

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

Bezpieczeństwo systemów i usług informatycznych

Laboratorium 2

Komunikator - klient

SPRAWOZDANIE

PROWADZĄCY:

dr inż. Dominik Żelazny

AUTORZY:

Mikołaj Szymczyk, 248881

Wrocław 12.11.2022

Spis treści

| | |
|--|----------|
| Spis treści | 2 |
| 1 Wstęp | 3 |
| 1.1 Cel i zakres laboratorium | 3 |
| 1.2 Harmonogram prac | 3 |
| 1.3 Technologie | 3 |
| 2 System weryfikacji serwera | 4 |
| 2.1 Ustalenie portu sesji | 4 |
| 2.2 HandShake | 4 |
| 3 Komunikacja | 7 |
| 3.1 Opis komunikacji pomiędzy klientem a serwerem | 7 |
| 3.2 Implementacja komunikacji w aplikacji klienckiej | 9 |
| 3.2.1 Opcje dla niezalogowanego użytkownika | 9 |
| 3.2.2 Opcje dla zalogowanego użytkownika | 10 |
| 3.3 Bazowe implementacja zapytania | 13 |

Rozdział 1

Wstęp

1.1 Cel i zakres laboratorium

Celem zadania na laboratorium jest stworzenie bezpiecznego komunikatora. Aby to osiągnąć zaprojektowano system weryfikacji połączenia pomiędzy klientem a serwerem oraz zaimplementowano komunikację po stronie klienta komunikatora.

1.2 Harmonogram prac

1. Ustalenie koncepcji weryfikacji połączenia oraz systemu komunikacji klienta i serwera.
2. Wstępna implementacja weryfikacji i komunikacji z serwerem.
3. Test komunikacji z serwerem.
4. Poprawienie implementacji weryfikacji oraz implementacja pozostałych akcji.

1.3 Technologie

Klient komunikatora został napisany w języku Java 17 z wykorzystaniem frameworka Spring. Komunikacja została zaimplementowana za pomocą gniazd (ang. *Socket's*)

Rozdział 2

System weryfikacji serwera

2.1 Ustalenie portu sesji

Klient łączący się z serwerem wysyła wiadomość powitalną na ustalony port serwera - 16123. Po czym klient odsyła port, na którym będzie odbywać się najpierw weryfikacja serwera, która jeżeli będzie pomyślna to również na tym porcie będzie przebiegać dalsza komunikacja. Odebranie nowego portu sesji można zaobserwować w logach działania aplikacji co przedstawiono m.in. na rysunkach 2.3 oraz 2.2. Mechanikę sesji wprowadzono aby umożliwić łączenie się kilku klientów do jednego serwera.

```
private void getSessionPort() {  
    //get session port  
    startConnection(ip, SERVER_PORT);  
    sessionPort = Integer.parseInt(sendMessage("Hi"));  
    LOGGER.info("Session port is: " + sessionPort);  
    stopConnection();  
}
```

