: Logowanie do systemu - diagram sekwencji.

4.2.2 Wysyłanie wiadomości

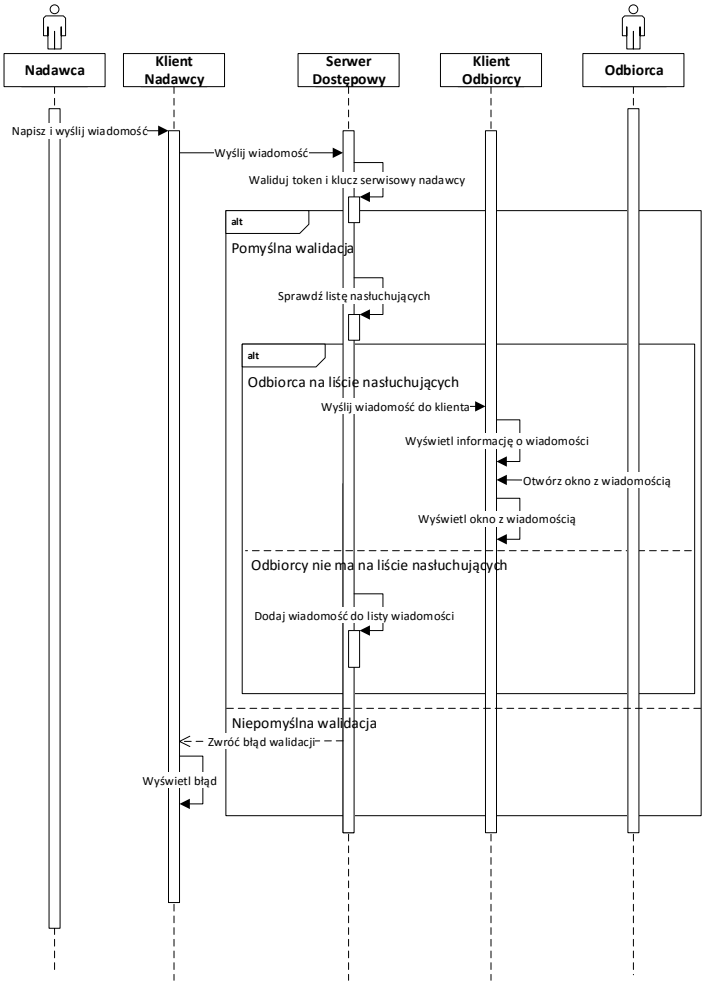
Użytkownik ma możliwość wysyłania wiadomości tekstowych do innych użytkowników systemu. Aby wysłać wiadomość z klienta należy być zalogowanym do systemu. Klient wysyła żądanie REST do serwera dostępowego, w treści którego znajduje się wiadomość, a w nagłówku klucz serwisowy oraz token. Serwer dostępowy następnie autoryzuje użytkownika oraz sprawdza listę nasłuchujących użytkowników. Jeżeli adresat wiadomości znajduje się na tej liście to serwer dostępowy przekazuje mu wiadomość od nadawcy. Jeżeli użytkownika nie ma na liście, to wiadomość dodawana jest do listy wiadomości i będzie tam aż do chwili zalogowania się adresata wiadomości do systemu, po czym zostanie mu przekazana. Na rysunkach 9 i 10 został przedstawiony proces, który zachodzi po wysłaniu wiadomości z aplikacji klienta.

Diagram aktywności dla wysyłania wiadomości



Rysunek 9: Wysyłanie wiadomości - diagram aktywności.

Diagram sekwencji dla wysyłania wiadomości

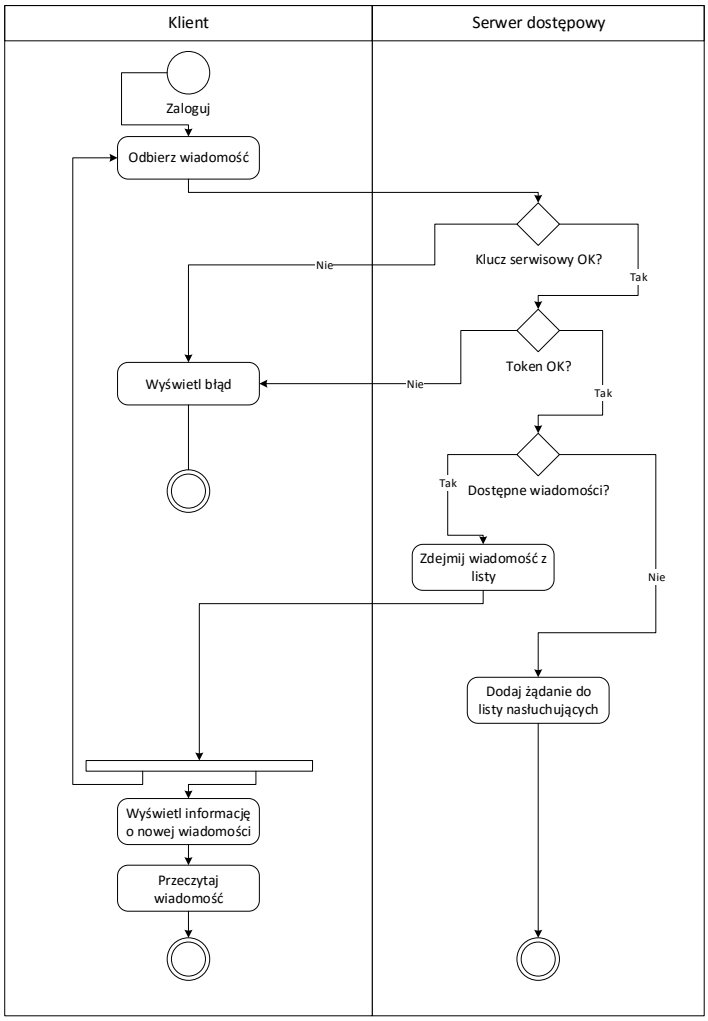


Rysunek 10: Wysyłanie wiadomości - diagram sekwencji.

4.2.3 Odbiór wiadomości

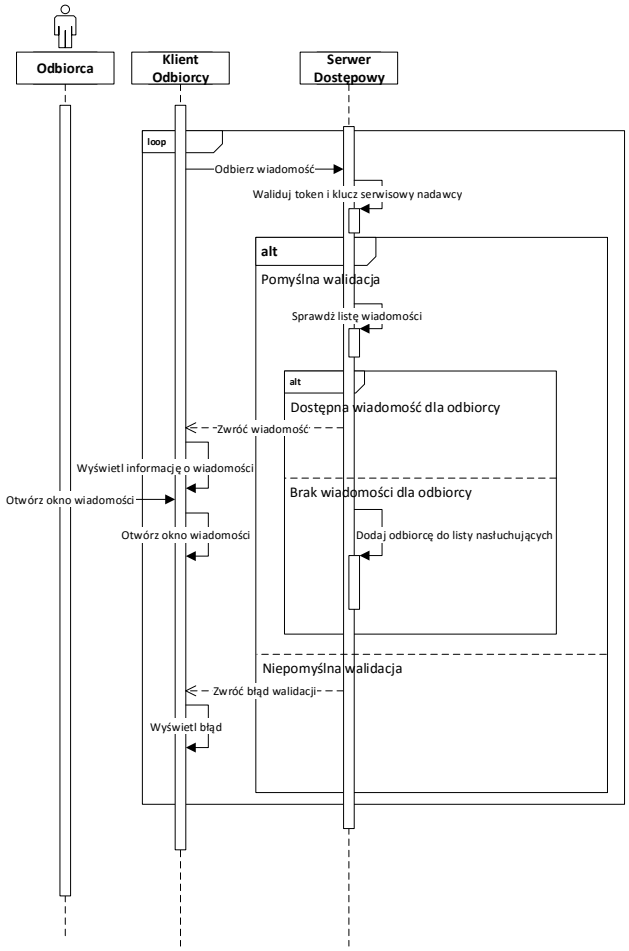
Odbiór wiadomości odbywa się również przy pomocy asynchronicznego zapytania REST. Po zalogowianiu klient wysyła do serwera dostępowego żądanie odbioru wiadomości z nagłówkiem zawierającym klucz serwisowy i token. Po pomyślnej autoryzacji serwer dotępowy sprawdza listę wiadomości w poszukiwaniu wiadomości, której adresat jest użytkownikiem wysyłającym żądanie. W przypadku znalezienia takiej wiadomości zostaje ona dołączona do odpowiedzi i jest zwracana doklienta. W przypadku, gdy na liście wiadomości nie ma żadnej wiadomości do użytkownika, to żądanie zapisywane jest na liście nasłuchujących i znajduje się tam aż do chwili pojawienia się wiadomości do użytkownika. Każde żądanie wysyłające wiadomość sprawdza najpierw listę nasłuchujących. W chwili pojawienia się na serwerze dostępowym nowego żądania wysłania wiadomości z wiadomością do użytkownika, którego zapytanie znajduje się na liście nasłuchujących, wiadomość zostaje dołączona do żądania oczekującego, a żądanie zostaje usunięte z listy nasłuchujących i jest odsyłane do klienta jako odpowiedź na żądnaie asynchroniczne. Aplikacja klient po otrzymaniu odpowiedzi wyświetla na liście kontatków obok kontatku, który jest adresatem wiadomości informację o nowej wiadomości. Proces w aplikacji klient do odbioru wiadomości uruchamiany jest jako osobny wątek w tle. Proces ten działa w pętli tak, że po wysyłaniu żądania odbioru i otrzymaniu odpowiedzi wysyła ponowne żądanie odbioru wiadomości. Na rysunkach 11 i 12 przedstawiony został proces odbioru wiadomości.

Diagram aktywności dla odbioru wiadomości



Rysunek 11: Odbiór wiadomości - diagram aktywności.

Diagram sekwencji dla odbioru wiadomości

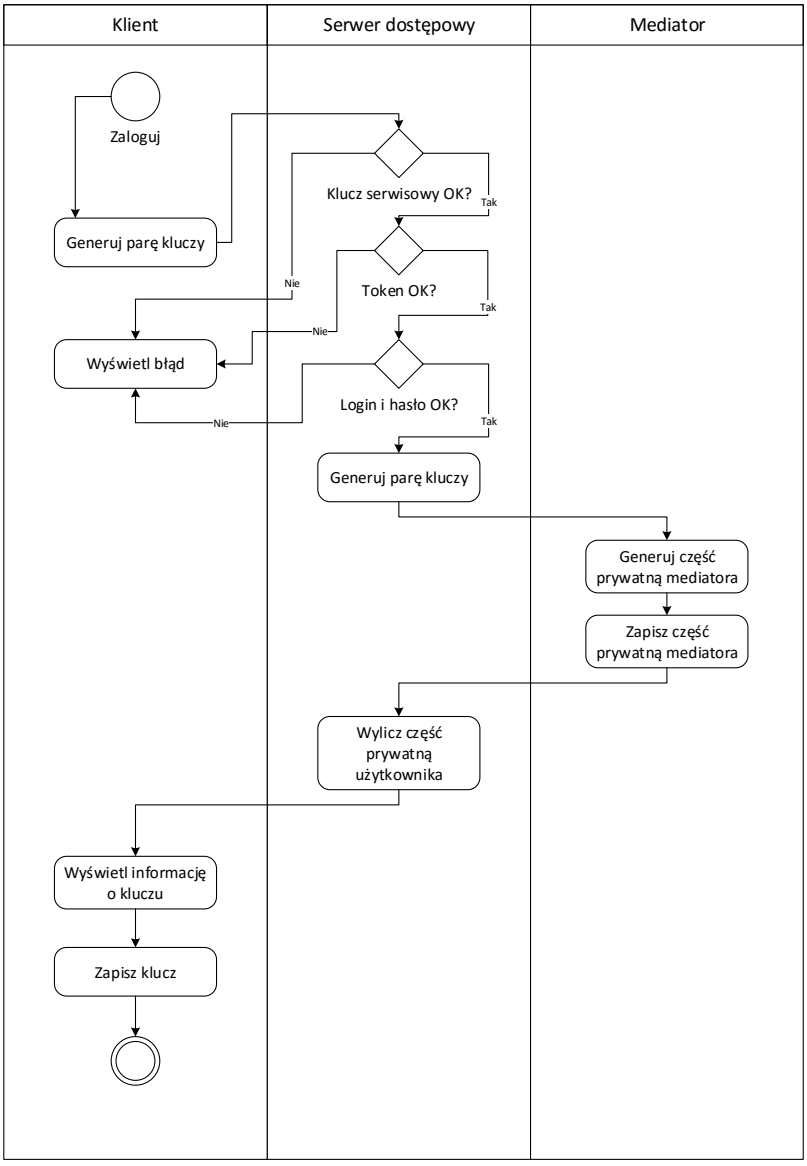


Rysunek 12: Odbiór wiadomości - diagram sekwencji.

