

Bezp. syst. i usług inform. 2

Komunikator z szyfrowaniem

Mateusz Mańka 209895

22 października 2016

1. Cel projektu

Celem projektu jest przygotowanie komunikatora w architekturze klient-serwer wspierającego bezpieczną wymianę sekretu wg protokołu Diffiego-Hellmana oraz obsługujący zadany format komunikacji.

2. Wymagania

2.1. Protokół komunikacji

W celu zapewnienia kompatybilności z innymi realizacjami tego projektu, ustalony został protokół komunikacji, według którego powinna odbywać się wymiana wiadomości w aplikacji. Wiadomości przesyłane pomiędzy klientem a serwerem powinny być zgodne z formatem danych opartych o JSON przedstawiony:

Stage	Klient	Serwer
1	{ "request": "keys" }	
2		{ "p": 123, "g": 123 }
3	{ "a": 123 }	{ "b": 123 }
4	{ "encryption": "none" }	
5	{ "msg": "...", "from": "John" }	{ "msg": "...", "from": "Anna" }

Podczas wymiany wiadomości należy uwzględnić podane wymagania:

- W kroku 3. wiadomości od serwera i klienta mogą nastąpić w dowolnej kolejności.
- Krok 4. tabeli 1 jest opcjonalny. W przypadku braku wysłania wiadomości o metodzie szyfrowania należy przyjąć wartość domyślną none.

2.2. Szyfrowanie

Wiadomości przesyłane między klientami a serwerem powinny być szyfrowane według metody wybranej przez użytkownika. Komunikator powinien wspierać następujące metody szyfrowania:

- none - brak szyfrowania (domyślne)
- xor - szyfrowanie OTP xor jednobajtowe (należy użyć najmłodszego bajtu sekretu)
- cezar - szyfr cezara

Komunikator powinien zapewniać możliwość zmiany metody szyfrowania w dowolnym momencie. Treść wiadomości powinna być zakodowana za pomocą base64 przed umieszczeniem jej w strukturze JSON:

```
b a s e 6 4 ( e n c r y p t ( u s e r m e s s a g e ) )
```

3. Projekt i implementacja

Aplikacja wykonana została w języku Java(wersja 1.8) z wykorzystaniem Swinga do stworzenia interfejsów a także technologii Maven w celu łatwiejszego zarządzania pluginami, pluginem Gson do tworzenia pluginu. Projekt został zaimplementowany w środowisku IntelliJ IDEA.

3.1. Koncepcja Rozwiązania

Część serwerowa i kliencka aplikacji zostaną wykonane w ramach jednego projektu lecz różnych pakietów (Client , Server). Aby wybrać funkcje klienta należy odpalić plik Client.java , aby odpalić funkcje Serwera należy odpalić plik Server.java. Po skonfigurowaniu parametrów połączenia aplikacja rozpocznie działanie zależnie od wybranej funkcji według jednego ze schematów:

Klient :

- Aplikacja tworzy połączenie z zadaniem adresem na wskazanym porcie(W pliku Commons.java) .
- Po utworzeniu połączenia rozpoczyna się wymiana kluczy według protokołu opisanego w punkcie 2.1.
- Po ustaleniu kluczy szyfrowania aplikacja oczekuje na interakcję użytkownika.
- Użytkownik może wybrać