Listing 2.1: Metoda pobierająca numer portu sesji

2.2 HandShake

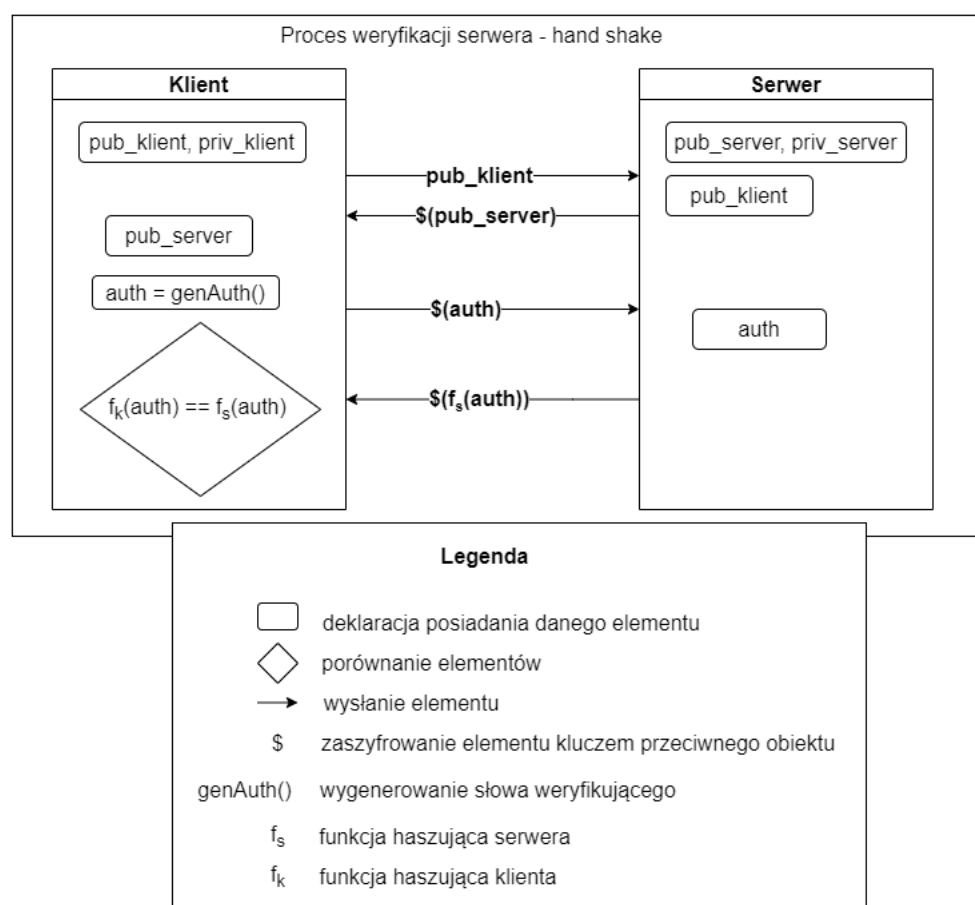
Następnie by zweryfikować serwer zaprojektowano proces weryfikacji serwera - *Hand Shake*. Proces ten zilustrowano na rysunku 2.1.

Przebieg działania tego procesu jest następujący. Klient oraz serwer generuje klucze RSA. Następnie klient wysyła swój klucz publiczny. Na co serwer odpowiada zaszyfrowanym przez klucz publiczny klienta klucz serwera. Klient odszyfrowuje wiadomość swoim kluczem prywat-

nym. Od tego momentu dalsza komunikacja podczas weryfikacji jest szyfrowana za pomocą klucza publicznego przeciwnej strony weryfikacji.

Kolejnym krokiem jest wygenerowanie przez klienta słowa weryfikacyjnego, które jest przesyłane do serwera. Po czym serwer odsyła zahaszowane połączenie słowa weryfikacyjnego i klucza publicznego klienta. Ostatnim krokiem weryfikacji serwera jest porównanie otrzymanego haszu z zahaszowanym połączeniem słowa weryfikacyjnego klienta i klucza publicznego klienta. Jeżeli hasze się nie zgadzają lub na to znaczy, że jest to fałszywy serwer i komunikacja jest przerywana. W przeciwnym przypadku klient generuje klucz szyfru AES oraz wysyła go do serwera. Od tego momentu procesu dalsza komunikacja jest szyfrowana kluczem AES.

Haszowanie odbywa się za pomocą funkcji haszującej SHA256.



Rysunek 2.1: Schemat procesu weryfikacji serwera

```

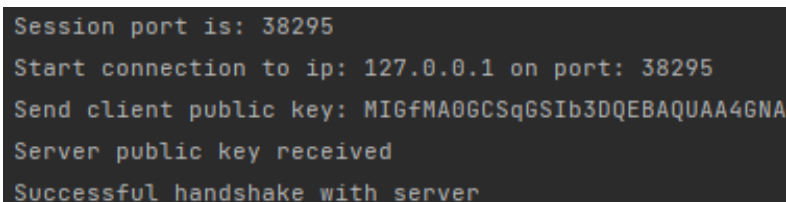
private void makeHandShake() throws Exception {
    //send public key and receive server encrypted public key
    var clientPublicKeyStr =
        cipherUtility.encodeKey(clientPublicKey);
    LOGGER.info("Send client public key: " + clientPublicKeyStr);
    Thread.sleep(1000);
}
  
