Bezp. syst. i usług inform. 2
Komunikator z szyfrowaniem

Adam Zimny 209787

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 przedstawionym w tabeli 1. W celu rozpoczęcia komunikacji klient i serwer muszą wymienić następujące wiadomości:

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" }

Tabela 1: Struktura wiadomości protokołu komunikacyjnego aplikacji

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 wspirać 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:

```
base64(encrypt(user_message))
```

3 Projekt i implementacja

Aplikacja wykonana została w języku Java w wersji 8, z wykorzystaniem technologii JavaFX oraz Swing do stworzenia interfejsów użytkownika odpowiednio dla aplikacji klienckiej i serwerowej. Do implementacji projektu wykorzystane zostało środowisko programistyczne IntelliJ IDEA. Projekt został wykonany z wykorzystaniem repozytorium kodu w serwisie GitHub.

3.1 Koncepcja rozwiązania

Część serwerowa i kliencka aplikacji zostaną wykonane w ramach jednego projektu, pozwalając uzytkownikowi wybrać funkcje uruchomionego programu na początku jego działania. Po skonfigurowaniu parametrów połączenia aplikacja rozpocznie działanie zależnie od wybranej funkcji według jednego ze schematów:

Klient

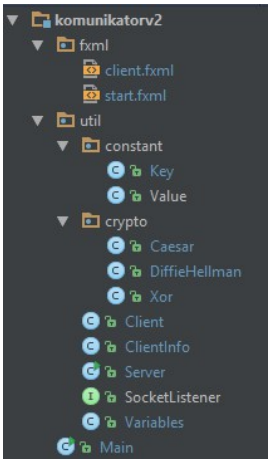
1. Aplikacja tworzy połączenie z zadanym adresem na wskazanym porcie.
2. Po utworzeniu połączenia rozpoczyna się wymiana kluczy według protokołu opisanego w punkcie 2.1.
3. Po ustaleniu kluczy szyfrowania aplikacja oczekuje na interakcję użytkownika.

Serwer

1. Aplikacja tworzy nowy wątek oczekujący na połączenia.
2. Po ustanowieniu połączenia z klientem serwer tworzy dla niego wątek nasłuchujący i odpowiada na zapytania ze strony klienta według ustalonego protokołu.
3. Po otrzymaniu wiadomości z jednego z aktywnych wątków klienckich, serwer odszyfrowuje wiadomość według metody wybranej przez nadawcę.
4. Serwer rozgłasza wiadomość do wszystkich klientów, ponownie szyfrując ją indywidualnie dla każdego klienta w wykorzystaniem jego preferowanej metody szyfrowania.

3.2 Struktura projektu

Projekt składa się z 11 klas i 2 plików deklaracji interfejsu o rozszerzeniu **.fxml** odpowiednim dla technologii JavaFX. Pliki zostały podzielone w pakiety według funkcjonalności, jakie implementują. Struktura projektu przedstawiona jest na rysunku 3.2.



Rysunek 1: Struktura plików projektu

Zawartość istatniejszych pakietów i plików oraz ich funkcje w projekcie opisuje poniższa lista:

fxml - pakiet zawierający pliki interfejsu użytkownika

constant - pakiet zawierający definicje stałych dla kluczy i wartości używanych w protokole komunikacyjnym aplikacji

crypto - pakiet zawierający klasy realizujące funkcjonalność szyfrowania wiadomości

Client - klasa implementująca funkcjonalności klienta, tworzy połączenie z serwerem, nasłuchuje nowych wiadomości z gniazda oraz zapewnia obsługę zdarzeń interfejsu użytkownika

ClientInfo - struktura przechowująca wartości parametrów używanych w protokole Diffiego-Hellmana

Server - klasa implementująca funkcjonalności serwera, nasłuchuje nowych wiadomości i rozgłasza je do wszystkich podłączonych klientów nakładając odpowiednie dla klienta szyfrowanie

SocketListener - pakiet zawierający klasy realizujące funkcjonalność szyfrowania wiadomości

Variables - klasa przechowująca statyczne wartości parametrów aplikacji, takich jak adres IP lub port, w celu łatwiejszego ich przekaywania wewnątrz aplikacji