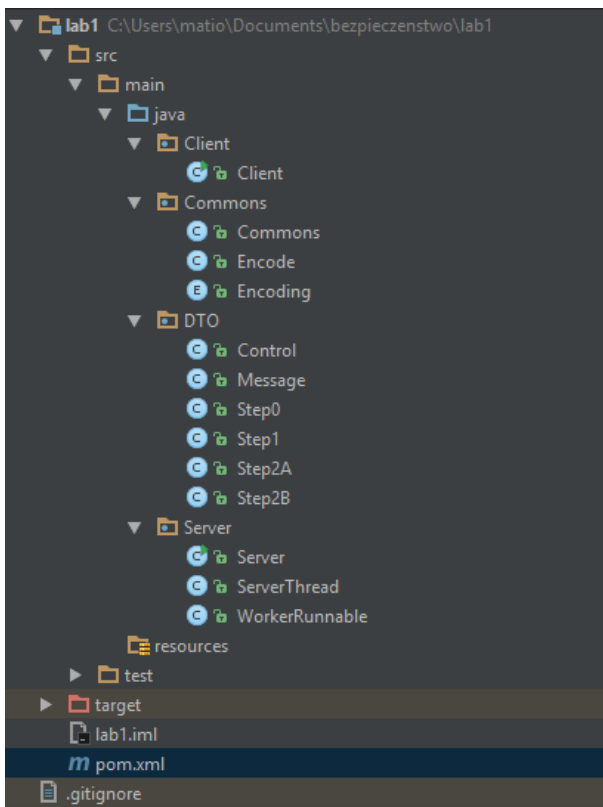
Serwer :

- Aplikacja tworzy nowy wątek oczekujący na połączenia.
- Po ustanowieniu połączenia z klientem serwer tworzy dla niego wątek nasłuchujący i odpowiada na zapytania ze strony klienta według ustalonego protokołu.
- Po otrzymaniu wiadomości z jednego z aktywnych wątków klienckich, serwer odszyfrowuje wiadomość według metody wybranej przez nadawcę.
- Serwer wyświetla wiadomość w konsoli a następnie zwraca ją, ponownie szyfrując ją.

3.2. Struktura projektu

Projekt składa się z 4 pakietów 13 klas i pliku pom.xml do konfiguracji mavena i pluginów.

Pakiety :



1. Client zawiera Klase:
 - 1.1. Client która zawiera wszystkie metody do komunikacji z serwerem wraz z GUI w Swingu
2. Commons klasy wspólne :
 - 2.1. Commons wspólne wartości odnośnie adresu serwera a także wspólne metody dla socketów
 - 2.2. Encode wspólne metody kodujące
 - 2.3. Encoding Enum metod kodowania
3. DTO Data transwear Objects wykorzystywane do budowania Jsonów opisanych w punkcie 2.1
4. Server zawiera klasy :
 - 4.1. Server klasa z metodą main do uruchomienia serwera
 - 4.2. ServerThread : Nasłuchiwanie na nowy socket
 - 4.3. WorkerRunnable : Obsługiwanie konkretnego klienta (Wątek)

3.3. Nawiązywanie połączenia i wymiana kluczy :

Uruchomienie serwera:

```
ServerThread server = new ServerThread(PORT);  
new Thread(server).start();  
while(!server.isStopped){  
}
```

Utworzenie wątku ServerThread ,
uruchomienie go (metoda start wywołująca
metode run) i oczekiwanie na zamknięcie.

Metoda run w klasie ServerThread :

```
public void run() {  
    System.out.println("Server Starting");  
    synchronized (this) {  
        this.runningThread = Thread.currentThread();  
    }  
    openServerSocket();  
    while (!isStopped()) {  
        try {  
            Socket clientSocket = this.serverSocket.accept();  
            new Thread(new WorkerRunnable(clientSocket, "Multithreaded Server")).start();  
        } catch (IOException e) {  
            if (isStopped()) {  
                System.out.println("Server Stopped.");  
                return;  
            }  
            throw new RuntimeException("Error accepting client connection", e);  
        }  
    }  
    System.out.println("Server Stopped.");  
}
```

Metoda ta za pomocą metody klasy Socket accept() oczekuje na połączenie z klienta , w przypadku połączenia tworzy nowy wątek WorkerRunnable i uruchamia go , a następnie oczekuje na kolejne połączenie do momentu zatrzymania (pętla while)

WorkerRunnable :

```
public WorkerRunnable(Socket clientSocket, String serverText) {  
    System.out.println("Połączono");  
    this.clientSocket = clientSocket;  
    this.serverText = serverText;  
    gson = new Gson();  
    generateBNumber();  
    generateKeys();  
}
```

WorkerRunnable generuje swoją liczbę b dla tego klienta a także generuje klucze p i g.