```

```
//receive server public key
var receivedKey = sockets.sendMessage(clientPublicKeyStr);
LOGGER.info("Server public key received");
serverPublicKey = cipherUtility.decodePublicKey(
    cipherUtility.decrypt(receivedKey, clientPrivateKey));
LOGGER.debug("Server public key is: " + serverPublicKey);

//send authentication word
var auth = controlMessageGenerator(100);
var fAuth = cipherUtility.decrypt(
    sockets.sendMessage(cipherUtility.encrypt(auth,
        serverPublicKey)), clientPrivateKey);

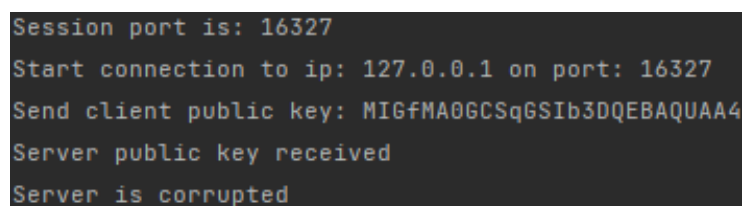
//check if it is good server
var encodedHash = Hashing.sha256().hashString(
    auth + cipherUtility.encodeKey(clientPublicKey),
    StandardCharsets.UTF_8).toString();
if (!fAuth.equals(encodedHash)) {
    sockets.stopConnection();
    LOGGER.error("Server is corrupted");
    throw new RuntimeException("Server is corrupted");
}
LOGGER.info("Successful handshake with server");
}
```

Listing 2.2: Implementacja weryfikacji serwera



```
Session port is: 38295
Start connection to ip: 127.0.0.1 on port: 38295
Send client public key: MIGfMA0GCSqGSIB3DQEBAQUAA4GNA
Server public key received
Successful handshake with server
```

Rysunek 2.2: Pomyślna weryfikacja serwera



```
Session port is: 16327
Start connection to ip: 127.0.0.1 on port: 16327
Send client public key: MIGfMA0GCSqGSIB3DQEBAQUAA4
Server public key received
Server is corrupted
```

Rysunek 2.3: Niepomyślna weryfikacja serwera

Rozdział 3

Komunikacja

3.1 Opis komunikacji pomiędzy klientem a serwerem

Komunikacja z serwerem polega na utworzeniu zapytania do serwera oraz otrzymaniu odpowiedzi na dane zapytanie. Zapytanie oraz odpowiedź są zdefiniowane w formacie JSON. Obiekt zapytania składa się z pola *action* oraz pola *body*. W polu *action* określany jest typ wykonywanej akcji, a w polu *body* dodatkowe parametry wymagane przy danych akcjach.

Dostępne akcje to:

- *register* - rejestracja użytkownika,
- *login* - logowanie użytkownika,
- *message* - wysłanie wiadomości w zadanej konwersacji,
- *listUsers* - pobranie listy użytkowników,
- *listConversations* - pobranie listy konwersacji,
- *createConversation* - stworzenie nowej konwersacji,
- *getMessages* - pobranie wiadomości z danej konwersacji,
- *getConversationMessages* - pobranie wszystkich nieprzeczytanych wiadomości.

```
{  
  "action": "createConversation",  
  "body": {  
    "name": "string",  
    "users": [ "login1", "login2"]  
  }  
}
```

Listing 3.1: Zawartość zapytania *createConversation*

Obiekt odpowiedzi składa się z pól:

- *status* - zawiera status HTTP odpowiedzi,
- *response* - zawiera opis odrzucenia zapytania lub dodatkowy opis statusu zapytania,
- *body* - zawiera dodatkowe informacje, ich specyfikacja zależy od zadanego zapytania.

```
{  
  "status": "int",  
  "response": "string",  
  "body": {  
    "users": [ "login1", "login2"]  
  }  
}
```

Listing 3.2: Zawartość odpowiedzi na zapytanie *listUsers*

Dokładny opis zapytań i odpowiedzi można znaleźć w dołączonych plikach.