4.2.4 Generowanie pary kluczy

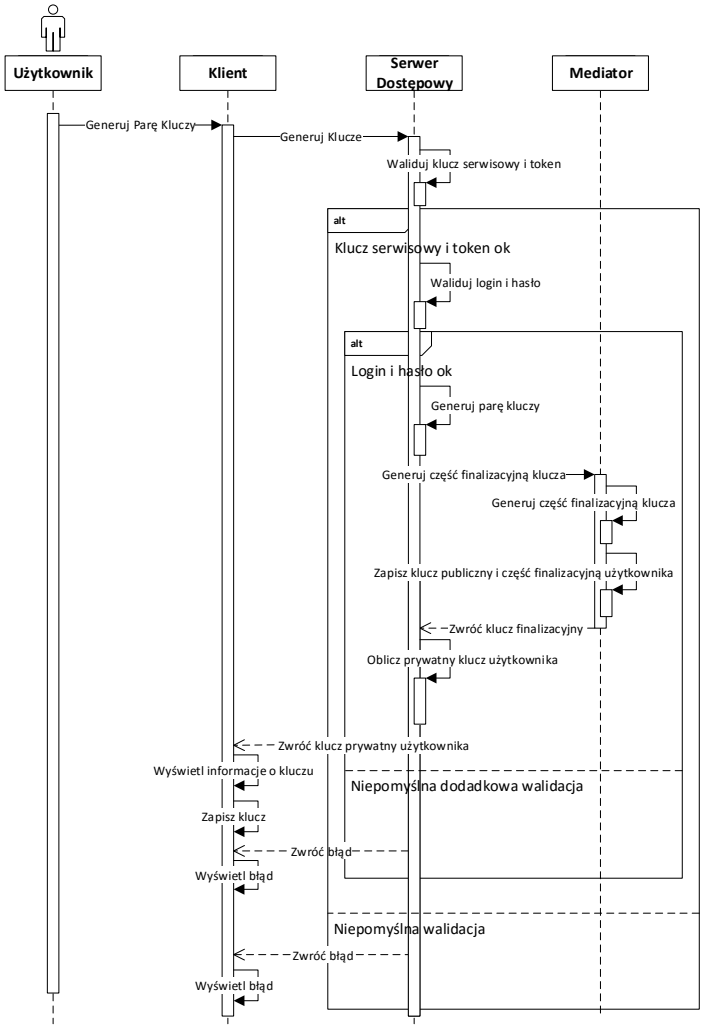
Proces generowania kluczy odbywa się w oparciu o model matematyczny przedstawiony w rozdziale 3. Nowe klucze generowane są na żadanie użytkownika. W aplikacji klienckiej żądanie generowania pary kluczy to jedyne żądanie oprócz logowania, które w celu dodadkowego zabezpieczenia, wymaga podania loginu i hasła. Żądanie generowania kluczy wysyłane jest do serwera dostępowego. Serwer po pomyślnej autoryzacji użytkownika generuje nową parę kluczy: klucz publiczny oraz klucz prywatny. Klucz publiczny oraz klucz serwisowy wysyłany jest następnie do mediatora. Mediator na podstawie długości bitowej klucza publicznego, własnego klucza prywatnego oraz wartości losowej generuje w oparciu o swój algorytm prywatny klucz finalizujący. Mediator zapisuje klucz publiczny użytkownika oraz prywatny klucz finalizujący, następnie prywatny klucz finalizujący odsyłany jest do serwera dostępowego. Serwer na podstawie wcześniej wygenerowanego klucza prywatnego oraz prywatnego klucza finalizującego oblicza prywatny klucz użytkownika. Para kluczy składająca się z prywatnego klucz użytkownika oraz wcześniej wygenerowanego klucza publicznego odsyłana jest do klienta jako odpowiedź na asynchroniczne zapytanie REST. Klient po zakończeniu żądania zapisuje swoją parę kluczy do plików. W przypadku generowania ponownie pary kluczy, stara para zostaje nadpisana na mediatorze oraz w aplikacji klienta. W takim przypadku, należy zabezpieczyć istniejący plik zawierający klucz publiczny w celu późniejszej weryfikacji plików podpisanych przy pomocy odpowiadającemu mu kluczowi prywatnemu. Rysunki 13 i 14 przedstawiają proces generowania pary kluczy.

Diagram aktywności dla generowania pary kluczy



Rysunek 13: Diagram aktywności - generowanie pary kluczy.

Diagram sekwencji dla generowania pary kluczy

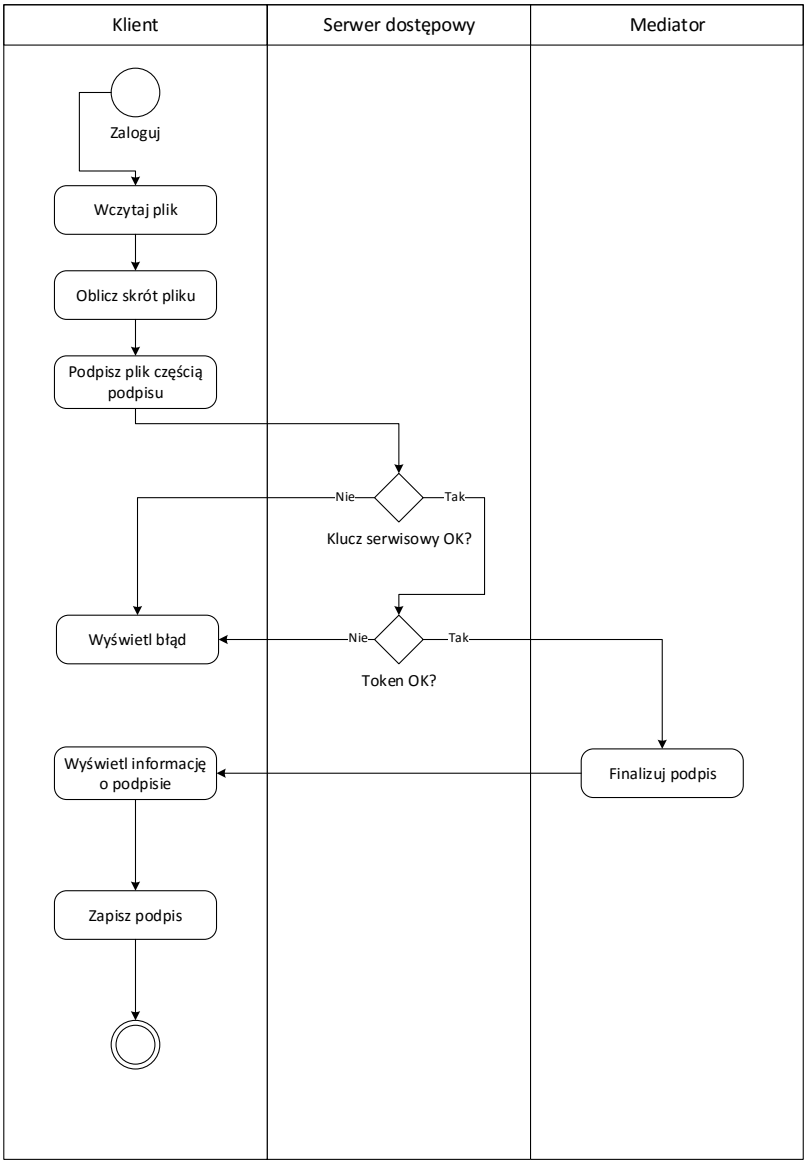


Rysunek 14: Diagram sekwencji - generowanie pary kluczy.

4.2.5 Podpisywanie pliku

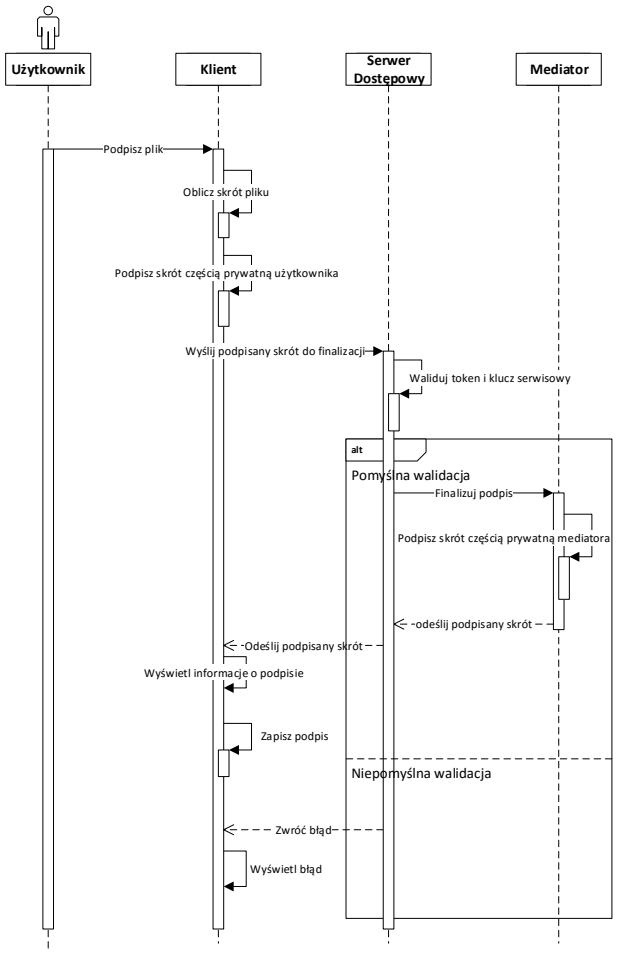
Proces podpisywania pliku również odbywa się w oparciu o model matematyczny przedstawiony w rozdziale 3. Aby podpisać plik użytkownik powinien posiadać wcześniej wygenerowaną parę kluczy. Następnie plik do podpisu powinien zostać wczytany w aplikacji klient. Z załadowanego wcześniej pliku zostaje obliczony skrót kryptograficzny SHA512. Skrót ten jest następnie podpisywany kluczem prywatnym użytkownika. Częściowo podpisany plik oraz jego skrót wysyłane są w żądaniu podpisu do serwera dostępowego. Nagłówek żądania zawiera klucz serwisowy i token. Serwera dostępowy po pomyślnym zautoryzowaniu użytkownika przesyła do mediatora otrzymany podpis i skrót oraz klucz serwisowy użytkownika. Mediator podpisuje otrzymany skrót przy pomocy prywatnego klucza finalizującego, a następnie składa go z podpisem dokonanym przez klienta. Złożony podpis odsyłany jest jako odpowiedź asynchroniczna do serwera dostępowego, który odsyła go do klienta. Podpis zapisywany jest automatycznie do pliku. Proces podpisywania pliku został przedstawiony na rysunkach 15 i 16

Diagram aktywności dla podpisywania pliku



Rysunek 15: Diagram aktywności - podpisywanie pliku.

Diagram sekwencji dla podpisywania pliku



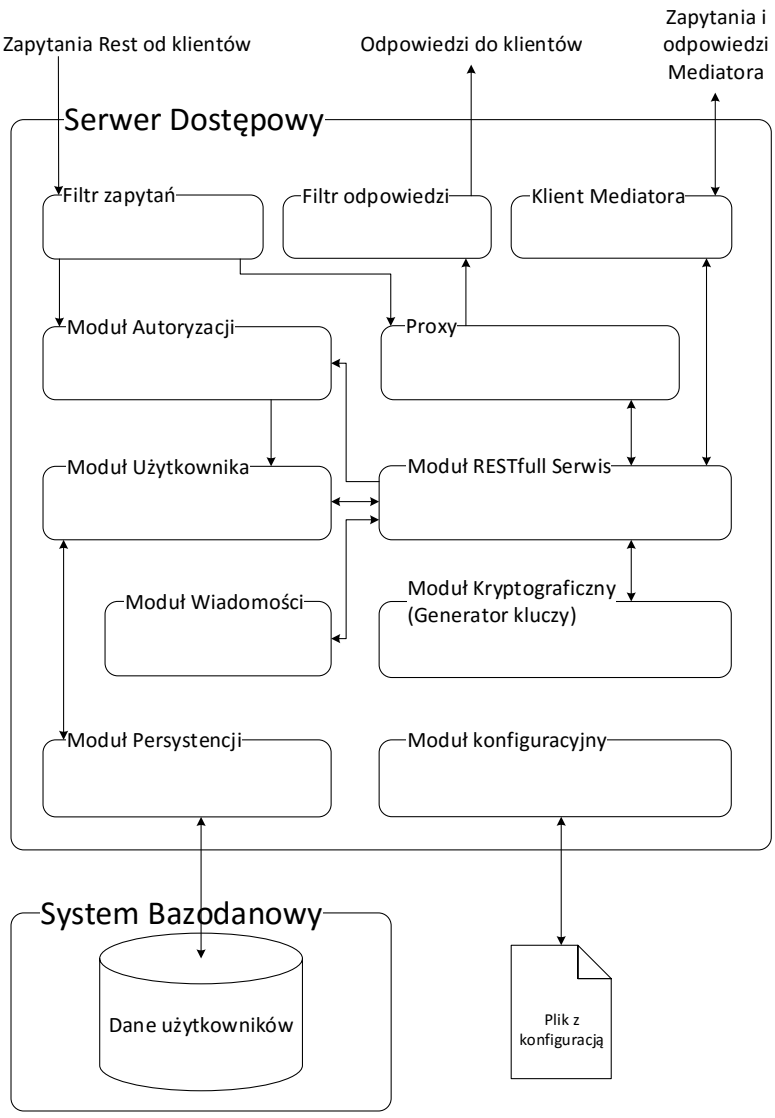
Rysunek 16: Diagram sekwencji - podpisywanie pliku.