Klient :

```
Socket skt = new Socket(HOST, PORT);  
InputStream input = skt.getInputStream();  
OutputStream output = skt.getOutputStream();  
System.out.println("Connected");  
generateMyAValue();  
System.out.println("Generated A");  
sendRequestForKeys(output, input);  
Step2B step2B = sendMyAValueAndWaitForB(output, input);  
calculateSecret(step2B);  
createGUI(output, input);
```

Tworząc obiekt Socket tworzymy
połączenie z serwerem.
następnie generujemy wartość a
(prywatną), wysyłamy prośbę o
klucze i czekamy.
w sendMyValueAndWaitForB
obliczamy wartość A wysyłamy i
oczekujemy na wartość B i
obliczamy secret.

Serwer :

```
InputStream input = clientSocket.getInputStream();
OutputStream output = clientSocket.getOutputStream();
waitForKeys(input);
sendKeys(output, step1);
Step2A step2A = sendBAndWaitForA(input, output);
calculateSecret(step2A);
waitForMessagesOrControl(input, output);
```

Oczekujemy na prośbę na klucze i wysyłamy je, następnie oczekujemy na wartość A i wysyłamy swoją obliczoną wartość B i obliczamy Secret . I oczekujemy na wiadomość lub zmianę kodowania.

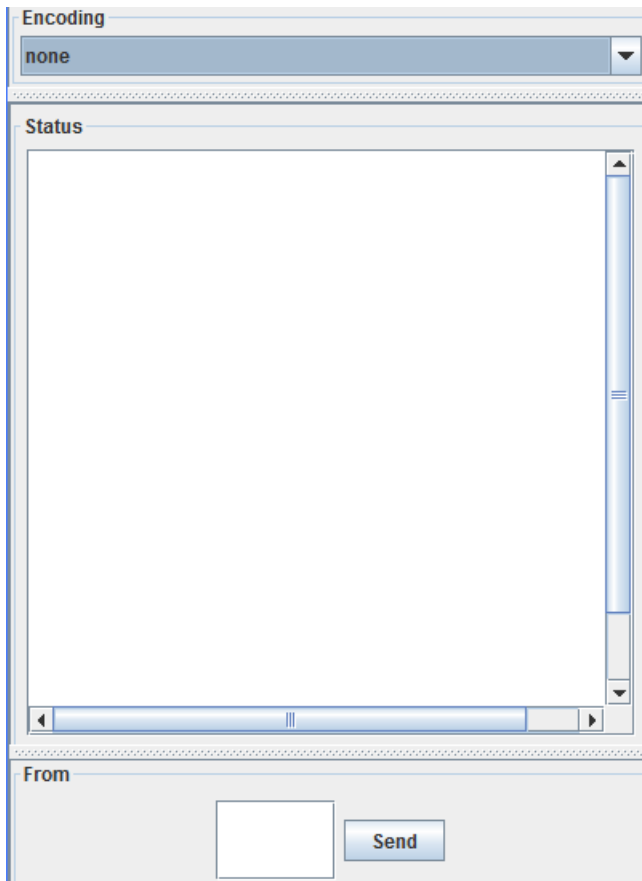
3.4. Obsługa wiadomości :

Serwer :

```
while(running){
    String msg = readJsonAndSendOne(input, null, null);
    System.out.println("Dostałem : " + msg);
    Message message = gson.fromJson(msg, Message.class);
    if(message == null || message.getMsg() == null){
        handleControlMessage(msg);
    }else{
        handleMessage(output, message);
    }
}
```

Oczekuje wiadomości następnie stara się ją sparsować na Message jeśli się nie uda oznacza to że jest to wiadomość kontrolna.

Klient:



Za pomocą gui może zmienić kodowanie za pomocą ComboBox Encoding
Lub napisać wiadomość w miejscu Status, odpisać swoje imię w miejscu FROM i wysłać wiadomość za pomocą klawisza Send, wiadomość zwrotna zostanie wyświetlona w miejscu status.