3.2 Implementacja komunikacji w aplikacji klienckiej

3.2.1 Opcje dla niezalogowanego użytkownika

Po otrzymaniu numeru portu sesji i zweryfikowaniu serwera w terminalu wyświetlają się opcje rejestracji użytkownika oraz logowania do wcześniej już utworzonego konta. W obu procesach należy wpisać wybrany login oraz hasło.

Proces rejestracji

W przypadku rejestracji, jeżeli konto o zadanym loginie już istnieje to zostaniemy poinformowani o takim stanie rzeczy oraz powrócimy do początkowego menu. Przypadek ten załączono na rysunku 3.2. Natomiast jeżeli takowy login jest dostępny to użytkownik jest automatycznie logowany i przenoszony do opcji zalogowanego użytkownika (rys 3.1).

```
Dostępne opcje:
1.Rejestracja
2.Logowanie
Twój wybór: 1

Rozpoczecie procesu  rejestracji!

Wprowadz login: test2

Wprowadz hasło: test
Konto zostało utworzone pomyślnie.
```

Rysunek 3.1: Pomyślna rejestracja konta

```
Dostępne opcje:
1.Rejestracja
2.Logowanie
Twój wybór: 1

Rozpoczecie procesu  rejestracji!

Wprowadz login: test2

Wprowadz hasło: test
Niepowodzenie wykonania operacji. Powod: Login is occupied
```

Rysunek 3.2: Niepomyślna rejestracja konta

Proces logowania

W przypadku logowania, jeżeli istnieje już taki użytkownik z podanym loginem i hasłem to użytkownik otrzymuje dostęp do opcji zalogowanego użytkownika (rys. 3.9). W przeciwnym wypadku jest informowany nie pomyślnym logowaniu i wraca do początkowego menu (rys. 3.3).

```
Dostępne opcje:
1.Rejestracja
2.Logowanie
Twój wybór: 2

Rozpoczęcie procesu logowania!

Wprowadz login: t

Wprowadz hasło: t

Niepowodzenie wykonania operacji. Powód: The authorization has not been successful
```

Rysunek 3.3: Niepomyślna rejestracja konta

3.2.2 Opcje dla zalogowanego użytkownika

Po zalogowaniu użytkownik ma dostęp do poniżej dostępnych opcji 3.9.

```
Dostępne opcje:
1.Utworzenie konwersacji
2.Wylistowanie listy konwersacji
3.Wyświetlenie wiadomości w wybranej konwersacji
4.Wysłanie wiadomości w wybranej konwersacji
5.Wyświetlanie dostępnych użytkowników
6.Wyloguj konto
Twój wybór:
```

Rysunek 3.4: Opcje zalogowanego użytkownika

Utworzenie konwersacji

Aby użytkownik mógł wysłać wiadomość do jednego lub kilku innych użytkowników musi utworzyć konwersację za pomocą tej opcji. Wprowadza się tutaj nazwę danej konwersacji, która musi być unikalna dla wszystkich konwersacji oraz pozostałych użytkowników tej konwersacji. Listę użytkowników można podejrzeć za pomocą innej opcji menu dla zalogowanego użytkownika (roz. 3.2.2), a cały proces zaprezentowano na rysunku 3.5.

```
Wprowadz nazwe konwersacji: convTest1

Wprowadz nazwe uzytkownika: test
```

Rysunek 3.5: Proces tworzenia konwersacji

Lista konwersacji

Sprawdzenie listy dostępnych konwersacji jest możliwe za pomocą tej opcji. Jej wykonanie zaprezentowano na rysunku 3.6.

```
Dostępne opcje:
1.Utworzenie konwersacji
2.Wylistowanie listy konwersacji
3.Wyświetlenie wiadomości w wybranej konwersacji
4.Wysłanie wiadomości w wybranej konwersacji
5.Wyświetlanie dostępnych użytkowników
6.Wyloguj konto
Twój wybór: 2

[Conversation{name='convTest1', users=[test]}, Conversation{name='convTest2', users=[test]}]
```