3.3 Nawiązanie połączenia

Do nawiązania połączenia wykorzystane zostały klasy `Socket` oraz `ServerSocket` z pakietu `java.net` . Nawiązywanie połączenia w aplikacji należy rozpocząć od uruchomienia serwera. Listing 1 przedstawia procedurę uruchomienia. Po uruchomieniu wywoływana jest pętla, która akceptuje kolejne połączenia i tworzy dla nich wątki obsługujące zdarzenia. Wątki gromadzone są w kolekcji, aby umożliwić późniejszy dostęp do nich w celu rozgłaszania wiadomości. Na ekranie wyświetlany jest komunikat o adresie IP i porcie, na który powinnny łączyć się aplikacje klienckie.

```
public void start(int port) {
    try {
        ServerSocket server = new ServerSocket(port);
        DiffieHellman.initPrimes(DiffieHellman.primeSize);
        display("Waiting for client connections on "
                + InetAddress.getLocalHost().getHostAddress() + ":"
                + server.getLocalPort());
        while (true) {

            Socket conn = server.accept();

            ClientThread t = new ClientThread(conn);
            cList.add(t);
            t.start();
        }
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

Listing 1: Metoda nasłuchująca połączeń klienckich

Wątek `ClientThread` utworzony po połączeniu z klientem pobiera z utworzonego gniazda strumienie wejściowy i wyjściowy, a także inicjalizuje parametry protokołu Diffiego-Hellmana. Konstruktor klasy przedstawiono na listingu 2. Wątek nasłuchuje wiadomości wysyłanych przez uzyskany strumień i odpowiada na nie poprzez wywołanie metody `onMessage` interfejsu `SocketListener` , której implementację przedstawia listing 6 w punkcie 3.4. Pętla nasłuchująca przedtawiona została na listingu 3.

```
ClientThread(Socket socket) {

        try {
            info.setId(++uniqueId);
            info.setSecretB(DiffieHellman.getInitialSecret());
            info.setP(DiffieHellman.generateP());
            info.setG(DiffieHellman.generateG());
            info.setB(DiffieHellman.makeB(info));
```

```
                this.socket = socket;
                in = new BufferedReader(new InputStreamReader(
                        socket.getInputStream()));
                out = new PrintWriter(socket.getOutputStream(), true);
                display("User " + info.getId() + " connected from "
                        + socket.getRemoteSocketAddress() + ".");
            } catch (IOException ioe) {
                display("Error creating in/out in ClientThread");
            } catch (NoSuchAlgorithmException | InvalidParameterSpecException e) {
                e.printStackTrace();
            }
        }
```

Listing 2: Konstruktor wątku ClientThread

```
    public void run() {
            String line;
            while (true) {
                try {
                    line = in.readLine();
                    System.out.println("Received message:\n\t" + line);
                    onMessage(line);
                } catch (IOException e) {
                    System.out.println(e);
                    break;
                }
            }
        }
```

Listing 3: Metoda nasłuchująca strumienia w wątku ClientThread

Po stronie klienta po ustanowieniu połączenia również otwierane są strumienie danych oraz tworzony jest wątek nasłuchujący wiadomości wysyłanych przez serwer. Implementację tej funkcjonalności przedstawia listing 4. Po otrzymaniu wiadomości wywoływana jest metoda `onMessage` zaimplementowana w klasie Client, pokazana w punkcie 3.4, w listingu 5.

```
public void startClient(String ip, int port) {
        info = new ClientInfo();
        info.setPort(port);
        info.setSecretA(DiffieHellman.getInitialSecret());
        updateInfo();
        try {
            socket = new Socket(ip.trim(), port);
            in = new BufferedReader(new InputStreamReader(
                    socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
            new ChatListener(this).start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


private class ChatListener extends Thread {
        SocketListener listener;
        ChatListener(SocketListener listener) {
            this.listener = listener;
        }

        @Override
        public void run() {
            send(Key.REQUEST, Value.KEYS);

            final String line[] = new String[1];
```

```
            while (true) {
                try {
                    line[0] = in.readLine();
                    System.out.println("Received message:\n\t" + line[0]);
                    Platform.runLater(() -> listener.onMessage(line[0]));
                } catch (IOException e) {
                    e.printStackTrace();

                    return;
                }
            }
        }
    }
```

Listing 4: Metody tworząca gniazdo po stronie klienta

3.4 Obsługa wiadomości

Obsługą wiadomości przesyłanych w aplikacji zajmuje się interfejs `SocketListener` . Jego implementacje znajdują się w klasach `Client` oraz `ClientThread` . Interfejs posiada jedną metodę `onMessage( String json)` , której parametrem jest wiadomość otrzymana ze strumienia. Otrzymany ciąg znaków interpretowany jest przez klasę `JSONObject` z biblioteki `org.json` , a następnie na podstawie kluczy znajdujących się w obiekcie podejmowana jest odpowiednia akcja. Implementacje tej metody w wymienionych klasach przedstawiono poniżej.

3.4.1 Client

```
@Override
public void onMessage(String line) {
    JSONObject json = new JSONObject(line);
    if (json.has(Key.MESSAGE)) {
        String message = json.getString(Key.MESSAGE);
        message = new String(Base64.getDecoder().decode(message),
                             StandardCharsets.UTF_8);
        chatWindow.getItems().add(json.get(Key.FROM)+": "+decrypt(message));
    }
    if (json.has(Key.A_KEY)) {