4.2.6 Weryfikacja podpisu

Aby zweryfikować podpisany `plik` należy wczytać go w aplikacji `klienckiej`, następnie należy wczytać `podpis` pliku oraz `klucz publiczny` należący do `użytkownika`, który ten plik podpisał. Weryfikacja następuje wtedy w całości po stronie klienta. Proces ten został dokładniej przedstawiony w projekcie aplikacji klienta.

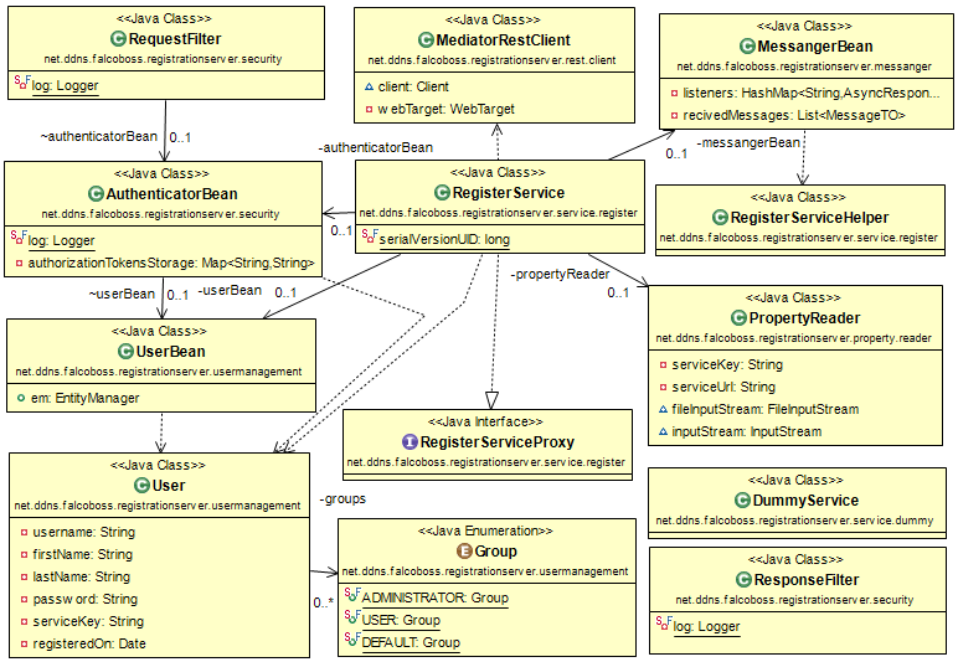
4.3 Projekt serwera dostępowego

4.3.1 Komponenty serwera dostępowego

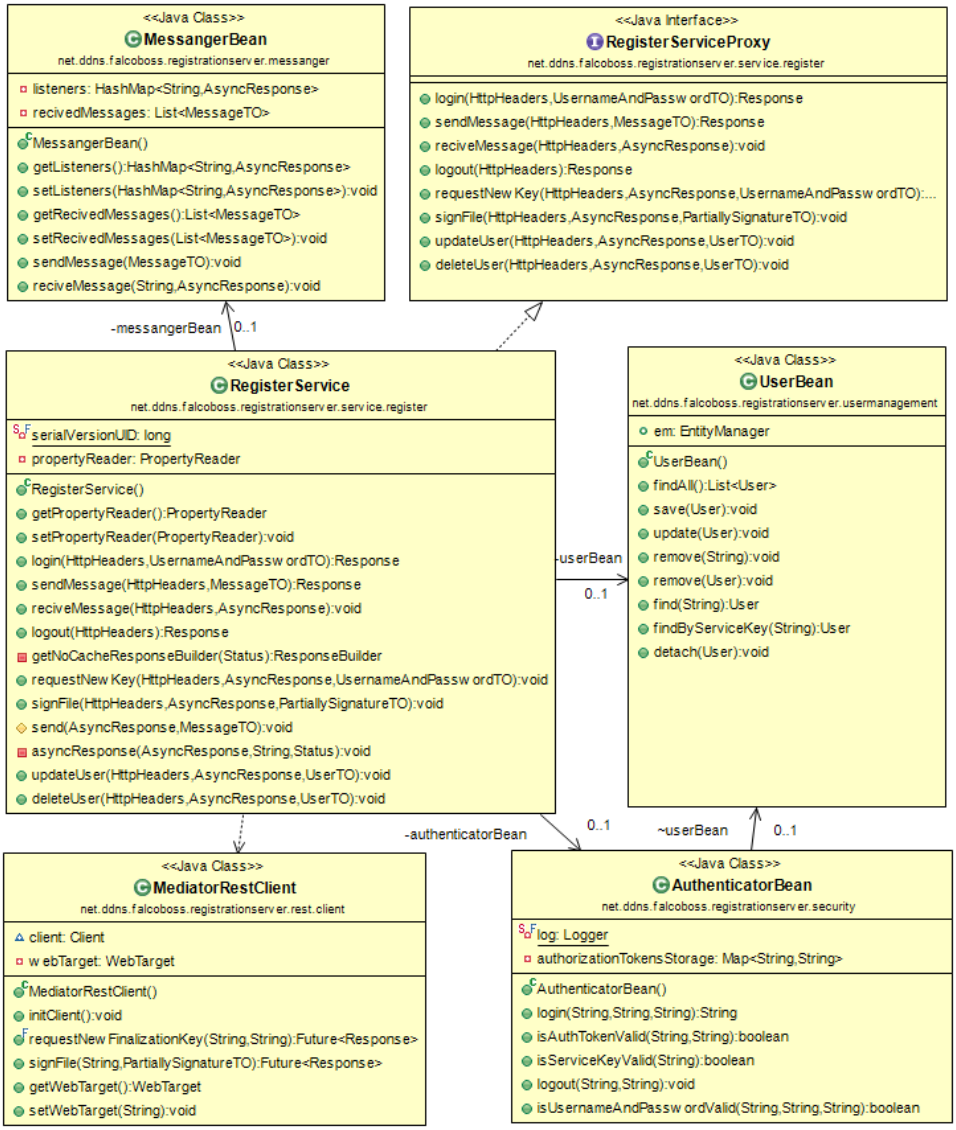


Rysunek 17: Interakcje między komponentami serwera dostępowego.

4.3.2 Diagram klas



Rysunek 18: Uproszczony diagram klas serwera dostępowego.



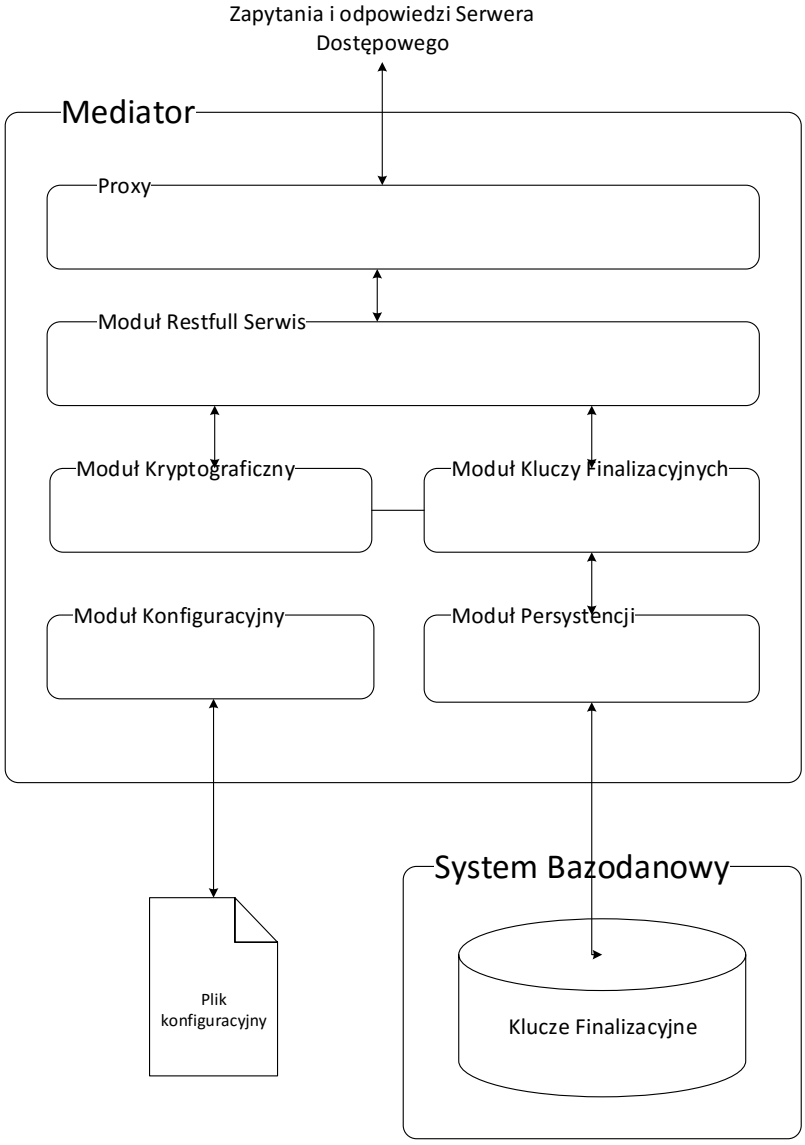
Rysunek 19: Diagram wybranych klas serwera dostępowego.

4.3.3 Diagramy sekwencji

TODO diagram sekwencji dla wybranej metody

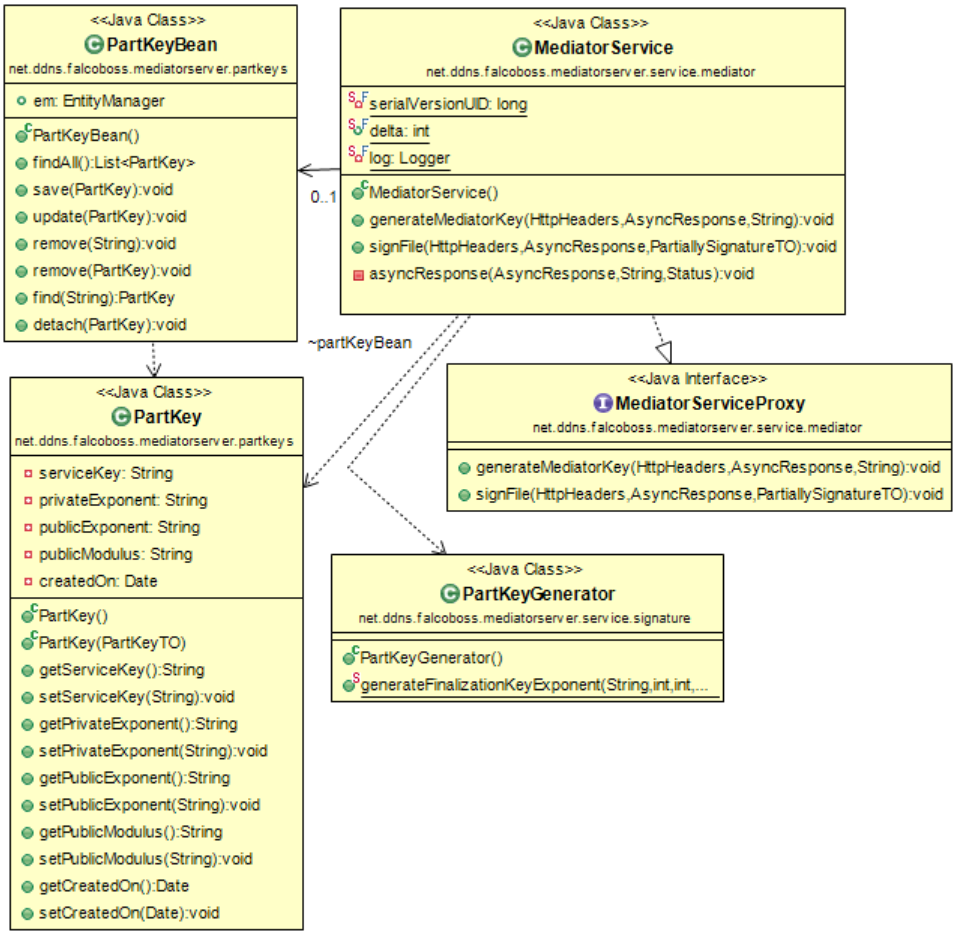
4.4 Projekt serwera mediatora

4.4.1 Komponenty mediatora



Rysunek 20: Interakcje między komponentami mediatora.