Rysunek 3.6: Listowanie konwersacji

Wyświetlenie wiadomości w wybranej konwersacji

Po podaniu nazwy konwersacji wykonywane jest wyświetlenie wszystkich wiadomości w konwersacji (rys. 3.7).

```
1.Utworzenie konwersacji
2.Wylistowanie listy konwersacji
3.Wyświetlenie wiadomości w wybranej konwersacji
4.Wysłanie wiadomości w wybranej konwersacji
5.Wyświetlanie dostępnych użytkowników
6.Wyloguj konto
Twój wybór: 3

Wprowadz nazwe konwersacji: convTest1

[Message{author='test2', dateTime='2022-11-09T16:24:44.344624+01:00', content='halo'}]
```

Rysunek 3.7: Wyświetlenie wiadomości w wybranej konwersacji

Wysłanie wiadomości w wybranej konwersacji

Aby wysłać wiadomość należy wybrać tę opcję, wskazać docelową konwersację oraz wprowadzić zawartość wiadomości (rys. 3.8).

```
1.Utworzenie konwersacji
2.Wylistowanie listy konwersacji
3.Wyświetlenie wiadomości w wybranej konwersacji
4.Wysłanie wiadomości w wybranej konwersacji
5.Wyświetlanie dostępnych użytkowników
6.Wyloguj konto
Twój wybór: 4

Wprowadz nazwe konwersacji: convfest1

Wprowadz wiadomosc: halo
```

Rysunek 3.8: Wysłanie wiadomości w wybranej konwersacji

Wyświetlenie dostępnych użytkowników

Wyświetlenie wybranych użytkowników odbywa się poprzez wybór tej opcji. Jej działa nie zaprezentowano na rysunku 3.2.2.

```
Dostępne opcje:
1.Utworzenie konwersacji
2.Wylistowanie listy konwersacji
3.Wyświetlenie wiadomości w wybranej konwersacji
4.Wysłanie wiadomości w wybranej konwersacji
5.Wyświetlanie dostępnych użytkowników
6.Wyloguj konto
Twój wybór: 5

Lista dostępnych klientów:
[root, test, test2]
```

Rysunek 3.9: Wyświetlenie dostępnych użytkowników

Wylogowanie z konta

Ostatnią opcją jest wylogowanie z konta, czego skutkiem jest powrót do menu dla nie zalogowanego użytkownika (3.2.1).

3.3 Bazowe implementacja zapytania

Poniżej załączono fragment bazowej klasy dla zapytań z serwerem. Przedstawiony kod przedstawia dwie deklaracje przeciążenia metody *run*, która odpowiada za tworzenie zapytań do serwerów z wykorzystaniem wcześniej stworzonego obiektu *body* lub z wykorzystaniem informacji z obiektu *CommunicatorClient* do stworzenia nowego obiektu *body*. Dodatkowo w klasie umieszczono metodę komunikującą się z serwerem.

```
/**
 * Abstract class for actions with server
 */
public abstract class AbstractAction {
    [...]
    /**
     * Abstract method which will create JSON Objects of action
     * using @see Body Creator
     * @param communicatorClient reference to communicator to get
     * some useful data
     * @return readied response from server
     */
    public abstract RespAbstract run(CommunicatorClient
        communicatorClient);

    /**
     * Abstract method which will create JSON Objects of action
     * using given body
     * @param body reference to body
     * @return readied response from server
     */
    public abstract RespAbstract run(Body body);

    /**
     * Method which sends action to server and read response.
     * @param data which will be sent
     * @return response from server
     */
    protected String communication(RequestData data) {
        try {
```

```
        var encryptedMsg = aes.encrypt(new
            JSONObject(data).toString());
        return
            aes.decrypt(connectionLayer.sendMessage(encryptedMsg));
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Listing 3.3: Bazowa klasa dla zapytań z serwerem