        System.out.println("Client should never receive A value!");
    }
    if (json.has(Key.B_KEY)) {
        info.setB(json.getBigInteger("b"));
        if (info.isReady()) {
            info.setS(DiffieHellman.makeClientSecret(info));
        }
        updateInfo();
    }
    if (json.has(Key.P_KEY)) {

        info.setP(json.getBigInteger("p"));
        info.setG(json.getBigInteger("g"));
        info.setA(DiffieHellman.makeA(info));
        JSONObject aJson = new JSONObject();
        aJson.put(Key.A_KEY, info.getA());
        out.println(aJson.toString());
        out.flush();
        updateInfo();
    }
    if (json.has(Key.ENCRYPTION)) {
        if (info.isReady()) {
            info.setS(DiffieHellman.makeClientSecret(info));
```

```
                updateInfo();
            }
        }
    }
```

Listing 5: Metoda obsługi wiadomości w klasie Client

3.4.2 ClientThread

```
  @Override
public void onMessage(String line) {
        JSONObject json = new JSONObject(line);
        display(line);
        if (json.has(Key.MESSAGE)) {
                String encoded = json.getString(Key.MESSAGE);
                String encrypted = new String(Base64.getDecoder().decode(encoded),
                                              StandardCharsets.UTF_8);
                String message = decrypt(encrypted);
                String name = json.getString(Key.FROM);
                broadcast(message,name);
        }
   if (json.has(Key.REQUEST)) {
                if (json.getString(Key.REQUEST).equals(Value.KEYS)) {
                        JSONObject pgJson = new JSONObject();
                        try {
                                pgJson.put(Key.P_KEY, DiffieHellman.generateP());
                                pgJson.put(Key.G_KEY, DiffieHellman.generateG());
                        } catch (NoSuchAlgorithmException |
                        InvalidParameterSpecException e) {
                                e.printStackTrace();
                        }
                        sendJson(pgJson.toString(), this);
                        try {
                                sleep(500);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        JSONObject bJson = new JSONObject();
                        bJson.put(Key.B_KEY, info.getB());
                        sendJson(bJson.toString(), this);
                }
        }
        if (json.has(Key.A_KEY)) {
                info.setA(json.getBigInteger("a"));
                if (info.isReady()) {
                        info.setS(DiffieHellman.makeServerSecret(info));
                        display(info.toString());
                }
        }
        if (json.has(Key.B_KEY)) {
                System.out.println("Server should never receive B value!");
        }
        if (json.has(Key.P_KEY)) {
                System.out.println("Server should never receive p or g value!");
        }
        if (json.has(Key.ENCRYPTION)) {
                String en = json.getString(Key.ENCRYPTION);
                if (en.equals(Value.CAESAR) || en.equals(Value.NONE) ||
                        en.equals(Value.XOR))
                        info.setEncryption(en);
        }
}
```

Listing 6: Metoda obsługi wiadomości w klasie ClientThread

3.5 Szyfrowanie

W aplikacji zaimplementowane zostały dwie metody szyfrowania wiadomości: szyfrowanie XOR oraz szyfr cezara. Implementacje metod szyfrujących przedstawione zostały na listingach 7 oraz 8 znajdujących się poniżej. Ze względu na odwracalność operacji `xor` szyfrowanie i deszyfrowanie wiadomości można wykonać przy uzyciu tej samej metody.

```
public class Xor {

    public static byte[] encrypt(String string, BigInteger secret) {
        byte[] a = string.getBytes();
        byte[] key = secret.toByteArray();
        byte[] out = new byte[a.length];
        for (int i = 0; i < a.length; i++) {
            out[i] = (byte) (a[i] ^ key[key.length-1]);
        }
        return out;
    }
}
```

Listing 7: Szyfrowanie XOR

```
public class Caesar {
    public static String encrypt(String string, BigInteger shift) {
        StringBuilder encrypted = new StringBuilder();

        for (char c : string.toCharArray()) {
            int ascii = (int) c;
            ascii += (shift.mod(BigInteger.valueOf(26))).intValue();
            if (ascii > (Character.isLowerCase(c) ? 'z' : 'Z')) ascii -= 26;
            encrypted.append((char) ascii);
        }
        return encrypted.toString();
    }

    public static String decrypt(String string, BigInteger shift) {

        StringBuilder decrypted = new StringBuilder();
        for (char c : string.toCharArray()) {
            int ascii = (int) c;
            ascii -= (shift.mod(BigInteger.valueOf(26))).intValue(); if (ascii < (
                Character.isLowerCase(c) ? 'a' : 'A')) ascii += 26;
            decrypted.append((char) ascii);
        }
        return decrypted.toString();
    }
}
```

Listing 8: Szyfr cezara

4 Podsumowanie

Przygotowana aplikacja spełnia wymagania projektu:

- Aplikacja poprawnie łączy się przez sieć z wykorzystaniem gniazd,
- Zaimplmenetowany został bezpieczny protokół wymiany kluczy Diffiego-Hellmana,
- Zaimplementowane zostały dwie metody szyfrowania,
- 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,