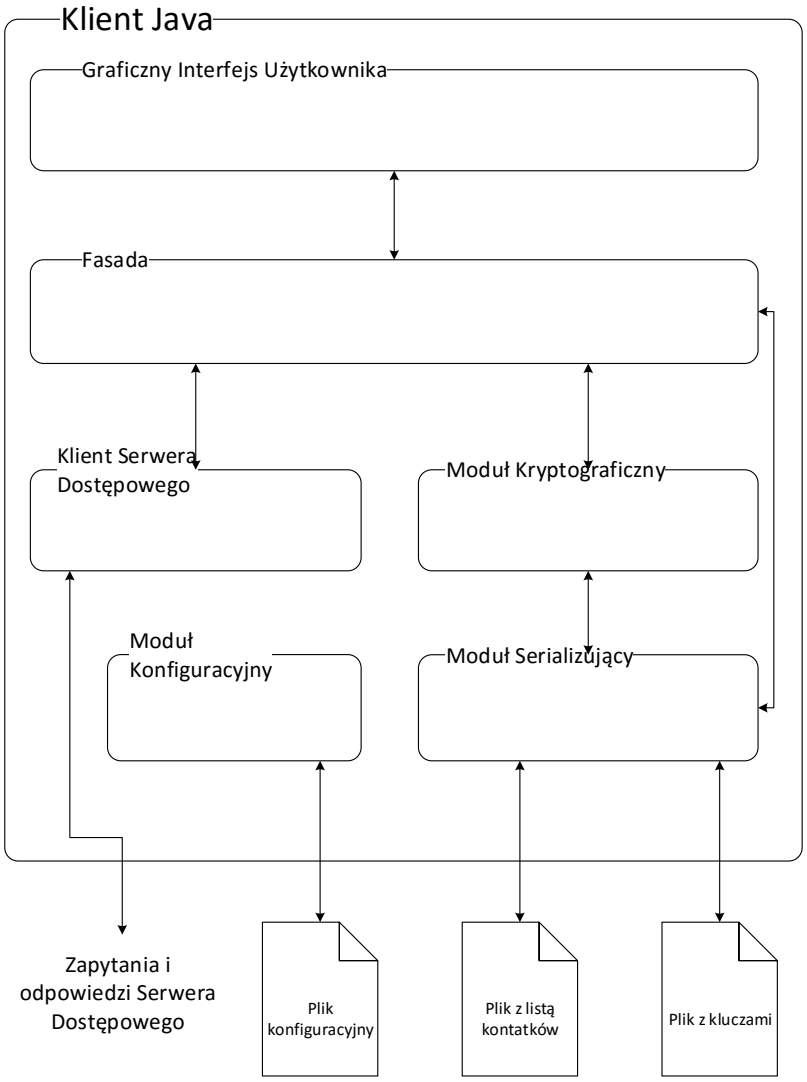
4.4.2 Diagram klas



Rysunek 21: Diagram wybranych klas mediatora.

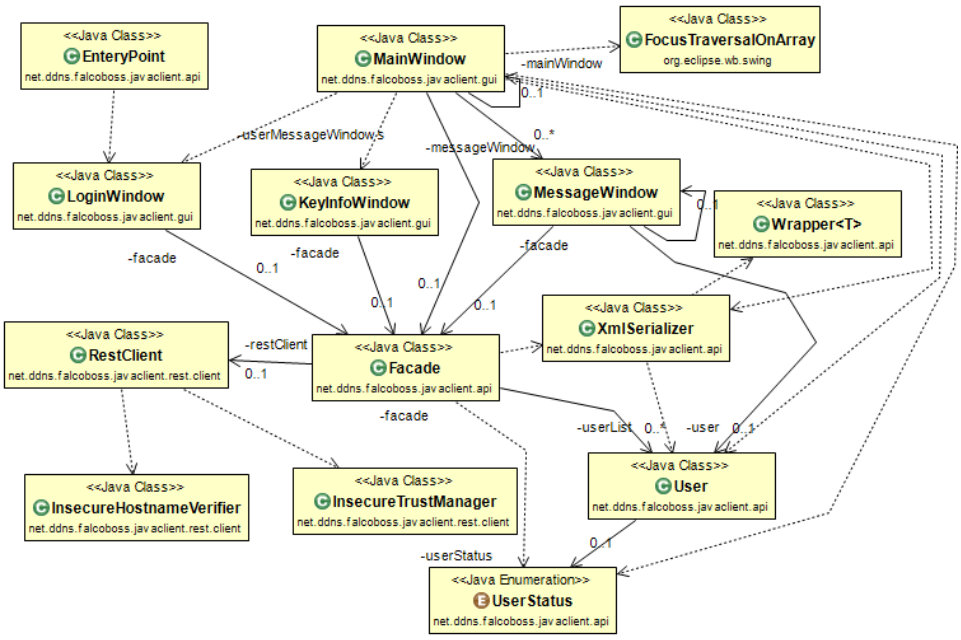
4.5 Projekt aplikacji klienta

4.5.1 Komponenty klienta

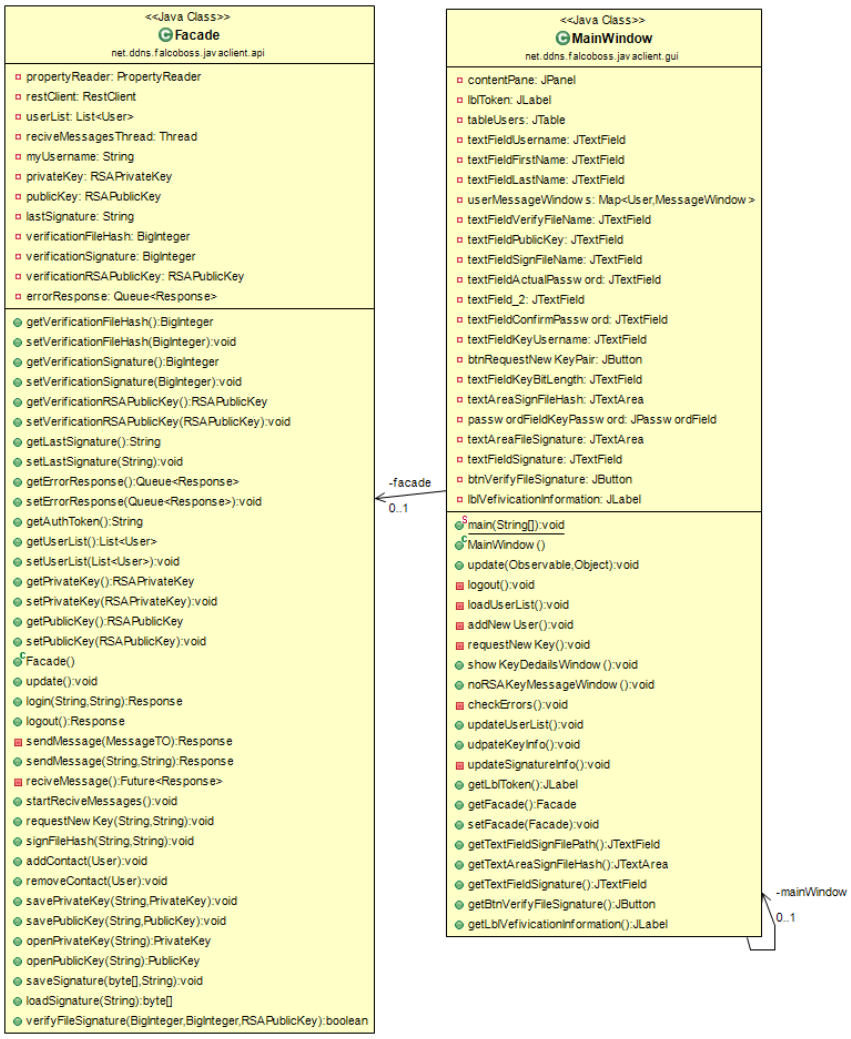


Rysunek 22: Interakcje między komponentami klienta.

4.5.2 Diagram klas



Rysunek 23: Uproszczony diagram klas klienta.



Rysunek 24: Diagram wybranych klas klienta.

5 Implementacja elementów systemu

Do zaimplementowana elementów systemu użyto zintegrowanego środowiska programistycznego `Eclipse Java EE Mars 4.5.1`. System składa się z następujących projektów:

- `common` – projekt zawierający wspólne klasy, klasy niektóre klasy pomocnicze i obiekty transportowe,
- `integration-test` – projekt zawierający testy integracyjne,
- `docs` – projekt zawierający dokumentację w formacie LaTeX, projekty diagramów klass, projekty Microsoft Visio oraz screeny z testów,
- `java-client` – projekt zawierający `aplikację klienta`,
- `java-client-admin` – projekt zawierający `apliikację administratora`,
- `mediator-server` – projekt zawierający aplikację `mediatora`,
- `registration-server` – projekt zawierający aplikację `serwera dostępowego`,
- `security-systems-ee` – projekt łączący wszystkie projekty, ułatwiający budowanie systemu.

Przy rozwijaniu projektu użyty został system kontroli wersji `Git`. Wszystkie kody źródłowe są dostępna pod adresem:

- `https://github.com/Michal-Lange/SecureSystemsEE`.

Projekty zostały budowane przy użyciu narzędzia `Apache Maven 3.3.3`. Konfiguracja projektów znajduje się w plikach `pom`. W plikach `pom` zdefiniowane zostały zależności pomiędzy projektami, zawarta konfiguracja projektów, określone niezbędne biblioteki. Na listingu 1 przedstawiony został przykładowy plik `pom` dla projektu `common`.

Źródła większości projektów podzielone są na 4 katalogi:

- `src/main/java` – katalog zawierający źródła, głównie pliki z rozszerzeniem java,
- `scr/test/java` – katalog zawierający testy jednostkowe dla danego projetku,
- `src/main/resources` – katalog zawierający zasoby aplikacji,
- `scr/test/resources` – katalog zawierający zasoby dla testów.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
      /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>net.ddns.falcoboss.common</groupId>
  <artifactId>common</artifactId>
  <name>Common</name>
  <description>Common Classes</description>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <parent>
    <groupId>net.ddns.falcoboss.enterprise.systems.ee</groupId>
    <artifactId>security-systems-ee</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../SecuritySystemsEE</relativePath>
  </parent>
</project>
```

Listing 1: Plik `pom.xml` dla projektu `common`.

5.1 Implementacja serwera dostępowego

5.1.1 Realizacja modułu ...

5.1.2 Realizacja filtrów zapytań

```
1  package net.ddns.falcoboss.registrationserver.security;
2  import java.io.IOException;
3  import java.util.logging.Logger;
4  import javax.ejb.EJB;
5  import javax.ejb.Local;
6  import javax.ejb.Stateless;
7  import javax.ws.rs.container.ContainerRequestContext;
8  import javax.ws.rs.container.ContainerRequestFilter;
9  import javax.ws.rs.container.PreMatching;
10 import javax.ws.rs.core.Response;
11 import javax.ws.rs.ext.Provider;
12
13 import net.ddns.falcoboss.common.HTTPHeaderNames;
14
15 @Provider
16 @PreMatching
17 @Local
18 @Stateless
19 public class RequestFilter implements ContainerRequestFilter {
20
21     private final static Logger log =
22 Logger.getLogger(RequestFilter.class.getName());
23
24     @EJB
25     AuthenticatorBean authenticatorBean;
26
27     @Override
28     public void filter(ContainerRequestContext requestCtx) throws IOException {
29         String path = requestCtx.getUriInfo().getPath();
30         log.info( "Filtering␣request␣path:␣" + path );
31         String serviceKey =
32 requestCtx.getHeaderString(HTTPHeaderNames.SERVICE_KEY);
33
34         if (!authenticatorBean.isServiceKeyValid(serviceKey)) {
35             requestCtx.abortWith(Response.status(
36 Response.Status.UNAUTHORIZED).build());
37             return;
38         }
39
40         if (!path.startsWith( "service/login/" )) {
41             String authToken =
42 requestCtx.getHeaderString(HTTPHeaderNames.AUTH_TOKEN);
43             if (!authenticatorBean.isAuthTokenValid(serviceKey, authToken)) {
44                 requestCtx.abortWith(Response.status(
45 Response.Status.UNAUTHORIZED).build());
46             }
47         }
48     }
49 }
```

Listing 2: RequestFilter.java.

5.1.3 Filtr odpowiedzi