3.5. Tworzenie Jsona

Json zostaje tworzony za pomocą biblioteki GSON z wcześniej zdefiniowanych klas DTO

Przykład :

Wiadomość :

```
public class Message {  
    public Message(String msg, String from) {  
        this.msg = msg;  
        this.from = from;  
    }  
    private String msg;  
    private String from;  
}
```

Posiada pola msg i from który później uzupełniamy i parsujemy na JSON. To rozwiązanie ułatwia zmianę formy jsona w przyszłości.

Zamiana obiektu Message na Json:

```
gson.toJson(message)
```

3.6. Wysyłanie i odbieranie obiektu JSON

Wysyłanie

```
public static boolean writeJson(OutputStream output, String msg) {  
    System.out.println("Sending" +msg);  
    byte[] message = msg.getBytes();  
    try {  
        output.write(message);  
        output.flush();  
        return true;  
    } catch (IOException e) {  
        e.printStackTrace();  
        return false;  
    }  
}
```

Wysyłanie polega na zamianie obiektu String na ciąg bitów i wypisanie go na output stream(+ obsługa wyjątków)

Odbieranie

```
public static String readJsonAndSendOne(InputStream input, OutputStream output, String sendMsg) throws SocketException {
    List<Byte> bytes = new ArrayList<>();
    BufferedReader br = new BufferedReader(new InputStreamReader(input));
    String msg;
    StringBuilder json = new StringBuilder();
    try {
        boolean received = false;
        while(! (received && input.available() == 0)) {
            if (sendMsg != null) {
                writeJson(output, sendMsg);
                sendMsg = null;
            }
            int oneByte = input.read();
            if (oneByte > 0) {
                received = true;
                bytes.add((byte) oneByte);
            } else {
                break;
            }
            if (oneByte == 125)
                break;
        }
    } catch (SocketException e) {
        throw new SocketException();
    } catch (IOException e) {
        e.printStackTrace();
    }
    String received = decodeListOfBytes(bytes.toArray());
    System.out.println("Received : " + received);
    return received;
}
```

Jak widać metoda nie tylko odbiera ale także wysyła przed odebraniem wiadomość jeśli do metody zostanie jakaś przekazana. Metoda została tak utworzona aby nie powtarzać kodu i była używalna w taki sposób w wielu miejscach (np. przy wysyłaniu i odbieraniu A i B)

Metoda w pętli while czyta tak długo dopóki nie otrzymała chociaż jednego bitu i nie ma już więcej dostępnych bitów lub otrzymała znak zamknięcia JSON-a „}”. Opcja z „}” została dodana ponieważ podczas testów udało mi się wytworzyć sytuację w której w szybkim odstępie czasu dostałem dwa JSONy jeden z kluczem p i g a drugi z wyliczonym przez serwer B.

3.7. Testy

Zostało przeprowadzonych wiele testów manualnych w których wykryłem i poprawiłem wiele problemów aplikacji, dodatkowo w trakcie zajęć wraz z kolegą Adamem Zimnym przeprowadziliśmy test połączenia naszych dwóch aplikacji który odniósł sukces potwierdzając że nasze aplikacje działają zgodnie z protokołem ustalonym na zajęciach.

4. Wnioski :

- Używanie integera zamiast BigInteger (long) powodowało wiele problemów związanych z niedokładnością obliczeń (np. przy wyliczaniu wielkich potęg)
- Klasa BigInteger pozwala na szybsze obliczanie sekretu za pomocą wbudowanej funkcji powMod (potęga z modulo)
- Klasa Javy AlgorithmParameterGenerator pozwala w łatwy sposób wygenerować p i g

5. Podsumowanie :

Przygotowana aplikacja spełnia wymagania projektu:

- Aplikacja poprawnie łączy się przez sieć z wykorzystaniem gniazd,
- Zaimplementowany został bezpieczny protokół wymiany kluczy Diffiego-Hellmana,
- Zaimplementowane zostały dwie metody szyfrowania (xor, cezarski)

- Poprawnie obsługiwany jest zadany protokół komunikacyjny. Aplikacja poprawnie komunikuje się z innymi implementacjami przygotowanymi przez uczestników laboratorium,
- Metody szyfrowania można zmienić przed wysłaniem każdej wiadomości.