```
1  package net.ddns.falcoboss.registrationserver.security;
2  import java.io.IOException;
3  import java.util.logging.Logger;
4  import javax.ws.rs.container.ContainerRequestContext;
5  import javax.ws.rs.container.ContainerResponseContext;
6  import javax.ws.rs.container.ContainerResponseFilter;
7  import javax.ws.rs.container.PreMatching;
8  import javax.ws.rs.ext.Provider;
9  import net.ddns.falcoboss.common.HTTPHeaderNames;
10
11 @Provider
12 @PreMatching
13 public class ResponseFilter implements ContainerResponseFilter {
14
15     private final static Logger log = Logger.getLogger(ResponseFilter.class.getName
           ());
16
17     @Override
18     public void filter(ContainerRequestContext requestContext,
19         ContainerResponseContext responseContext) throws IOException {
20
21         log.info("Filtering␣REST␣Response");
22
23         responseContext.getHeaders().add("Access-Control-Allow-Origin", "*");
24         responseContext.getHeaders().add("Access-Control-Allow-Credentials", "true"
               );
25         responseContext.getHeaders().add("Access-Control-Allow-Methods", "GET,␣POST
               ,␣DELETE,␣PUT");
26         responseContext.getHeaders().add("Access-Control-Allow-Headers",
               HTTPHeaderNames.
27         SERVICE_KEY + ",␣" + HTTPHeaderNames.AUTH_TOKEN);
28     }
29 }
```

Listing 3: RequestFilter.java.

6 Testowanie i ocena jakości systemu

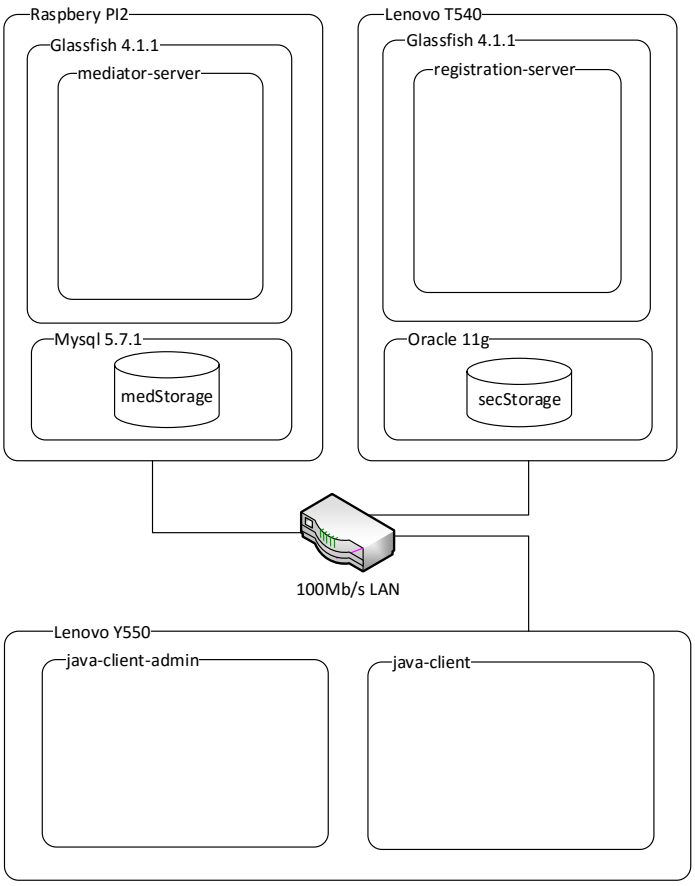
6.1 Instalowanie i konfiguracja systemu

6.1.1 Środowisko testowe

Środowisko testowe składa się z 3 komputerów.

- `Rasspbery PI 2`, 4-rdzeniowy procesor ARM Cortex-A7 900MHz, 1GB RAM;
- `Lenovo T540`, 4-rdzeniowy procesor i7-4700MQ 2,4GHz, 8GB RAM;
- `Lenovo Y550`, 2-rdzeniowy procesor Core 2 Duo T6500 2,1GHZ, 4GB RAM;

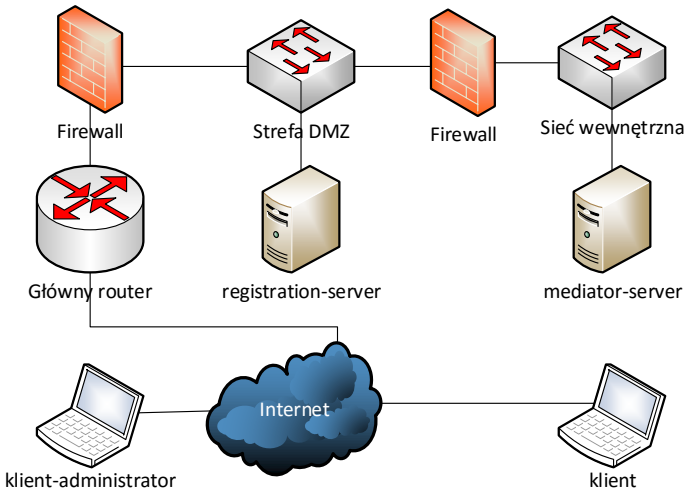
`Mediator` uruchomiony został na `Rasspbery PI2` z baz danych `MySql 5.7`. `Serwer dostępowy` uruchomiony został na `Lenovo T540` z baz danych `Oracle 11g`. `Klient` i `Klient administrator` działaj na `Lenovo Y550`. `Mediator` i `Serwer dostępowy` uruchamiane są wewnątrz kontenerów hostowanych przez serwer aplikacji `Glassfiash 4.1.1 Full Platform`. Na wszystkich komputerach zainstalowana została Maszyna wirtualna Javy `JVM 1.8.0_60-b27`. Komputery połączone są za pomocą sieci LAN o maksymalnej przepustowości 100MB/s. Domyślna biblioteka serwera `Glassfiash 4.1.1 Full Platform` odpowiadająca za persystencje obiektów `EclipseLink 2.6.1-RC1` zawiera błąd i uniemożliwia poprawne działanie z bazą danych, dlatego wymagana jest aktualizacja biblioteki do wersji 2.6.2. Na rysunku 25 przedstawiona została konfiguracja testowa systemu.



Rysunek 25: Środowisko testowe.

Przy wdrażaniu systemu w środowisku produkcyjnym w celu zapewnienia wydajności oraz bezpieczeństwa należy spełnić dodadkowe warunki:

- serwery i odpowiadające im bazy danych powinny znajdować się na tym samym hoście;
- `mediator` powinien znajdować się w bezpiecznej sieci wewnętrznej i być nieosiągalny z zewnątrz, a osiągalny jedynie dla `serwera dostępowego`;
- dla `serwera dostępowego` powinien zostać utworzony `certyfikat`, a zapytania powinny być obsługiwane jedynie bezpiecznym kanałem za pomocą protokołu `HTTPS`;
- `serwera dostępowy` powinien być osiągalny z zewnątrz np. sieci internet, ale jednocześnie być chroniony firewallem, powinien znajdować się w tzw. strefie zdemilitaryzowanej (DMZ).



Rysunek 26: Przykładowe środowisko produkcyjne.

6.1.2 przygotowanie bazy danych

Na zainstalowaną bazę danych należy się zalogować, utworzyć `użytkownika` oraz dodać mu prawa dostępowe do bazy. Na listingu 4 przedstawio utworzenie `użytkownika` dla `serwera dostępowego` oraz nadanie mu odpowiednich praw. Analogicznie należy utworzyć `użytkownika` dla mediatora.

```
1  -- połączenie z systemem bazodanowym jako root
2  mysql --host=192.168.1.20 --user=root -p
3
4  -- usuwanie bazy danych jeżeli istnieje
5  mysql> DROP DATABASE IF EXISTS secstorage;
6
7  -- tworzenie bazy danych
8  mysql> CREATE DATABASE secstorage;
9
10 -- polaczenie z utworzoną bazą danych
11 mysal> use secstorage;
12
13 -- usuwanie użytkownika
14 mysql> DROP USER 'mainStorage'@'localhost';
15
16 -- utworzenie użytkownika
17 mysql> CREATE USER 'mainStorage'@'localhost' IDENTIFIED BY 'registrationServer1';
18
19 -- nadanie utworzonemu użytkownikowi wszystkich praw do utworzonej bazy
20 mysql> GRANT ALL PRIVILEGES ON *.* TO 'mainStorage'@'localhost';
```

Listing 4: Tworzenie bazy danych i użytkownika bazy danych

6.1.3 Konfiguracja serwera Java EE

Połączenie do bazy danych konfigurowane jest w serwerze Java EE i dodawane do puli połączeń. Udostępnienie połączenia dla aplikacji następuje poprzez zdefiniowanie zasobu dla puli. Do poprawnego działania systemu należy utworzyć pule oraz zasoby dla bazy danych `serwera dostępowego` i `mediatora`. Na listingu 5 Przedstawione zostało utworzenie puli i zasobu dla `serwera dostępowego`. Analogiczną konfigurację należy przeprowadzić dla `mediatora`.

```
1
2  asadmin create-jdbc-connection-pool
3  --datasourceclassname=com.mysql.jdbc.jdbc2.optional.MysqlDataSource
4  --restype=javax.sql.DataSource
5  --property="user=mainStorage:password=registrationServer1:url=
6  jdbc\:mysql\://192.168.1.20\:3306/secstorage" mysqlRegPool
7
8  asadmin create-jdbc-resource
9  --connectionpoolid mysqlRegPool jdbc/secstorage
```

Listing 5: Tworzenie zasobu JDBC.

6.2 Realizacja testów

W celu zapewnienia jakości systemu utworzonych zostało szereg testów:

- jednostkowych - testy automatyczne,
- integracyjnych - testy automatyczne,
- akceptacyjnych - testy manualne,
- wydajnościowych - testy automatyczne,
- bezpieczeństwa - testy automatyczne i manualne,

Wszystkie testy automatyczne zostały napisane przy pomocy biblioteki `Junit 4`. Testy matualne zostały wykonane ręczne, a ich wyniki zostały udokumentowane w postaci screenów z programu. Przedstawione zostaną jedynie wybrane metody testujące.

6.2.1 Testy jednostkowe

W celu przetestowania pojedynczych modułów systemu utworzony został szereg testów jednostkowych. Każdy projekt posiada własne testy jednostkowe testujące jego własne moduły. Do realizacji testów jednostkowych została użyta biblioteka `Junit 4`. Wszystkie testy na wszystkich elementach systemu przechodzą pomyślnie.

Krytycznym elementem dla działania systemu jest podział wygenerowanego `klucza prywatnego` na `klucz prywatny użytkownika` oraz `klucz prywatny mediatora`. Na listingu 6 przedstawiona została metoda testująca powyższą procedurę. W metodzie generowana jest `para kluczy`, następnie wywoływana jest metoda, która jako parametry dostaje `klucz serwisoy`, długość bitową `klucza publicznego`, wartości `delta` oraz `własny klucz mediatora`. Wartość `delta` oraz `własny klucz mediatora` zostały zakodowane w metodzie testującej. Metoda ta zwraca `klucz prywatny mediatora`. Sprawdzane jest czy długość bitowa wygenerowanego `klucza prywatnego mediatora` jest dłuższa od długości `klucza publicznego`. W kolejnych krokach wyciągane są liczby pierwsze z klucza prywatnego oraz liczona wartość fi. Następnie od `klucza prywatnego` wygenerowanego na początku odejmowany jest modulo `fi klucz prywatny mediatora` a wynik zapisywany jest jako `klucz prywatny użytkownika`. Na końcu `klucz prywatny mediatora` i `klucz prywatny użytkownika` są dodawane modulo `fi`. następnie wykonywane jest porównanie otrzymanej sumy do wygenerowanego na początku wartości `klucza prywatnego`.
Test wykonuje się pomyślnie.

```
1 package net.ddns.falcoboss.mediatorserver.service.signature;
2
3 \\Import bibliotek
4 import java.math.BigInteger;
5 import java.security.InvalidKeyException;
6 import java.security.KeyFactory;
```

```
7  import java.security.KeyPair;
8  import java.security.NoSuchAlgorithmException;
9  import java.security.PrivateKey;
10 import java.security.PublicKey;
11 import java.security.SignatureException;
12 import java.security.interfaces.RSAPrivateKey;
13 import java.security.interfaces.RSAPublicKey;
14 import java.security.spec.InvalidKeySpecException;
15 import java.security.spec.RSAPrivateCrtKeySpec;
16 import org.junit.Assert;
17 import org.junit.Before;
18 import org.junit.Test;
19 import net.ddns.falcoboss.common.cryptography.KeyHelper;
20 import net.ddns.falcoboss.common.cryptography.PublicKeyCryptography;
21
22 public class PartKeyGeneratorTest {
23   public final static int delta = 120;
24   KeyPair newKeyPair = null;
25   PrivateKey mediatorOwnKey = null;
26
27   \\Metoda dostarczająca klucze.
28   @Before
29   public void generateKey() throws Exception {
30
31     \\Generowanie klucza do podziału.
32     newKeyPair = PublicKeyCryptography.createKeyPair();
33
34     \\Wartość moduł klucza własnego mediatora.
35     String mediatorOwnModulus = "APGBsXYWAjLQewbQWxmoyQaY+EllH1MdXoGVUW0rmfP+
           g8XMw2AqAXgW5NHZK6dwIZ0CVzkBDf0M8JMWMDqUsGho0H+N1q+
           NIiwADmg061AEoS2rL9Jm5ulm7VPmShnoRuNog7s+
           PR3F8xSJMGbHctBQ1gRdqGst3q9NsehLdmJ731YiNCMhrC/X4TRgbudCL0CDVJ+
           J6mntf11HNzXO+c5EvAmyaf4zJDWIU0veosOCK6EMcFYv5HQWkmwEcIT5DdKdM8af10tMx36uC3
           +UlXYQDWHd96OupWRBLStyZIWcHJqWPrrf2GsJf5MoYt1ZHmZZQ4ee2d9vJA+zAzt6qjc=";
36
37     \\Wartość eksponenty klucza własnego mediatora.
38     String mediatorOwnPrivateExponent = "VBXV1cl/5
           nVUAGFW9q4fn95uxA8jQur81p1Ihnwh1CQPeTT76WV2sXs3HCFC479U1LfV6pEFb8+
           ri2q0TBEtAo1L2r1lvCWlejBi08FpFKkn/SCXO+h8CVO+2fFaZ37J/6+J/
           g2DdfRP2ByT75UN0p3yhf6d/wMvf1XL1ZdAlrThNuW/
           ju9JtLGlRnHLNStbofCwNV4c7NzmPT7KCSlgrDRfXQX7XLx+D+hFpZPWypErV+UlgRM0x2cfp+
           K2gE6Y+tYCBXLsZZ3I3q+bTDwmiOte+2sn0uop7HuC7CxvK+
           wUxqcOgkOMTWy2oA4U7OrtQXCL6hAHya/DlEGmCyH0UQ==";
39
40     \\Własny klucz prywatny mediatora.
41     mediatorOwnKey = KeyHelper.getPrivateKeyFromBase64ExponentAndModulus(
           mediatorOwnPrivateExponent,
42         mediatorOwnModulus);
43   }
44
45   \\Metoda testująca.
46   @Test
47   public void testKeySplit()
48       throws NoSuchAlgorithmException,
49       InvalidKeySpecException, InvalidKeyException, SignatureException {
50
51     PublicKey commonPublicKey = newKeyPair.getPublic();
52     RSAPrivateKey commonPrivateKey = (RSAPrivateKey) newKeyPair.getPrivate();
53     BigInteger modulus = ((RSAPublicKey) commonPublicKey).getModulus();
54     BigInteger commonPrivateExponent = commonPrivateKey.getPrivateExponent();
55
56     //Wywoływanie metody generującej klucz prywatny mediatora.
57     BigInteger mediatorPrivateExponent =
58     PartKeyGenerator.generateFinalizationKeyExponent("testServiceKey",
```

```
59      modulus.bitLength(), delta, mediatorOwnKey);
60
61      //Sprawdzanie długości wygenerowanego klucza.
62      Assert.assertTrue(modulus.bitLength() < mediatorPrivateExponent.bitLength());
63
64      //Wyciąganie liczb pierwszych z klucza prywatnego.
65      KeyFactory keyFactory = KeyFactory.getInstance("RSA");
66      RSAPrivateCrtKeySpec pkSpec = keyFactory.getKeySpec(commonPrivateKey,
67      RSAPrivateCrtKeySpec.class);
68      BigInteger pMinusOne = pkSpec.getPrimeP().subtract(BigInteger.ONE);
69      BigInteger qMinusOne = pkSpec.getPrimeQ().subtract(BigInteger.ONE);
70
71      //Obliczanie wartości fi.
72      BigInteger fi = (pMinusOne.multiply(qMinusOne));
73
74      //Składanie klucza.
75      BigInteger userPrivateExponent =
76      (commonPrivateExponent.subtract(mediatorPrivateExponent)).mod(fi);
77
78      //Porównanie wartości złożonego klucza do klucza wygenerowanego na początku.
79      Assert.assertEquals(commonPrivateKey.getPrivateExponent(),
80          (userPrivateExponent.add(mediatorPrivateExponent)).mod(fi));
81    }
82  }
```

Listing 6: PartKeyGeneratorTest.java.

Kolejna metoda testuje podpisywanie danych przy pomocy częściowych kluczy. Początek metody nie różni się niczym od metody testującej generowanie klucza częściowego przedstawionej na listingu 6. W dalszej części metody przedstawionej na listingu 7 od linii 80 tworzony jest `skrót SHA512` z pola tekstowego. `Skrót` ten jest następnie podpisywany przy pomocy `klucza prywatnego` wygenerowanego na początku. Następnie `skrót` jest podpisywany osobno przy pomocy `klucza prywatnego użytkownika` i `klucz prywatny mediatora`. W kolejnym etapie dwa ostanie `podpisy` składane są w jeden przy pomocy operacji mnożenia modulo. Na samym końcu następuje test złożonego `podpisu`, który polega na porównaniu go do podpisu złożonego przy pomocy `klucza prywatnego` przed podziałem. Test kończy się sukcesem.

```
80      //Utworzenie skrótu z tekstu.
81      byte[] message =
82      SHA512.hashText("RSA␣Signature␣Test␣String").getBytes();
83      BigInteger bigIntegerFileHash =
84      new BigInteger(1,message);
85
86      //Podpis przy pomocy klucza przed podziałem.
87      BigInteger commonSignatureB =
88      bigIntegerFileHash.modPow(commonPrivateExponent, modulus);
89
90      //Podpis przy pomocy klucza prywatnego użytkownika.
91      BigInteger userSignatureB =
92      bigIntegerFileHash.modPow(userPrivateExponent, modulus);
93
94      //Podpis przy pomocy klucza prywatnego mediatora.
95      BigInteger mediatorSignatureB =
96      bigIntegerFileHash.modPow(mediatorPrivateExponent, modulus);
97
```

```
98        //Porównanie złożonego podpisu
99        //do podpisu złożonego przy pomocy niepodzielonego klucza prywatnego.
100       Assert.assertEquals(commonSignatureB,
101       (userSignatureB.multiply(mediatorSignatureB)).mod(modulus));
102     }
103 }
```

Listing 7: SignatureAssemblyTest.java.

6.2.2 Testy integracyjne

W celu wykrycia błędów w interfejsach oraz zapewnienia bezbłędnej komunikacji pomiędzy modułami, utworzonych zostało szereg testów integracyjnych. Aby testy automatyczne zakończyły się pomyślnie, system musi zostać uruchomiony oraz porawnie skonfigurowany. Wszystkie testy integracyjne znajdują się w projekcie `integration-test`.

Na listingu 8 przedstawiona została klasa abstrakcyjna, która służy jako baza do testów interfejsu `serwera dostępowego`. Klasa odpowiedzialna jest za stworzenie i zainicjowanie obiektu `klienta serwisu REST` oraz skonfigurowanie biblioteki odpowiedzialnej za serjalizację obiektów do formatu JSON. Klasa ustawia klientowi adres `serwera dostępowego` oraz przechowuje obliczone skróty haseł testowych.

```
1  package net.ddns.falcoboss.integration.test.messanger;
2
3  import javax.ws.rs.client.Client;
4  import javax.ws.rs.client.ClientBuilder;
5  import javax.ws.rs.client.WebTarget;
6
7  import org.eclipse.persistence.jaxb.rs.MOXyJsonProvider;
8  import org.glassfish.jersey.client.ClientConfig;
9  import org.json.JSONObject;
10 import org.junit.Before;
11
12 import net.ddns.falcoboss.common.cryptography.SHA512;
13
14 public abstract class AbstractConnectionTest {
15     Client client;
16     WebTarget webTarget;
17     JSONObject authToken;
18
19   String password1Hash;
20   String password2Hash;
21
22     @Before
23     public void initClient() throws Exception {
24       password1Hash = SHA512.hashText("password1");
25       password2Hash = SHA512.hashText("password2");
26
27     ClientConfig cc = new ClientConfig();
28     cc.register(MOXyJsonProvider.class);
29
30         this.client = ClientBuilder.newClient(cc);
31         this.webTarget =
32 this.client.target(
33 "http://localhost:8080/registration-server/rest/service/");
34     }
35 }
```

Listing 8: PartKeyGeneratorTest.java.

Z klasy z listingu 8 dziedziczy klasa testowa przedstawiona na listingu 9. Klasa ta testuje proces logowania oraz wylogowywania z systemu symulując `klienta`. Przy logowaniu porównywany jest kod odpowiedzi serwera do wartości 200 (oznacza to poprawne zalogowanie). Następnie sprawdzany jest token. Przy wylogowywaniu sprawdzany jest jedynie kod odpowiedzi, wartość 204 odpowiada poprawnemu wylogowaniu się z serwra. Test kończy się sukcesem.

```
1  package net.ddns.falcoboss.integration.test.messanger;
2
3
4  import javax.ws.rs.client.Entity;
5  import javax.ws.rs.core.MediaType;
6  import javax.ws.rs.core.Response;
7
8  import org.json.JSONObject;
9  import org.junit.After;
10 import org.junit.Assert;
11 import org.junit.Test;
12
13 import net.ddns.falcoboss.common.cryptography.SHA512;
14 import net.ddns.falcoboss.common.transport.objects.HTTPHeaderNames;
15 import net.ddns.falcoboss.common.transport.objects.UsernameAndPasswordTO;
16
17 public class ServiceRegisterLoginLogoutTest extends AbstractConnectionTest{
18   private String serviceKey = "f80ebc87-ad5c-4b29-9366-5359768df5a1";
19
20     @Test
21     public void testConnection() throws Exception{
22       String passwordHash = SHA512.hashText("password1");
23       UsernameAndPasswordTO usernameAndPasswordBean = new UsernameAndPasswordTO("
             username1",passwordHash);
24       Response response = webTarget.path("login/").request().header(HTTPHeaderNames
             .SERVICE_KEY, serviceKey).
25       accept(MediaType.APPLICATION_JSON_TYPE).post(Entity.entity(
             usernameAndPasswordBean, MediaType.APPLICATION_JSON_TYPE));
26         Assert.assertEquals(200, response.getStatus());
27         authToken = new JSONObject(response.readEntity(String.class));
28         Assert.assertNotNull(authToken.get("auth_token"));
29     }
30
31     @After
32     public void logout(){
33       Response response = webTarget.path("logout/").request().header(
             HTTPHeaderNames.SERVICE_KEY, serviceKey).header(HTTPHeaderNames.
             AUTH_TOKEN, authToken.get("auth_token")).post(null);
34       Assert.assertEquals(204, response.getStatus());
35     }
36 }
```

Listing 9: ServiceRegisterLoginLogoutTest.java.

6.2.3 Testy akceptacyjne

Wszystkie testy akceptacyjne zostały przeprowadzone manualnie oraz udokumentowane w postaci zrzutu ekranu aplikacji. Przetestowane zostały wybrane przypadki

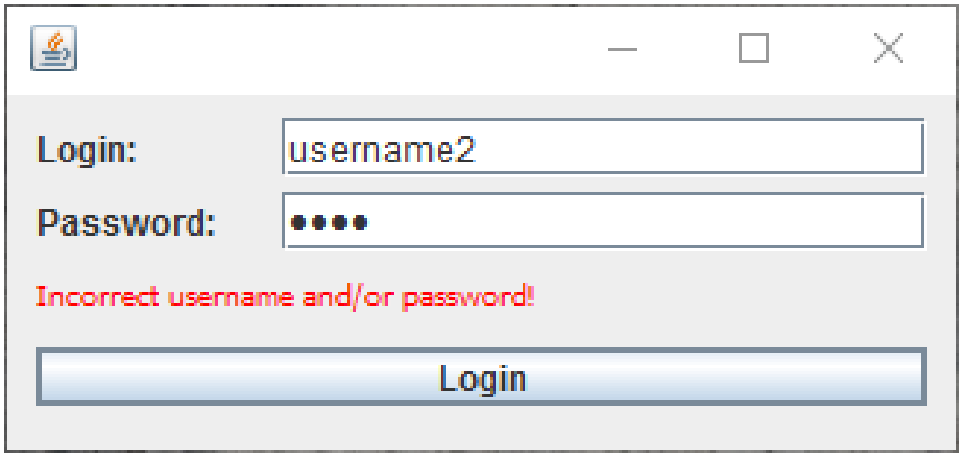
użycia.

Po uruchomieniu programu wyświetlone zostaje okno logowania z przyciskiem `Zaloguj`. Pomyślne zalogowanie do systemu następuje w przypadku podania prawidłowego `loginu` i `hasła` oraz poprawnej konfiguracji pliku `property.config`.

W następujących przypadkach nie ma możliwości zalogowania się, w oknie logowania zostaje wyświetlony komunikat z błędem:

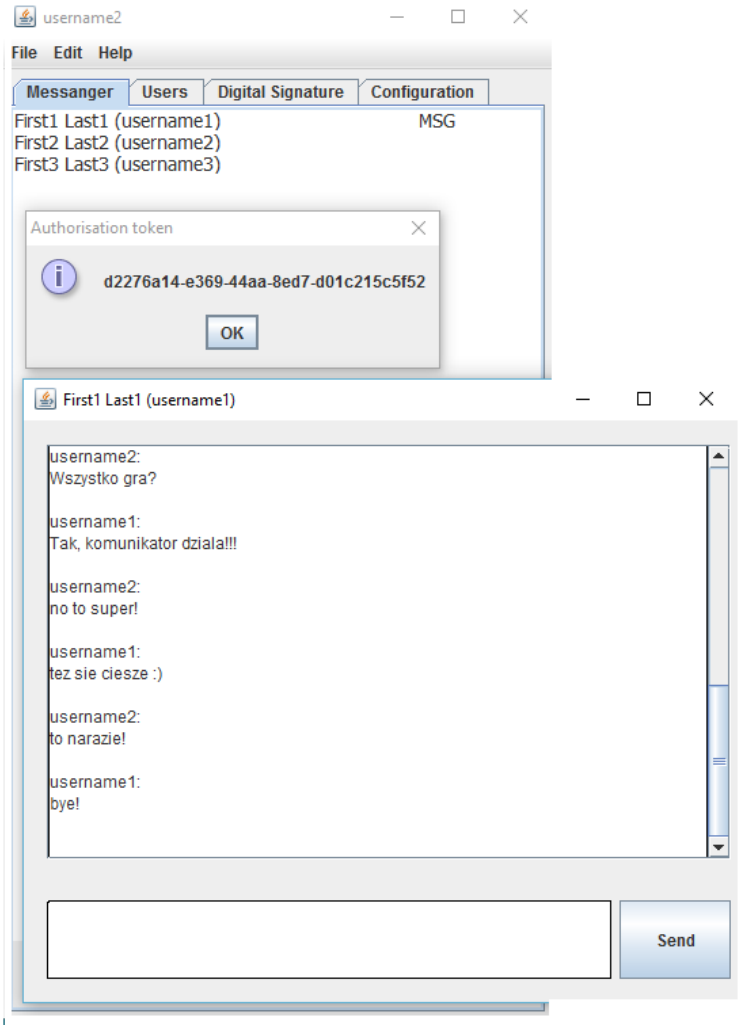
- brak `loginu` lub `hasła`,
- niepoprawny `login` i `hasło`,
- niepoprawny adres serwisu w pliku `property.config`,
- niepoprawny `klucz serwisowy` lub jego brak.

Na rysunku 27 przedstawione zostało okno aplikacj `klient`, po próbie zalogowania ze złym `loginem` lub `hasłem`, albo ze `kluczem serwisowym`.



Rysunek 27: Komunikat w przypadku błędnego loginu, hasła lub klucza serwisowego.

Po poprawnej autoryzacji aplikacja `klient` wyświetala okno z `listą kontaktów`. Po kliknięciu na `kontakt` możemy rozpocząć komunikacje tekstową. Na rysunku 28 przedstawione zostało główne okno programu, na nim okno wiadomości z aktywną rozmową, oraz okno z informacją o `tokenie`



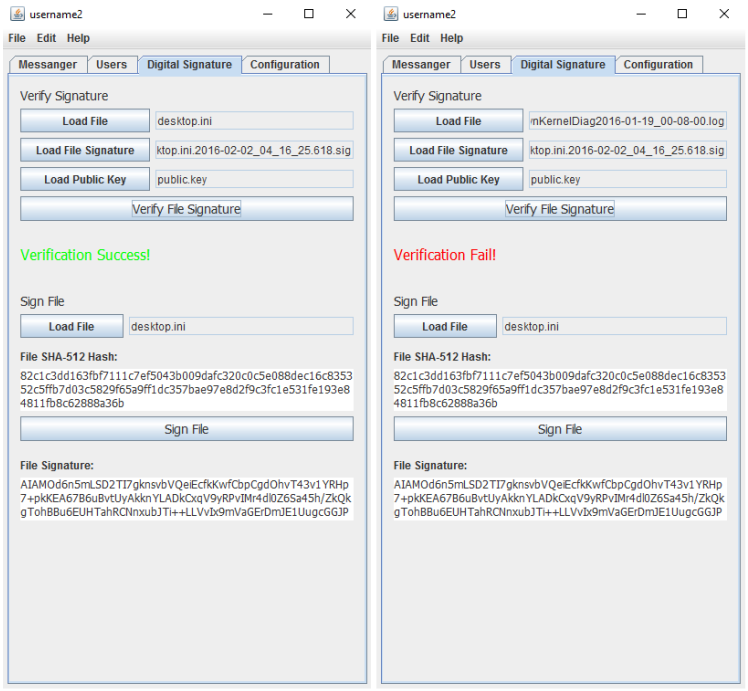
Rysunek 28: Główne okno programu, okno wiadomości i okno tokena.

Kolejny przeprowadzony test sprawdza możliwość składania oraz weryfikację podpisu cyfrowego. W zakładce `podpis cyfrowy` po wczytaniu pliku w okienu przedstawiony zostanie skrót pliku w postaci szestnastkowej. Następnie po kliknięciu przycisku `podpisz plik` system w sposób rozproszony wygeneruje podpis, który po chwili pojawia się w polu `podpis plku` zakodowany w postaci Base64. W katalogu głównym programu podpis zostaje automatycznie zapisany do plku, nazwa pliku generowana jest według następującego schematu:

- `<nazwa_wczytanego_plku>.<pieczątka_czasu>.sig`

Aby zweryfikować podpisany `plik` należy załadować go do programu wraz z jego `podpisem` oraz `kluczem publicznym` użytkownika, który go podpisał, a następnie kliknąć w przycisk `weryfikuj podpis`. W przypadku prawidłowego

`klucza publicznego`, `podpisu` oraz `pliku` wyświetlony zostaje komunikat o pomyślnej weryfikacji. W przypadku gdy `plik` został zmieniony, wczytany został niepoprawny `podpis` lub `klucz publiczny`, `klient` wyświetli informację o niepoprawnej weryfikacji. Na rysunku 29 przedstawione zostały obydwa przypadki.



Rysunek 29: Pomyślna oraz niepomyślna weryfikacja podpisu.

6.3 Ocena wydajności operacji generowania kluczy

System domyślnie generuje klucze 1024-bitowe. W środowiku produkcyjnym zalecane jest przejście do długości 2048-bitowej, taka długość klucza uważana jest powszechnie za bazpieczną i obecnie najczęściej stosowana. Wzrost mocy obliczeniowej komputerów powoduje, że w przyszłości standardem mogą stać się klucze 4096-bitowe. Na listingu 10 przedstawiony został test obliczający czasy generowania kluczy w zależności od długości bitowej klucza. Test oblicza średnią czasów z 6 prób. Wyniki testu zostały przedstawione na obrazku 30. Generowanie kluczy jest najbardziej zasobożerna operacja w całym systemie. W przypadku zażądania klucza w tym samym czasie przez większą ilość użytkowników może dojśc do znacznego spadku wydajności `serwera dostępowego`. Moduł do generowania kluczy stanowi osobną bibliotekę i może być w przyszłości łatwo wydzielony jako niezależy węzeł systemu.

```
1  @Test
2  public void testCreateKeyPairTimes() throws NoSuchAlgorithmException {
3     long timeStart;
4     long timeEnd;
5
6     long tt = 0;
7     for(int j=512; j<4097; j=j+512)
8     {
9       for(int i = 0;  i<6; i++){
10        try {
11          timeStart = System.currentTimeMillis();
12          KeyPairGenerator keygen = KeyPairGenerator.getInstance("RSA");
13          keygen.initialize(j);
14          keygen.generateKeyPair();
15          timeEnd = System.currentTimeMillis();
16          tt += (timeEnd - timeStart);
17        }
18        catch (Exception e) {
19          e.printStackTrace();
20        }
21      }
22      System.out.println("(" + j + "," + (tt/6) + ")");
23    }
24 }
```

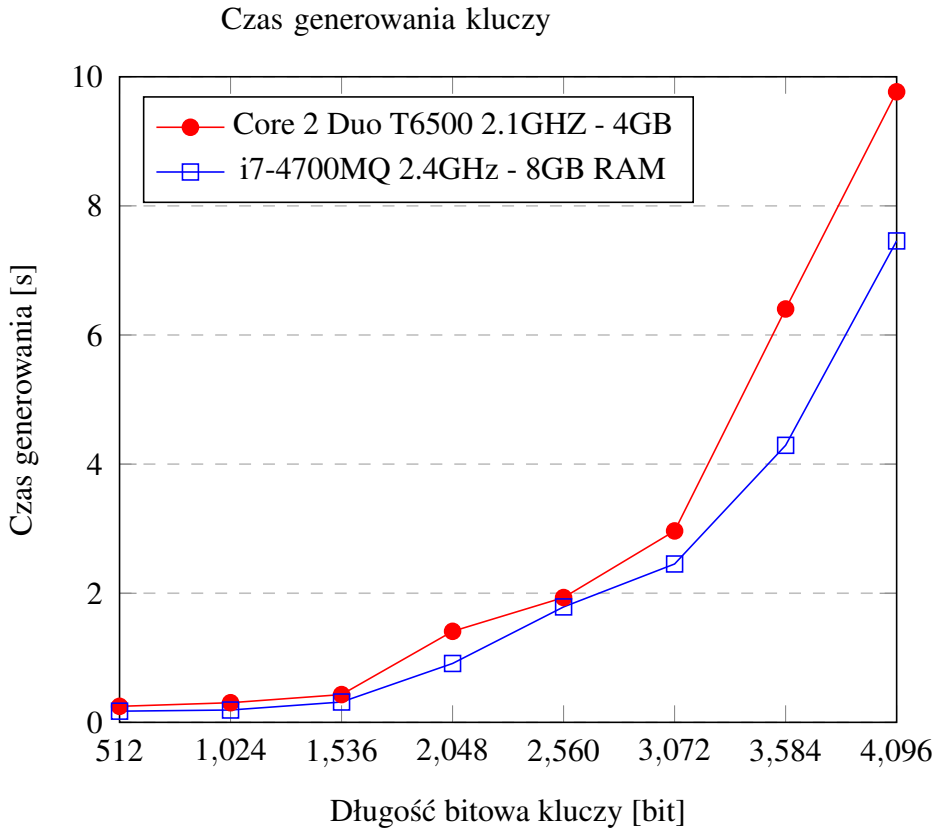
Listing 10: ServiceRegisterLoginLogoutTest.java.

6.4 Ocena wydajności transmisji danych

//TODO test wątki wysyłające losowe wiadomości.

6.5 Ocena zachowania w przypadku awarii

//TODO screeny z testów

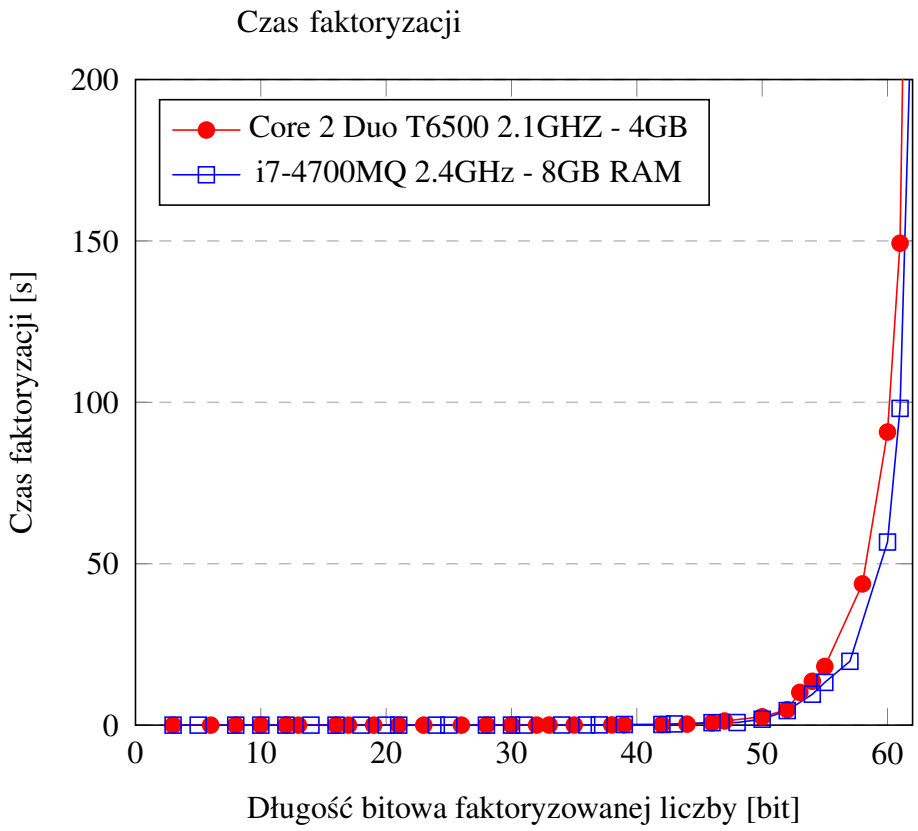


Rysunek 30: Czasy faktoryzacji liczb.

6.6 Ocena bezpieczeństwa

Bezpieczeństwo podpisu cyfrowego opiera się o trudność faktoryzacji, czyli rozkładu na czynniki pierwsze dużych liczb. Na listingu 11 przedstawiony został test realizujący to zadanie. Test w pętli generuje dwie liczby pierwsze, zaczynając od liczb 3 bitowych, następnie je mnoży, a otrzymany wynik mnożenia próbuje rozłożyć na czynniki pierwsze. Program liczy średnią czasów dla 6 prób. Wyniki zostały przedstawione na rysunku 31. Jak widać na rysunku czas rośnie bardzo szybko do w stosunku do długości bitowej liczby. Rozkład liczby 62-bitowej na komputerze 2 rdzeniowym trwa już ok 6 minut. Pojedyńcza próba faktoryzacji liczby 64bitowej trwała ponad godzinę. Rozłożenie liczby 1024-bitowej jest niemożliwe w realnym czasie. Zagrożeniem dla bezpieczeństwa algorytmu RSA mogą być komputery kwantowe.

```
1  package net.ddns.falcoboss.integration.test.factorization;
2
3  import java.math.BigInteger;
4  import java.util.LinkedList;
5  import java.util.Random;
6
7  import org.junit.Test;
8
9  public class FactorBigIntegerTest {
10
11   public static LinkedList<BigInteger> tdFactors(BigInteger n) {
12     BigInteger tmp = n;
13     BigInteger two = BigInteger.valueOf(2);
14     LinkedList<BigInteger> factors = new LinkedList<BigInteger>();
15     if (tmp.compareTo(two) < 0) {
16       throw new IllegalArgumentException("Liczba␣musi␣by␣wieksza␣niz␣1!");
17     }
18     while (tmp.mod(two).equals(BigInteger.ZERO)) {
19       factors.add(two);
20       tmp = tmp.divide(two);
21     }
22     if (tmp.compareTo(BigInteger.ONE) > 0) {
23       BigInteger f = BigInteger.valueOf(3);
24       while (f.multiply(f).compareTo(tmp) <= 0) {
25         if (tmp.mod(f).equals(BigInteger.ZERO)) {
26           factors.add(f);
27           tmp = tmp.divide(f);
28         } else {
29           f = f.add(two);
30         }
31       }
32       factors.add(tmp);
33     }
34     return factors;
35   }
36
37   @Test
38   public void testFactorization() {
39     BigInteger nx = BigInteger.valueOf(100);
40     BigInteger n;
41     long startTime;
42     long endTime;
43     long totalTime;
44     System.out.println("start...");
45     for (BigInteger i = BigInteger.valueOf(2); nx.compareTo(i) >
```



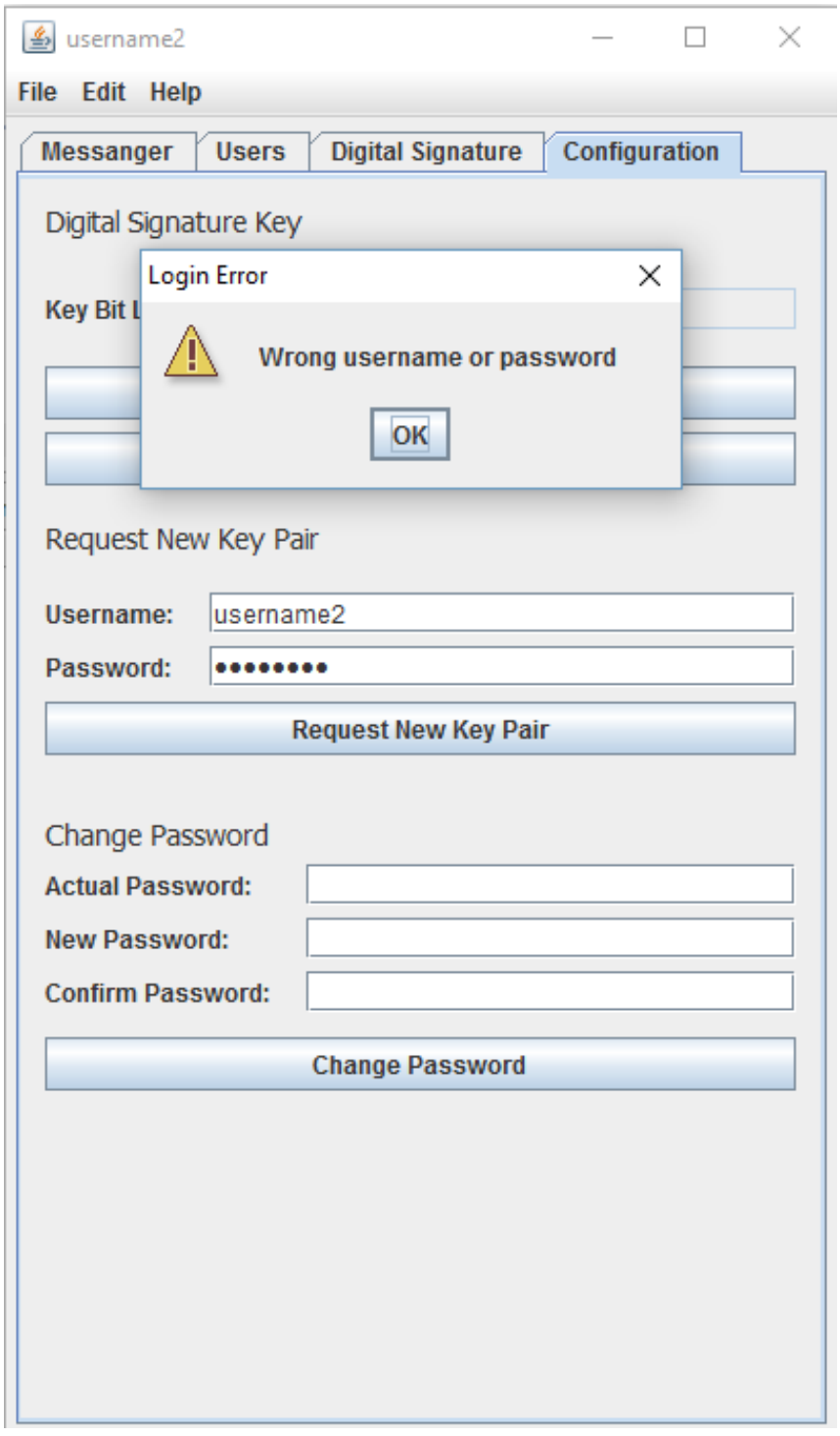
Rysunek 31: Czasy generowania kluczy.

```
46          0; i = i.add(BigInteger.valueOf(1))) {
47          long tt = 0;
48          for(int j=0; j<6; j++)
49          {
50            BigInteger p = BigInteger.probablePrime(i.intValue(), new Random());
51            BigInteger q = BigInteger.probablePrime(i.intValue(), new Random());
52            n = p.multiply(q);
53            startTime = System.currentTimeMillis();
54            LinkedList<BigInteger> lst = tdFactors(n);
55            endTime = System.currentTimeMillis();
56            tt += totalTime = endTime - startTime;
57          }
58          System.out.println("(" + n.bitLength() + "," + tt/6 + ")");
59        }
60      }
61  }
```

Listing 11: FactorBigIntegerTest.java.

Komunikacja między węzłami może być szyfrowana przy pomocy protokołu HTTPS, który opiera się na TLS, a ten korzysta również z kryptografi asymetrycznej. Wszystkie próby zalogowania się do systemu z nieprawidłowym `loginem` lub `hasłem` kończą się niepowodzeniem, dodadkowym elementem podnoszącym bezpieczeństwo jest `klucz serwisowy`. Generowanie `pary kluczy` to jedyny oprócz

logowania przypadek użycia wymagający dodadkowej walidacji. Również tam w przypadku nieprawidłowego `loginu` lub `hasła` wyświetlany jest komunikat z błędem, który przedsationy został na rysunku refwrongPasswordKey1. Komunikat wyświetlany jest na tle zakładki do `generowania pary kluczy`.



Rysunek 32: Generowanie klucza - złe hasło.

7 Podsumowanie

Utworzony system pozwala na swobodną komunikację pomiędzy użytkownikami, zapewnia też podstawową funkcjonalność podpisu cyfrowego. Bezpieczeństwo tego systemu podniesione jest dzięki dodadkowej stronie podpisującej. Nawet jeżeli osoba trzecia zdobędzie klucz prywatny użytkownika, to nie będzie w stanie podpisać nim dokumentów. Utworzony system może posłużyć jako baza do rozwoju bardziej zaawansowanych rozwiązań. ,dzięki modułowej budowie może być łatwo modyfikowany i rozszerzany.

//TODO dokonczyc

Literatura

[1] Arun Gupta. Java EE 6 Leksykon kieszonkowy, Wydawnictwo HELION, Gliwice, 2013. ISBN 978-83-246-6640-9.

[2] Eric Jendrock, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. Java EE 6 Przewodnik, Wydanie IV, Wydawnictwo HELION, Gliwice, 2012. ISBN 978-83-246-3847-5.

[3] Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Devika Gollapudi, Kim Haase, William Markito, Chinmayee Srivathsa. Java EE 6 Zaawansowany Przewodnik, Wydanie IV, Wydawnictwo HELION, Gliwice, 2012. ISBN 978-83-246-7393-3.

[4] Bhakti Mehta. REST Najlepsze praktyki i wzorce w języku Java, Wydawnictwo HELION, Gliwice, 2015. ISBN 978-83-283-0644-8.

[5] Andrew S. Tanenbaum. Sieci Komputerowe, Wydawnictwo HELION, Gliwice, 2004. ISBN 83-7361-557-1. Bezpieczeństwo w sieciach komputerowych, s.641-740.

[6] Bernd Bruegge, Allen H. Dutoit. Inżynieria oprogramowania w ujęciu obiektowym. UML, wzorce projektowe i Java, Wydawnictwo HELION, Gliwice, 2012. ISBN 978-83-246-2872-8.

[7] William Stallings. Kryptografia i bezpieczeństwo sieci komputerowych. Matematyka szyfrów i techniki kryptologii, Wydawnictwo HELION, Gliwice, 2010. ISBN 978-83-246-2986-2.

[8] Mirosław Kutylowski, Przemysław Kubiak. Mediated RSA cryptography specification for additive private key splitting (mRSAA), Internet Szkic, 2013 [dostęp 15 grudnia 2015] Dostępne w internecie: `https://tools.ietf.org/html/draft-kutylowski-mrsa-algorithm-03`

[9] Oracle® Security Developer Tools Reference, Źródło, 2006 [dostęp 15 grudnia 2015] Dostępne w internecie: `https://docs.oracle.com/cd/B28196_01/idmanage.1014/b28165/crypto.htm`

[10] Kryptografia asymetryczna i jej zastosowanie w algorytmach komunikacji, Źródło, 2006 [dostęp 29 grudnia 2015] Dostępne w internecie: `http://kis.pwszchelm.pl/publikacje/V/Bartyzel.pdf`

[11] Systemy rozproszone, Wykład, 2006 [dostęp 28 grudnia 2015] Dostępne w internecie: `http://wazniak.mimuw.edu.pl/index.php?title=Systemy_rozproszone`

[12] Zofia Kruczkiewicz. Modelowanie i analiza systemów informatycznych 1, Wykłady, 2014 [dostęp 28 grudnia 2015] Dostępne w internecie: http://zofia.kruczkiewicz.staff.iiar.pwr.wroc.pl/wyklady/analizasi/

[13] Java Platform, Enterprise Edition: The Java EE Tutorial, Źródło, 2014 [dostęp 15 grudnia 2015] Dostępne w internecie: https://docs.oracle.com/javaee/7/JEETT.pdf

[14] Filters and Handlers Revisited, Źródło, 2011 [dostęp 24 stycznia 2015] Dostępne w internecie: https://java.net/projects/jax-rs-spec/pages/PreMatchingFilters

[15] Jersey, Źródło, 2015, [dostęp 24 stycznia 2016] Dostępne w internecie: https://jersey.java.net/documentation/latest/index.html